Scalable Data Mining for Rules

by

Mohammed Javeed Zaki

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Dr. Wei Li

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

1998

To my parents and brothers.

Read! in the name of thy Lord and Cherisher, Who created
Created man, out of a (mere) clot of congealed blood:
Read! And thy Lord is Most Bountiful,
He Who taught (the use of) the pen,
Taught man that which he knew not...
Holy Quran, Sura 96, Ayat 1-5

Curriculum Vitae

Mohammed Javeed Zaki was born in Hyderabad, India, on August 24th, 1971. He attended Angelo State University, San Angelo, TX, from 1989 to 1993, and graduated with a Bachelor of Science degree in 1993 with a dual major in Computer Science and Mathematics. He came to the University of Rochester in the Fall of 1993 and began graduate studies in Computer Science. He pursued his research in parallel data mining and knowledge discovery under the direction of Dr. Wei Li and received the Master of Science degree in 1995.

Acknowledgments

I want to thank Dr. Wei Li for getting me started on data mining and for guiding me through this thesis. Special thanks go to Dr. Mitsunori Ogihara, who was like a co-advisor on this thesis, and to the other members of my thesis committee, Dr. Alexander Albicki, Dr. Thomas LeBlanc, and Dr. Michael Scott for their helpful comments. Thanks are also due to Dr. James Allen for providing summer support, and to Dr. Howard Ho and Dr. Rakesh Agrawal for a valuable summer spent with the Quest data mining group at IBM Almaden Research Center. Thanks to Dr. Mitsunori Ogihara, Srini Parthasarathy and Dr. Neal Lesh for the collaborations on various papers.

The five years at Rochester were very enjoyable. I'll always remember my classmates Wagner Meira, Jim Vallino, Garbis Salgian, Mark Core, Aaron Kaplan, Umesh Berry, Colm O'Rian, George Kardaras, Mauricio Marengoni, Mike Marchetti, and Terry Riopka. Thanks to my raquetball and squash partners Jim Vallino, Olac Fuentes, Srini Parthasarathy and Dana Ballard. The desi gang, which included Harmit Malik, Srini Parthasarathy, Amit Singhal, Samantha Baker, Rajesh Rao and Ramesh Sarukkai, made life in Rochester unforgettable.

Finally, I am grateful to my parents for always encouraging me to do my best, and also to my brothers for their encouragement and support. All thanks and praises to God Almighty for guiding and blessing me in this endeavor.

This research was supported in part by NSF research initiation award no. CCR-9409120, NSF research grant no. CCR-9705594, ARPA contract no. F19628-94-C-0057, and U.S. Air Force/Rome Labs research contract no. F30602-95-1-0025. The rights to the work on mining classification rules are owned by IBM Corporation.

Abstract

Data Mining is the process of automatic extraction of novel, useful, and understandable patterns in very large databases. High-performance scalable and parallel computing is crucial for ensuring system scalability and interactivity as datasets grow inexorably in size and complexity. This thesis deals with both the algorithmic and systems aspects of scalable and parallel data mining algorithms applied to massive databases. The algorithmic aspects focus on the design of efficient, scalable, disk-based parallel algorithms for three key rule discovery techniques — *Association Rules*, *Sequence Discovery*, and *Decision Tree Classification*. The systems aspects deal with the scalable implementation of these methods on both sequential machines and popular parallel hardware ranging from shared-memory systems (SMP) to hybrid hierarchical clusters of networked SMP workstations.

The association and sequence mining algorithms use lattice-theoretic combinatorial properties to decompose the original problem into small independent sub-problems that can be solved in main memory. Using efficient search techniques and simple intersection operations all frequent patterns are enumerated in a few database scans. The parallel algorithms are asynchronous, requiring no communication or synchronization after an initial set-up phase. Furthermore, the algorithms are based on a hierarchical parallelization, utilizing both shared-memory and message-passing primitives. In classification rule mining, we present disk-based parallel algorithms on shared-memory multiprocessors, the first such study. Extensive experiments have been conducted for all three problems, showing immense improvement over previous approaches, with linear scalability in database size.

Table of Contents

List of Tables

List of Figures

1 Introduction

Data Mining and Knowledge Discovery in Databases (KDD) is a new interdisciplinary field merging ideas from statistics, machine learning, databases, and parallel computing. It has been engendered by the phenomenal growth of data in all spheres of human endeavor, and the economic and scientific need to extract useful information from the collected data. The key challenge in data mining is the extraction of knowledge and insight from massive databases, and it can be defined as follows.

> Data Mining is the process of automatic extraction of novel, useful, and understandable patterns in large databases.

Data mining refers to the overall process of discovering new patterns or building models from a given dataset. There are many steps involved in the KDD enterprise which include data selection, data cleaning and preprocessing, data transformation and reduction, data-mining task and algorithm selection, and finally post-processing and interpretation of discovered knowledge [Fayyad *et al.*, 1996a; Fayyad *et al.*, 1996b]. This KDD process tends to be highly iterative and interactive. Figure 1.1 illustrates some of the KDD steps.

Typically data mining has the two high level goals of *prediction* and *description* [Fayyad *et al.*, 1996a]. In prediction, we are interested in building a model that will predict unknown or future values of attributes of interest, based on known values of some attributes in the database. In KDD applications, the description of the data in human-understandable terms is equally if not more important than prediction. Two main forms of data mining can be identified [Simoudis, 1996]. In *verification-driven* data mining the user postulates a hypothesis, and the system tries to validate it. The common verification-driven operations include query and reporting, multidimensional analysis, and statistical analysis. *Discovery-driven* mining, on the other hand automatically extracts new information, and it forms the main focus of this thesis. The typical discovery-driven tasks include:

- **Association Rules:** Given a database of transactions, where each transaction consists of a set of items, association discovery finds all the item sets that frequently occur together, and also the rules among them [Agrawal *et al.*, 1993b; Agrawal *et al.*, 1996]. An example of an association could be that, “40% of people who buy Jane Austen’s *Pride and Prejudice* also buy *Sense and Sensibility.*”

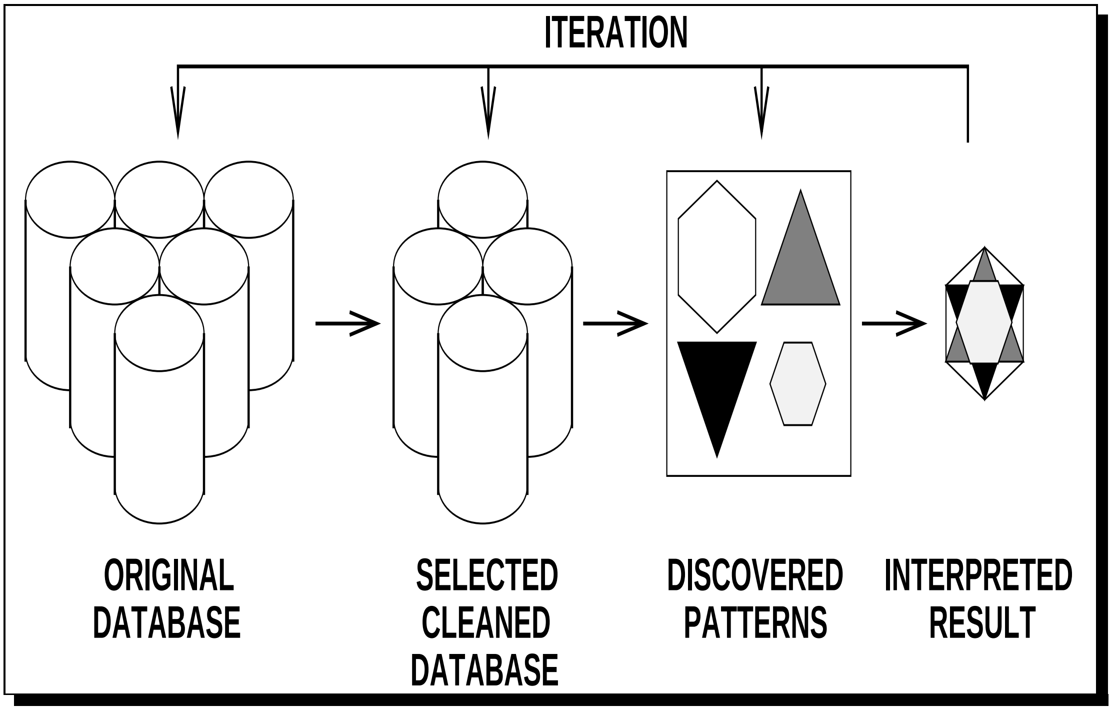


Figure 1.1: Data Mining Process

Potential application areas include catalog design, store layout, customer segmentation, telecommunication alarm diagnosis, etc.

- **Sequential Patterns:** Sequence discovery aims at extracting sets of events that commonly occur over a period of time [Agrawal and Srikant, 1995]. An example of a sequential pattern could be that "70% of the people who buy Jane Austen's *Pride and Prejudice* also buy *Emma* within a month". Such patterns are useful in domains such as retail sales, telecommunications (*e.g.* alarm diagnosis) and medicine (*e.g.* identifying symptoms that precede diseases).

- **Classification and Regression:** Classification aims to assign a new data item to one of several predefined categorical classes [Weiss and Kulikowski, 1991; Michie *et al.*, 1994]. Since the field being predicted is pre-labeled, classification is also known as supervised induction. While there are several classification methods including neural networks [Lippmann, 1987] and genetic algorithms [Goldberg, 1989], decision trees [Breiman *et al.*, 1984; Quinlan, 1993] are particularly suited to data mining, since they can be constructed relatively quickly, and are simple and easy to understand. While classification predicts a categorical value, regression is applied if the field being predicted comes from a real-valued domain. Common applications of classification include credit card fraud detection, insurance risk analysis, bank loan approval, etc.

- **Clustering:** Clustering is used to partition a database into subsets or clusters, such that elements of a cluster share a set of common interesting properties that distinguish them from other clusters [Jain and Dubes, 1988; Cheeseman *et al.*, 1988; Fisher, 1987; Michalski and Stepp, 1983]. Unlike classification which has predefined labels, clustering must in essence automatically come up with the labels. For this reason clustering is also called unsupervised induction. Applications of clustering include demographic or market segmentation for identifying common traits of groups of people, discovering new types of stars in datasets of stellar objects, and so on.

- **Similarity Search:** Similarity search is performed on a database of objects to find the object(s) that are within a user-defined distance from the queried object, or to find all pairs within some distance of each other [Agrawal *et al.*, 1993c; Faloutsos *et al.*, 1994]. This kind of search is especially applicable to temporal and spatial databases. Example applications include discovering stock with similar price movements, identifying companies with similar sales patterns, etc.

1.1 Thesis Contributions

This thesis focuses on some of the key techniques in discovery-driven data mining, and their intersection with high-performance scalable and parallel computing, i.e., *Scalable Data Mining.* We develop new sequential and parallel algorithms that are efficient, disk-based, and that scale to very large databases. The main contributions of this thesis are:

1. We look at three key rule discovery techniques that include Association Rules, Sequence Discovery, and Decision Tree Classification.

2. Our algorithms scale to large disk-resident databases

3. Our techniques are based on a sound lattice-theoretic framework.

4. We experimentally evaluate our approach on major parallel platforms which include shared-memory multiprocessors, and hierarchical clusters of SMP workstations.

We will now briefly outline our contributions to each of the three rule mining tasks.

1.1.1 Mining Association Rules

Most of the extant algorithms for association mining [Agrawal *et al.*, 1993b; Agrawal *et al.*, 1996; Houtsma and Swami, 1995; Park *et al.*, 1995a] are iterative and make repeated passes over the database, incurring high I/O overhead. They use complex hash structures which suffer from poor locality. Furthermore, most parallel schemes [Park *et al.*, 1995b; Agrawal and Shafer, 1996; Cheung *et al.*, 1996b; Shintani and Kitsuregawa, 1996; Han *et al.*, 1997] involve repeated exchanges of frequency information or remote database partitions, incurring high communication and synchronization cost.

This thesis has been successful in overcoming most of these limitations. We present new algorithms that utilize lattice-theoretic techniques to decompose the original problem into smaller sub-problems, which can be independently solved in main memory using efficient search techniques, and using simple join operations on inverted item lists. All associations are discovered in a few passes over the database. The parallel algorithms also selectively replicate the database on each node, so that after the initial setup there is no more communication or synchronization. The new algorithms have been implemented on a cluster of 8 DEC Alpha 4-processor SMP machines (32 processors in all) with the fast Memory Channel network. The parallel implementation is hierarchical in nature, exploiting SMP parallelism within a node and message-passing among nodes. Extensive experiments have been conducted, showing immense improvement over previous algorithms, with linear scalability in database size. Another desirable property of the new algorithms is that only simple intersection operations are used in computing frequent item sets, making them suitable for direct implementation on general purpose database management systems (DBMS), using SQL primitives.

We also present the computational complexity of itemset enumeration, and develop a lattice-theoretic framework for association mining that can not only aid the development of efficient algorithms, but can also help in the visualization of discovered associations, and can provide a unifying framework for reasoning about some common data mining tasks. Parts of the work on associations have appeared in [Zaki *et al.*, 1996; Zaki *et al.*, 1997c; Zaki *et al.*, 1997d; Zaki *et al.*, 1997a; Zaki *et al.*, 1997b; Zaki *et al.*, 1997e; Zaki and Ogihara, 1998; Parthasarathy *et al.*, 1998].

1.1.2 Mining Sequence Rules

Existing solutions to this problem share most of the features and limitations of association mining, namely that they tend to make repeated database scans, and use complex hash structures [Agrawal and Srikant, 1995; Srikant and Agrawal, 1996b]. In this thesis we develop a fast new algorithm for sequence discovery. As in the association case, the new algorithm partitions the search space into small independent pieces, which are solved in main memory. The new algorithm retains the simple intersection operations, scales linearly with data size and outperforms previous approaches. Parts of the work on sequences have appeared in [Zaki, 1998; Zaki *et al.*, 1998b].

1.1.3 Mining Classification Rules

Classification is a well-studied problem [Weiss and Kulikowski, 1991; Michie *et al.*, 1994]; however, most of the current algorithms require that all or a portion of the dataset remain permanently in memory [Mehta *et al.*, 1996; Shafer *et al.*, 1996]. This requirement makes it difficult to mine large databases. This thesis focuses on the development of new scalable parallel algorithms targeting shared-memory systems, the first such study. The algorithms span the spectrum of data and task parallelism. The data parallelism scheme is based on attribute scheduling among processors. This scheme is extended with task pipelining and dynamic load balancing to yield more complex schemes. The task parallel approach uses dynamic subtree partitioning among processors. All of these algorithms are disk-based and achieve excellent scalability. Part of this work has appeared in [Zaki *et al.*, 1998a; Zaki *et al.*, 1999].

1.2 Thesis Outline

We begin by presenting new algorithms for mining association rules in Chapter 2. New hierarchical parallel algorithms for association mining on a cluster of SMP machines are described in Chapter 3. A theoretical foundation for association rules is formulated in Chapter 4. Chapter 5 presents an efficient algorithm for mining sequence rules, and Chapter 6 describes parallel algorithms for inducing classification rules on shared-memory multi-processors. Finally, Chapter 7 summarizes the main contributions of this thesis, and suggests avenues for future work.

2 Mining Association Rules

2.1 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 authors or 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 chapter 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 single database scan (with some pre-processed information), 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*, *Top-Down*, and *AprClique*. Our new algorithms not only minimize I/O costs by making only one 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. 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 chapter is organized as follows: In Section 2.2 we describe the association discovery problem. We look at related work in Section 2.3. In section 2.4 we develop our lattice-based approach for problem decomposition and pattern search. Section 2.5 describes our new algorithms. Some previous methods, used for experimental comparison, are described in more detail in Section 2.6. An experimental study is presented in Section 2.7, and we conclude in Section 2.8.

2.2 Problem Statement

The association mining task, first introduced in [Agrawal *et al.*, 1993b], 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]:

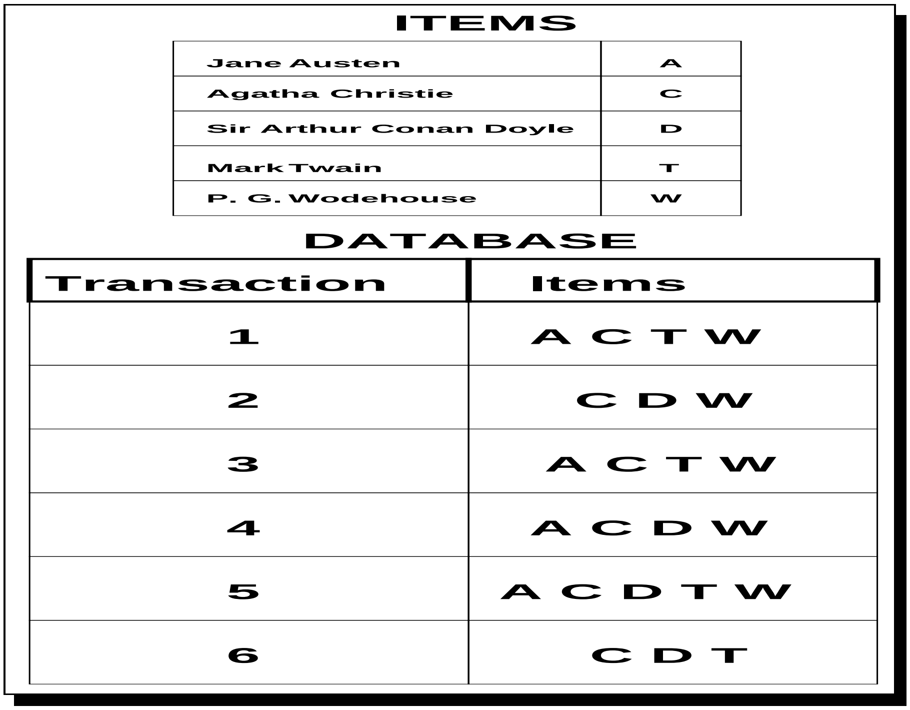
1. Find all frequent itemsets. This step is computationally and I/O intensive. Given m items, there can be potentially 2^m frequent itemsets. Efficient methods are needed to traverse this exponential itemset search space to enumerate all the frequent itemsets. Thus frequent itemset discovery is the main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

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 2.1. 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 2.2 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 maximal elements have been identified, by making an additional database pass, and gathering the support of all uncounted subsets.

2.3 Related Work

Several algorithms for mining associations have been proposed in the literature [Agrawal *et al.*, 1993b; Agrawal *et al.*, 1996; Brin *et al.*, 1997; Houtsma and Swami, 1995; Lin and Kedem, 1998; Lin and Dunham, 1998; Mueller, 1995; Park *et al.*, 1995a; Savasere *et al.*, 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 *et al.*, 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 *et al.*, 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 and 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



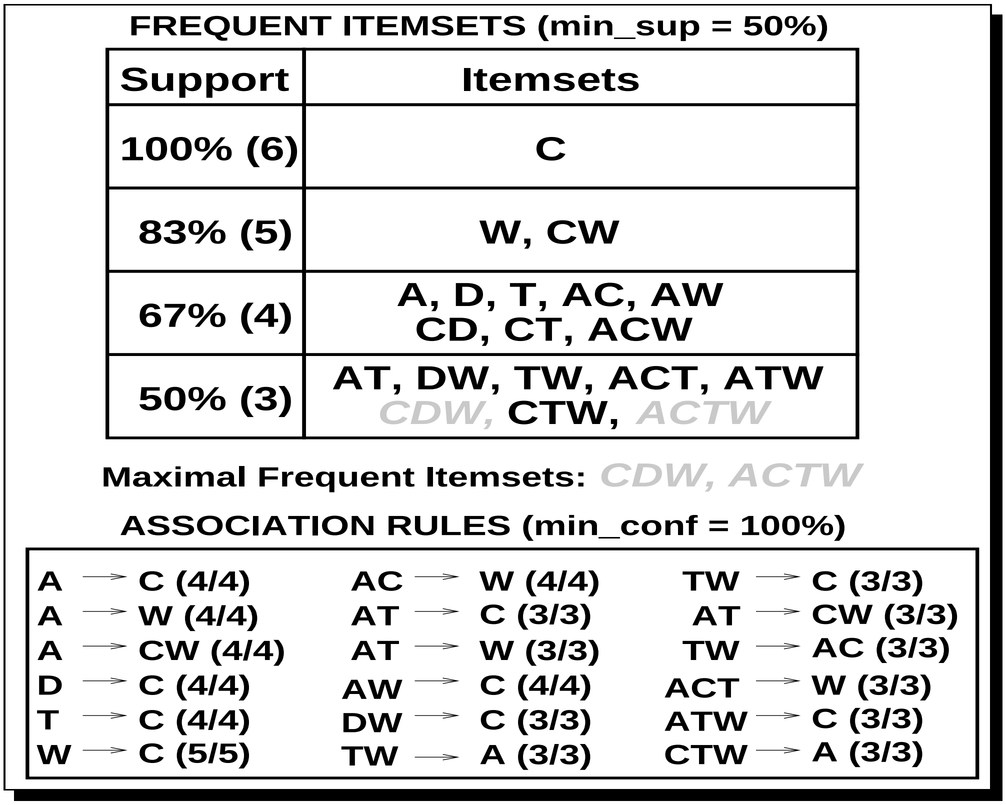
ITEMS

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

Figure 2.1: A Bookstore Database



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

Figure 2.2: Frequent Itemsets and Strong Rules

et al., 1997b], and [Toivonen, 1996] presents an exact algorithm that finds all rules using sampling. The AS-CPA algorithm and its sampling versions [Lin and 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 and Swami, 1995].

All the above algorithms generate all possible frequent itemsets. Methods for finding the maximal elements include *All-MFS* [Gunopulos *et al.*, 1997b], which is a randomized algorithm to discover maximal frequent itemsets. The *Pincer-Search* algorithm [Lin and 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. Our new algorithms [Zaki *et al.*, 1997c; Zaki *et al.*, 1997d] range from those that generate all frequent itemsets to those that generate the maximal frequent itemsets.

2.4 Itemset Enumeration: Lattice-Based Approach

Before embarking on the algorithm description, we will briefly review some terminology from lattice theory (see [Davey and Priestley, 1990] for a good introduction).

Definition 2.1 *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 2.2 *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 2.3 *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 $\perp$, is called the* **bottom element***.*

Definition 2.4 *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 $X \wedge Y \in M$.*

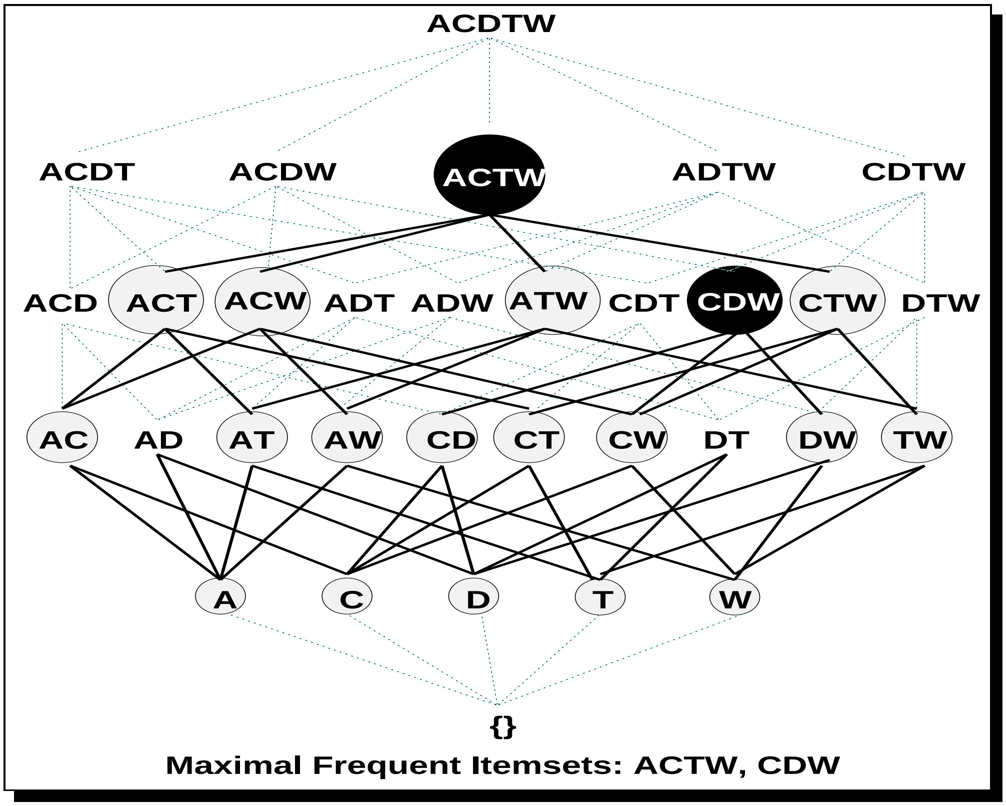


Figure 2.3: The Complete Powerset Lattice $\mathcal{P}(\mathcal{I})$

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 2.3 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 2.1 *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 *et al.*, 1995a; Savasere *et al.*, 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.

Lemma 2.2 *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.

2.4.1 Support Counting

Definition 2.5 *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 2.6 *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 2.7 *A lattice L is called a* **Boolean lattice** *if*

1) It is distributive.

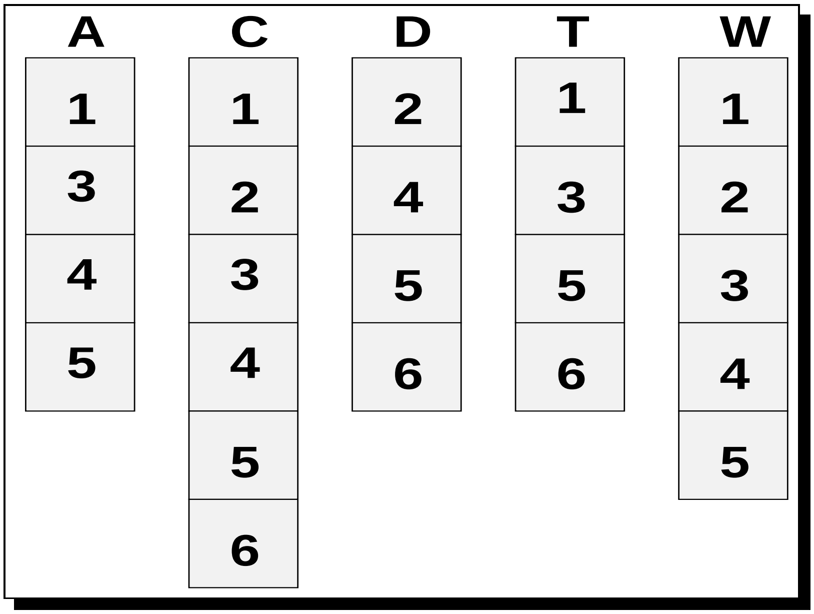
2) It has $\top$ and $\bot$ elements.

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 2.4 shows the tid-lists for the atoms in our example database. For example consider atom A. Looking at the database in Figure 2.1, we see that A occurs in transactions 1, 3, 4, and 5. This forms the tid-list for atom A.

Lemma 2.3 ([Davey and 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



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 2.4: Tid-List for the Atoms

Lemma 2.4 *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 2.5 *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 2.5 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 2.6 *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

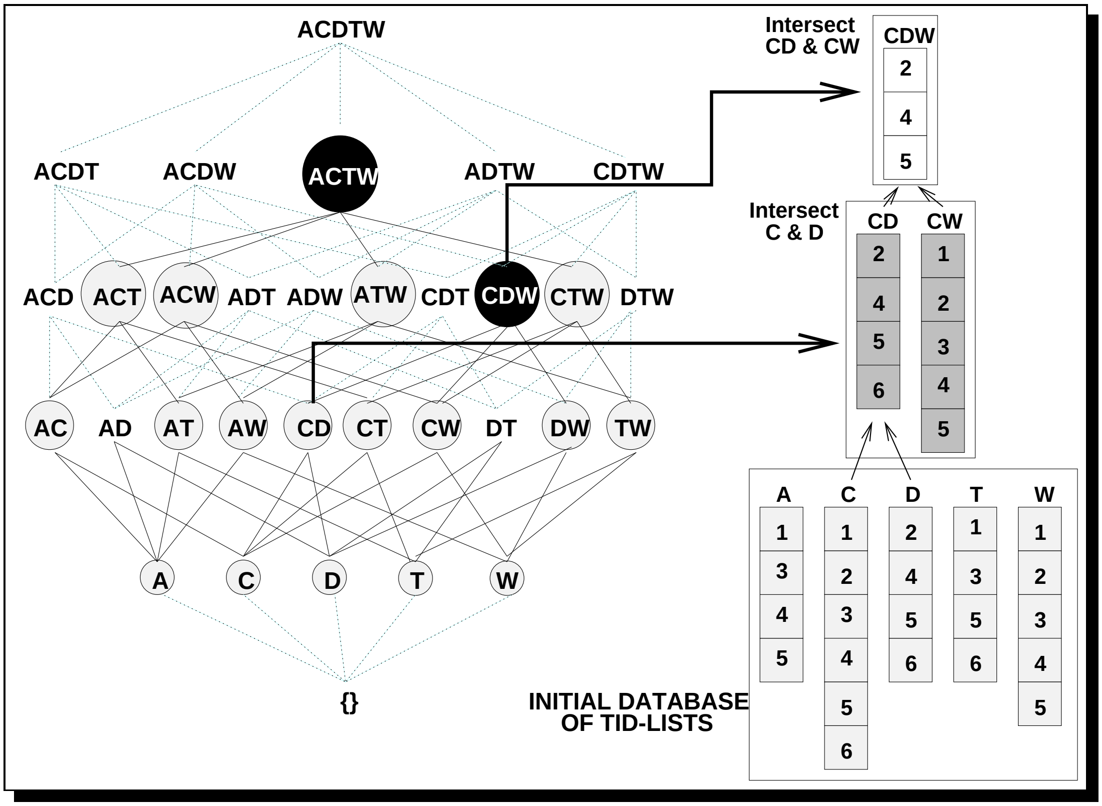


Figure 2.5: Computing Support of Itemsets via Tid-List Intersections

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.

2.4.2 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 2.8 *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 2.6 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 2.7 *Each equivalence class $[X]_{\theta_k}$ induced by the equivalence 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 2.4, and 2.5, 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 2.1), 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 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 2.7 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 2.8. 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.

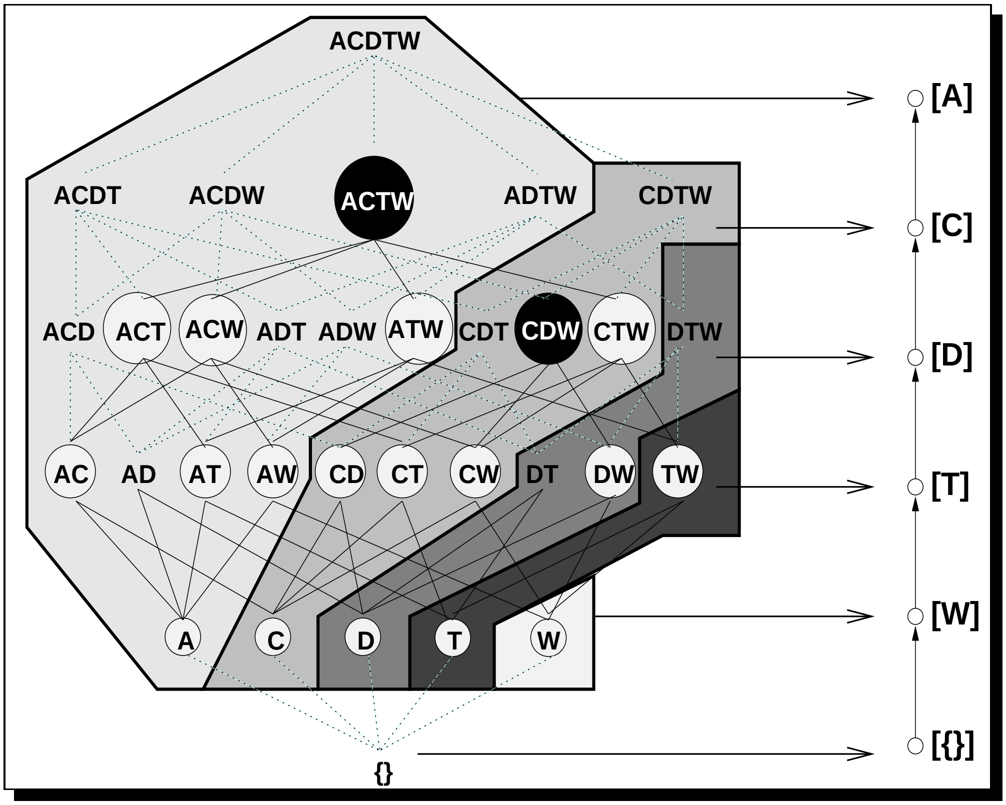


Figure 2.6: Equivalence Classes of $\mathcal{P}(\mathcal{I})$ Induced by θ_1

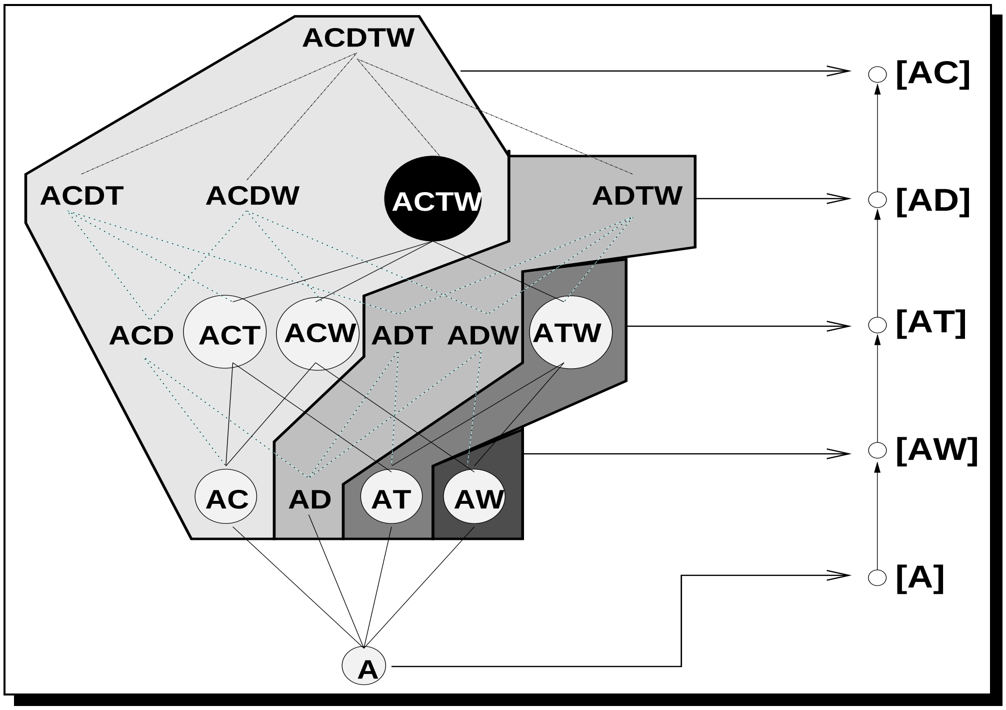


Figure 2.7: Equivalence Classes of $[A]_{\theta_1}$ Induced by θ_2

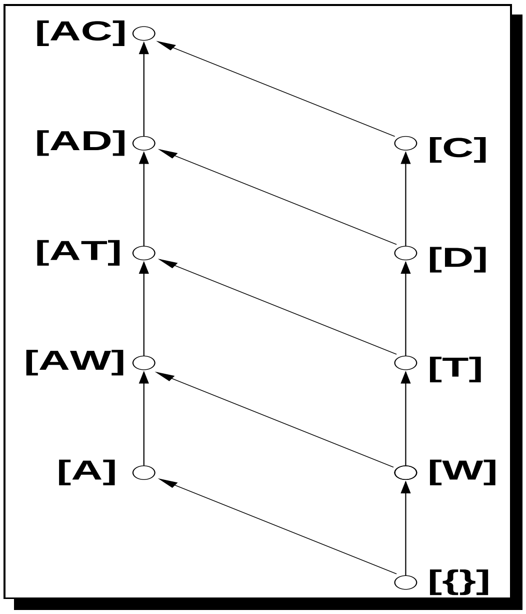


Figure 2.8: Final Lattice of Independent Classes

2.4.3 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 2.9 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 chapter 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.

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

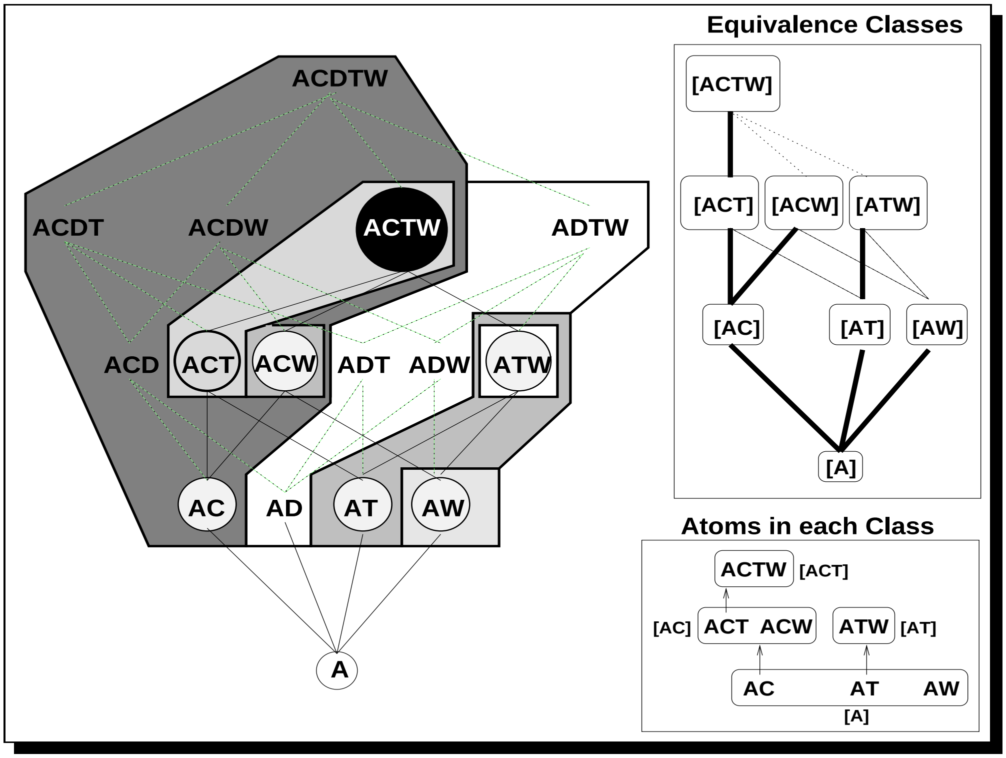


Figure 2.9: Bottom-Up Search

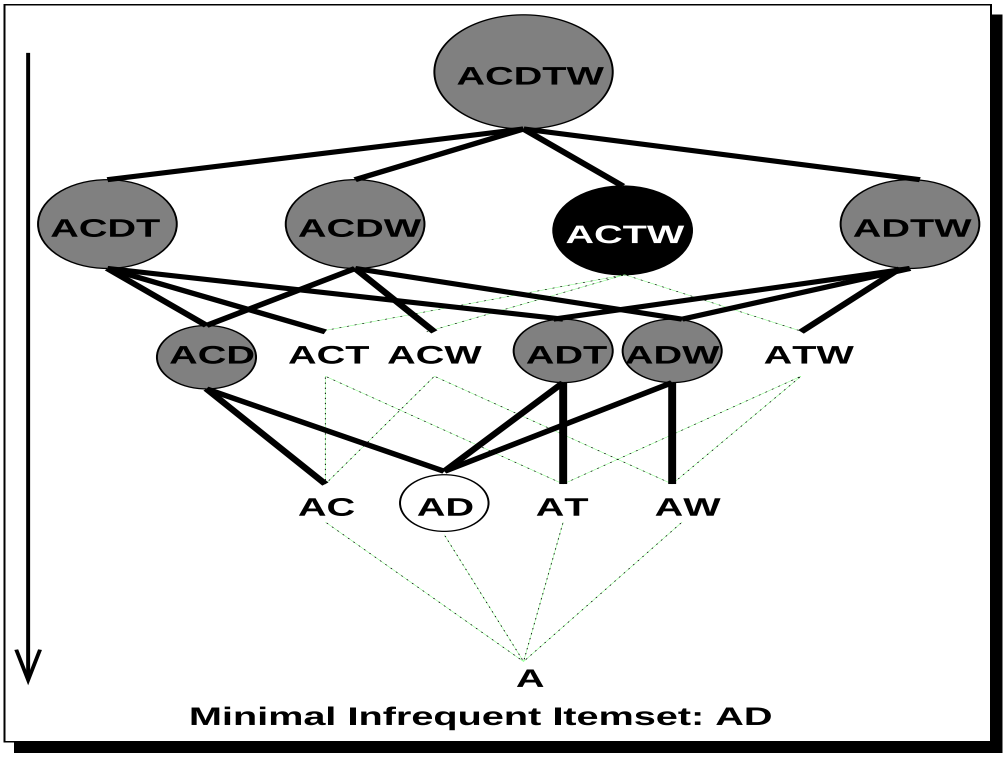


Figure 2.10: Top-Down Search

done. Otherwise, we check each subset at the next level. This process is repeated until we have identified all minimal infrequent itemsets. Figure 2.10 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.

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 2.11 illustrates this approach (just for this case, to better show the bottom-up phase, we have assumed that AD and ADW are also frequent). 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 all the maximal elements only.

2.4.4 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 2.9 *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 2.10 *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**.

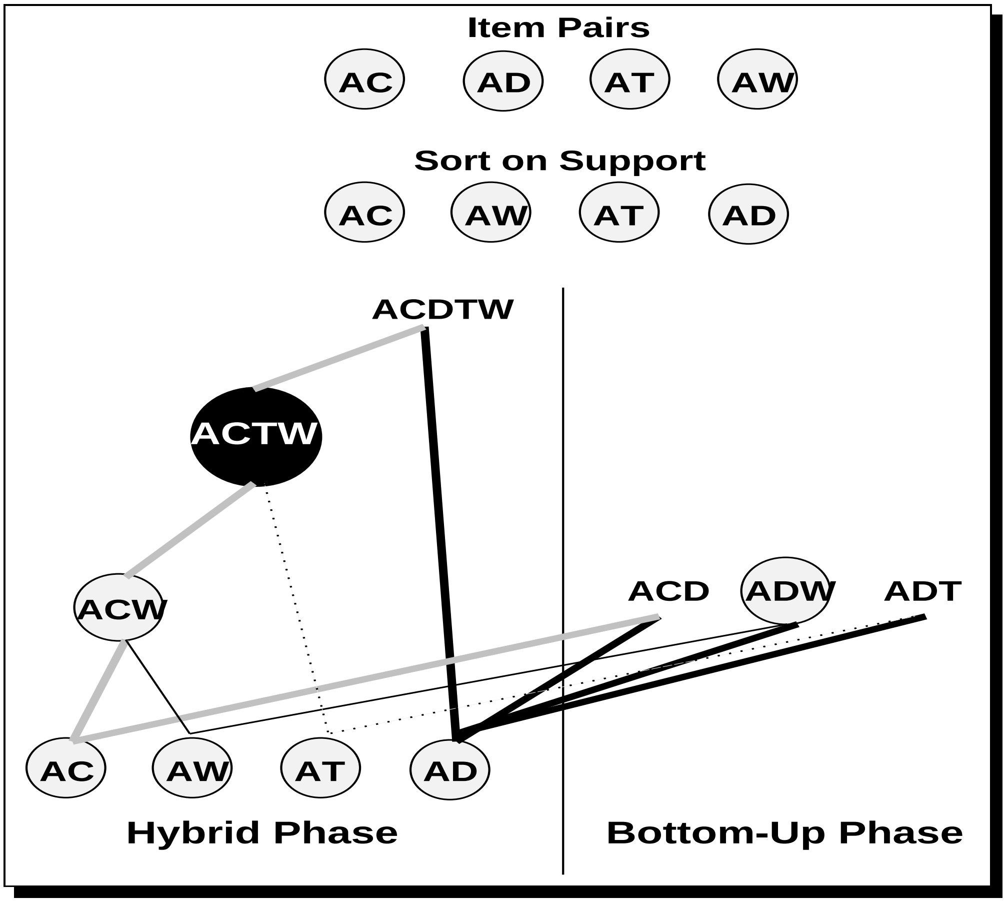


Figure 2.11: Hybrid Search

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 2.12 shows the association graph G_1, and its maximal clique set $M_1 = \{ACTW, CDW\}$.

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 2.8 *Each pseudo-class $[X]_{\phi_k}$ induced by the pseudo-equivalence 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,

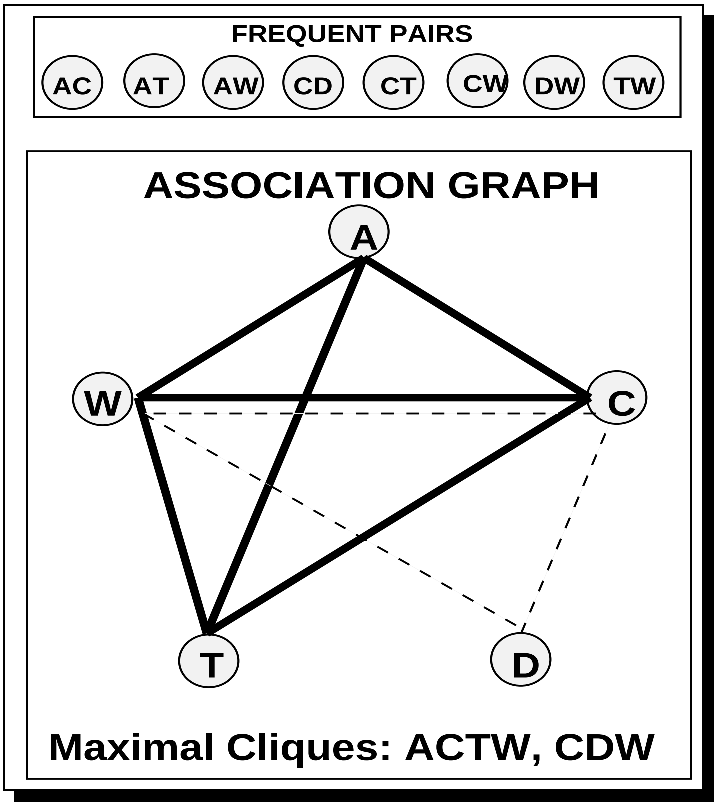


Figure 2.12: Maximal Cliques of the Association Graph

$U \cup V \subseteq C$, 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 2.4, and 2.5 on the atoms, and using any of the three search strategies described above.

Lemma 2.9 *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

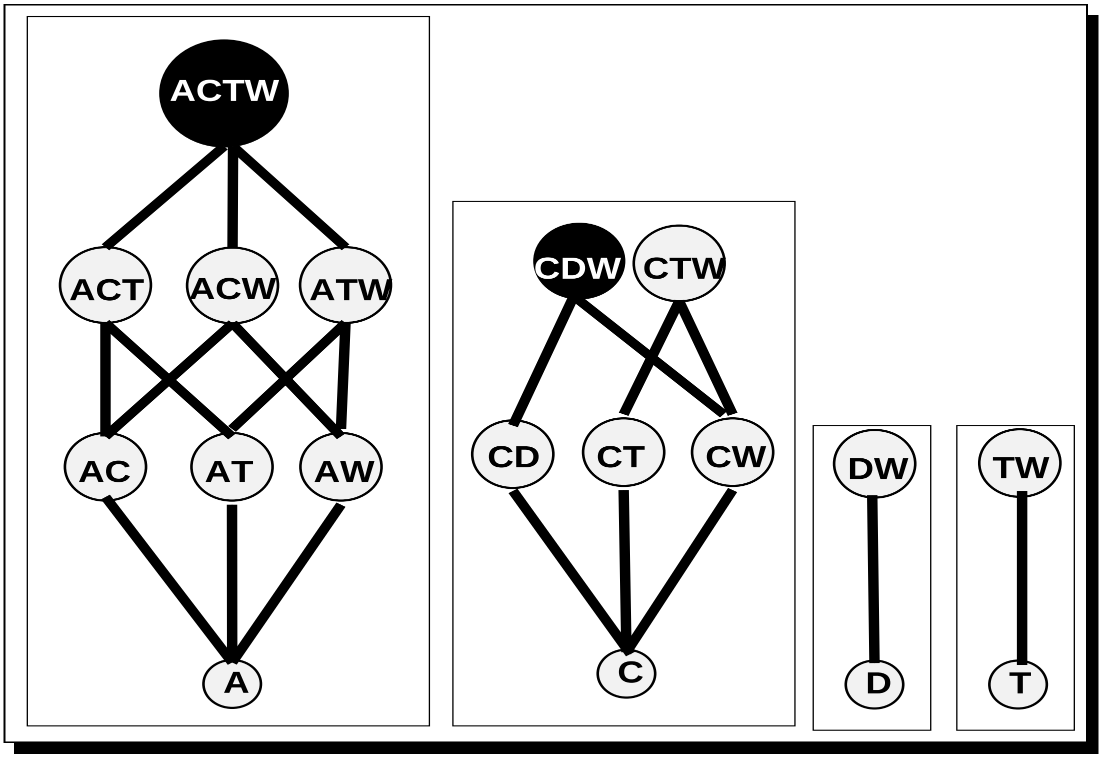


Figure 2.13: Maximal-Clique-Based Sub-lattices Induced by ϕ_1

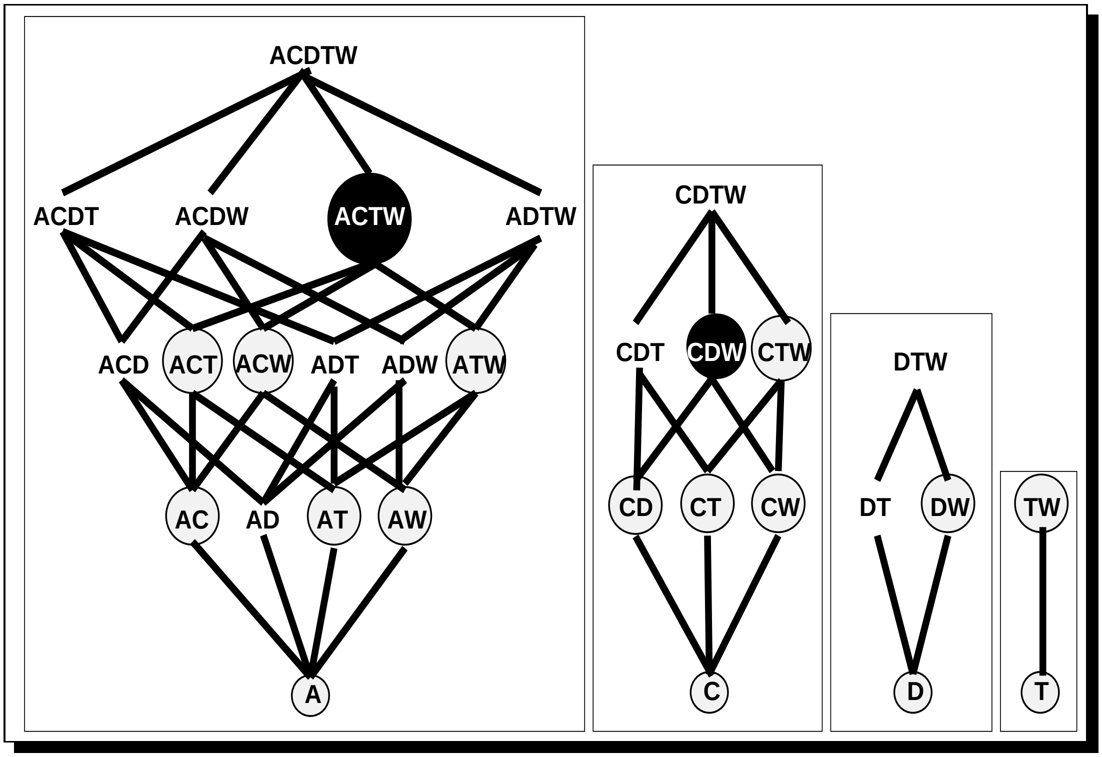


Figure 2.14: Prefix-Based Sub-lattices Induced by θ_1

independently using any of the three search strategies described above. Figure 2.13 and Figure 2.14 contrast the sub-lattices generated by ϕ_1 and θ_1. It is apparent that ϕ_1 generates smaller sub-lattices. The increased precision 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 and 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 effect of these parameters is studied in the experimental section.

Maximal Clique Generation

We used a modified version of the Bierstone's algorithm [Mulligan and Corneil, 1972] for generating maximal cliques in the k-association graph. A number of other faster clique generating algorithms [Bron and Kerbosch, 1973; Chiba and Nishizeki, 1985; Tsukiyama *et al.*, 1977] can also be used.

2.5 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 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.

2.5.1 Computing Frequent 1-Itemsets and 2-Itemsets

Most of the current association algorithms [Agrawal *et al.*, 1996; Brin *et al.*, 1997; Lin and Dunham, 1998; Park *et al.*, 1995a; Savasere *et al.*, 1995; Toivonen, 1996] assume a *horizontal* database layout, such as the one shown in Figure 2.1, 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 2.4, 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 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.

2.5.2 Search Implementation

Bottom-Up Search Figure 2.15 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 2.15: 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 2.6, 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($S$):**
$R = \bigcup\{A_i \in S\}$;
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 2.16: Pseudo-code for Top-Down Search

```
**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 2.17: Pseudo-code for Hybrid Search

Top-Down Search The code for top-down search is given in Figure 2.16. 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 2.17 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.

2.5.3 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 single database scan after computing $\mathcal{F}_2$, in contrast to the current approaches which require multiple scans.

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 θ_1 along with bottom-up search. It enumerates all frequent itemsets.

2. **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.

3. **Clique**: It uses maximal-clique-based pseudo-equivalence relation ϕ_1 along with bottom-up search. It enumerates all frequent itemsets.

4. **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.

5. **TopDown**: It uses maximal-clique-based pseudo-equivalence relation ϕ_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 θ_1 is not likely to be efficient.

```
AprClique():
for all sub-lattices $S_i$ induced by $\phi_1$ do
    $R = \bigcup\{A_j \in S_i\}$;
    for all $k > 2$ and $k \leq |R|$ do
        Insert each $k$-subset of $R$ in $C_k$;
end
for all transactions $t \in \mathcal{D}$ do
    for all $k$-subsets $s$ of $t$, with $k > 2$ and $k \leq |t|$ do
        if $(s \in C_k)$ $s.count++$;
end
$\mathcal{F}_k = \{c \in C_k | c.count \geq \text{minsup}\}$;
Set of all frequent itemsets $= \bigcup_k \mathcal{F}_k$;
```

Figure 2.18: Pseudo-code for AprClique Algorithm

6. **AprClique**: It uses maximal-clique-based pseudo-equivalence relation ϕ_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* [Agrawal *et al.*, 1996], 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 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 2.18.

2.6 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* [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 2.19. 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 to facilitate 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 2.19: The *Apriori* Algorithm

Partition Algorithm *Partition* [Savasere *et al.*, 1995] 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.

2.7 Experimental Results

Our experiments used a 100MHz MIPS processor with 256MB main memory, 16KB primary cache, 1MB secondary cache and a non-local 2GB disk.

Synthetic Databases We used different synthetic databases that have been used as benchmark databases for many association rules algorithms [Agrawal *et al.*, 1993b; Agrawal *et al.*, 1996; Brin *et al.*, 1997; Houtsma and Swami, 1995; Lin and Kedem, 1998; Lin and Dunham, 1998; Park *et al.*, 1995a; Savasere *et al.*, 1995; Zaki *et al.*, 1997c]. The dataset generation procedure is described in [Agrawal *et al.*, 1996], and the code is publicly available from IBM [IBMa].

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 and Srikant, 1994] 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 2.1.

For fair comparison, all algorithms discover frequent k-itemsets for $k \geq 3$, since the supports of 2-itemsets are known from the preprocessing step. Figure 2.20 shows the number of frequent itemsets of different sizes for the databases used in our experiments.

Comparative Performance In Figure 2.21 and Figure 2.22 we compare our new algorithms against *Apriori* and *Partition* (with 10 database partitions) for decreasing values of minimum support on the different databases. We compare *Eclat* against *Apriori*, *AprClique* and *Partition* in the left column, and we compare *Eclat* against the other new algorithms in the right column to highlight the differences among them. As the support decreases, the size and the number of frequent itemsets increases. *Apriori* thus has to make multiple passes over the database, and performs poorly. *Partition* performs worse than *Apriori* on small databases and for high support, since the database is only scanned once or twice at these points. However, as the support is lowered,

Database	T	I	D	Size
T5.I2.D100K	5	2	100,000	2.8MB
T10.I2.D100K	10	2	100,000	4.8MB
T10.I4.D100K	10	4	100,000	4.8MB
T20.I2.D100K	20	2	100,000	8.8MB
T20.I4.D100K	20	4	100,000	8.8MB
T20.I6.D100K	20	6	100,000	8.8MB
T10.I4.D250K	10	4	250,000	12.0MB
T10.I4.D500K	10	4	500,000	24.0MB
T10.I4.D1000K	10	4	1,000,000	48.0MB
T10.I4.D2500K	10	4	2,500,000	120.0MB
T10.I4.D5000K	10	4	5,000,000	240.0MB

Table 2.1: Database Parameter Settings

Partition wins out over *Apriori*, since it only scans the database twice. These results are in agreement with previous experiments comparing these two algorithms [Savasere *et al.*, 1995]. 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. *AprClique* scans the database only once, and out-performs *Apriori* in almost all cases, and generally lies between *Apriori* and *Partition*. *AprClique* is very sensitive to the quality of maximal cliques (sub-lattices) that are generated. For small support, or small average maximal potential frequent itemset size I for fixed T, or with increasing average transaction size T for fixed I, the edge density of the k-association graph increases, consequently increasing the size of the maximal cliques. *AprClique* is unlikely to perform well under these conditions. *Eclat* performs significantly better than all these algorithms in all cases. It out-performs *Apriori* by more than an order of magnitude, and *Partition* by more than a factor of five. *Eclat* makes only once database scan, requires no hash trees, uses only simple intersection operations to generate frequent itemsets, avoids redundant computations, and since it deals with one cluster at a time, has excellent locality.

To summarize, there are several reasons why *Eclat* outperforms previous approaches:

1. *Eclat* uses 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 [Parthasarathy *et al.*, 1998]. On the other hand *Eclat* has 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* on the other hand

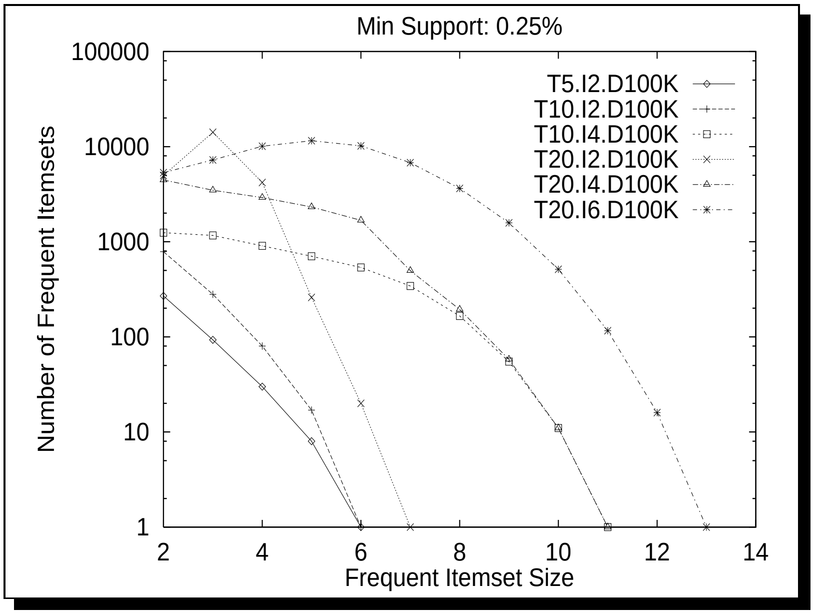


Figure 2.20: Number of Frequent Itemsets

restricts itself to usually only one scan. It thus cuts down the I/O costs.

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

Table 2.2: Number of Joins: T20.I6.D100K (0.25%)

The right hand columns in Figure 2.21 and Figure 2.22 presents the comparison among the other new algorithms. *Clique* uses the maximal-clique-based decomposition, which generates smaller classes with fewer number of candidates, and therefore performs better than *Eclat*. The graphs for *MaxEclat* and *MaxClique* indicate that the reduction in search space by performing the hybrid search also provides significant gains. Both the maximal strategies outperform their normal counterparts. *TopDown* usually outperforms both *Eclat* and *MaxEclat*, since it also only generates the maximal frequent itemsets. Like *AprClique*, it is very sensitive to the size of the maximal cliques, and it suffers as the cliques become larger. The best scheme for the databases we considered is *MaxClique* since it benefits from the smaller sub-lattices and the hybrid search scheme. Table 2.2 gives the number of intersections performed by the different algorithms on T20.I6.D100K. *MaxClique* cuts down the candidate search space drastically, by more than a factor of 4 over *Eclat*. In terms of raw performance *MaxClique* outperforms *Apriori* by a factor of 40, *Partition* by a factor of 20, and *Eclat* by a factor of 2.5 for this case. However, as previously mentioned, whenever the edge density of the association graph increases, the number and size of the cliques becomes large and there is

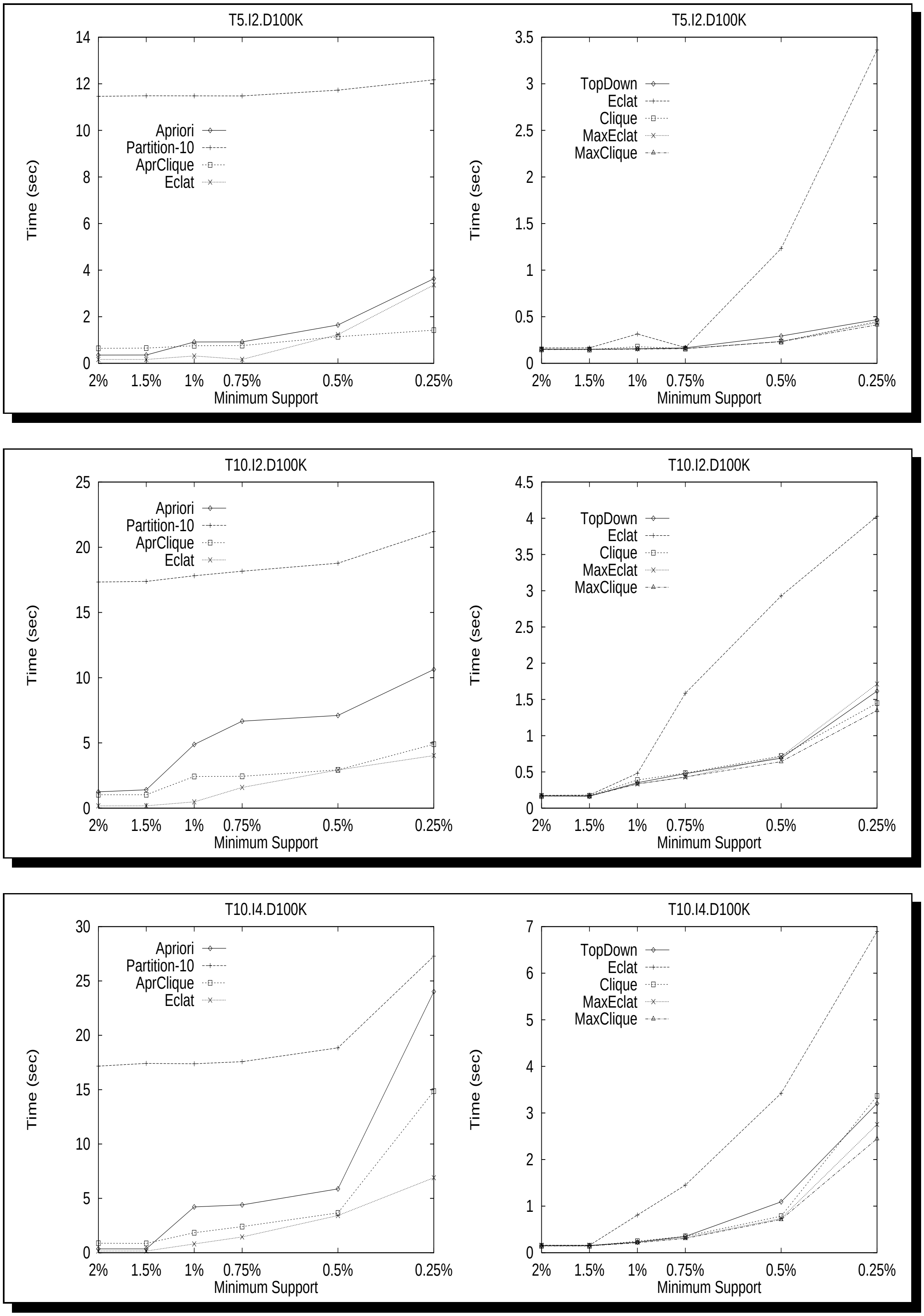


Figure 2.21: Execution Time

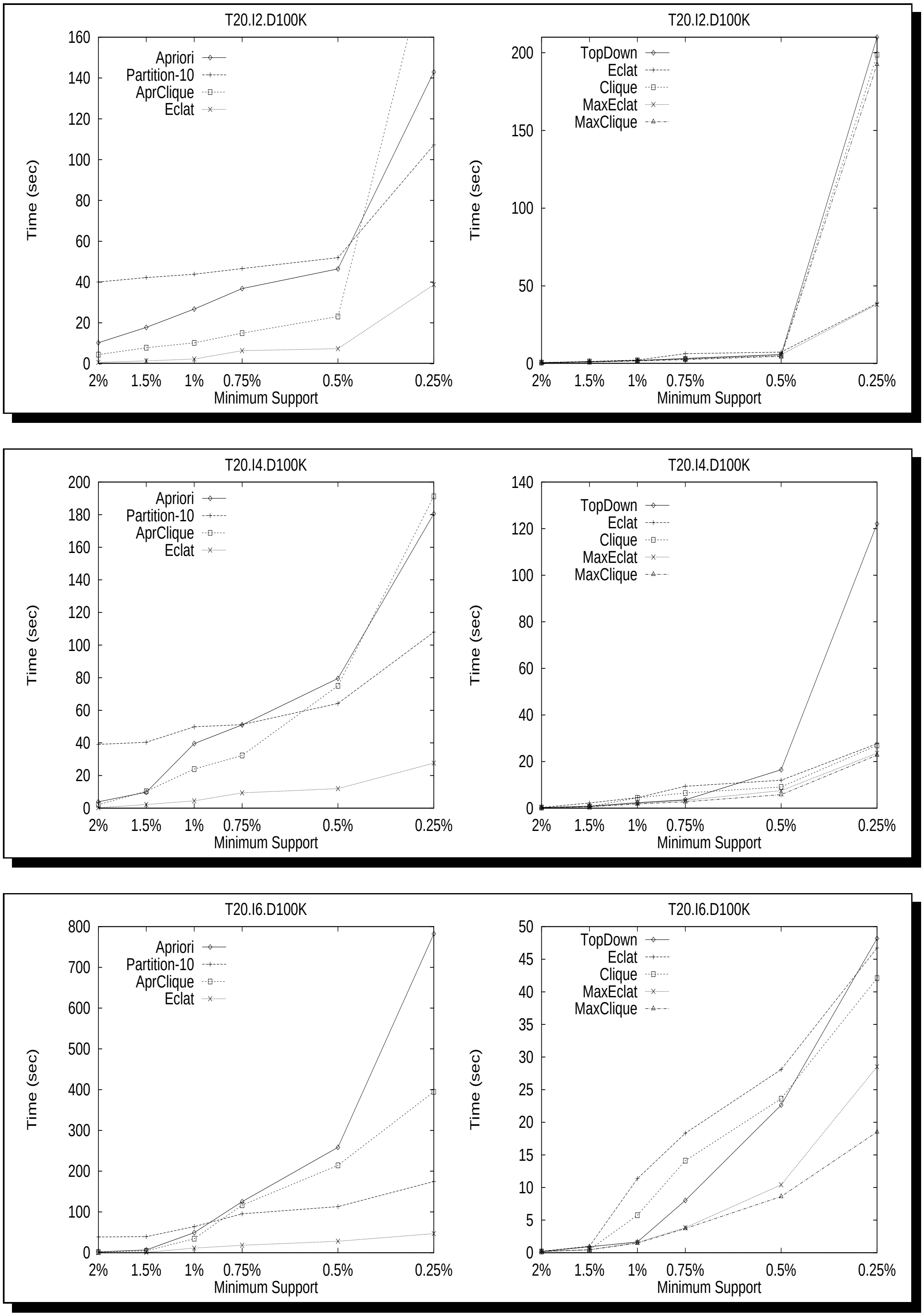


Figure 2.22: Execution Time

a significant overlap among different cliques. In such cases the clique based schemes start to suffer. For example, consider Figure 2.24. As the transaction size increases for a fixed support value the clique based algorithms start performing worse. We expect to reduce the overhead of the clique generation by implementing the faster clique finding algorithms proposed in [Bron and Kerbosch, 1973; Chiba and Nishizeki, 1985; Tsukiyama *et al.*, 1977], instead of the modified Bierstone's algorithm [Mulligan and Corneil, 1972] used in our current implementation.

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 2.23 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 2.24 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.

Memory Usage Figure 2.25 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 θ_1 and ϕ_1 are small enough, so that all intermediate tid-lists for a class can be kept in main-memory. We expect the maximal-clique-based algorithms to use even less memory.

2.8 Conclusions

In this chapter 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-

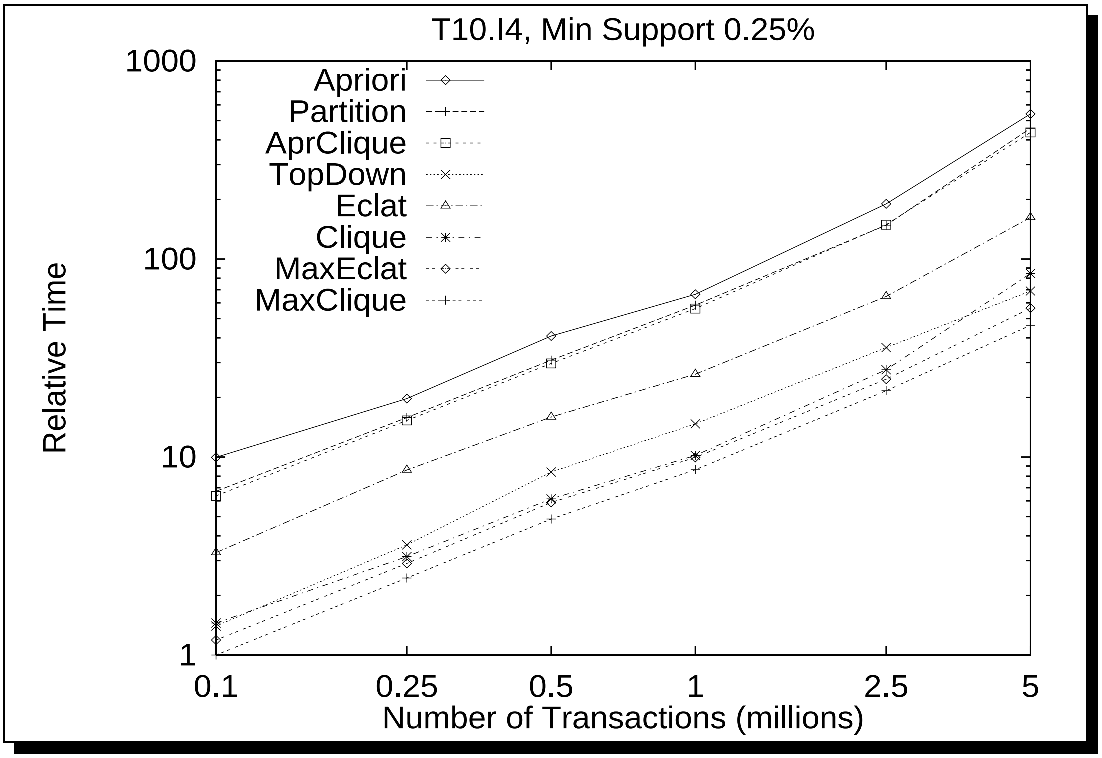


Figure 2.23: Scale-up Experiments: Number of Transactions

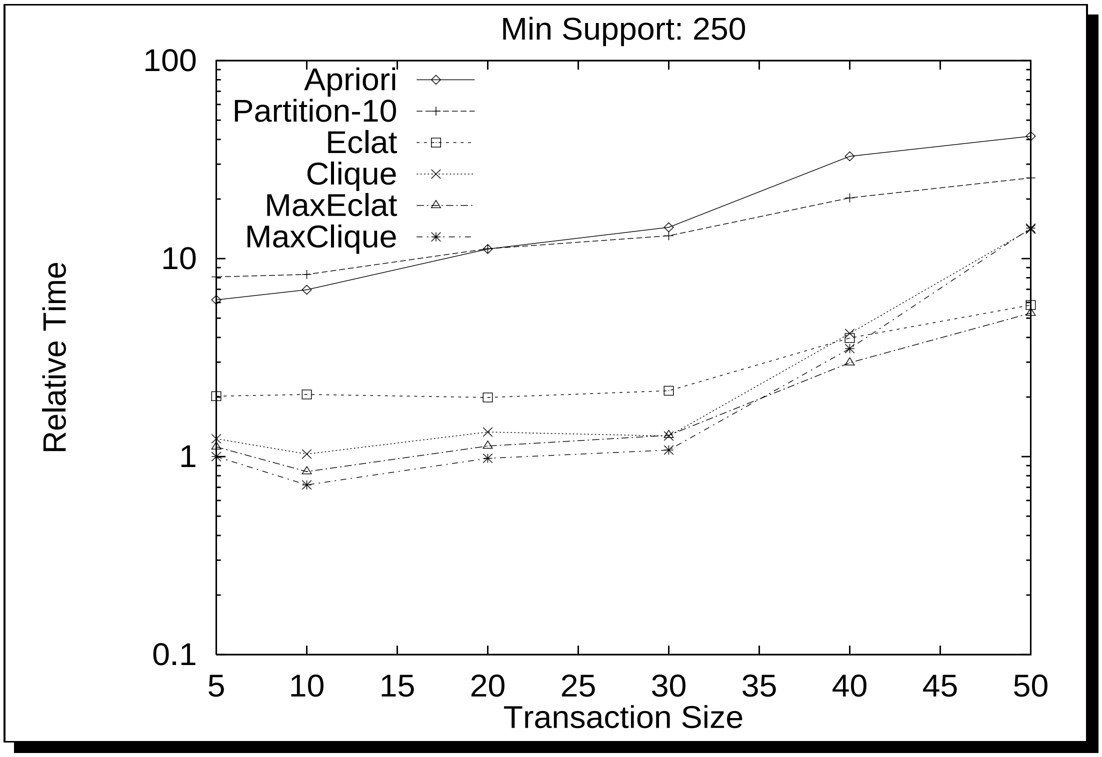


Figure 2.24: Scale-up Experiments: Transaction Size

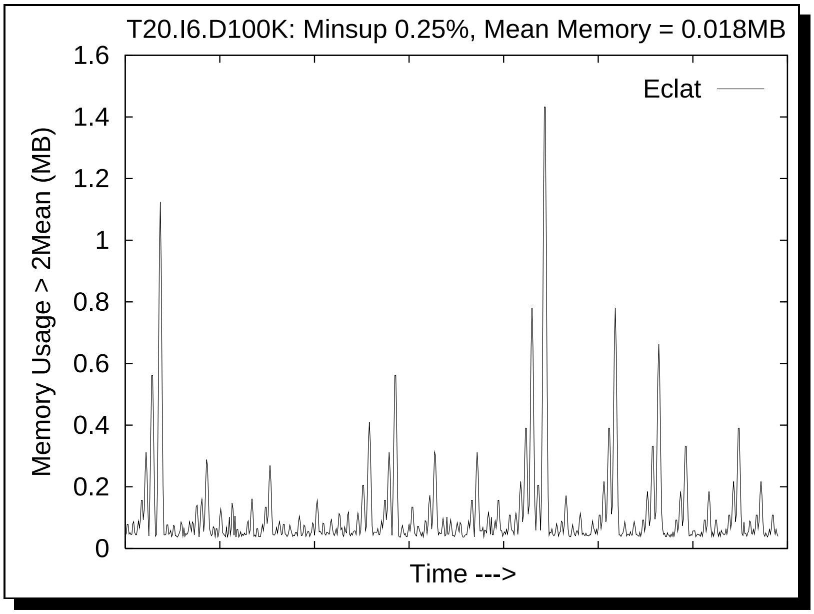


Figure 2.25: *Eclat* Memory Usage

up, top-down, or a hybrid search procedure, and the entire process usually takes only one database scan.

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.

3 Parallel Association Mining

3.1 Introduction

The discovery of association rules is a very computational and I/O intensive task, and beyond a certain database size, it is crucial to leverage the combined computational power of multiple processors for fast response and scalability. In this chapter we present new parallel algorithms for frequent itemset discovery.

Most previous parallel algorithms [Park *et al.*, 1995b; Zaki *et al.*, 1996; Agrawal and Shafer, 1996; Cheung *et al.*, 1996b; Cheung *et al.*, 1996a; Shintani and Kitsuregawa, 1996] make repeated passes over the disk-resident database partition, incurring high I/O overhead, and exchange either the counts of candidates or the remote database partitions during each iteration, resulting in high communication and synchronization overhead. The previous algorithms also use complicated hash structures which entail additional overhead in maintaining and searching them, and typically suffer from poor cache locality [Parthasarathy *et al.*, 1998].

Our work contrasts to these approaches in several ways. We present new parallel algorithms based on concepts outlined in the last chapter. The new algorithms use the prefix-based and clique-based problem decomposition techniques, and efficient bottom-up and top-down search, along with the vertical database format. The 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 experimental testbed is a 32-processor DEC Alpha SMP cluster (8 hosts, 4 processors/host) inter-connected 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. Extensive experiments have been conducted, showing immense improvement over existing algorithms, with linear scalability in database size. We also present performance results on the speedup, sizeup, communication cost and memory usage of the new algorithms.

The rest of the chapter is organized as follows. We begin with related work in Section 3.2. Section 3.3 describes some of the previous *Apriori* based parallel algorithms.

Section 3.4 describes the design and implementation of the new parallel algorithms. Our experimental study is presented in Section 3.5, and our conclusions in Section 3.6.

3.2 Related Work

Distributed-Memory Machines Three different parallelizations of *Apriori* on IBM-SP2, a distributed-memory machine, were presented in [Agrawal and 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 and 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; Cheung *et al.*, 1996a; Han *et al.*, 1997]. The PDM algorithm [Park *et al.*, 1995b] presents a parallelization of the DHP algorithm [Park *et al.*, 1995a]. The hash based parallel algorithms NPA, SPA, HPA, and HPA-ELD, proposed in [Shintani and Kitsuregawa, 1996] are similar to those in [Agrawal and 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 *et al.*, 1997a] 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. Parts of this chapter have appeared in [Zaki *et al.*, 1997e].

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 *et al.*, 1998] is an asynchronous parallel algorithm for shared-memory machines based on the DIC algorithm [Brin *et al.*, 1997].

3.3 *Apriori*-based Parallel Algorithms

In this section we will look at some previous parallel algorithms. These algorithms assume that the database is partitioned among all the processors in equal-sized blocks, which reside on the local disk of each processor.

The *Count Distribution* algorithm [Agrawal and Shafer, 1996] is a simple 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. The algorithm is pictorially depicted in Figure 3.1. 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. The implementation of *Count Distribution* used for comparison in our experiments differs slightly from the above description and is optimized for our testbed configuration. Only one copy of the hash tree resides on each of the 8 hosts in our cluster. All the 4 processors on each host share this hash tree. Each processor still has its own local database portion and uses a local array to gather the local candidate support. The sum-reduction is accomplished in two steps. The first step performs the reduction only among the local processors on each host. The second step performs the reduction among the hosts. We also utilize some optimization techniques such as hash-tree balancing and short-circuited subset counting [Zaki *et al.*, 1996] to further improve the performance of *Count Distribution*.

The *Data Distribution* algorithm [Agrawal and Shafer, 1996] was designed to utilize the total system memory by generating disjoint candidate sets on each processor. 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 high communication overhead, and performs poorly when compared to *Count Distribution*.

The *Candidate Distribution* algorithm [Agrawal and Shafer, 1996] uses a property of frequent itemsets [Agrawal and Shafer, 1996; Zaki *et al.*, 1996] to partition the candidates during iteration l, so that each processor can generate disjoint candidates independent of other processors. At the same time the database is selectively replicated so that a processor can generate global counts independently. The choice of the redistribution pass involves a trade-off between decoupling processor dependence as soon as possible and waiting until sufficient load balance can be achieved. In their experiments the repartitioning was done in the fourth pass. After this the only dependence a processor has on other processors is for pruning the candidates. Each processor asynchronously broadcasts the local frequent set to other processors during each iteration. This pruning information is used if it arrives in time, otherwise it is used in the next iteration. Note that each processor must still scan its local data once per iteration. Even though it uses problem-specific information, it performs worse than *Count Distribution* [Agrawal and Shafer, 1996]. *Candidate Distribution* pays the cost of redistributing the database, and

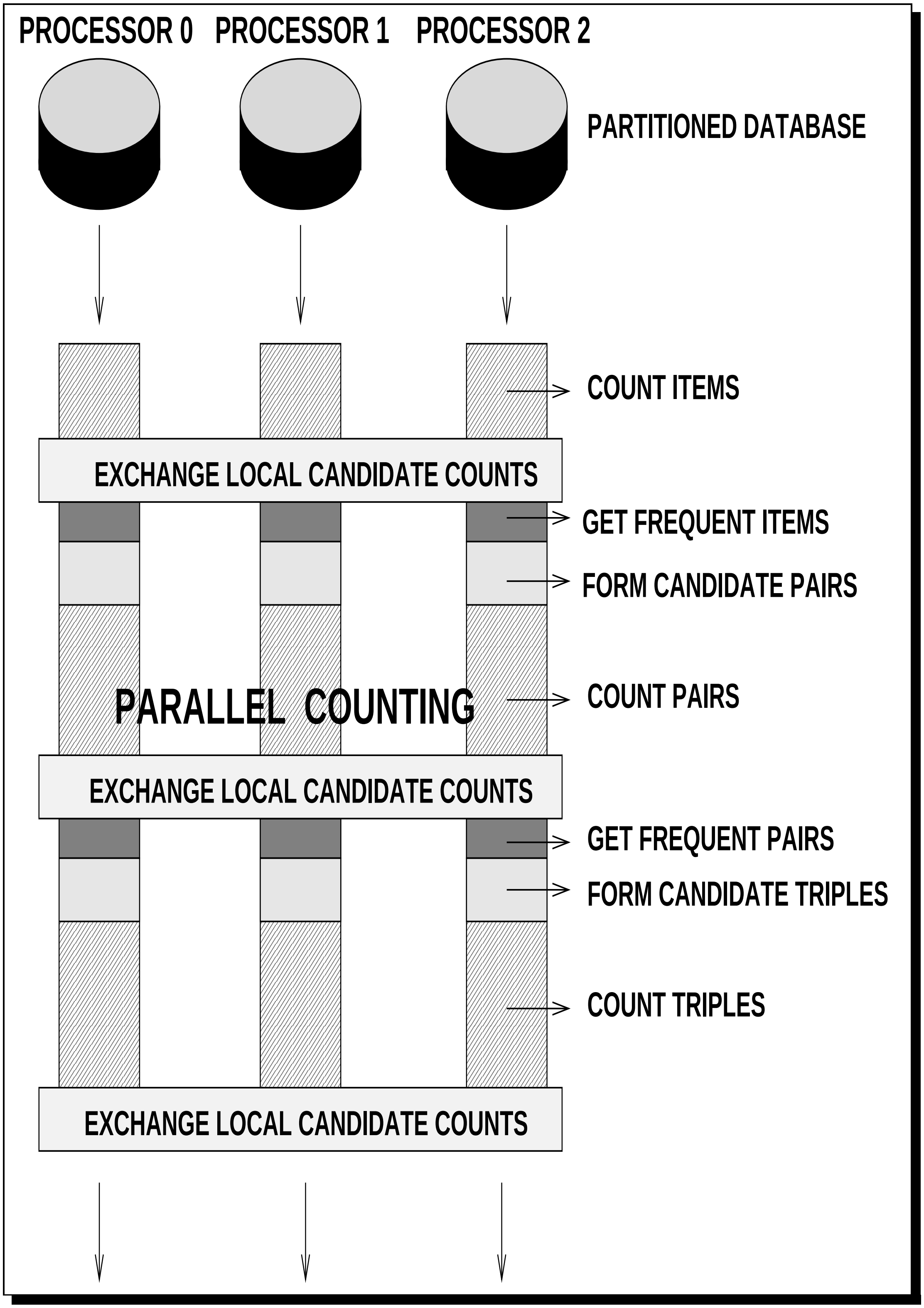


Figure 3.1: The *Count Distribution* Algorithm

it then scans the local database partition repeatedly.

3.4 Algorithm Design and Implementation

In this section we will discuss the design and implementation of new parallel algorithms for mining frequent itemsets.

3.4.1 New Parallel Algorithms

We present four new parallel algorithms, depending on the decomposition relation used to generate independent classes, and the lattice search scheme used. Each algorithm is based on its sequential counterpart discussed in the last chapter.

- *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.

3.4.2 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 and 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 3.3 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 3.2: Pseudo-code for the New Parallel Algorithms

3.4.3 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 3.2, and a pictorial representation of the different phases is shown in Figure 3.3.

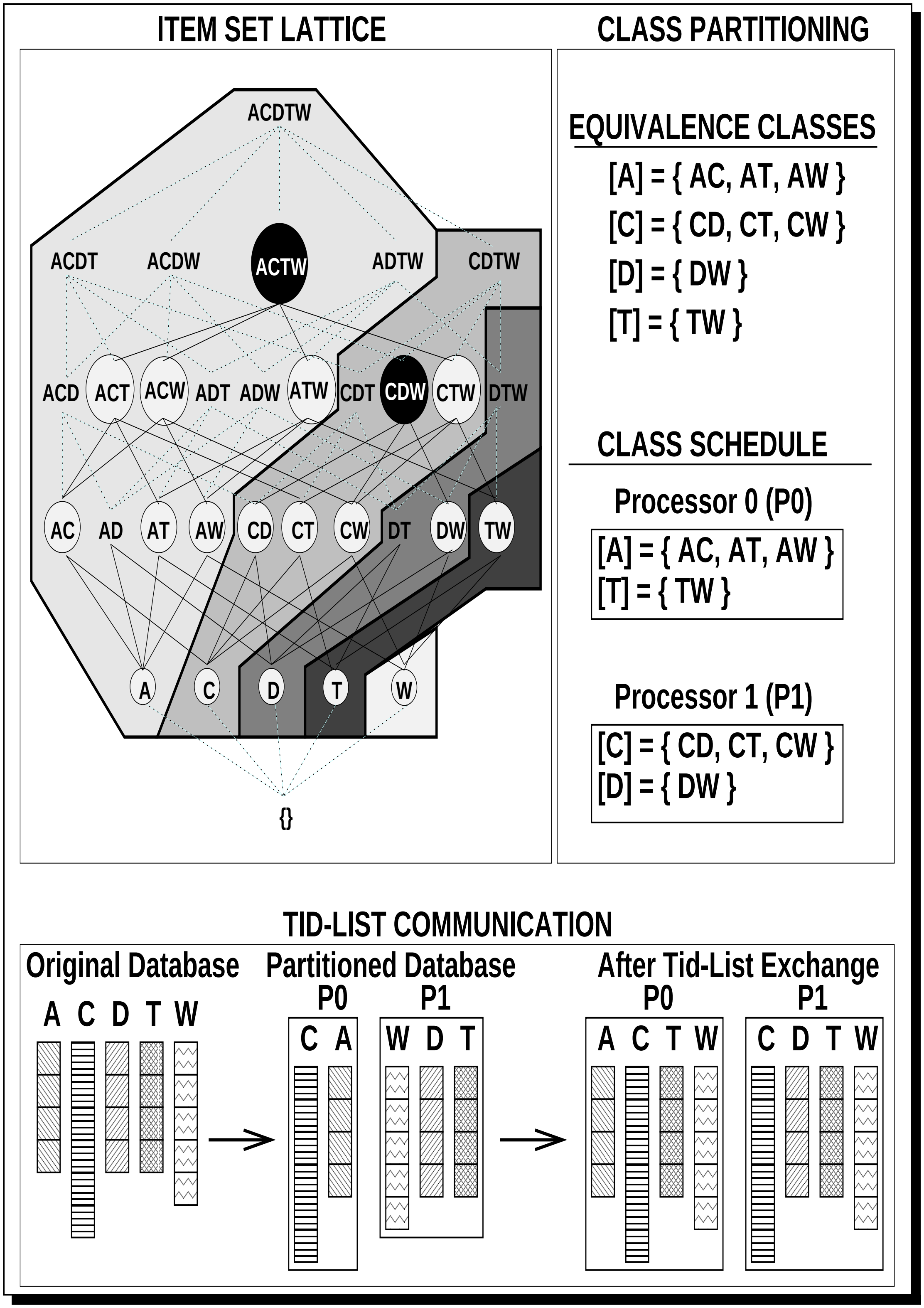


Figure 3.3: Database Partitioning and Class Scheduling

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.

Class Scheduling We first partition $\mathcal{F}_2$ into equivalence classes as described above. 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 3.3 shows how the prefix-based classes of our example database (from Figure 2.1) 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. We believe that decoupling processor performance right in the beginning holds promise, even though it may cause some load imbalance, since the repartitioning cost can be amortized over later iterations. Deriving better heuristics for scheduling the classes, which minimize the load imbalance and communication, is part of ongoing research.

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 3.3 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 chapter (see Figure 2.15 and Figure 2.16). As an illustration of the various steps, the *Par-Eclat* algorithm is shown in Figure 3.4. At the end of the asynchronous phase we accumulate all the results from each processor and print them out.

Pruning Candidates Recall that both *Count* and *Candidate Distribution* use a pruning step to eliminate unnecessary candidates. This step is essential in those algorithms to reduce the size of the hash tree. Smaller trees lead to faster support counting, since each subset of a transaction is tested against the tree. However, with the vertical database layout we found the pruning step to be of little or no help. This can be attributed to several factors. First, there is additional space and computation overhead in constructing and searching hash tables. This is also likely to degrade locality. Second, there is extra overhead in generating all the subsets of a candidate. Third, there is extra communication overhead in communicating the frequent itemsets in each iteration, even though it may happen asynchronously. Fourth, because the average size of tid-lists decreases as the itemsets size increases, intersections can be performed very quickly.

3.4.4 Salient Features of the New Algorithms

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

- Unlike *Count Distribution*, 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.

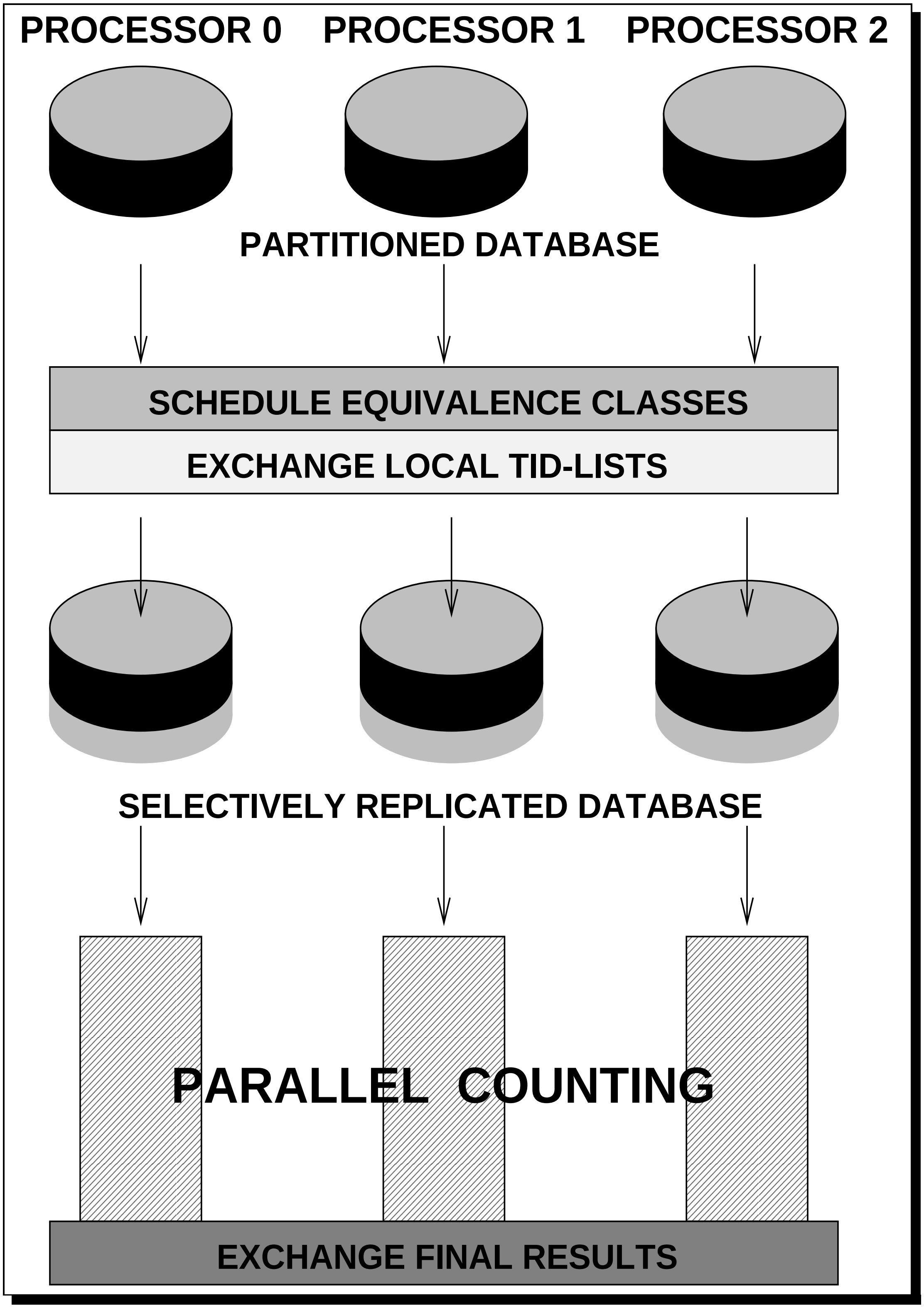


Figure 3.4: The *Par-Eclat* Algorithm

- 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 two times on each processor. The first scan for communicating the tid-lists, and the second for obtaining the frequent itemsets. In contrast, both *Count* and *Candidate Distribution* scan 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 *et al.*, 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.

3.5 Experimental Results

All the experiments were performed on a 32-processor (8 hosts, 4 processors/host) Digital Alpha cluster inter-connected 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 attached to it, 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 3.5 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.

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

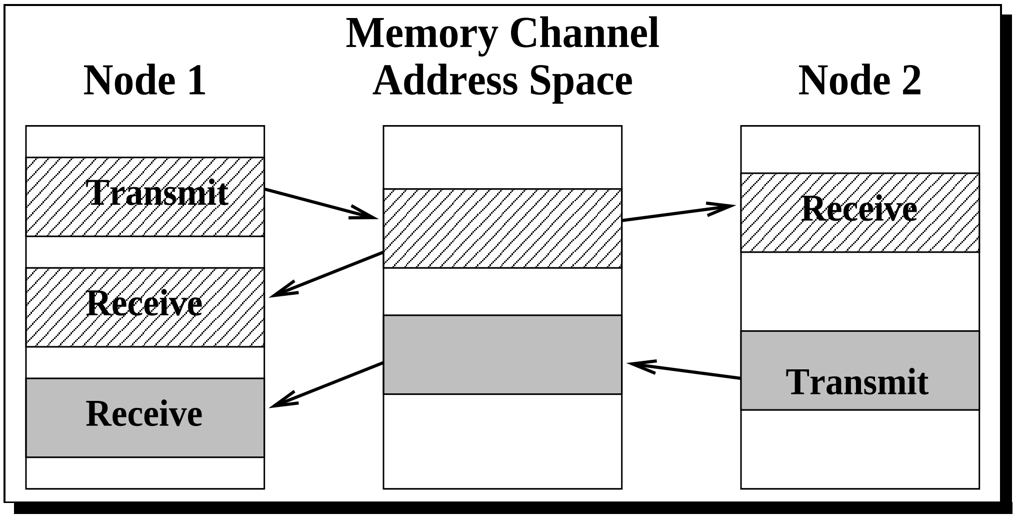


Figure 3.5: The Memory Channel Space

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 3.1: Database Properties

Synthetic Databases All the partitioned databases reside on the local disks of each processor. We used different synthetic databases, generated using the procedure described in the last chapter. Table 3.1 shows the databases used and their properties. The number of transactions is denoted as $\mathcal{D}_r$, where r is the replication factor. 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 $N = 1000$. We refer the reader to [Agrawal and Srikant, 1994] for more detail on the database generation. All the experiments were performed with a minimum support value of 0.25%. The number of large itemsets discovered are shown in

Figure 3.6. 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.

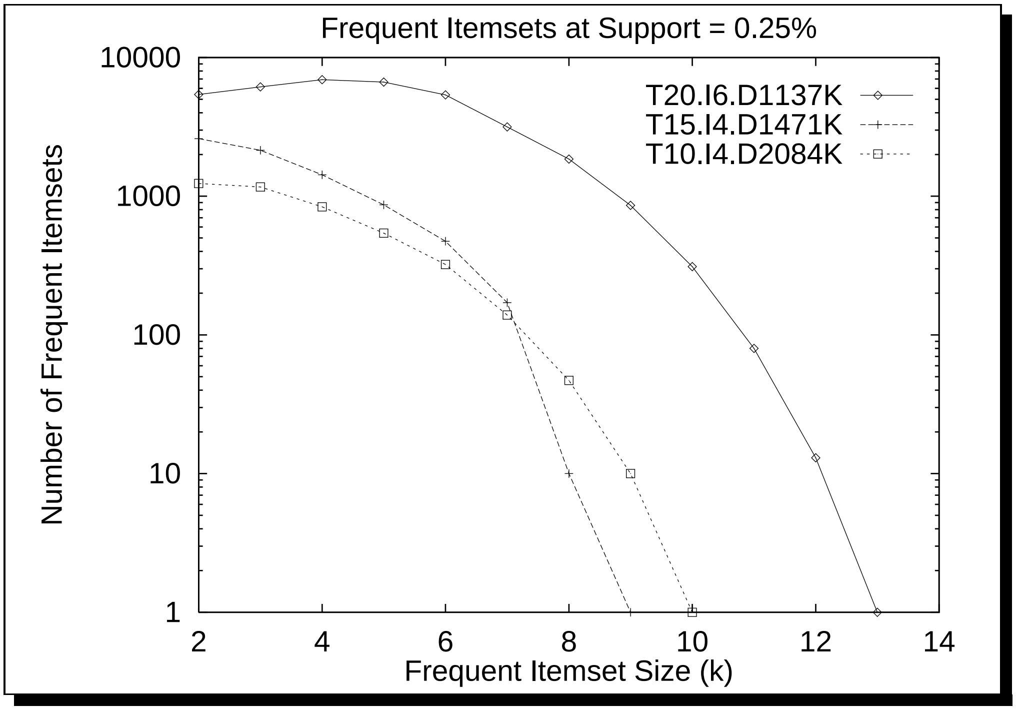


Figure 3.6: Number of Frequent Itemsets

3.5.1 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 and 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 3.7 and Figure 3.8 show 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. Figure 3.7 compares *Par-MaxClique*, the best new algorithm, with *Par-Eclat* and CD, while Figure 3.8 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 *et al.*, 1997a]. An interesting trend in the figures is that the performance gap seems to decrease at larger configurations, with CD actually performing better on the 32 pro-

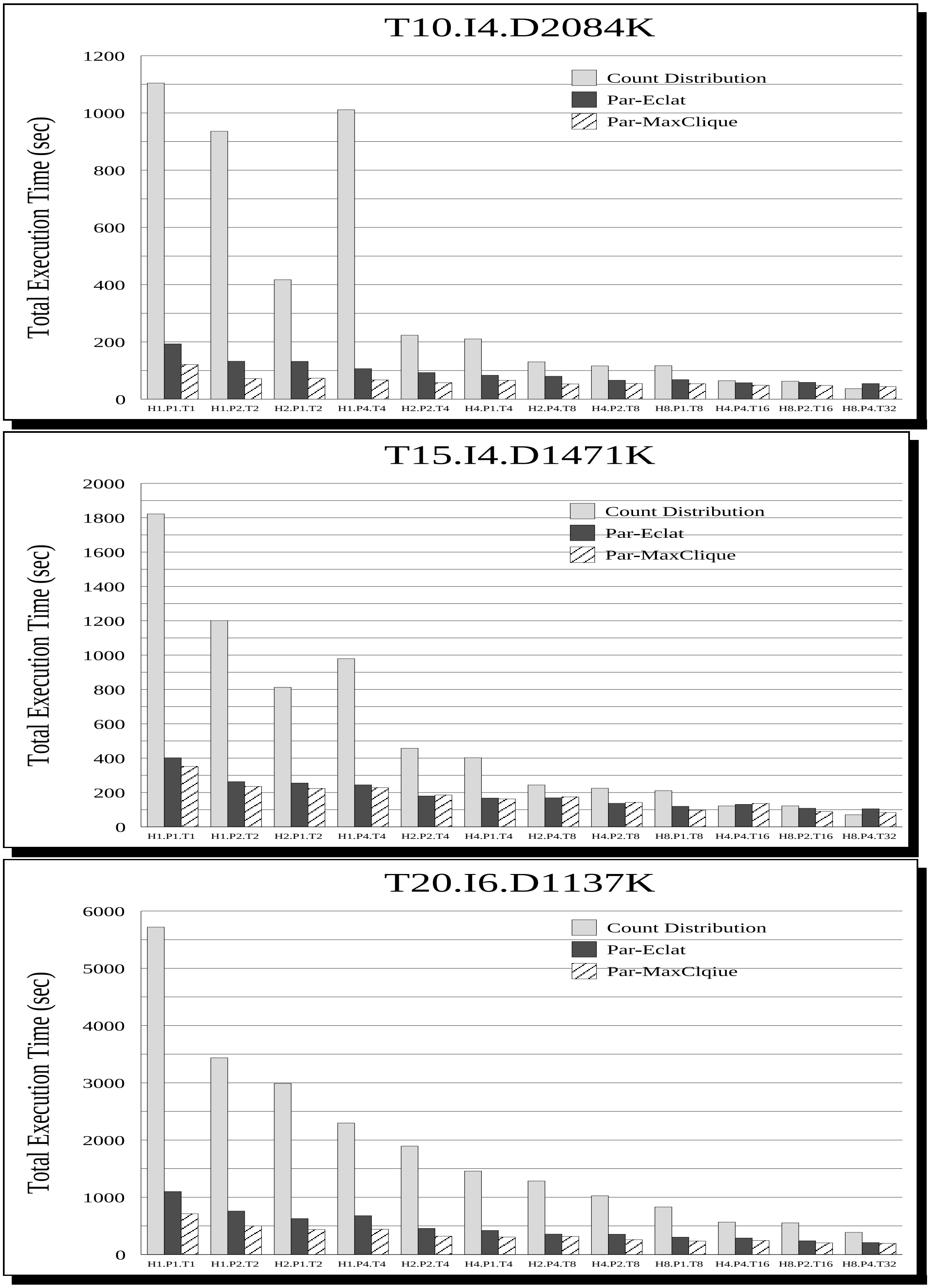


Figure 3.7: Parallel Performance: *Par-Eclat* vs *Count Distribution*

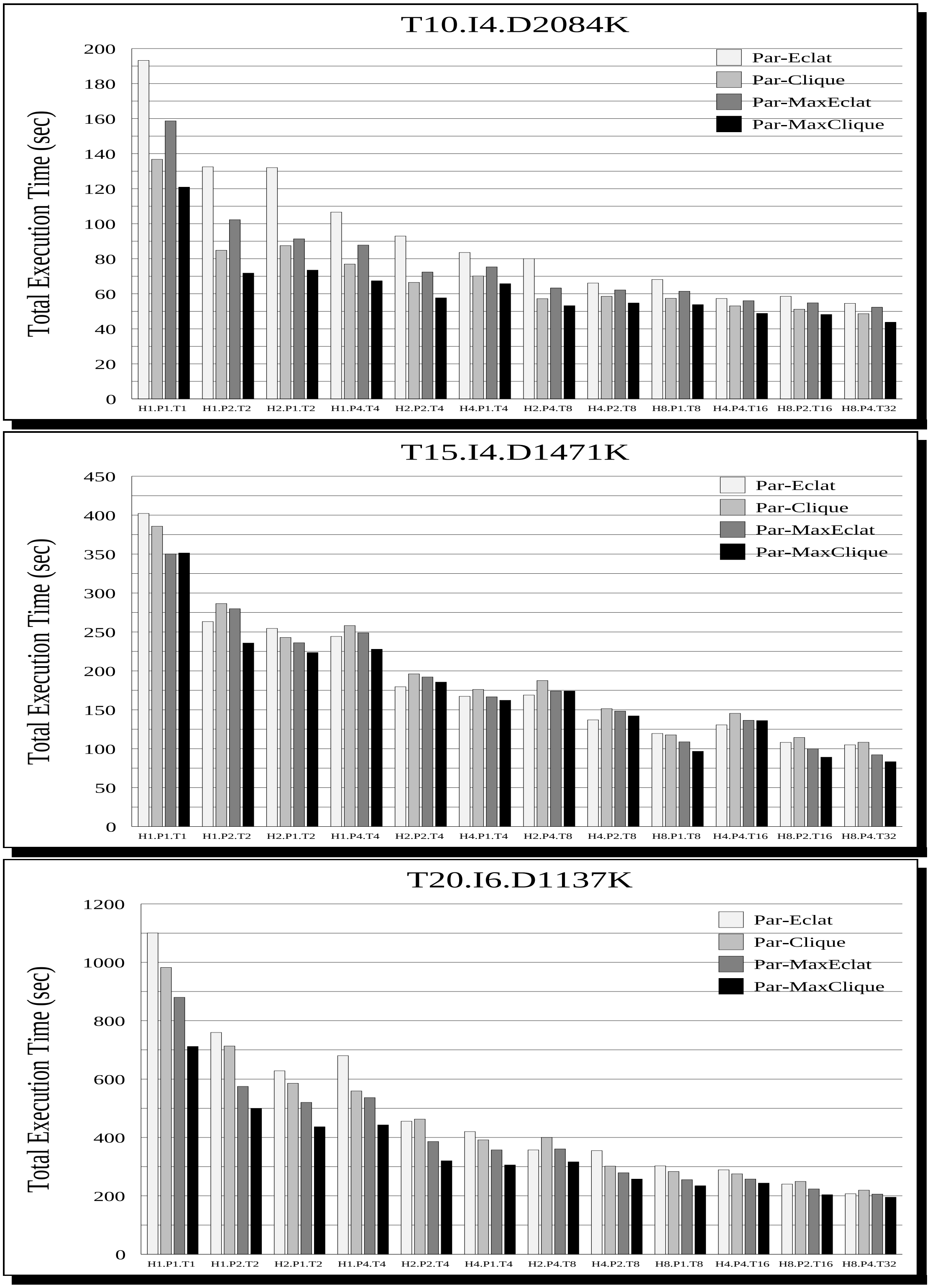


Figure 3.8: Parallel Performance: New Algorithms

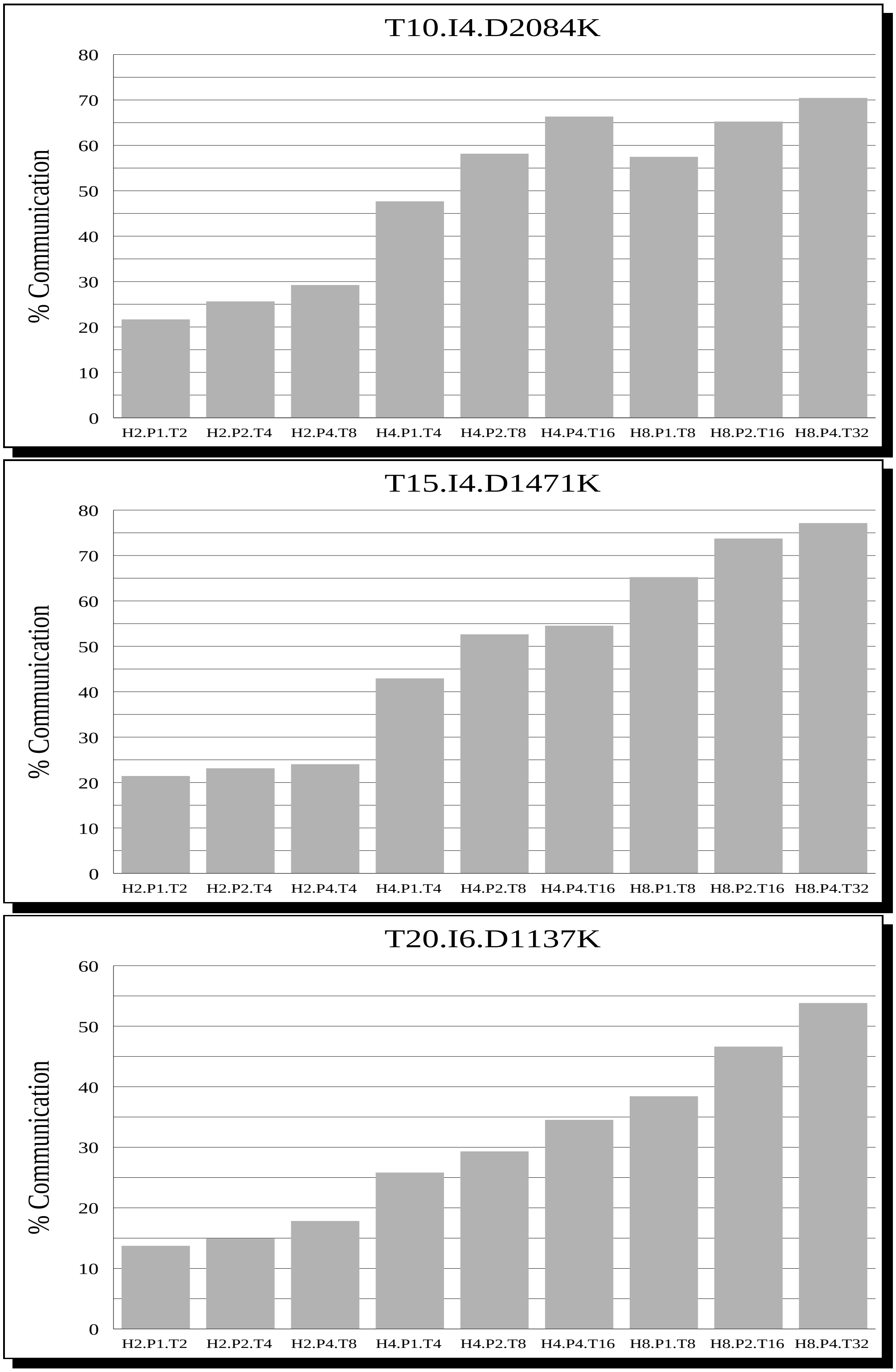


Figure 3.9: Communication Cost in *Par-Eclat*

cessor configuration H8.P4.T32 for the databases T10.I4.D2084K and T15.I4.D1471K. To see why, consider Figure 3.6, which shows the total number of frequent itemsets of different sizes for the different databases. Also from Figure 3.9, 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 3.4, *Par-Eclat* was designed to scan the database only once during frequent itemset computation, and would thus benefit more with larger database size.

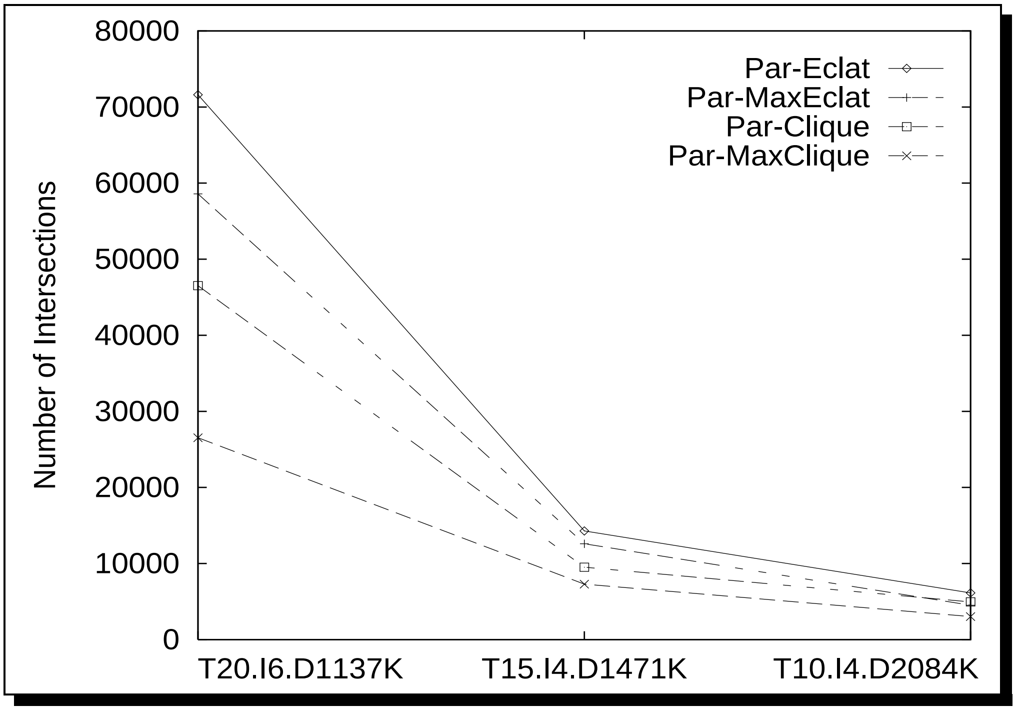


Figure 3.10: Number of Intersections

Figure 3.8 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 3.10, 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 3.6), *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%.

3.5.2 Memory Usage

Figure 3.11 shows the total memory usage of the *Par-Eclat* algorithm as the computation of frequent itemsets progresses. The mean memory usage for the tid-lists is less than 0.7MB for all databases, even though the database itself is over 360MB. The figure only shows the cases where the memory usage was more than twice the mean. The peaks in the graph are usually due to the initial construction of all the 2-itemset tid-lists within each class. The maximum amount of memory consumed by the prefix-based approach of *Par-Eclat* was 35MB, which is still less than 10% of the database. For the maximal-clique-based algorithms, we expect these peaks to be lower, since the classes are smaller.

3.5.3 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 3.12, 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 3.6), and that about 50% to 70% of the time is spent in tid-list communication (see Figure 3.9), 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

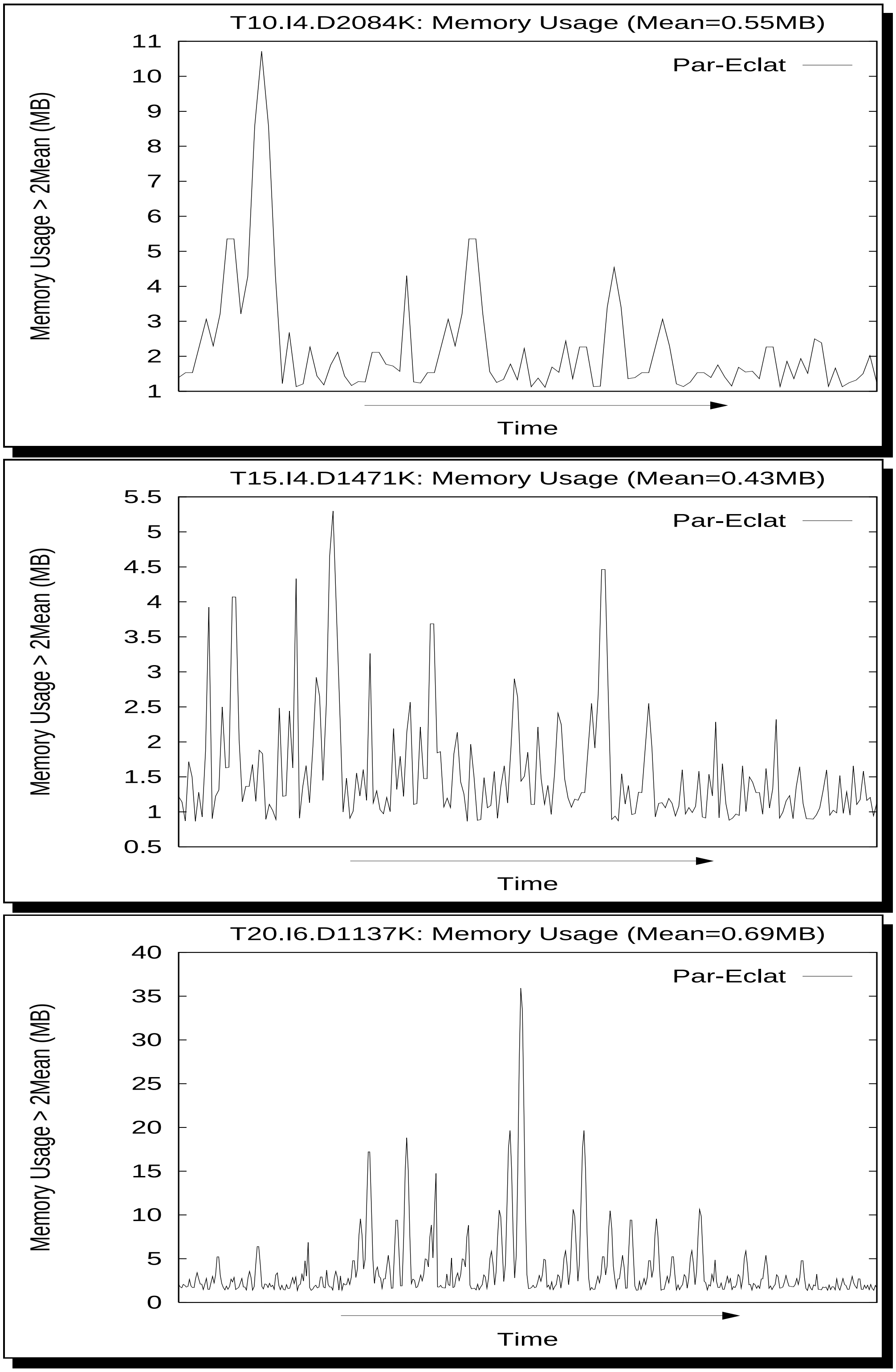


Figure 3.11: Memory Usage in *Par-Eclat* (H1.P1.T1)

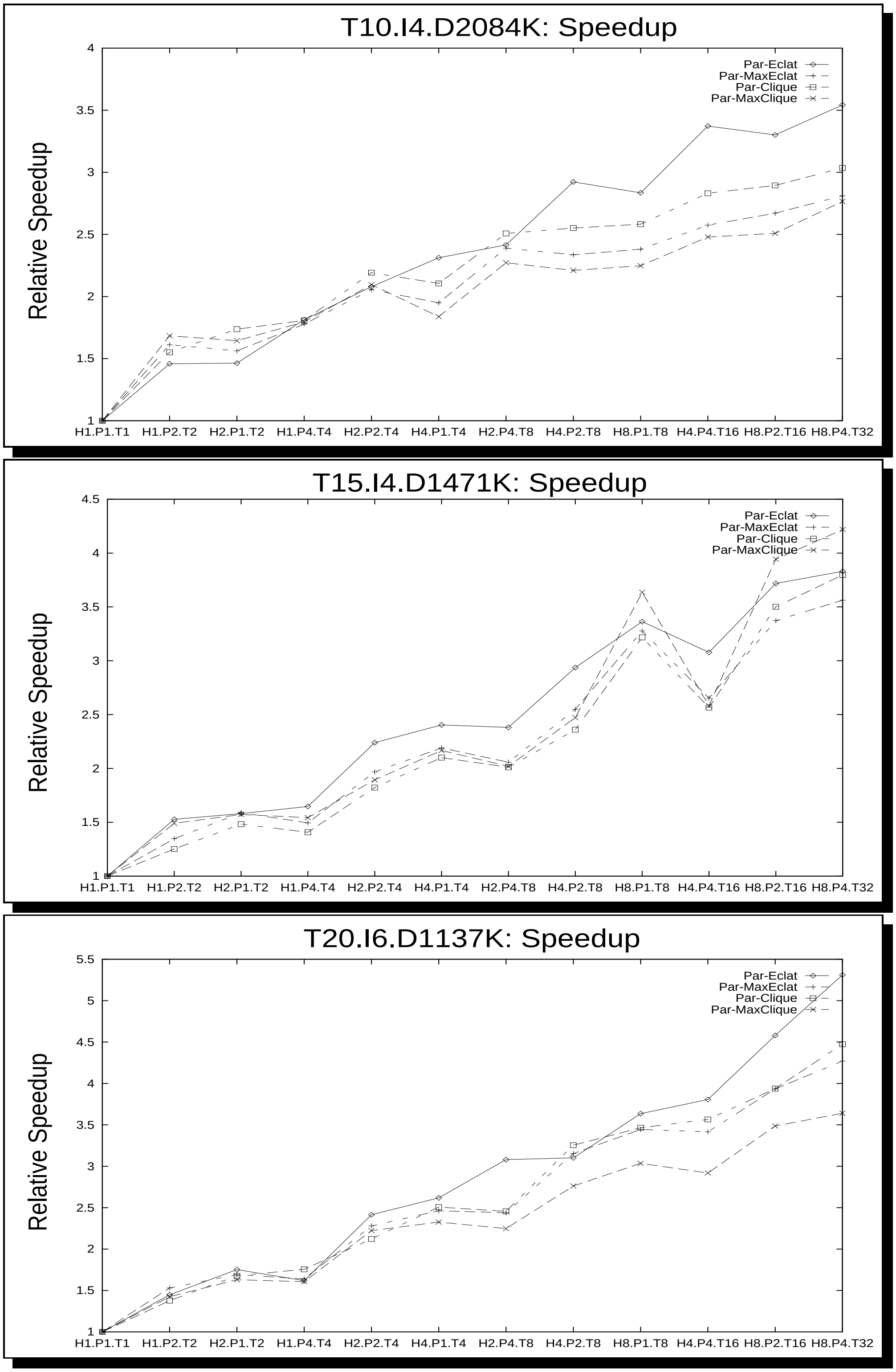


Figure 3.12: Parallel Speedup (H4.P1.T4)

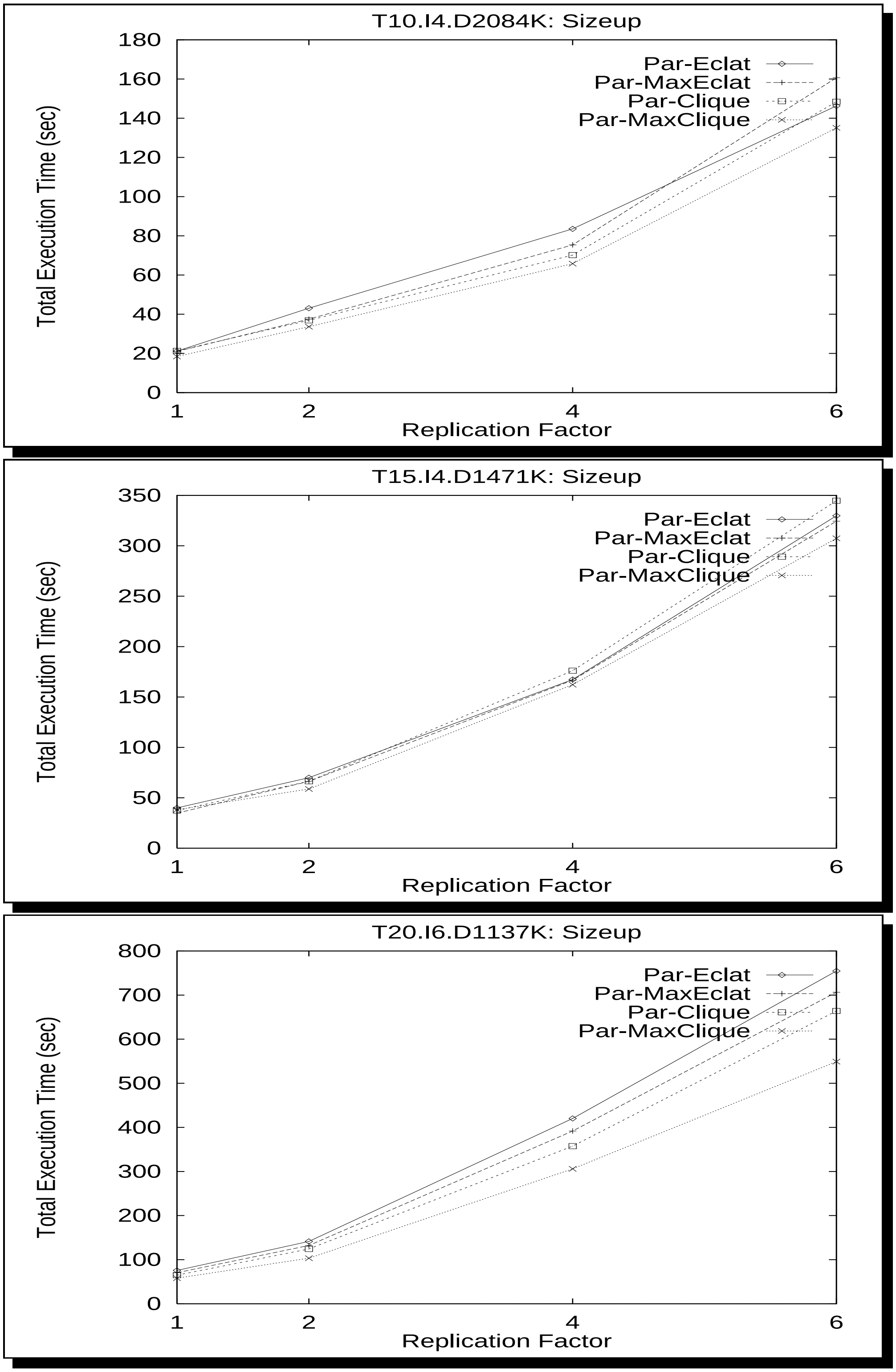


Figure 3.13: Parallel Sizeup (H4.P1.T4)

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 3.13 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.

3.6 Conclusions

In this chapter we proposed new parallel algorithms for the discovery of association rules. The algorithms use the prefix-based and maximal-clique-based decomposition techniques, and the bottom-up and hybrid search schemes. 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 and Shafer, 1996]. Experimental results indicate that our best parallel algorithm *Par-MaxClique* outperformed *Count Distribution* by upto an order of magnitude.

4 Theoretical Foundations of Association Rules

4.1 Introduction

In the previous chapters we developed efficient algorithms for mining association rules. Most of the extant research effort has also focused on the algorithmic task [Agrawal *et al.*, 1993b; Agrawal *et al.*, 1996; Bayardo, 1998; Brin *et al.*, 1997; Houtsma and Swami, 1995; Lin and Kedem, 1998; Lin and Dunham, 1998; Park *et al.*, 1995a; Savasere *et al.*, 1995; Toivonen, 1996; Yen and Chen, 1996; Zaki *et al.*, 1997c]. On the other hand, there has been little work in formulating a theory of associations. Such a theory can help in estimating the complexity of the mining task, and also in developing a unified framework for common data mining problems.

This chapter begins by presenting some complexity results based on the connection between frequent itemsets and bipartite cliques. We then place association rule mining within the lattice-theoretic framework of formal concept analysis introduced by Wille [Wille, 1982]. Given a binary relation, a *concept* consists of an *extent* (transactions) and an *intent* (attributes), such that all objects in the extent share the attributes in the intent, and vice versa. We show that all frequent itemsets are uniquely determined by the set of *frequent concepts*. We then tackle the problem of generating a *base*, a minimal rule set, from which all the other association rules can be inferred. The concept lattice framework can not only aid in the development of efficient algorithms, but can also help in the visualization of discovered associations, and can provide a unifying framework for reasoning about associations, and supervised (classification) and unsupervised (clustering) concept learning [Carpineto and Romano, 1993; Carpineto and Romano, 1996; Godin *et al.*, 1991].

The rest of the chapter is organized as follows. We discuss related work in Section 4.2. A graph-theoretic view of the association mining problem is given in Section 4.3. Section 4.4 casts association mining as a search for frequent concepts, and Section 4.5 looks at the problem of generating rule bases. We conclude in Section 4.6.

4.2 Related Work

In the previous chapters we have already discussed the astonishing amount of research in developing efficient algorithms for mining frequent itemsets [Aggarwal and

Yu, 1998; Agrawal *et al.*, 1993b; Agrawal *et al.*, 1996; Bayardo, 1998; Brin *et al.*, 1997; Houtsma and Swami, 1995; Lin and Kedem, 1998; Lin and Dunham, 1998; Park *et al.*, 1995a; Savasere *et al.*, 1995; Toivonen, 1996; Zaki *et al.*, 1997c]. In [Gunopulos *et al.*, 1997b; Gunopulos *et al.*, 1997a], the connection between associations and hypergraph transversals was made. They also presented a model of association mining as the discovery of maximal elements of theories, and gave some complexity bounds.

A lot of algorithms have been proposed for generating the Galois lattice of concepts [Carpineto and Romano, 1993; Ganter, 1987; Godin *et al.*, 1991; Guenoche, 1990; Kuznetsov, 1993]. An incremental approach for building the concepts was studied in [Carpineto and Romano, 1996; Godin *et al.*, 1991]. These algorithms will have to be adapted to enumerate only the frequent concepts. Further, they have only been studied on small datasets. It remains to be seen how scalable these approaches are compared to the association mining algorithms. Finally, there has been some work in pruning discovered association rules by forming rule covers [Toivonen *et al.*, 1995]. However, the problem of constructing a base or generating set has not been studied previously. Some parts of this chapter have appeared in [Zaki and Ogihara, 1998].

4.3 Itemset Discovery: Bipartite Graphs

Definition 4.1 *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 is called a* **bipartite clique**.

Definition 4.2 *An* **independent set** *of a graph* $G = (V, E)$ *is a set of vertices* $I \subseteq V$*, such that no two vertices in* I *are connected by an edge.*

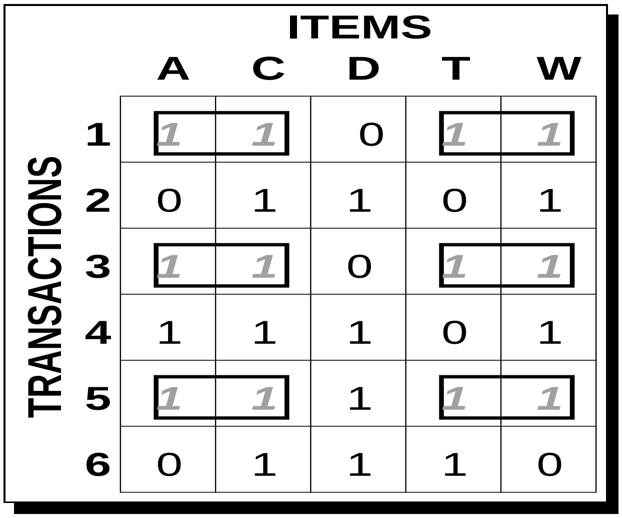
Definition 4.3 *A* **hypergraph** ([Berge, 1989]) *on* $\mathcal{I}$ *is a family* $H = \{E_1, E_2, ..., E_n\}$ *of edges or subsets of* $\mathcal{I}$*, such that* $E_i \neq \emptyset$*, and* $\bigcup_{i=1}^{n} E_i = \mathcal{I}$*.*

Definition 4.4 *Let* $H = \{E_1, E_2, ..., E_n\}$ *be a hypergraph on a set* $\mathcal{I}$*. A set* $T \subset \mathcal{I}$ *is a* **transversal** *of* H *if it intersects all the edges, that is to say:* $T \cap E_i \neq \emptyset$*, for all* E_i*.*

A hypergraph is a generalization of a graph in the following sense. While a graph has edges connecting only two vertices, a hypergraph has edges connecting a set of vertices. A transversal in a hypergraph is called a vertex cover in a graph.

Lemma 4.1 *There is a one-to-one correspondence between bipartite graphs, hypergraphs, and binary matrices.*

Lemma 4.2 *A clique of a graph* $G = (V, E)$ *corresponds to an independent set in the complementary graph* $\overline{G} = (V, E')$*, with* $E' = \{(u, v) \mid (u, v) \notin E\}$*. In a bipartite graph, an independent set is the complement of a vertex cover* ([Harary, 1969])*, i.e., if* $C \subset V$ *is a vertex cover, then* $I = V \backslash C$ *is an independent set.*



TRANSACTIONS	ITEMS				
	A	C	D	T	W
1	1	1	0	1	1
2	0	1	1	0	1
3	1	1	0	1	1
4	1	1	1	0	1
5	1	1	1	1	1
6	0	1	1	1	0

Figure 4.1: Maximal Unit Sub-Matrix of a Binary Matrix

The input database for association mining, as shown in Figure 2.1, 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, $I \times T$, where $I \subseteq U$, $T \subseteq V$, and $|T| \geq \mathit{min\_sup}$. Due to the one-to-one correspondence between bipartite graphs, binary matrices and hypergraphs, one can also view it as the problem of enumerating all (maximal) unit sub-matrices in a binary matrix, or as the problem of enumerating all (minimal) transversals of a hypergraph, satisfying the support constraints. Figures 4.1 and 4.2 show the bookstore database as a binary matrix, and as a bipartite graph, respectively. They also show the maximal

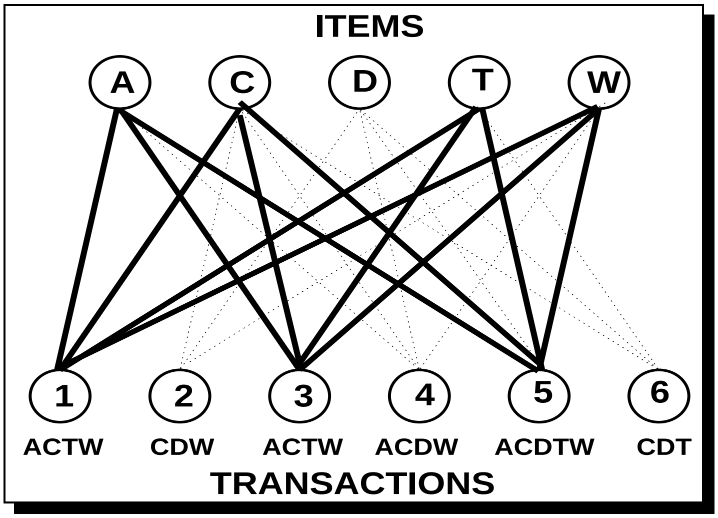


Figure 4.2: Maximal Constrained Bipartite Clique

unit sub-matrix and the maximal constrained bipartite clique $ACTW \times 135$ (i.e., the maximal frequent itemset $ACTW$).

Definition 4.5 *Let Σ^* denote strings composed of the symbols of a finite set Σ. Let $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation on strings. R is called* **polynomially decidable** *if there is a deterministic Turing Machine deciding the language $\{X; Y \mid (X,Y) \in R\}$ in polynomial time. R is* **polynomially balanced** *if $(X,Y) \in R$ implies $|Y| \leq |X|^k$ for some $k \geq 1$.*

Lemma 4.3 (see, e.g. [Papadimitriou, 1994]) *Let $L \subseteq \Sigma^*$ be a language. L is in the class* **NP** *if and only if there is a polynomially decidable and polynomially balanced relation R, such that $L = \{X \mid (X,Y) \in R \text{ for some } Y\}$.*

Definition 4.6 *Let C be a complexity class, and let L be a language in C. L is called* **C-Complete** *if any language $L' \in C$ is* **polynomial-time reducible** *to L, i.e., if there exists a polynomial-time computable function f such that $X \in L'$ iff $f(X) \in L$.*

Definition 4.7 ([Valiant, 1979]) *Let Q be a polynomially balanced, polynomial-time decidable binary relation. The* **counting problem** *for Q is the following: Given X, how many Y are there such that $(X,Y) \in Q$?* **#P** *is the class of all counting problems associated with polynomially balanced polynomial-time decidable relations.*

The above definitions specify important tools for determining the computational complexity of various problems. For example, showing that a problem is NP-Complete establishes that it is unlikely to be solved in polynomial time. In this sense, these problems are hard and one should perhaps concentrate on developing approximation algorithms for the problem, or on developing solutions for special cases, etc. 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.

Having given the basic notions of computational complexity, we now direct our attention to the complexity of mining frequent itemsets. Table 4.1 shows the complexity of decision problems for maximal bipartite cliques (itemsets) with restrictions on the size of I (items) and 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 hand, the problem whether there exists a maximal bipartite clique such that $|I| + |T| = K$ is NP-Complete [Kuznetsov, 1989]. 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 [Kashiwabara *et al.*, 1992], which has $O((|U| + |V|)^{2.5})$ complexity. The following theorem says that determining the number of maximal cliques in a bipartite graph is extremely hard.

Theorem 4.1 ([Kuznetsov, 1989]) *Determining the number of maximal bipartite cliques in a bipartite graph is #P-Complete.*

	$\leq K$	$= K$	$\geq K$
$\|I\|$	P	NP-Complete	P
$\|T\|$	P	NP-Complete	P
$\|I\| + \|T\|$	NP-Complete	NP-Complete	P

Table 4.1: Mining Complexity

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.

Definition 4.8 *The* **arboricity** $r(G)$ *of a graph is the minimum number of forests into which the edges of G can be partitioned. The arboricity of a graph is given as*

$$r(G) = \max_{H \subset G} \{e(H)/(n(H) - 1)\}$$

where $n(H)$ *is the number of vertices and* $e(H)$ *the number of edges of an non-trivial subgraph* $H \subset G$.

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 $I \times T$ is a maximum bipartite clique. Furthermore, if we assume $|I| \ll |T|$ (since we want large support), then $r(G) \approx |I|$, i.e., the arboricity of the association database is given by the maximum sized frequent itemset. For sparse graphs, of bounded arboricity $|I|$, the complexity of finding all maximal bipartite cliques is linear in number of items and transactions:

Theorem 4.2 ([Eppstein, 1994]) *All maximal bipartite cliques can be enumerated in time* $O(|I|^3 \cdot 2^{2|I|} \cdot (|U| + |V|))$.

Even though the above algorithm has linear complexity, it is not practical for large databases due to the large constant overhead ($|I|$ can easily be around 10 to 20 in practice). Nevertheless, the result is very encouraging, and gives a reason why all current association mining algorithms exhibit linear scalability in database size. This result also says that at least in theory the association mining algorithms should scale linearly in the number of items or attributes, a very important feature if practicable.

Theorem 4.3 ([Kashiwabara *et al.*, 1992]) *All maximum independent sets can be listed in* $O((|U| + |V|)^{2.5} + \gamma)$ *time, where* γ *is output size.*

The above theorem states that all the maximum (largest) bipartite cliques (independent sets in complimentary 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 input polynomial time. 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.

4.4 Itemset Discovery: Formal Concept Analysis

In this section we will show that association mining is very closely related to *formal concept analysis*, which was introduced in a seminal paper by Wille [Wille, 1982].

Definition 4.9 *Let S be a set. A function $c : \mathcal{P}(S) \mapsto \mathcal{P}(S)$ is a* **closure operator** *on S if, for all $X, Y \subseteq S$, c satisfies the following properties:*

1) Extension: $X \subseteq c(X)$.

2) Monotonicity: if $X \subseteq Y$, then $c(X) \subseteq c(Y)$.

3) Idempotency: $c(c(X)) = c(X)$.

A subset X of S is called **closed** *if $c(X) = X$.*

Definition 4.10 *A* **context** *is a triple (G, M, I), where G and M are sets and $I \subseteq G \times M$. The elements of G are called* **objects**, *and the elements of M are called* **attributes**. *For an arbitrary $g \in G$, and $m \in M$, we note gIm, when g is related to m, i.e., $(g, m) \in I$.*

Definition 4.11 *Let (G, M, I) be a context with $X \subseteq G$, and $Y \subseteq M$. Then the mappings*

$$s : G \mapsto M, s(X) = \{m \in M \mid \ (\forall g \in X) \ gIm\}$$

$$t : M \mapsto G, t(Y) = \{g \in G \mid \ (\forall m \in Y) \ gIm\}$$

define a **Galois connection** *between $\mathcal{P}(G)$ and $\mathcal{P}(M)$, the power sets of G and M, respectively.*

The set $s(X)$ is the set of attributes common to all the objects in X and $t(Y)$ is the set of objects common to all the attributes in Y. We note that $X_1 \subseteq X_2 \Rightarrow s(X_2) \subseteq s(X_1)$, for $X_1, X_2 \subseteq G$ (and “dually” for function t on M). Furthermore, the compositions $c = s \circ t$ and dually, $t \circ s$ are closure operators.

Definition 4.12 *A* **concept** *of the context (G, M, I) is defined as a pair (X, Y), where $X \subseteq G$, $Y \subseteq M$, $s(X) = Y$, and $t(Y) = X$. In other words, a concept (X, Y) consists of the closed sets X and Y, since $X = t(Y) = t(s(X)) = s \circ t(X) = c(X)$, and similarly $Y = c(Y)$. X is also called the* **extent** *and Y the* **intent** *of the concept (X, Y).*

Definition 4.13 *The concept generated by a single attribute $m \in M$ given as $\alpha(m) = (t(m), c(m))$ is called an* **attribute concept**, *while the concept generated by a single object $g \in G$ given as $\beta(g) = (c(g), s(g))$ is called an* **object concept**.

Definition 4.14 *A concept (X_1, Y_1) is a* **subconcept** *of (X_2, Y_2), denoted as $(X_1, Y_1) \leq (X_2, Y_2)$, iff $X_1 \subseteq X_2$ (iff $Y_2 \subseteq Y_1$).*

Notice that the mappings between the closed sets of G and M are anti-isomorphic, i.e., concepts with large extents have small intents, and vice versa. The set of all concepts of the context is denoted by $\mathcal{B}(G, M, I)$. The different concepts can be organized as a hierarchy of concepts based on the superconcept-subconcept partial order.

Definition 4.15 *A subset P of an ordered set Q is* **join-dense** *if $\forall q \in Q$, there exists $Z \subseteq P$, such that $q = \bigvee_Q Z$ (and dually we can define* **meet-dense***).*

The fundamental theorem of formal concept analysis can now be stated as follows:

Theorem 4.4 ([Wille, 1982]) *Let (G, M, I) be a context. Then $\mathcal{B}(G, M, I)$ is a complete lattice with join and meet given by*

$$\bigvee_j (X_j, Y_j) = (c(\bigcup_j X_j), \bigcap_j Y_j) \qquad \bigwedge_j (X_j, Y_j) = (\bigcap_j X_j, c(\bigcup_j Y_j))$$

Conversely, if L is a complete lattice then L is isomorphic to $\mathcal{B}(G, M, I)$ iff there are mappings $\gamma : G \mapsto L$, and $\mu : M \mapsto L$, such that $\gamma(G)$ is join-dense in L, $\mu(M)$ is meet-dense in L, and gIm is equivalent to $\gamma(g) \leq \mu(M)$ for all $g \in G$ and $m \in M$. In particular L is isomorphic to $\mathcal{B}(L, L, \leq)$.

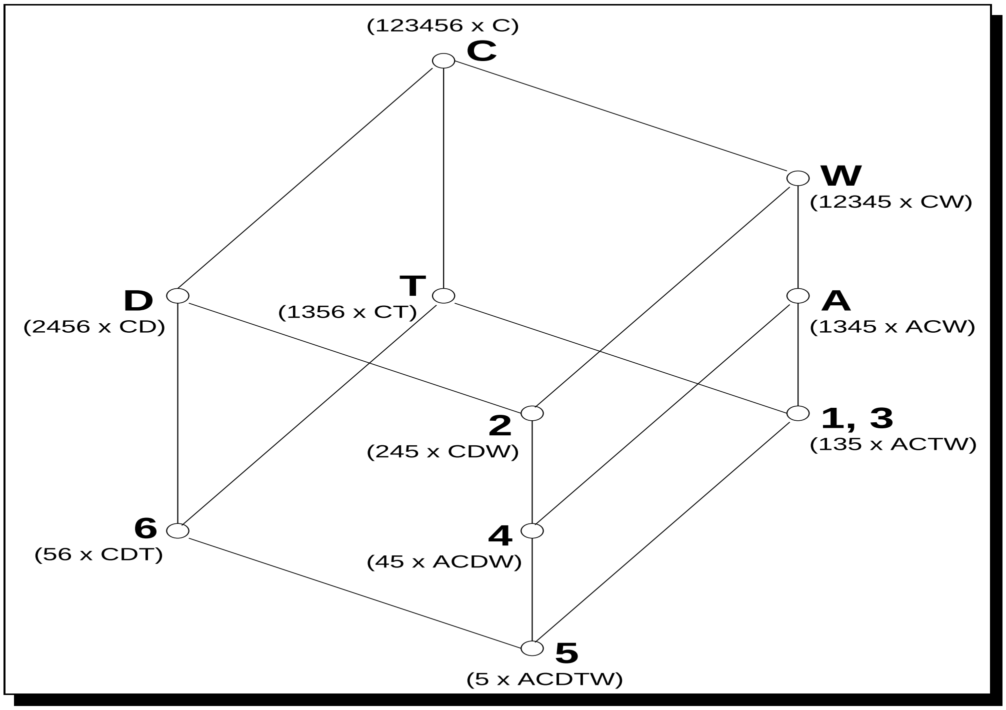


Figure 4.3: Galois Lattice of Concepts

The complete lattice $\mathcal{B}(G, M, I)$ is called the *Galois* lattice of the context. The concept lattice can be represented graphically by a *Hasse diagram*, where each concept is a circle, and for concepts $c_1 \leq c_2$, there is a line joining them, with c_1 being lower than c_2. For example, Figure 4.3 shows the Galois lattice for our example database.

It is shown with a *minimal labeling*, where the intent (extent) of a concept can be reconstructed by considering all labels reachable above (below) that concept. In other words, each concept is labeled with an attribute (object) if it is an attribute (object) concept. It is clear that an appropriately drawn diagram can aid in visualizing and understanding the relationships among the attributes and objects (i.e., associations).

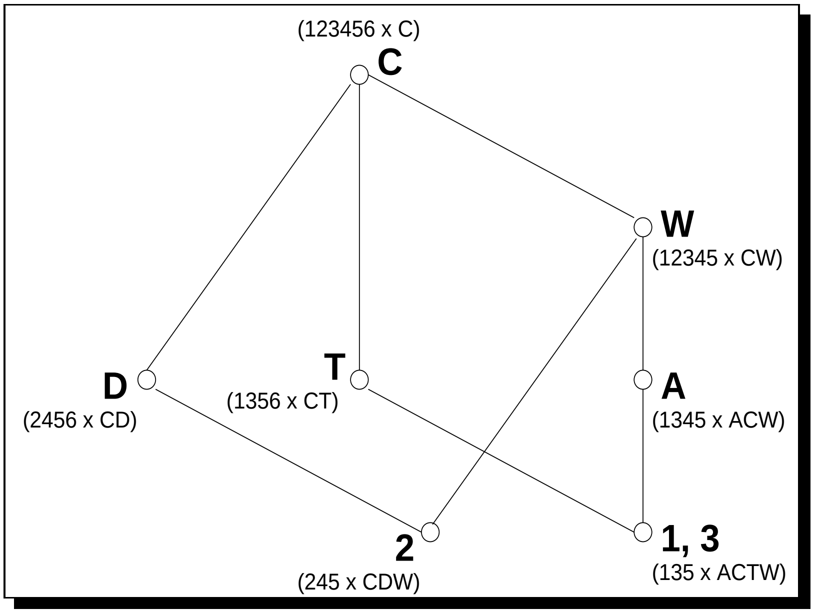


Figure 4.4: Frequent Concepts of the Galois Lattice

Define a *frequent concept* as a concept (X, Y) with $X \subseteq G$, $Y \subseteq M$, and $|X| \geq$ *min_sup*. Figure 4.4 shows all the frequent concepts with *min_sup* = 50%. All frequent itemsets can be determined by the meet operation on attribute concepts. For example, since meet of attribute concepts D and T, $\alpha(D) \wedge \alpha(T)$, doesn't exist, DT is not frequent, while $\alpha(A) \wedge \alpha(T) = (135, ACTW)$, thus AT is frequent. Furthermore, the support of AT is given by the cardinality of the resulting concept's extent, i.e., $\sigma(AT) = |\{1,3,5\}| = 3$. Thus all frequent itemsets are uniquely determined by the frequent concepts. This observation can possibly aid the development of efficient algorithms since we need to enumerate only the closed frequent itemsets, instead of enumerating all frequent itemsets like most current algorithms. In our example database, there are only 7 closed frequent itemsets versus 19 frequent itemsets.

4.5 Rule Generation

Association rules were originally proposed in [Agrawal *et al.*, 1993b]. However, we will show below that association rules are exactly the *partial implications*, satisfying support and confidence constraints, proposed in an earlier paper [Luxenburger, 1991].

Let (G, M, I) be a context. A *partial implication rule* $X \overset{p}{\Rightarrow} Y$ is a triple (X, Y, p), where $X, Y \subseteq M$ are sets of attributes, and the *precision* $p = P(Y|X) = |t(X \cup$

$Y)|/|t(X)|$. Clearly, association rules correspond to partial implications meeting the support and confidence constraints, i.e., with $|t(X \cup Y)| \geq min\_sup$, and $p \geq min\_conf$, respectively.

Definition 4.16 *Let $\mathcal{S} = \{X \overset{p}{\Rightarrow} Y \mid X, Y \subseteq M, p = P(Y|X)\}$, be a set of partial implications. A rule $K \overset{p}{\Rightarrow} L$ can be* **derived** *from a set of rules $\mathcal{S}$, denoted $\mathcal{S} \vdash K \overset{p}{\Rightarrow} L$, iff we can obtain $K \overset{p}{\Rightarrow} L$ from $\mathcal{S}$ by applying certain inference rules. In this case we also call $K \overset{p}{\Rightarrow} L$ a* **redundant** *rule according to $\mathcal{S}$.*

Definition 4.17 *A set $\mathcal{R} \subseteq \mathcal{S}$ is called a* **generating set** *for $\mathcal{S}$ iff $\mathcal{R} \vdash \mathcal{S}$. A minimal-by-inclusion generating set is called a* **base**.

Since the set of all partial implications (i.e., association rules) can be very large, we are interested in finding a base for it. This means that only a small and easily understandable set of rules can be presented to the user, who can later selectively derive other rules of interest. The set of partial implications can be broken into two parts: implications with $p = 1$, called the *global implications*, and those with $p < 1$, called the *proper partial implications*. A base for all partial implications can be obtained by combining the bases of these two sets.

Global Implications A *global implication rule* is denoted as $X \Rightarrow Y$, where $X, Y \subseteq M$, and $t(X) \subseteq t(Y)$, i.e., all objects related to X are also related to Y. It can be shown that $t(X) \subseteq t(Y) \Leftrightarrow P(Y|X) = 1$. Thus, global implications are precisely the association rules with 100% confidence. A global implication can be directly discerned from the Hasse diagram of the concept lattice, since in this case the meet of attribute concepts in X is less than (lies below) the meet of attribute concepts in Y. For example, consider the frequent concepts in Figure 4.6. $AD \Rightarrow CW$, since $\alpha(A) \wedge \alpha(D) = (45, ACDW) \leq \alpha(C) \wedge \alpha(W) = (12345, CW)$. The problem of finding a base of all global implication rules has been well studied [Ganter, 1987; Guigues and Duquenne, 1986; Maier, 1983; Wild, 1991]. One characterization of a base is given as follows:

Theorem 4.5 ([Ganter, 1987]) *The set $\{X \Rightarrow c(X)\backslash X \mid X$ is a pseudo-intent$\}$ is a base for all global implications, where X is a* **pseudo-intent** *if $X \neq c(X)$, and for all pseudo-intents $Q \subset X$, $c(Q) \subseteq X$.*

For example, $\{A, D, T, W, CTW\}$ is the set of pseudo-intents in our example database. A base of global implications is thus given by the set $\mathcal{R} = \{A \Rightarrow CW,\ D \Rightarrow C,\ T \Rightarrow C,\ W \Rightarrow C,\ CTW \Rightarrow A\}$. This set is shown in Figure 4.5. All other global implications can be derived from $\mathcal{R}$ by application of simple inference rules such as those given in [Maier, 1983, pp. 47], 1) Reflexivity: $X \subseteq Y$ implies $Y \Rightarrow X$, 2) Augmentation: $X \Rightarrow Y$ implies $XZ \Rightarrow YZ$, 3) Transitivity: $X \Rightarrow Y$ and $Y \Rightarrow Z$ implies $X \Rightarrow Z$, and so on.

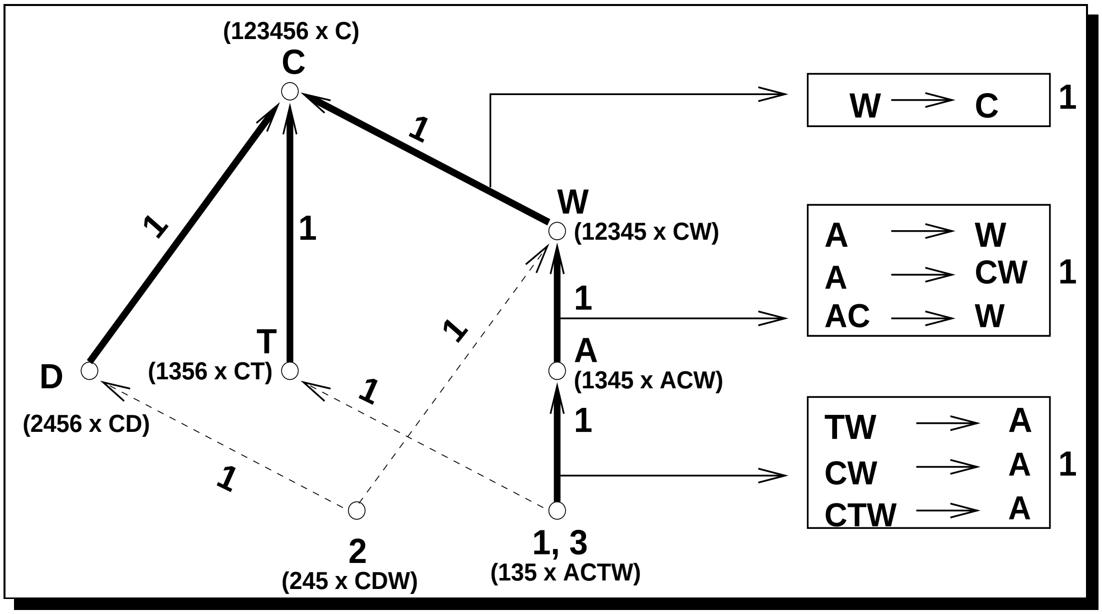


Figure 4.5: A Base for Global Implications

Proper Partial Implications We now turn to the problem of finding a base for *proper* partial implications with $p < 1$, i.e., association rules with confidence less than 100%. Note that for any $Z \subseteq X$, $P(Y|X) = P(Y \cup Z|X)$, and thus $X \overset{p}{\Rightarrow} Y$ iff $X \overset{p}{\Rightarrow} Y \cup Z$. In particular, $X \overset{p}{\Rightarrow} Y$ iff $X \overset{p}{\Rightarrow} X \cup Y$. We thus only discuss the rules $X \overset{p}{\Rightarrow} Y$, with $X \subseteq Y$. Furthermore, it can be shown that $X \overset{p}{\Rightarrow} Y$ iff $c(X) \overset{p}{\Rightarrow} c(Y)$. We can thus restrict ourselves to only the rules where X and Y are intents of a frequent concept. The set of all proper partial implications is given by $\mathcal{S}^{<1}(\mathcal{B}(G,M,I)) = \{K \overset{p}{\Rightarrow} L \mid K \subset L \text{ are intents of } \mathcal{B}(G,M,I)\}$. The following theorem states that unlike global implications, partial implications satisfy transitivity and commutativity only under certain conditions.

Theorem 4.6 ([Luxenburger, 1991]) *Let $M_1, M_2, M_3, M_4 \subseteq M$ be intents with $M_1 \subseteq M_2 \subseteq M_4$ and $M_1 \subseteq M_3 \subseteq M_4$. Then $P(M_2|M_1) \cdot P(M_4|M_2) = P(M_4|M_1) = P(M_3|M_1) \cdot P(M_4|M_3)$ (i.e., $M_1 \overset{p}{\Rightarrow} M_2$ and $M_2 \overset{q}{\Rightarrow} M_4$ implies $M_1 \overset{pq}{\Rightarrow} M_4$).*

Consider the Hasse diagram of the frequent concepts with the precision on the edges, shown in Figure 4.6. The edge between attribute concepts C and W corresponds to the implication $C \overset{5/6}{\Rightarrow} W$. The reverse implication $W \Rightarrow C$ has precision 1 by definition. Only the implications between adjacent concepts need to be considered, since the other implications can be derived from the above theorem. For example, $C \Rightarrow A$ has precision $p = 4/6$, since $P(A|C) = P(W|C) \cdot P(A|W) = 5/6 \cdot 4/5 = 4/6$. The diagram provides a wealth of embedded information; the link joining attribute concept T and object concept $1, 3$ corresponds to the rule $T \Rightarrow A$. Immediately we can see that it has the same confidence (3/4) as the rules $T \Rightarrow W$, $T \Rightarrow AC$, $T \Rightarrow AW$, $T \Rightarrow ACW$, $CT \Rightarrow A$, $CT \Rightarrow W$, and $CT \Rightarrow AW$. All these other rules are thus redundant! On the other

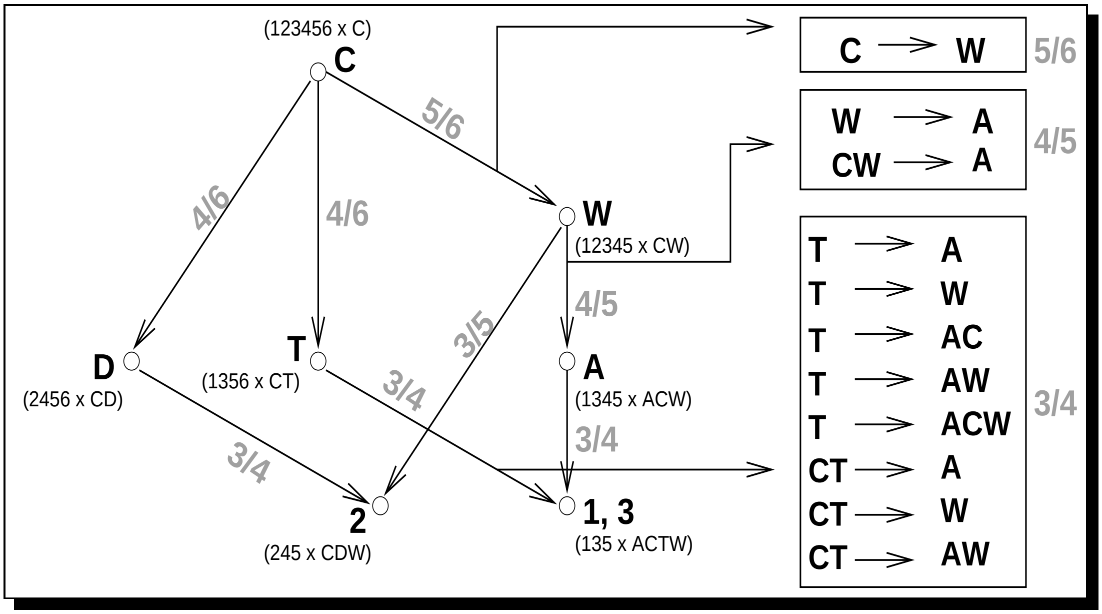


Figure 4.6: Frequent Concepts with Edge Precisions

hand the link from A to $1, 3$ corresponds to the rule $A \Rightarrow T$, which generates another set of redundant rules.

Definition 4.18 ([Rota, 1964]) *The* **Möbius** *function on an ordered set* L *is defined as* $\mu_L : L \times L \mapsto \mathcal{Z}$*, where* $\mathcal{Z}$ *is the set of Integers, and for all* $X, Y, Z \in L$*,*

1) $\mu_L(X, X) = 1$.

2) $\mu_L(X, Y) = -\sum_{X<Z\leq Y} \mu_L(Z, Y)$ *if* $X < Y$*, otherwise* $\mu_L(X, Y) = 0$.

Theorem 4.7 ([Luxenburger, 1991]) *Let* $\mathcal{K} = \mathcal{B}(G, M, I)$ *be the set of concepts of the context* (G, M, I)*, and let* $X, Y \subseteq M$ *be intents. Then the partial implications also satisfy the following properties:*

1) Let $X \subseteq Y$ *and* $P(Y|X) = 0$*. Then* $Y = M$*.*

2) $\sum_{W \in J} P(W|X) \cdot \mu_{\mathcal{K}}((t(W), W), (t(X), X)) \geq 0$*, where* $J = \{Y \mid X \subseteq Y = c(Y)\}$*.*

3) $1 - P(X|X \cap Y) - P(Y|X \cap Y) + P(X|X \cap Y) \cdot P(c(X \cup Y)|X) \geq 0$*.*

For $\mathcal{S}' \subseteq \mathcal{S}^{<1}(\mathcal{B}(G, M, I))$, define the graph $\mathcal{G}(\mathcal{S}') = (V, E)$, with vertex set $V = \{N \subseteq M \mid N \text{ is an intent}\}$, and edge set $E = \{(K, L) \in V \times V \mid K \overset{p}{\Rightarrow} L \in \mathcal{S}'\}$.

Lemma 4.4 ([Luxenburger, 1991]) *If there exists a cycle in* $\mathcal{G}(\mathcal{S}')$*, then there exists a partial implication* $K \in \mathcal{S}'$ *such that* $\mathcal{S}' \backslash K \vdash K$*.*

As a consequence of this lemma, one rule in every cycle is redundant, and it can be discarded. The next theorem gives a more precise characterization of a generating set.

Theorem 4.8 ([Luxenburger, 1991]) *$\mathcal{S}'$ is a generating set if*

1) $Gr(\mathcal{S}')$ is a spanning tree.

2) M is a consequent of only one partial implication in $\mathcal{S}'$.

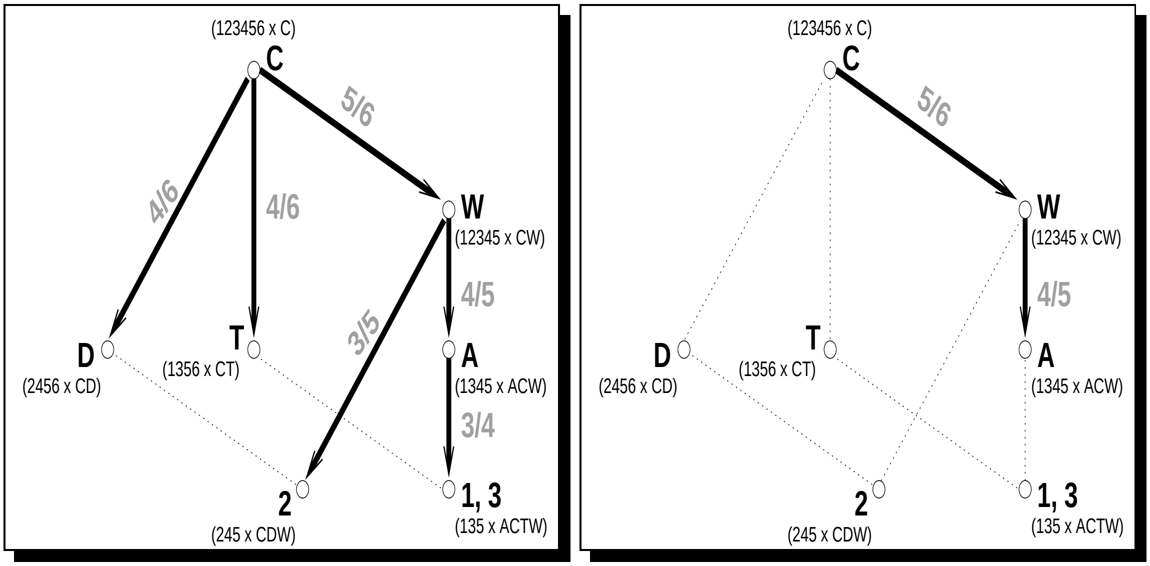


Figure 4.7: a) Generating Set for Precisions Between 50% and 100%; b) Generating Set for Precisions Between 80% and 100%

Figure 4.7a shows a generating set (a minimal spanning tree) for all the proper partial implications in our example. We can derive the precision of a redundant rule by multiplying the precisions of the other rules involved in the cycle (except, we need to invert the precision if we go from a lower concept to a higher concept in the cycle). For example, in Figure 4.7a, the precision of the missing edge $D \Rightarrow W$ can be obtained by multiplying the inverted precision on the edge from D to C, with the precisions on the edges from C to W, and from W to 2, i.e., $\frac{6}{4} \cdot \frac{5}{6} \cdot \frac{3}{5} = \frac{3}{4}$.

To obtain the rules satisfying a given value of min_conf, one can simply discard all edges in the diagram with $p < min\_conf$ For example, Figure 4.7b shows the generating set for rules with $min\_conf = 80\%$.

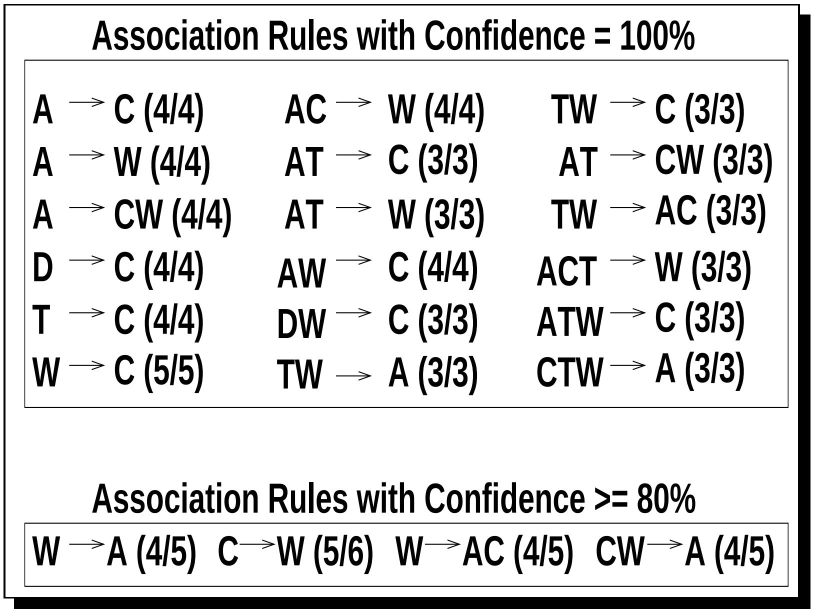
Definition 4.19 *A element $x \in L$ of the lattice L, is called* **join-irreducible** *(dually* **meet-irreducible***) if it has exactly one lower (dually upper) neighbor.*

Let $\mathcal{J}(L)$ and $\mathcal{M}(L)$ denote the set of all join- and meet-irreducible elements, respectively. Any finite lattice $(L, \leq)$ is uniquely determined (up to isomorphism) by $\mathcal{J}(L)$ and $\mathcal{M}(L)$, and restricting the order relation to the set $\mathcal{J}(L) \cup \mathcal{M}(L)$. For the lattice in Figure 4.3, $\mathcal{J}(L) = \{2, 4, 6, 13\}$, and $\mathcal{M}(L) = \{A, D, T, W\}$. This observation can thus help in reducing the size of the original database. The following theorem gives upper and lower bounds on the size of a base:

Theorem 4.9 ([Luxenburger, 1991]) *If* $\mathcal{S}' \subseteq \mathcal{S}^{<1}(\mathcal{K} = \mathcal{B}(G, M, I))$ *is a base, then*

$$1/2 \cdot |\mathcal{J}(\mathcal{K}) \cap \mathcal{M}(\mathcal{K})| \quad \leq \quad |\mathcal{S}'| \quad \leq \quad |\mathcal{K}| - 1$$

The bad news is that the upper bound is tight for a large number of lattices, and thus for such concept lattices the construction of a base will not lead to a reduction in storage over a generating set. Furthermore, the lower limit is not very interesting since there exist lattices with $\mathcal{J}(\mathcal{K}) \cap \mathcal{M}(\mathcal{K}) = \emptyset$.



Association Rules with Confidence = 100%

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

Association Rules with Confidence >= 80%

W → A (4/5) C → W (5/6) W → AC (4/5) CW → A (4/5)

Figure 4.8: Association Rules with 50% Min. Support and 80% Min. Confidence

The problem of finding a canonical base for all partial implications is thus open. Nevertheless, the generating set obtained by the application of Theorem 4.8 should be a good substitute for a base in practice. For example, by combining the base for rules with $p = 1$, shown in Figure 4.5 and the generating set for rules with $p \geq 0.8$, shown in Figure 4.7b, we obtain a generating set for all association rules with $min\_sup = 50\%$, and $min\_conf = 80\%$: $\{A \overset{1}{\Rightarrow} CW,\ D \overset{1}{\Rightarrow} C,\ T \overset{1}{\Rightarrow} C,\ W \overset{1}{\Rightarrow} C,\ CTW \overset{1}{\Rightarrow} A,\ C \overset{5/6}{\Rightarrow} W,\ W \overset{4/5}{\Rightarrow} A\}$. It can be easily verified that all the association rules shown in Figure 4.8, for our example database from Figure 2.1, can be derived from this set. If we consider all possible association rules with $min\_sup = 50\%$ (and consequently $min\_conf = 50\%$), then there are 72 possible association rules versus only 11 in the generating set obtained by combining the base from Figure 4.5 and the generating set from Figure 4.7a.

4.6 Conclusions

In this chapter we presented a lattice-theoretic foundation for the task of mining associations based on formal concept analysis. We showed that the set of frequent concepts uniquely determines all the frequent itemsets. The lattice of frequent concepts can also be used to obtain a rule generating set from which all associations can be derived. We showed that while there exists a characterization of a base for rules with 100% confidence, the problem of constructing a base for all associations is still open.

5 Mining Sequence Rules

5.1 Introduction

The sequence mining task is to discover a set of attributes, shared across time 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 authors or books. Let's say that the database records the books bought by each customer over a period of time. The discovered patterns are the sequences of books most frequently bought by the customers. An example could be that "70% of the people who buy Jane Austen's *Pride and Prejudice* also buy *Emma* within a month." Stores can use these patterns for promotions, shelf placement, etc. Consider another example of a web access database at a popular site, where an object is a web user and an attribute is a web page. The discovered patterns are the sequences of most frequently accessed pages at that site. This kind of information can be used to restructure the web-site, or to dynamically insert relevant links in web pages based on user access patterns. Other domains where sequence mining has been applied include identifying plan failures [Zaki *et al.*, 1998b], finding network alarm patterns [Hatonen *et al.*, 1996], and so on.

The task of discovering all frequent sequences in large databases is quite challenging. The search space is extremely large. For example, with m attributes there are $O(m^k)$ potentially frequent sequences of length k. With millions of objects in the database the problem of I/O minimization becomes paramount. However, most current algorithms are iterative in nature, requiring as many full database scans as the longest frequent sequence, 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 effect performance. Furthermore, most approaches use very complicated internal data structures which have poor locality [Parthasarathy *et al.*, 1998], and add additional space and computation overheads. Our goal is to overcome all of these limitations.

In this chapter we present a new algorithm, called SPADE (**S**equential **PA**ttern **D**iscovery using **E**quivalence classes), for discovering the set of all frequent sequences. The key features of our approach are as follows:

1. We use a *vertical id-list* database format, where we associate with each sequence a list of objects in which it occurs, along with the time-stamps. We show that all frequent sequences can be enumerated via simple id-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. Our approach usually requires three database scans, or only a single scan with some pre-processed information, thus minimizing the I/O costs.

3. We decouple the problem decomposition from the pattern search. We propose two different search strategies for enumerating the frequent sequences within each sub-lattice: breadth-first and depth-first search.

SPADE not only minimizes I/O costs by reducing database scans, but also minimizes computational costs by using efficient search schemes. The vertical id-list based approach is also insensitive to data-skew. An extensive set of experiments shows that SPADE outperforms previous approaches by a factor of two, and by an order of magnitude if we have some additional off-line information. Furthermore, SPADE scales linearly in the database size, and a number of other database parameters.

The rest of the chapter is organized as follows: In Section 5.2 we describe the sequence discovery problem and look at related work in Section 5.3. In Section 5.4 we develop our lattice-based approach for problem decomposition, and for pattern search. Section 5.5 describes our new algorithm. Section 5.6 describes how it can be parallelized efficiently. We briefly discuss a popular previous algorithm in Section 5.7, which is used as a base case for comparison against SPADE. An experimental study is presented in Section 5.8. Finally, we conclude in Section 5.9.

5.2 Problem Statement

The problem of mining sequential patterns can be stated as follows: Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be a set of m distinct attributes, also called *items*. An *itemset* is a non-empty unordered collection of items (without loss of generality, we assume that items of an itemset are sorted in lexicographic order). A *sequence* is an ordered list of itemsets. An itemset i is denoted as $(i_1 i_2 \cdots i_k)$, where i_j is an item. An itemset with k items is called a *k-itemset*. A sequence α is denoted as $(\alpha_1 \mapsto \alpha_2 \mapsto \cdots \mapsto \alpha_q)$, where the sequence *element* α_j is an itemset. A sequence with k items ($k = \sum_j |\alpha_j|$) is called a *k-sequence*. For example, $(B \mapsto AC)$ is a 3-sequence. An item can occur only once in an itemset, but it can occur multiple times in different itemsets of a sequence.

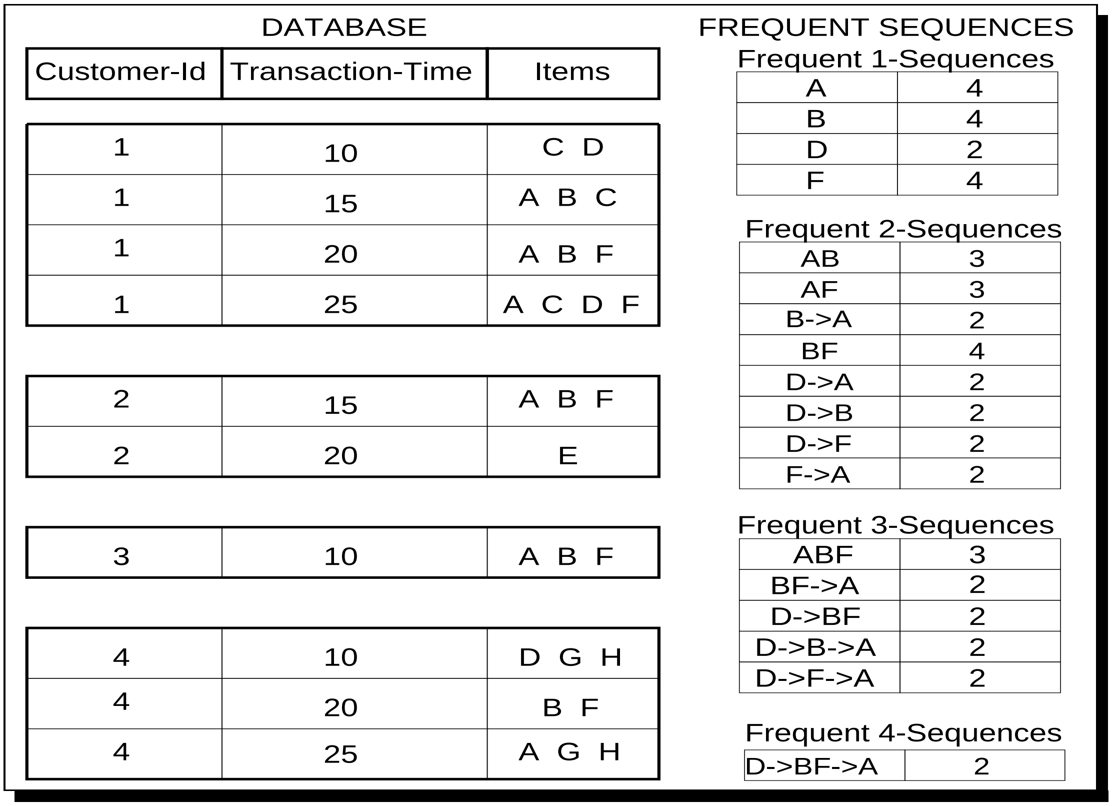
A sequence $\alpha = (\alpha_1 \mapsto \alpha_2 \mapsto \cdots \mapsto \alpha_n)$ is a *subsequence* of another sequence $\beta = (\beta_1 \mapsto \beta_2 \mapsto \cdots \mapsto \beta_m)$, denoted as $\alpha \preceq \beta$, if there exist integers $i_1 < i_2 < \cdots < i_n$ such that $a_j \subseteq b_{i_j}$ for all a_j. For example the sequence $(B \mapsto AC)$ is a subsequence of $(AB \mapsto E \mapsto ACD)$, since the sequence elements $B \subseteq AB$, and $AC \subseteq ACD$. On the other hand the sequence $(AB \mapsto E)$ is not a subsequence of (ABE), and vice versa. We say that α is a proper subsequence of β, denoted $\alpha \prec \beta$, if $\alpha \preceq \beta$ and $\beta \not\preceq \alpha$. A sequence is *maximal* if it is not a subsequence of any other sequence. A subsequence of length k is called a k-subsequence.

A *transaction* $\mathcal{T}$ has a unique identifier and *contains* a set of items, i.e., $\mathcal{T} \subseteq \mathcal{I}$. A *customer* $\mathcal{C}$ has a unique identifier and has associated with it a list of transactions

$\{\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_n\}$. We assume that no customer has more than one transaction with the same time-stamp, so that we can use the transaction-time as the transaction identifier. We also assume that a customer's transaction list is sorted by the transaction-time, forming a sequence $\mathcal{T}_1 \mapsto \mathcal{T}_2 \mapsto \cdots \mapsto \mathcal{T}_n$ called the *customer-sequence*. The database $\mathcal{D}$ consists of a number of such customer-sequences.

A customer-sequence $\mathcal{C}$ is said to *contain* a sequence α, if $\alpha \preceq \mathcal{C}$, i.e., if α is a subsequence of the customer-sequence $\mathcal{C}$. The *support* or *frequency* of a sequence, denoted $\sigma(\alpha)$, is the the total number of customers that contain this sequence. Given a user-specified threshold called the *minimum support* (denoted *min_sup*), we say that a sequence is *frequent* if occurs more than *min_sup* times. The set of frequent k-sequences is denoted as $\mathcal{F}_k$.

Given a database $\mathcal{D}$ of customer sequences and *min_sup*, the problem of mining sequential patterns is to find all frequent sequences in the database. For example, consider the customer database shown in Figure 5.1 (used as a running example throughout this chapter). The database has eight items (A to H), four customers, and ten transactions in all. The figure also shows all the frequent sequences with a minimum support of 50% or 2 customers. In this example we have a unique maximal frequent sequence $D \mapsto BF \mapsto A$.



DATABASE

Customer-Id	Transaction-Time	Items
1	10	C D
1	15	A B C
1	20	A B F
1	25	A C D F
2	15	A B F
2	20	E
3	10	A B F
4	10	D G H
4	20	B F
4	25	A G H

FREQUENT SEQUENCES

Frequent 1-Sequences

A	4
B	4
D	2
F	4

Frequent 2-Sequences

AB	3
AF	3
B->A	2
BF	4
D->A	2
D->B	2
D->F	2
F->A	2

Frequent 3-Sequences

ABF	3
BF->A	2
D->BF	2
D->B->A	2
D->F->A	2

Frequent 4-Sequences

D->BF->A	2

Figure 5.1: Original Customer-Sequence Database

5.2.1 Sequence Rules

```
**RuleGen**($\mathcal{F}$, *min_conf*):
  **for** all frequent sequences $\beta \in \mathcal{F}$ **do**
    **for** all subsequences $\alpha \prec \beta$ **do**
      $conf = fr(\beta)/fr(\alpha)$;
      **if** ($conf \geq min\_conf$) **then**
        output the rule $\alpha \Rightarrow \beta$, and *conf*
```

Figure 5.2: Rule Generation Algorithm

Once the frequent sequences are known, they can be used to obtain rules that describe the relationship between different sequence items. For example, the sequence (BF) occurs in four customers, while (ABF) in three customers. We can therefore say that if BF occurs together, then there is a 75% chance that A also occurs. In other words we say that the rule $(BF) \Rightarrow (BFA)$ has a 75% *confidence.* Another example of a rule is that $(D \mapsto BF) \Rightarrow (D \mapsto BF \mapsto A)$. It has 100% confidence. Given a user-specified minimum confidence (*min_conf*), we can generate all rules that meet the condition by means of the simple algorithm shown in figure 5.2. Since the rule generation step is quite straightforward, in the rest of the chapter we will only concentrate on the frequent sequence discovery phase.

5.3 Related Work

The problem of mining sequential patterns was introduced in [Agrawal and Srikant, 1995]. They also presented three algorithms for solving this problem. The *AprioriAll* algorithm was shown to perform equal to or better than the other two approaches. In subsequent work [Srikant and Agrawal, 1996b], the same authors proposed the GSP algorithm that outperformed *AprioriAll* by up to 20 times. They also introduced maximum gap, minimum gap, and sliding window constraints on the discovered sequences.

The problem of finding *frequent episodes* in a sequence of events was presented in [Mannila *et al.*, 1995]. An episode consists of a set of events and an associated partial order over the events. Our definition of a sequence can be expressed as an episode, however their work is targeted to discover the frequent episodes in a single long event sequence, while we are interested in finding frequent sequences across many different customer-sequences. They further extended their framework in [Mannila and Toivonen, 1996] to discover *generalized episodes*, which allows one to express arbitrary unary conditions on individual episode events, or binary conditions on event pairs. The MEDD and MSDD algorithms [Oates *et al.*, 1997] discover patterns in multiple event sequences. However, they only find sequences of length two with a given window size and a time-gap.

Sequence discovery can essentially be thought of as association discovery [Agrawal *et al.*, 1996] over a temporal database. While association rules discover only intratransaction patterns (itemsets), we now also have to discover inter-transaction patterns (sequences). The set of all frequent sequences is a superset of the set of frequent itemsets. Due to this similarity sequence mining algorithms like *AprioriAll*, GSP, etc., utilize some of the ideas initially proposed for the discovery of association rules [Agrawal *et al.*, 1996; Savasere *et al.*, 1995]. Our new algorithm is based on the fast association mining techniques presented by us in [Zaki *et al.*, 1997c]. Nevertheless, the sequence search space is much more complex and challenging than the itemset space, and thus warrants specific algorithms. Pars of this chapter have appeared in [Zaki, 1998].

5.4 Sequence Enumeration: Lattice-based Approach

Lemma 5.1 *Let $\mathcal{S}$ be the set of all sequences on the items in $\mathcal{I}$. Then the subsequence relation $\preceq$, defines a partial order on $\mathcal{S}$.*

Theorem 5.1 *Given a set $\mathcal{I}$ of items, the ordered set $\mathcal{S}$ of all possible sequences on the items, is a complete lattice in which join and meet are given by union and intersection, respectively:*

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

From our example database (Figure 5.1), we have $\mathcal{F}_1 = \{A, B, D, F\}$. Figure 5.3 shows the sequence lattice $\mathcal{S}$ spanned by the subsequence relation on these four frequent items, which are also the atoms. The bottom element of the sequence lattice is $\perp = \{\}$, but the top element is undefined, since in the abstract the sequence lattice is infinite. Thus only some parts of the lattice are shown. The figure shows the complete set of 2-sequences, the 3-sequences that can be generated from $A \mapsto A$ and AB, and the possible 4-sequences that can be generated from $A \mapsto A \mapsto A$. The recursive combinatorial structure of the subsequence lattice should be apparent. For example consider the set of sequences generated from the item A, and the sequence $A \mapsto A$. The two sets are identical except for the extra $A \mapsto$ prefix in the latter set.

Lemma 5.2 *Let n denote the total number of frequent items. Then the total number of k-sequences is given as:* $\sum_{i_1=1}^{k} \binom{n}{i_1} \sum_{i_2=1}^{k-i_1} \binom{n}{i_2} \cdots \sum_{i_k=1}^{k-i_1-\cdots-i_{k-1}} \binom{n}{i_k}$

Proof: We will count the number of ways in which a k sequence can be constructed, and then assign items for each arrangement. The number of ways a k-sequence can be constructed is given by the number of ways we can obtain k as a sum of integers. For example, Table 5.1 shows the number of ways we can obtain 4 as a sum of integers. The integers in the sum are interpreted to be the sizes of the itemsets comprising a k length sequence. We now assign items to each such itemset. For an itemset of length i, we have $\binom{n}{i}$ item assignments. Multiplying the choices for each case, and adding all the cases we obtain the total number of k-sequences shown above. ■

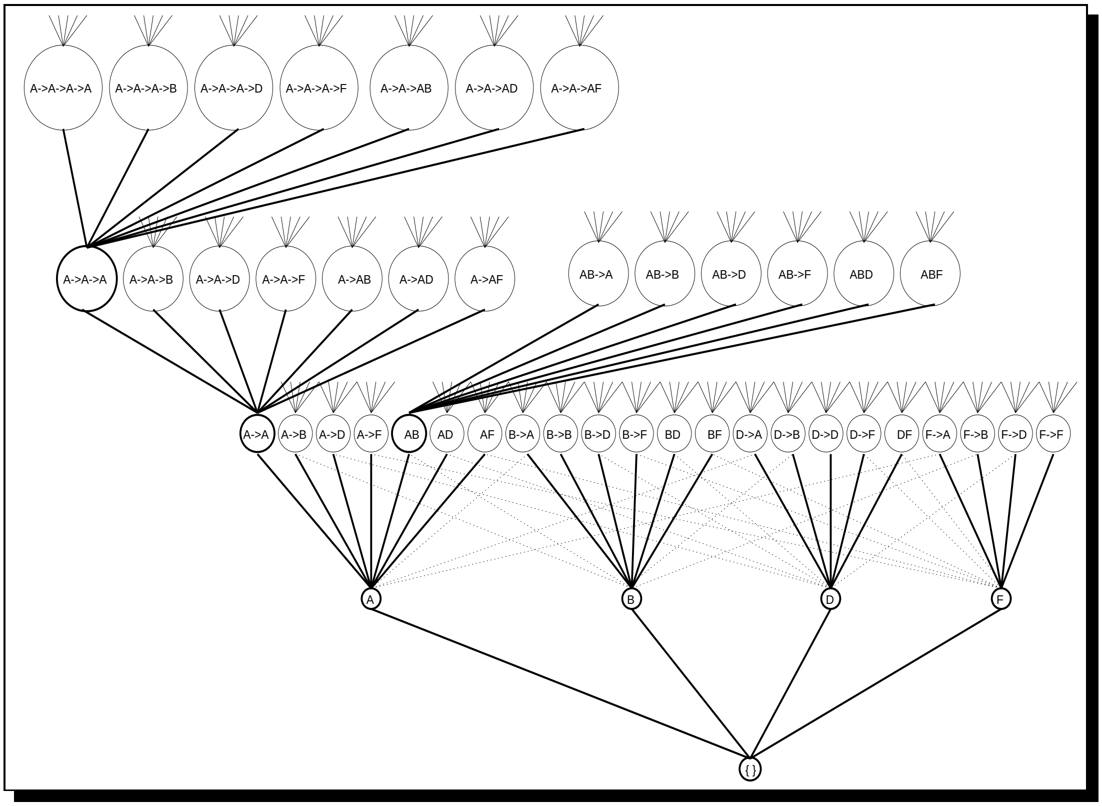


Figure 5.3: Sequence Lattice Spanned by Subsequence Relation

As mentioned above, in the abstract case the lattice of sequences is infinite. Fortunately in all practical cases it is bounded. The number of sequence elements (itemsets) is bounded above by the maximum number of transactions per customer (say C). Since the size of an itemset is bounded above by the maximum transaction size (say T), a sequence can have at most $C \cdot T$ items, and hence the subsequence lattice is bounded above by $C \cdot T$. In our example, $C = 4$ and $T = 4$, so that the maximum possible sequence can have 16 items.

In all practical cases not only is the lattice bounded, but the set of frequent sequences is also very sparse (depending on the *min_sup* value). For example, consider Figure 5.4 which shows the sequence lattice induced by the maximal frequent sequence $D \mapsto BF \mapsto A$ in our example. The set of atoms $\mathcal{A}$ is given by the frequent items $\{A, B, D, F\}$. It is obvious that the set of all frequent sequences forms a meet-semilattice, because it is closed under the meet operation, i.e., if X and Y are frequent sequences, then the meet $X \cap Y$ is also frequent. However, it is not a join-semilattice, since it is not closed under joins, i.e., X and Y being frequent, doesn't imply that $X \cup Y$ is frequent. The closure under meet leads to the well known observation on sequence frequency:

Lemma 5.3 *All subsequences of a frequent sequence are frequent.*

The above lemma leads very naturally to a bottom-up search procedure for enumer-

i_1	i_2	i_3	i_4
1	1	1	1
1	1	2	
1	2	1	
1	3		
2	1	1	
2	2		
3	1		
4			

Table 5.1: Number of Ways to Obtain a 4-Sequence

ating frequent sequences, which has been leveraged in many sequence mining algorithms [Srikant and Agrawal, 1996b; Mannila *et al.*, 1995; Oates *et al.*, 1997]. In essence what the lemma says is that we need to focus only on those sequences whose subsequences are frequent. This leads to a very powerful pruning strategy, where we eliminate all sequences, at least one of whose subsequences is infrequent. However, the lattice formulation makes it apparent that we need not restrict ourselves to a purely bottom-up search. We can employ different search procedures, which we will discuss below.

5.4.1 Support Counting

Let's associate with each atom X in the sequence lattice its *id-list*, denoted $\mathcal{L}(X)$, which is a list of all customer (*cid*) and transaction identifier (*tid*) pairs containing the atom. Figure 5.5 shows the id-lists for the atoms in our example database. For example consider the atom D. In our original database in Figure 5.1, we see that D occurs in the following customer and transaction identifier pairs $\{(1, 10), (1, 25), (4, 10)\}$. This forms the id-list for item D.

Lemma 5.4 *For any $X \in \mathcal{S}$, let $J = \{Y \in \mathcal{A}(\mathcal{S}) \mid Y \leq X\}$. 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 sequence in $\mathcal{S}$ can be obtained as a union or join of some atoms of the lattice, and the support of the sequence can be obtained by intersecting the id-list of the atoms. This lemma is applied only to the atoms of the lattice. We generalize this for a set of sequences in the next lemma.

Lemma 5.5 *For any $X \in \mathcal{S}$, let $X = \cup_{Y \in J} Y$. Then $\sigma(X) =\mid \cap_{Y \in J} \mathcal{L}(Y) \mid$.*

This lemma says that if X is given as a union of a set of sequences in J, then its support is given as the intersection of id-lists of elements in J. In particular we can determine the support of any k-sequence by simply intersecting the id-lists of any two of its $(k-1)$ length subsequences. A simple check on the cardinality of the resulting id-list tells us

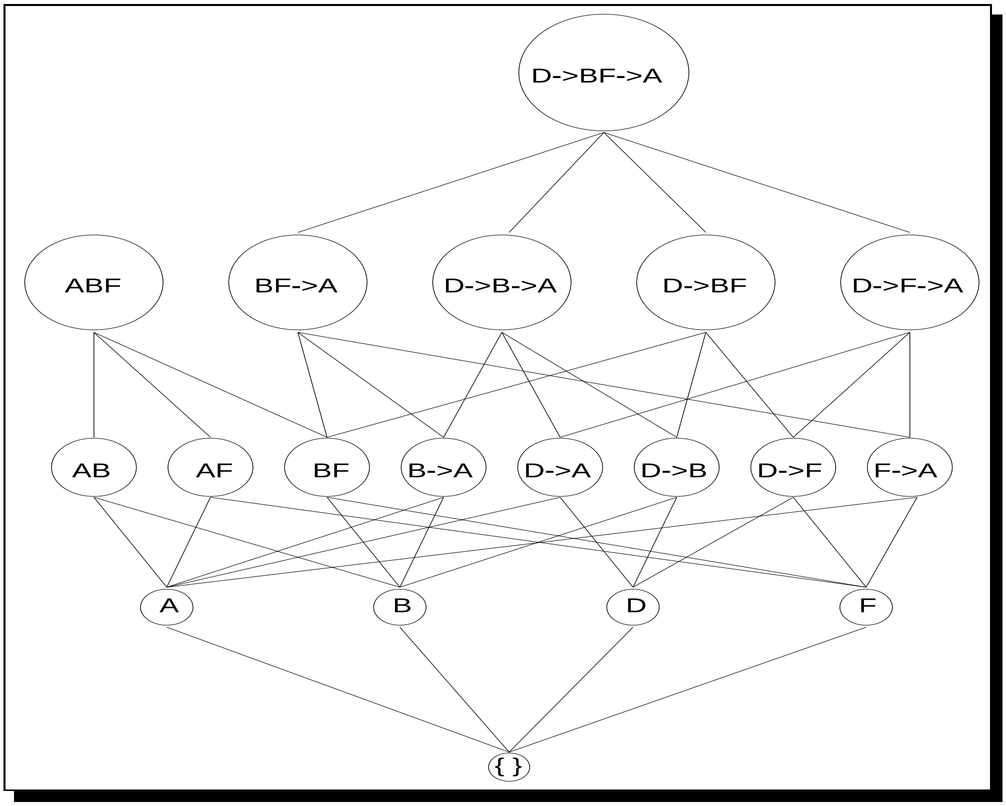


Figure 5.4: Lattice Induced by Maximal Frequent Sequence $D \mapsto BF \mapsto A$

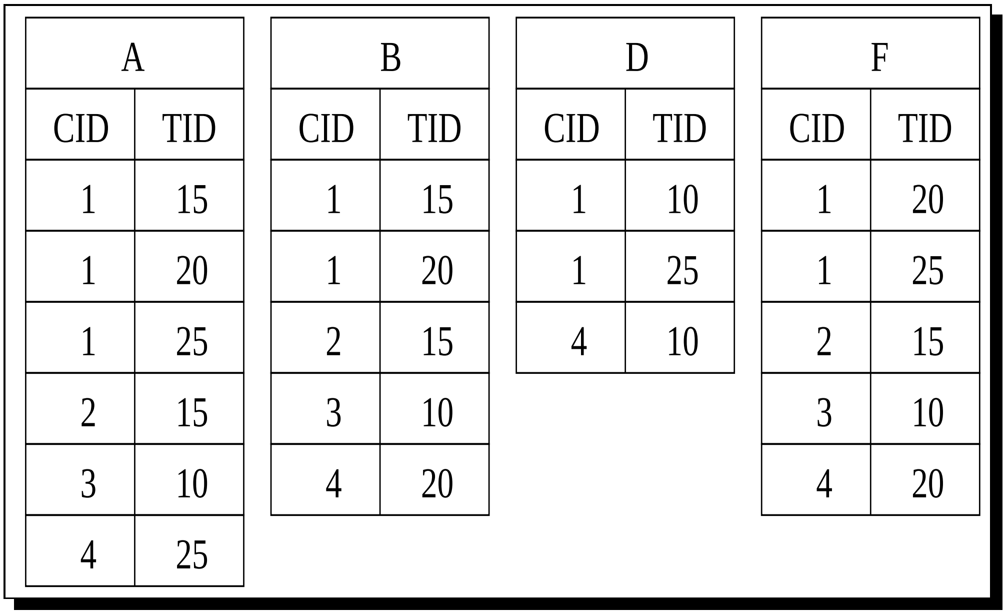
whether the new sequence is frequent or not. Figure 5.6 shows this process pictorially. It shows the initial vertical database with the id-list for each atom. The intermediate id-list for $D \mapsto A$ is obtained by intersecting the lists of atoms D and A, i.e., $\mathcal{L}(D \mapsto A) = \mathcal{L}(D) \cap \mathcal{L}(A)$. Similarly, $\mathcal{L}(D \mapsto BF \mapsto A) = \mathcal{L}(D \mapsto BF) \cap \mathcal{L}(D \mapsto B \mapsto A)$, and so on. Thus, only the lexicographically first two subsequences at the previous level are required to compute the support of a sequence at a given level.

Lemma 5.6 *Let X and Y be two sequences , with $X \preceq Y$. Then $\mathcal{L}(X) \supseteq \mathcal{L}(Y)$.*

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

5.4.2 Lattice Decomposition: Prefix-Based Classes

If we had enough main-memory, we could enumerate all the frequent sequences by traversing the lattice, and performing intersections to obtain sequence supports.



A	
CID	TID
1	15
1	20
1	25
2	15
3	10
4	25

B	
CID	TID
1	15
1	20
2	15
3	10
4	20

D	
CID	TID
1	10
1	25
4	10

F	
CID	TID
1	20
1	25
2	15
3	10
4	20

Figure 5.5: Id-lists for the Atoms

In practice, however, we only have a limited amount of main-memory, and all the intermediate id-lists will not fit in memory. This brings up a natural question: can we decompose the original lattice into smaller pieces such that each piece can be solved independently in main-memory. We address this question below.

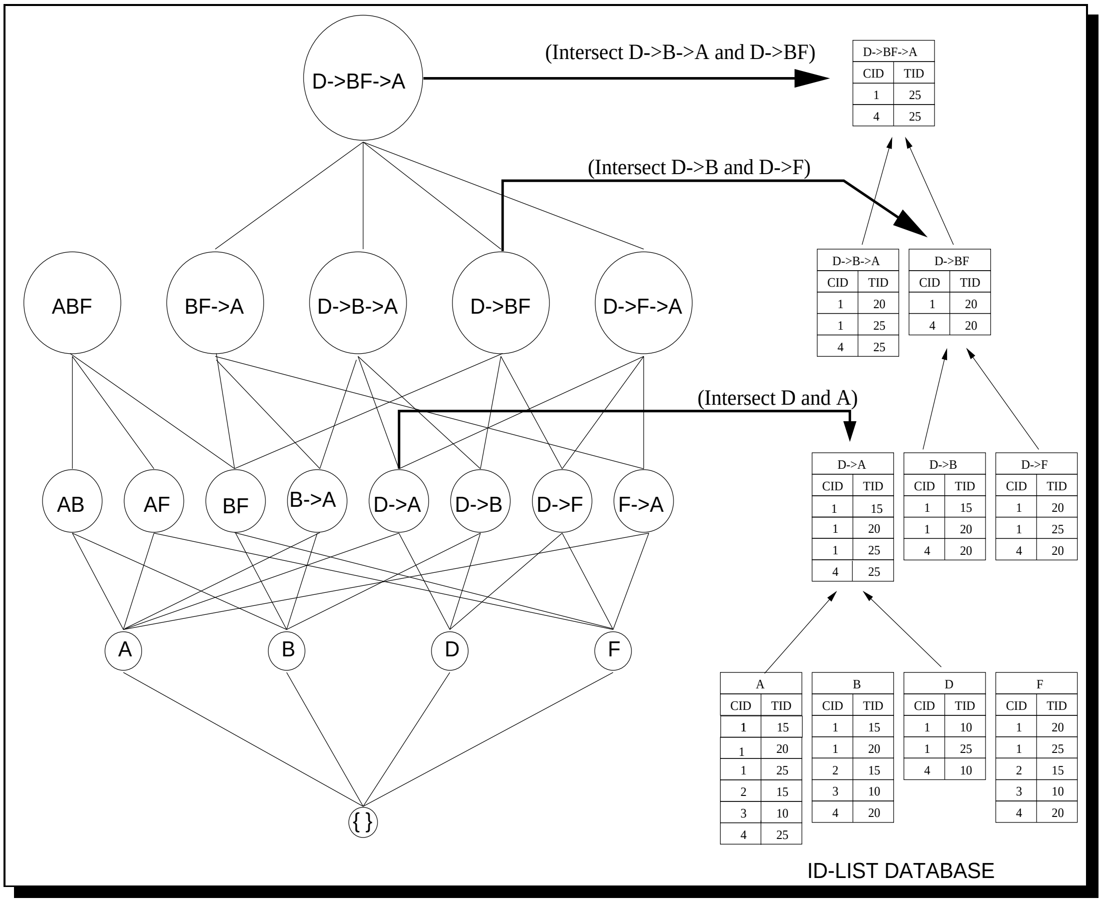
Define a function $p : \mathcal{S} \mapsto \mathcal{S}$ where $p(X, k) = X[1 : k]$. In other words, $p(X, k)$ returns the k length prefix of X. Define an equivalence relation θ_k on the lattice $\mathcal{S}$ as follows: $\forall X, Y \in \mathcal{S}$, we say that X is related to Y under θ_k, denoted as $X \equiv_{\theta_k} Y$ if and only if $p(X, k) = p(Y, k)$. That is, two sequences are in the same class if they share a common k length prefix.

Figure 5.7 shows the lattice induced by the equivalence relation θ_1 on $\mathcal{S}$, where we collapse all sequences with a common item prefix into an equivalence class. The resulting set of equivalence classes is $\{[A], [B], [D], [F]\}$. At the bottom of the figure, it also shows the links among the four classes. These links carry pruning information. In other words if we want to prune a sequence (if it has at least one infrequent subsequence) then we may need some cross-class information. We will have more to say about this later.

Lemma 5.7 *Each equivalence class $[X]_{\theta_k}$ induced by the equivalence relation θ_k is a sub-lattice of $\mathcal{S}$.*

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 sub-lattice of $\mathcal{S}$. ■

Each $[X]_{\theta_1}$ is thus a lattice with its own set of atoms. For example, the atoms of $[D]_{\theta_1}$ are $\{D \mapsto A, D \mapsto B, D \mapsto F\}$, and the bottom element is $\bot = D$. By the



D->BF->A	
CID	TID
1	25
4	25

D->B->A	
CID	TID
1	20
1	25
4	25

D->BF	
CID	TID
1	20
4	20

D->A	
CID	TID
1	15
1	20
1	25
4	25

D->B	
CID	TID
1	15
1	20
4	20

D->F	
CID	TID
1	20
1	25
4	20

A	
CID	TID
1	15
1	20
1	25
2	15
3	10
4	25

B	
CID	TID
1	15
1	20
2	15
3	10
4	20

D	
CID	TID
1	10
1	25
4	10

F	
CID	TID
1	20
1	25
2	15
3	10
4	20

Figure 5.6: Computing Support via Id-list Intersections

application of Lemmas 5.4 and 5.5, we can generate all the supports of the sequences in each class (sub-lattice) by intersecting the id-list of atoms or any two subsequences at the previous level. If there is enough main-memory to hold temporary id-lists for each class, then we can solve each $[X]_{\theta_1}$ independently.

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. Lets assume that $[D]$ is too large to fit in main-memory. Since $[D]$ is itself a lattice, it can be decomposed using the relation θ_2. Figure 5.8 shows the classes induced by applying θ_2 on $[D]$ (after applying θ_1 on $\mathcal{S}$). Each of the resulting six classes, $[A]$, $[B]$, $[D \mapsto A]$, $[D \mapsto B]$, $[D \mapsto F]$, and $[F]$, can be solved independently. Thus 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.

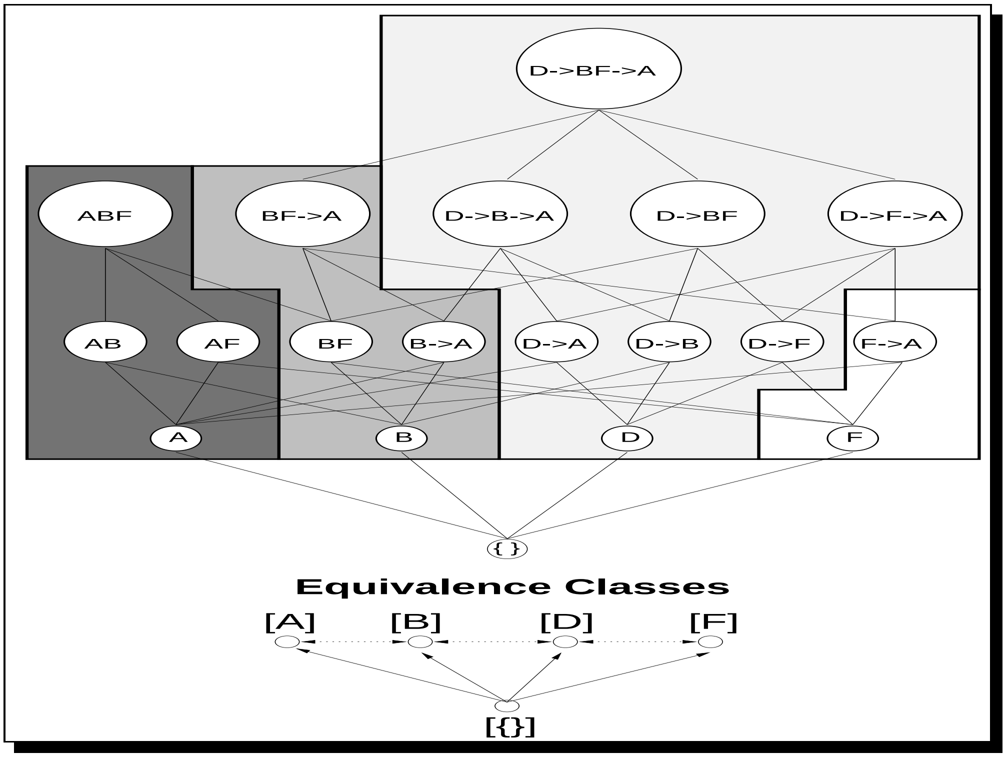


Figure 5.7: Equivalence Classes of $\mathcal{S}$ Induced by θ_1

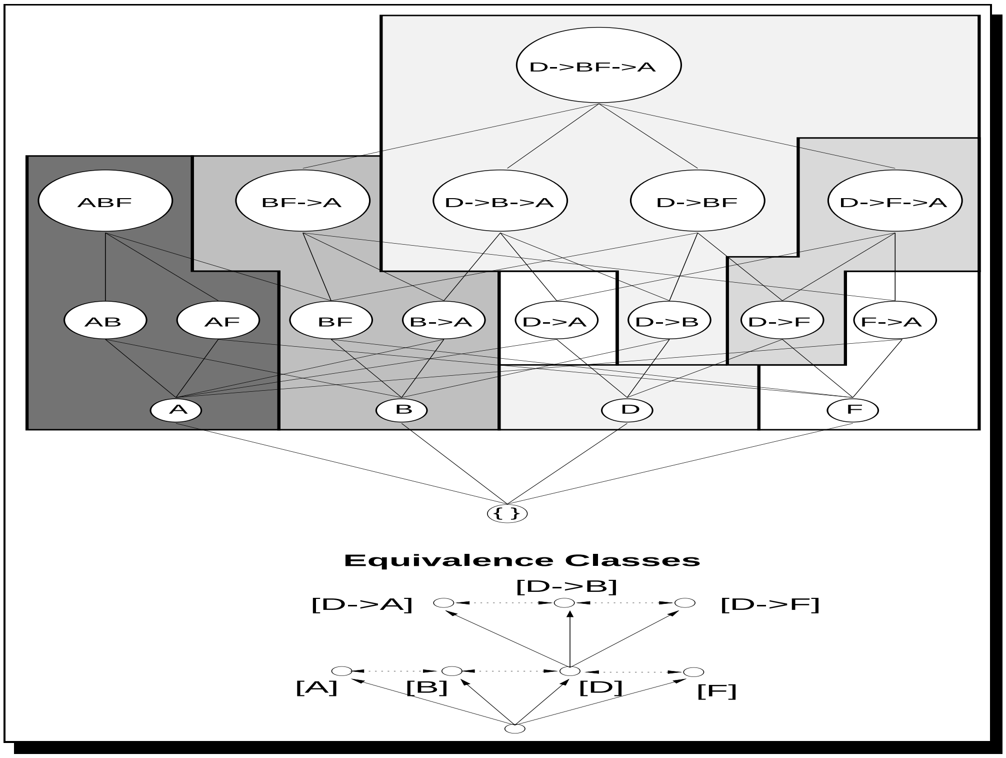


Figure 5.8: Equivalence Classes of $[D]_{\theta_1}$ Induced by θ_2

5.4.3 Search for Frequent Sequences

In this section we discuss efficient search strategies for enumerating the frequent sequences within each class. We will discuss two main strategies: breadth-first and depth-first search. Both these methods are based on a recursive decomposition of each class into smaller classes induced by the equivalence relation θ_k. Figure 5.9 shows the decomposition of $[D]_{\theta_1}$ into smaller and smaller classes, and the resulting lattice of equivalence classes.

Breadth-First Search (BFS) In a breadth-first search the lattice of equivalence classes generated by the recursive application of θ_k is explored in a bottom-up manner. We process all the child classes at a level before moving on to the next level. For example, in Figure 5.9, we process the equivalence classes $\{[D \mapsto A], [D \mapsto B], [D \mapsto F]\}$, before moving on to the classes $\{[D \mapsto B \mapsto A], [D \mapsto BF], [D \mapsto F \mapsto A]\}$, and so on.

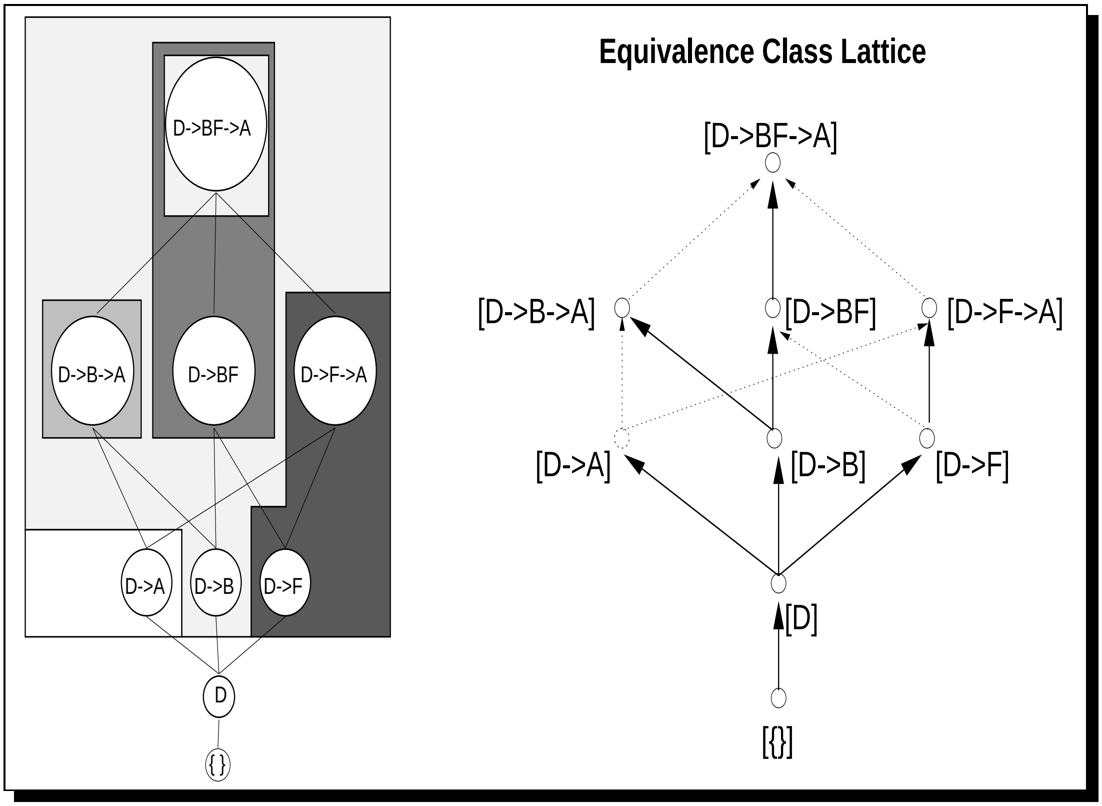


Figure 5.9: Recursive Decomposition of Class $[D]$ into Smaller Sub-Classes via θ_k

Depth-First Search (DFS) In a depth-first search, we completely solve all child equivalence classes along one path before moving on to the next path. For example, we process the classes in the following order $[D \mapsto A]$, $[D \mapsto B]$, $[D \mapsto B \mapsto A]$, $[D \mapsto BF]$, $[D \mapsto BF \mapsto A]$, and so on.

The advantage of BFS over DFS is that we have more information available for pruning. For example, we know the set of 2-sequences before constructing the 3-sequences, while this information is not available in DFS. On the other hand DFS requires less main-memory than BFS. DFS only needs to keep the intermediate id-lists for two consecutive classes along a single path, while BFS must keep track of id-lists for all the classes in two consecutive levels.

Besides BFS and DFS search, there are many other search possibilities. For example, in the DFS scheme, if we determine that $D \mapsto BF \mapsto A$ is frequent, then we do not have to process the classes $[D \mapsto F]$, and $[D \mapsto F \mapsto A]$, since they must necessarily be frequent. We are currently investigating such schemes for efficient enumeration of only the maximal frequent sequences.

5.5 SPADE: Algorithm Design and Implementation

In this section we describe the design and implementation of SPADE. Figure 5.10 shows the high level structure of the algorithm. The main steps include the computation of the frequent 1-sequences and 2-sequences, the decomposition into prefix-based equivalence classes, and the enumeration of all other frequent sequences via BFS or DFS search within each class. We will now describe each step in some more detail.

```
SPADE ($min\_sup, \mathcal{D}$):
    $\mathcal{F}_1 = \{$ frequent items or 1-sequences $\}$;
    $\mathcal{F}_2 = \{$ frequent 2-sequences $\}$;
    $\mathcal{E} = \{$ equivalence classes $[X]_{\theta_1}\}$;
    for all $[X] \in \mathcal{E}$ do Enumerate-Frequent-Seq([X]);
```

Figure 5.10: The SPADE Algorithm

5.5.1 Computing Frequent 1-Sequences and 2-Sequences

Most of the current sequence mining algorithms [Agrawal and Srikant, 1995; Srikant and Agrawal, 1996b] assume a *horizontal* database layout such as the one shown in Figure 5.1. In the horizontal format the database consists of a set of customers. Each customer has a set of transactions, along with the items contained in the transaction. In contrast our algorithm uses a *vertical* database format, where we maintain a disk-based id-list for each item. Each entry of the id-list is a (cid, tid) pair where the item occurs. This enables us to check support via simple id-list intersections.

Computing $\mathcal{F}_1$: Given the vertical id-list database, all frequent 1-sequences can be computed in a single database scan. For each database item, we read its id-list from the disk into memory. We then scan the id-list, incrementing the support for each new cid encountered.

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-sequences 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 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, with very little overhead. For each item i, we scan its id-list into memory. For each customer and transaction id pair, say (c, t) in $\mathcal{L}(i)$, we insert (i, t) in the list for customer c. For example, consider the id-list for item A, shown in Figure 5.5. We scan the first pair $(1, 15)$, and then insert $(A, 15)$ in the list for customer 1. Figure 5.11 shows the complete horizontal database recovered from the vertical item id-lists. Computing $\mathcal{F}_2$ from the recovered horizontal database is straight-forward. We form a list of all 2-sequences in each customer sequence, and update counts in a 2-dimensional array indexed by the frequent items.

cid	$(item, tid)$ pairs
1	(A 15) (A 20) (A 25) (B 15) (B 20) (C 10) (C 15) (C 25) (D 10) (D 25) (F 20) (F 25)
2	(A 15) (B 15) (E 20) (F 15)
3	(A 10) (B 10) (F 10)
4	(A 25) (B 20) (D 10) (F 20) (G 10) (G 25) (H 10) (H 25)

Figure 5.11: Vertical-to-Horizontal Database Recovery

5.5.2 Enumerating Frequent Sequences of a Class

Figure 5.12 shows the pseudo-code for the breadth-first and depth-first search. The input to the procedure is a set of atoms of a sub-lattice S, along with their id-lists. Frequent sequences are generated by intersecting the id-lists of all distinct pairs of atoms and checking the cardinality of the resulting id-list against *min_sup*. Before intersecting the id-lists a pruning step is inserted to ensure that all subsequences of the resulting sequence are frequent. If this is true, then we go ahead with the id-list intersection, otherwise we can avoid the intersection. The sequences found to be frequent at the current level form the atoms of classes for the next level. This recursive process is repeated until all frequent sequences have been enumerated. In terms of memory management it is easy to see that we need memory to store intermediate id-lists for at most two consecutive levels. The depth-first search requires memory for two classes

```
Enumerate-Frequent-Seq($S$):
    for all atoms $A_i \in S$ do
        $T_i = \emptyset$;
        for all atoms $A_j \in S$, with $j > i$ do
            $R = A_i \cup A_j$;
            if ($Prune$(R) == FALSE) then
                $\mathcal{L}(R) = \mathcal{L}(A_i) \cap \mathcal{L}(A_j)$;
                if $\sigma(R) \geq min\_sup$ then
                    $T_i = T_i \cup \{R\}$; $\mathcal{F}_{|R|} = \mathcal{F}_{|R|} \cup \{R\}$;
        end
        if (Depth-First-Search) then Enumerate-Frequent-Seq($T_i$);
    end
    if (Breadth-First-Search) then
        for all $T_i \neq \emptyset$ do Enumerate-Frequent-Seq($T_i$);
```

Figure 5.12: Pseudo-code for Breadth-First and Depth-First Search

on the two levels. The breadth-first search requires memory of all the classes on the two levels. Once all the frequent sequences for the next level have been generated, the sequences at the current level can be deleted.

Disk Scans Before processing each of equivalence classes from the initial decomposition, all the relevant item id-lists for that class are scanned from disk into memory. The id-lists for the atoms of each initial class are constructed by intersecting the item id-lists. All the other frequent sequences are enumerated as described above. If all the initial classes have disjoint set of items, then each item's id-list is scanned from disk only once during the entire frequent sequence 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 single database scan after computing $\mathcal{F}_2$, in contrast to the current approaches which require multiple scans.

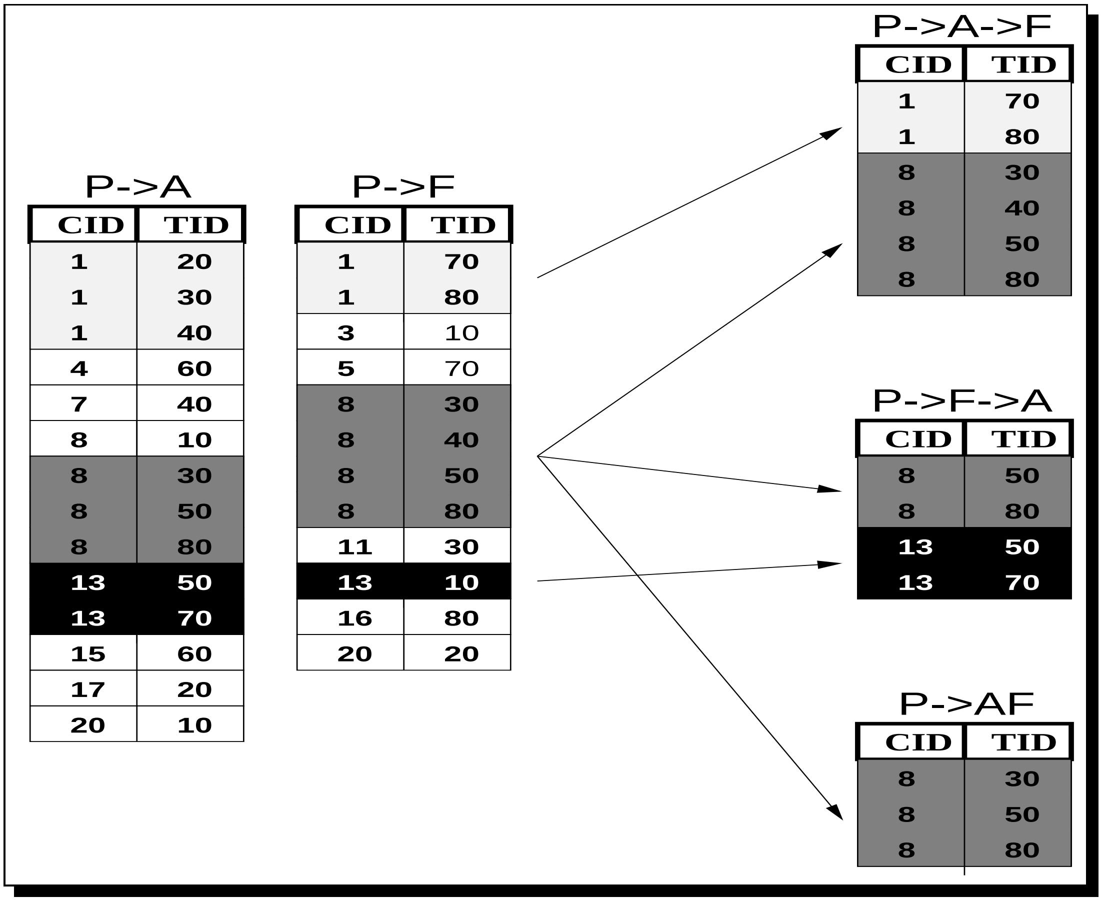
5.5.3 Id-List Intersection

We now describe how we perform the id-list intersections for two sequences. Consider an equivalence class $[B \mapsto A]$ with the atom set $\{B \mapsto AB, B \mapsto AD, B \mapsto A \mapsto A, B \mapsto A \mapsto D, B \mapsto A \mapsto F\}$. If we let P stand for the prefix $B \mapsto A$, then we can rewrite the class to get $[P] = \{PB, PD, P \mapsto A, P \mapsto D, P \mapsto F\}$. One can observe the class has two kinds of atoms: the itemset atoms $\{PB, PD\}$, and the sequence atoms $\{P \mapsto A, P \mapsto D, P \mapsto F\}$. We assume without loss of generality that the itemset atoms of a class always precede the sequence atoms. To extend the class it is sufficient to intersect the id-lists of all pairs of atoms. However, depending on the atom pairs being intersected, there can be upto three possible resulting frequent sequences:

1. **Itemset Atom vs Itemset Atom**: If we are intersecting PB with PD, then the only possible outcome is new itemset atom PBD.

2. **Itemset Atom vs Sequence Atom**: If we are intersecting PB with $P \mapsto A$, then the only possible outcome is new sequence atom $PB \mapsto A$.

3. **Sequence Atom vs Sequence Atom**: If we are intersecting $P \mapsto A$ with $P \mapsto F$, then there are three possible outcomes: a new itemset atom $P \mapsto AF$, and two new sequence atoms $P \mapsto A \mapsto F$ and $P \mapsto F \mapsto A$. A special case arises when we intersect $P \mapsto A$ with itself, which can only produce the new sequence atom $P \mapsto A \mapsto A$.



P->A

CID	TID
1	20
1	30
1	40
4	60
7	40
8	10
8	30
8	50
8	80
13	50
13	70
15	60
17	20
20	10

P->F

CID	TID
1	70
1	80
3	10
5	70
8	30
8	40
8	50
8	80
11	30
13	10
16	80
20	20

P->A->F

CID	TID
1	70
1	80
8	30
8	40
8	50
8	80

P->F->A

CID	TID
8	50
8	80
13	50
13	70

P->AF

CID	TID
8	30
8	50
8	80

Figure 5.13: Id-list Intersection

We now describe how the actual id-list intersection is performed. Consider Figure 5.13, which shows the hypothetical id-lists for the sequence atoms $P \mapsto A$ and $P \mapsto F$. To compute the new id-list for the resulting itemset atom $P \mapsto AF$, we simply need to check for *equality* of (cid,tid) pairs. In our example, the only matching pairs are $\{(8,30),(8,50),(8,80)\}$. This forms the id-list for $P \mapsto AF$. To compute the id-list for the new sequence atom $P \mapsto A \mapsto F$, we need to check for a *follows* relationship, i.e., for a given pair (c,t_1) in $\mathcal{L}(P \mapsto A)$, we check whether there exists a pair (c,t_2) in $\mathcal{L}(P \mapsto F)$ with the same cid c, but with $t_2 > t_1$. If this is true, it means that the item F follows the item A for customer c. In other words, the customer c contains the pattern $P \mapsto A \mapsto F$, and the pair (c,t_2) is added to the pattern's id-list. Finally, the id-list for $P \mapsto F \mapsto A$ can be obtained in a similar manner by reversing the roles of $P \mapsto A$ and $P \mapsto F$. The final id-lists for the three new sequences are shown in Figure 5.13. Since we only intersect sequences within a class, which have the same prefix (whose items have the same tid or time-stamp), we only need to keep track of the last item's tid for determining the equality and follows relationships. As a further optimization, we generate the id-lists of all the three possible new sequences in just one intersection.

5.5.4 Pruning Sequences

```
Prune ($\beta$):
  for all $(k-1)$-subsequences, $\alpha \prec \beta$ do
    if ([$\alpha_1$] has been processed, and $\alpha \notin \mathcal{F}_{k-1}$) then
        return TRUE;
  return FALSE;
```

Figure 5.14: Sequence Pruning

The pruning algorithm is shown in Figure 5.14. Let α_1 denote the first item of sequence α. Before generating the id-list for a new k-sequence β, we check whether all its k subsequences of length $k-1$ are frequent. If they all are frequent then we perform the id-list intersection. Otherwise, β is dropped from consideration. Note that all subsequences except the last are within the current class. For example consider a sequence $\beta = (D \mapsto BF \mapsto A)$. The first three subsequences, $(D \mapsto BF)$, $(D \mapsto B \mapsto A)$, and $(D \mapsto F \mapsto A)$ are all lie in the class $[D]$. However, the last subsequence $(BF \mapsto A)$ belongs to the class $[B]$. If $[B]$ has already been processed then we have complete subsequence information for pruning. Otherwise, if $[B]$ has not been processed, then we cannot determine whether $(BF \mapsto A)$ is frequent or not. Nevertheless, partial pruning based on the members of the same class is still possible. It is generally better to process the class in lexicographically descending order, since in this case at least for itemsets all information is available for pruning. This is because items of an itemset are kept sorted in increasing order. For example, if we wanted to test $\beta = ABDF$, then we would first check within its class $[A]$ if ADF is frequent, and since $[B]$ will have been processed if we solve the classes in reverse lexicographic order, we can also check if BDF is frequent.

5.6 Parallel Sequence Mining

We will now briefly discuss how the SPADE algorithm can be efficiently parallelized. Since SPADE retains most of the feature of *Par-Eclat* the parallelization can be essentially the same as that described in Section 3.4:

1. We initially partition the id-lists equally among the different processors.
2. We compute the set of frequent 2-sequences, and then generate independent classes by using the prefix-based decomposition. These classes are then scheduled among the processors using the greedy heuristic described in Section 3.4.
3. We exchange the id-lists among the processors so that the id-lists of all atoms of classes scheduled on a given processor are local to that processor.
4. Finally, we enter the asynchronous phase. Each processor can generate all the frequent sequences independently since each class is self-contained.

5.7 The GSP Algorithm

Below we describe the GSP algorithm [Srikant and Agrawal, 1996b] in some more detail, since we use it as a base against which we compare SPADE, and it is one of the best previous algorithms.

```
$\mathcal{F}_1$ = { frequent 1-sequences };
for ($k = 2; \mathcal{F}_{k-1} \neq \emptyset; k = k + 1$) do
   $C_k$ = Set of candidate $k$-sequences;
   for all customer-sequences $\mathcal{E}$ in the database do
      Increment count of all $\alpha \in C_k$ contained in $\mathcal{E}$
   $\mathcal{F}_k = \{\alpha \in C_k | \alpha.sup \geq min\_sup\}$;
Set of all frequent sequences = $\bigcup_k \mathcal{F}_k$;
```

Figure 5.15: The GSP Algorithm

GSP makes multiple passes over the database. In the first pass, all single items (1-sequences) are counted. From the frequent items a set of *candidate* 2-sequences are formed. Another pass is made to gather their support. The frequent 2-sequences are used to generate the candidate 3-sequences, and this process is repeated until no more frequent sequences are found. There are two main steps in the algorithm.

1. **Candidate Generation**: Given the set of frequent $(k-1)$-sequences $\mathcal{F}_{k-1}$, the candidates for the next pass are generated by joining $\mathcal{F}_{k-1}$ with itself. A pruning phase eliminates any sequence at least one of whose subsequences is not frequent. For fast counting, the candidate sequences are stored in a *hash-tree.*

2. **Support Counting**: To find all candidates contained in a customer-sequence $\mathcal{E}$, all k-subsequences of $\mathcal{E}$ are generated. For each such subsequence a search is made in the hash-tree. If a candidate in the hash-tree matches the subsequence, its count is incremented.

The GSP algorithm is shown in Figure 5.15. For more details on the specific mechanisms for constructing and searching hash-trees, please refer to [Srikant and Agrawal, 1996b].

5.8 Experimental Results

In this section we compare the performance of SPADE with the GSP algorithm. The GSP algorithm was implemented as described in [Srikant and Agrawal, 1996b]. For SPADE results are shown only for the BFS search. Experiments were performed on a 100MHz MIPS processor with 256MB main memory running IRIX 6.2. The data was stored on a non-local 2GB disk.

Dataset	C	T	S	I	D	Size (MB)
C10-T2.5-S4-I1.25-D100K	10	2.5	4	1.25	100,000	18.4
C10-T2.5-S4-I1.25-D200K	10	2.5	4	1.25	200,000	36.8
C10-T2.5-S4-I1.25-D500K	10	2.5	4	1.25	500,000	92.0
C10-T2.5-S4-I1.25-D1000K	10	2.5	4	1.25	1,000,000	184.0
C10-T5-S4-I1.25-D200K	10	5	4	1.25	200,000	56.5
C10-T5-S4-I2.5-D200K	10	5	4	2.5	200,000	54.3
C20-T2.5-S4-I1.25-D200K	20	2.5	4	1.25	200,000	76.7
C20-T2.5-S4-I2.5-D200K	20	2.5	4	2.5	200,000	66.5
C20-T2.5-S8-I1.25-D200K	20	2.5	8	1.25	200,000	76.4

Table 5.2: Synthetic Datasets

Synthetic Datasets The synthetic datasets are the same as those used in [Srikant and Agrawal, 1996b], albeit with twice as many customers. We used the publicly available dataset generation code from the IBM Quest data mining project [IBMa]. These datasets mimic real-world transactions, where people buy a sequence of sets of items. Some customers may buy only some items from the sequences, or they may buy items from multiple sequences. The customer sequence size and transaction size are clustered around a mean and a few of them may have many elements. The datasets are generated using the following process. First N_I maximal itemsets of average size I are generated by choosing from N items. Then N_S maximal sequences of average size S are created by assigning itemsets from N_I to each sequence. Next a customer of average C transactions is created, and sequences in N_S are assigned to different customer elements, respecting the average transaction size of T. The generation stops when D customers have been generated. Like [Srikant and Agrawal, 1996b] we set $N_S = 5000$, $N_I = 25000$

and $N = 10000$. The number of data-sequences was set to $D = 200,000$. Table 5.2 shows the datasets with their parameter settings. We refer the reader to [Agrawal and Srikant, 1995] for additional details on the dataset generation.

Plan Dataset The real-life dataset was obtained from a Natural Language Planning domain. The planner generates plans for routing commodities from one city to another. A "customer" corresponds to a plan identifier, while a "transaction" corresponds to an event in a plan. An event consists of an event identifier, an outcome (such as "success", "late", or "failure"), an action name (such as "move", or "load"), and a set of additional parameters specifying things such as origin, destination, vehicle type ("truck", or "helicopter"), weather conditions, and so on. The data mining goal is to identify the causes of plan failures. There are 77 items, 202071 plans (customers), and 829236 events (transactions). The average plan length is 4.1, and the average event length is 7.6.

5.8.1 Comparison of SPADE with GSP

Figure 5.16 compares our SPADE algorithm with GSP, on different synthetic datasets. Each graph shows the results as the minimum support is changed from 1% to 0.25%. Two sets of experiments are reported for each value of support. The bar labeled SPADE corresponds to the case where we computed $\mathcal{F}_2$ via the vertical-to-horizontal transformation method described in Section 5.5.1. The times for GSP and SPADE include the cost of computing $\mathcal{F}_2$. The bars labeled SPADE-F2 and GSP-F2 correspond to the case where $\mathcal{F}_2$ was computed in a pre-processing step, and the times shown don't include the pre-processing cost.

The figures clearly indicate that the performance gap between the two algorithms increases with decreasing minimum support. SPADE is about twice as fast as GSP at lower values of support. In addition we see that SPADE-F2 outperforms GSP-F2 by an order of magnitude in most cases. There are several reasons why SPADE outperforms GSP:

1. SPADE uses 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 [Parthasarathy *et al.*, 1998]. On the other hand SPADE has 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. GSP makes a complete dataset scan for each iteration. SPADE on the other hand restricts itself to usually only three scans. It thus cuts down the I/O costs.

Another conclusion that can be drawn from the SPADE-F2 and GSP-F2 comparison is that nearly all the benefit of SPADE comes from the improvement in the running time

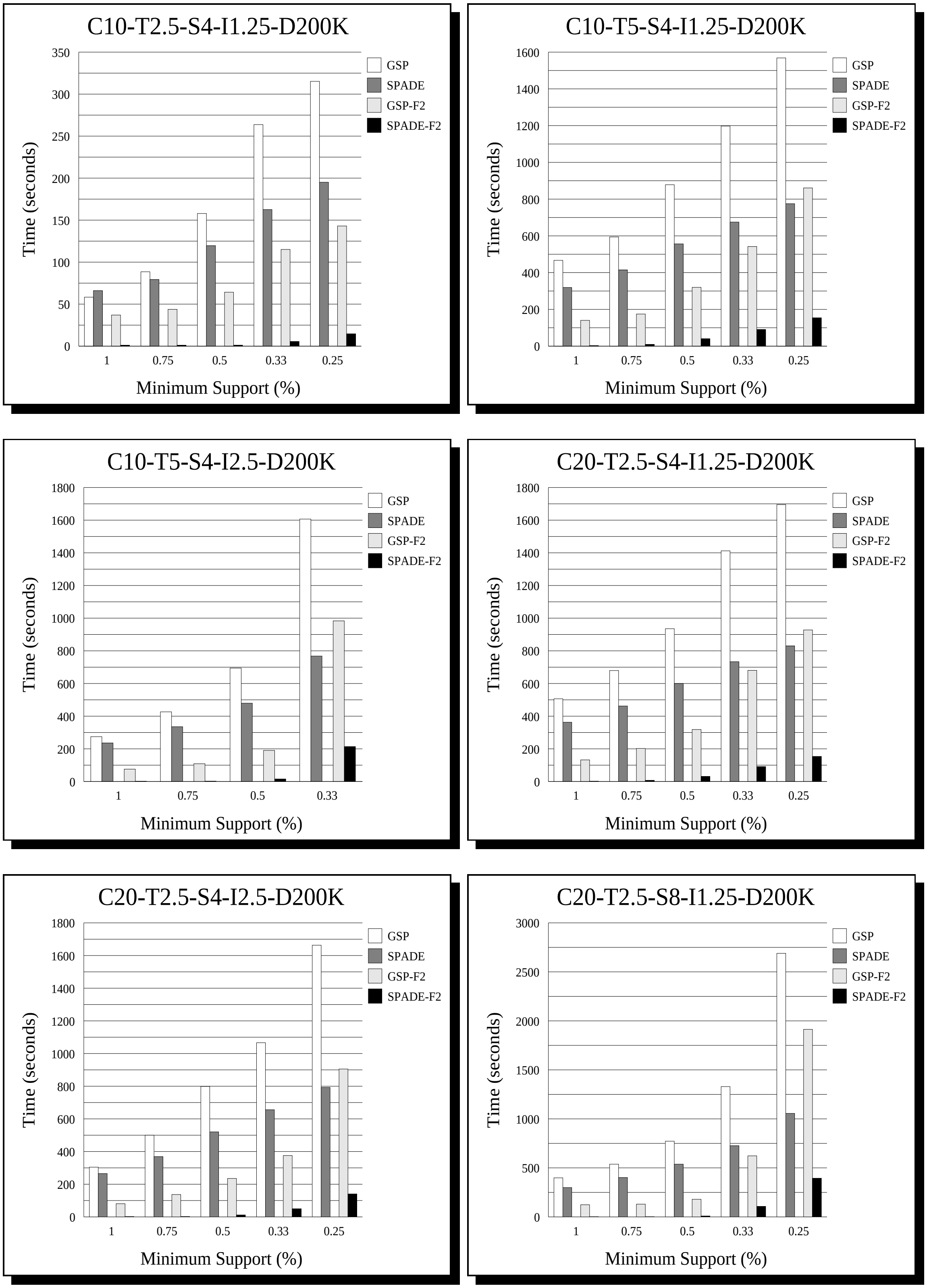


Figure 5.16: Performance Comparison: Synthetic Datasets

after the $\mathcal{F}_2$ pass since both algorithms spend roughly the same time in computing $\mathcal{F}_2$. Between $\mathcal{F}_3$ and $\mathcal{F}_k$, SPADE outperforms GSP anywhere from a factor of three to an order of magnitude.

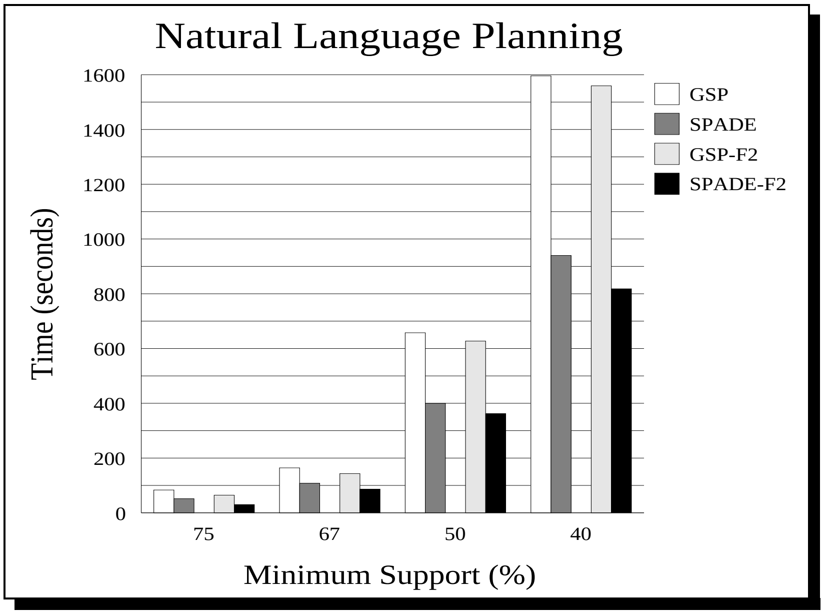


Figure 5.17: Performance Comparison: Planning Dataset

We also compared the performance of the two algorithms on the plan database. The results are shown in Figure 5.17. As in the case of synthetic databases, the SPADE algorithm outperforms GSP by a factor of two.

5.8.2 Scaleup

We first study how SPADE performs with increasing number of customers. Figure 5.18 shows how SPADE scales up as the number of customers is increased ten-fold, from 0.1 million to 1 million (the number of transactions is increased from 1 million to 10 million, respectively). All the experiments were performed on the *C10-T2.5-S4-I1.25* dataset with different minimum support levels ranging from 0.5% to 0.1%. The execution times are normalized with respect to the time for the 0.1 million customer dataset. It can be observed that SPADE scales quite linearly.

We next study the scale-up as we vary the dataset parameters in two ways: 1) keeping the average number of items per transaction constant, we increase the average number of transactions per customer; and 2) keeping the average number of transactions per customer constant, we increase the average number of items per transaction. The size of the datasets is kept nearly constant by ensuring that the product of the average transaction size, the average number of transactions per customer, and the number of customers $T \cdot C \cdot D$ remains the same. The aim of these experiments is to gauge the scalability with respect to the two test parameters, and independent of factors like database size or the number of frequent sequences.

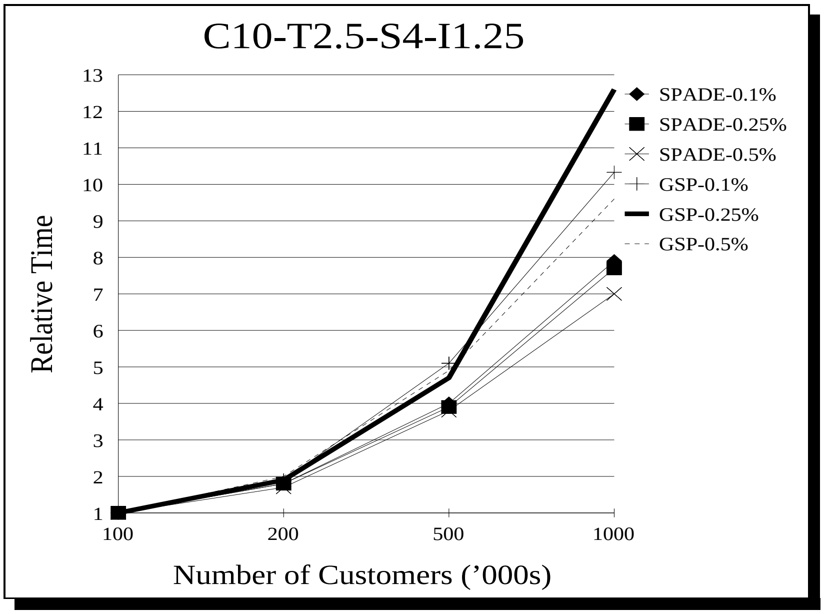


Figure 5.18: Scale-up: Number of Customers

Figure 5.19 shows the scalability results. To ensure that the number of frequent sequences doesn't increase by a great amount, we used an absolute minimum support value instead of using percentages (the graph legends indicate the value used). For both the graphs, we used *S4-I1.25*, and the database size was kept a constant at $T \cdot C \cdot D = 500K$. For the first graph we used $T = 2.5$, and varied C from 10 to 100 (D varied from 200K to 20K), and for the second graph we set $C = 10$, and varied T from 2.5 to 25 (D varied from 200K to 20K). It can be easily observed the the algorithm scales linearly with the two varying parameters. The scalability is also dependent on the minimum support value used, since for a lower minimum support relatively more frequent sequences are generated with increase in both the number of transactions and the transaction size, and thus it takes more time for pattern discovery in these cases.

We further study the scalability as we change the size of the maximal elements in two ways: 1) keeping all other parameters constant, we increase the average length of maximal potential frequent sequences; and 2) keeping all other parameters constant, we increase the average length of maximal potential frequent itemsets. The constant parameters for the first experiment were *C10-T2.5-I1.25-D200K*, and S was varied from 2 to 10. For the second experiment, the constant parameters were *C10-T5-S4-D200K*, and I was varied from 1 to 5.

Figure 5.20 shows how the algorithm scales with the two test parameters. For higher values of support the time starts to decrease with increasing maximal element size. This is because of the fact that the average transaction size and average number of customer transactions remains fixed, and increasing the maximal frequent sequence or itemset size means that fewer of these will fit in a customer-sequence, and thus fewer frequent sequences will be discovered. For lower values of support, however, a larger sequence

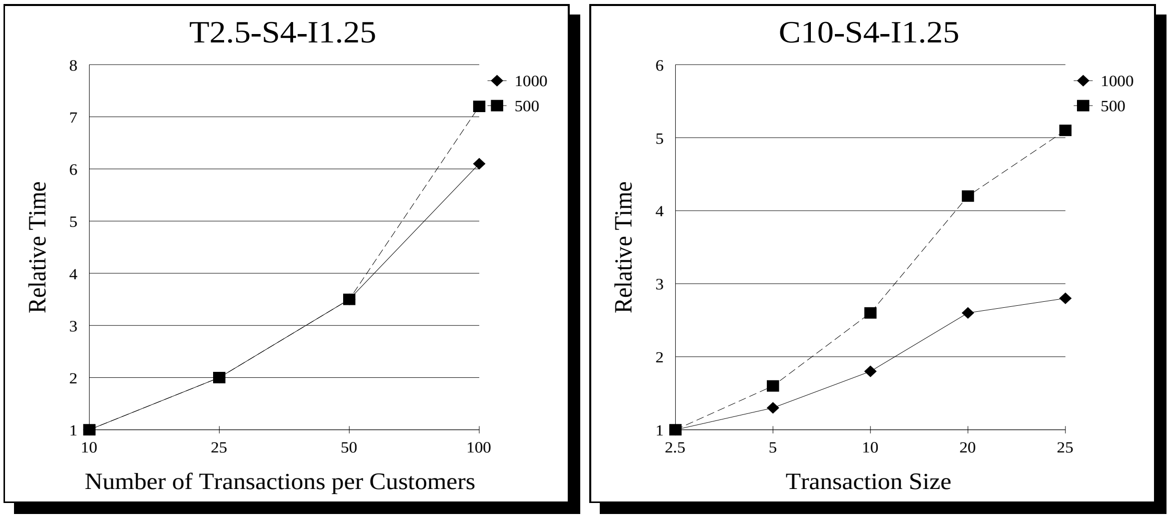


Figure 5.19: Scale-up: a) Number of Transactions/Customer; b) Transaction Size

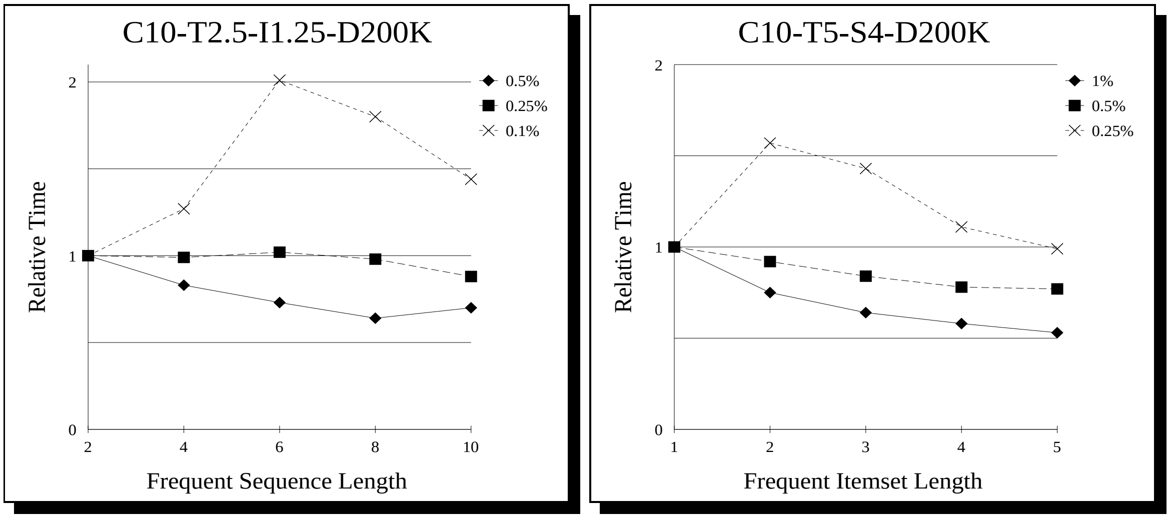


Figure 5.20: Scale-up: a) Frequent Sequence Length; b) Frequent Itemset Length

will introduce many more subsequences, thus the time starts to increase initially, but then decreases again due to the same reasons given above. The peak occurs at roughly the median values of $C10$ (at $S6$) for the sequences experiment, and of $T5$ (at $I2$) for the itemsets experiment.

5.9 Conclusions

In this chapter we presented SPADE, a new algorithm for fast mining of sequential patterns in large databases. Unlike previous approaches which make multiple database scans and use complex hash-tree structures that tend to have sub-optimal locality, SPADE decomposes the original problem into smaller sub-problems using equivalence classes on frequent sequences. Not only can each equivalence class be solved independently, but it is also very likely that it can be processed in main-memory. Thus SPADE usually makes only three database scans – one for frequent 1-sequences, another for frequent 2-sequences, and one more for generating all other frequent sequences. If the supports of 2-sequences is available then only one scan is required. SPADE uses only simple intersection operations, and is thus ideally suited for direct integration with a DBMS. An extensive set of experiments has been conducted to show that SPADE outperforms the best previous algorithm, GSP, by a factor of two, and by an order of magnitude with precomputed support of 2-sequences. It also has excellent scaleup properties with respect to a number of parameters such as the number of customers, the number of transactions per customer, the transaction size, and the size of potential maximal frequent itemsets and sequences.

6 Mining Classification Rules

6.1 Introduction

An important task of data mining can be thought of as the process of assigning things to predefined categories or classes – a process called *Classification*. Since the classes are predefined this is also known as *Supervised Induction*. The input for the classification system consists of a set of example records, called a *training set*, where each record consists of several fields or *attributes*. Attributes are either *continuous*, coming from an ordered domain, or *categorical*, coming from an unordered domain. One of the attributes, called the *classifying* attribute, indicates the *class* or label to which each example belongs. The induced model consists of patterns that are useful in class discrimination. Once induced, the model can help in the automatic prediction of new unclassified data. Classification has been identified as an important problem in the emerging field of data mining [Agrawal *et al.*, 1993a]. It has important applications in diverse domains such as retail target marketing, customer retention, fraud detection and medical diagnosis [Michie *et al.*, 1994].

Classification is a well-studied problem (see [Weiss and Kulikowski, 1991; Michie *et al.*, 1994] for excellent overviews) and several models have been proposed over the years, which include neural networks [Lippmann, 1987], statistical models like linear/quadratic discriminants [James, 1985], decision trees [Breiman *et al.*, 1984; Quinlan, 1993] and genetic algorithms [Goldberg, 1989]. Among these models, decision trees are particularly suited for data mining [Agrawal *et al.*, 1993a; Mehta *et al.*, 1996]. Decision trees can be constructed relatively fast compared to other methods. Another advantage is that decision tree models are simple and easy to understand [Quinlan, 1993]. Moreover, trees can be easily converted into SQL statements that can be used to access databases efficiently [Agrawal *et al.*, 1992]. Finally, decision tree classifiers obtain similar, and sometimes better, accuracy when compared with other classification methods [Michie *et al.*, 1994]. We have therefore focused on building scalable and parallel decision-tree classifiers.

While there has been a lot of research in classification in the past, the focus had been on memory-resident data, thus limiting their suitability for mining over large databases. Recent work has targeted the massive databases usual in data mining. Classifying larger datasets can enable the development of higher accuracy models. Various studies have confirmed this hypothesis [Catlett, 1991; Chan and Stolfo, 1993a; Chan and Stolfo, 1993b]. Examples of fast scalable classification systems include SLIQ [Mehta *et al.*, 1996], which for the first time was successful in handling disk-resident data. However, it did require some hashing information to be maintained in memory, restricting its scalability. The SPRINT [Shafer *et al.*, 1996] classifier was able to remove all such restrictions. It was also parallelized on the IBM SP2 parallel distributed-memory machine [Shafer *et al.*, 1996].

A continuing trend in data mining is the rapid and inexorable growth in the data that is being collected. The development of high-performance scalable data mining tools must necessarily rely on parallel computing techniques. Past work on parallel classification has utilized distributed-memory parallel machines. In such a machine, each processor has private memory and local disks, and communicates with other processors only via passing messages. Parallel distributed-memory machines are essential for scalable massive parallelism. However, shared-memory multiprocessor systems (SMPs), often called shared-everything systems, are also capable of delivering high performance for low to medium degree of parallelism at an economically attractive price. SMP machines are the dominant types of parallel machines currently used in industry. Individual nodes of parallel distributed-memory machines are also increasingly being designed to be SMP nodes. For example, an IBM SP2 parallel system may consist of up to 64 *high nodes*, where each high node is an 8-way SMP system with PowerPC 604e processors [IBMb]. A shared-memory system offers a single memory address space that all processors can access. Processors communicate through shared variables in memory and are capable of accessing any memory location. Synchronization is used to co-ordinate processes. Any processor can also access any disk attached to the system.

This chapter presents fast scalable decision-tree-based classification algorithms targeting shared-memory systems, the first such study. The algorithms are based on the sequential SPRINT classifier, and span the gamut of data and task parallelism. The data parallelism is based on attribute scheduling among processors. This is extended with task pipelining and dynamic load balancing to yield more complex schemes. The task parallel approach uses dynamic subtree partitioning among processors. These algorithms are evaluated on two SMP configurations: one in which data is too large to fit in memory and must be paged from a local disk as needed and the other in which memory is sufficiently large to hold the whole input data and all temporary files. For the local disk configuration, the speedup ranged from 2.97 to 3.86 for the build phase and from 2.20 to 3.67 for the total time on a 4-processor SMP. For the large memory configuration, the range of speedup was from 5.36 to 6.67 for the build phase and from 3.07 to 5.98 for the total time on an 8-processor SMP.

The rest of the chapter is organized as follows. We review related work in Section 6.2. In Section 6.3 we describe the sequential SPRINT decision-tree classifier, which forms the backbone of the new algorithms. This section is adapted from [Shafer *et al.*, 1996]. Section 6.4 describes our new SMP algorithms based on various data and task parallelization schemes. We give experimental results in Section 6.5 and conclude with a summary in Section 6.6.

6.2 Related Work

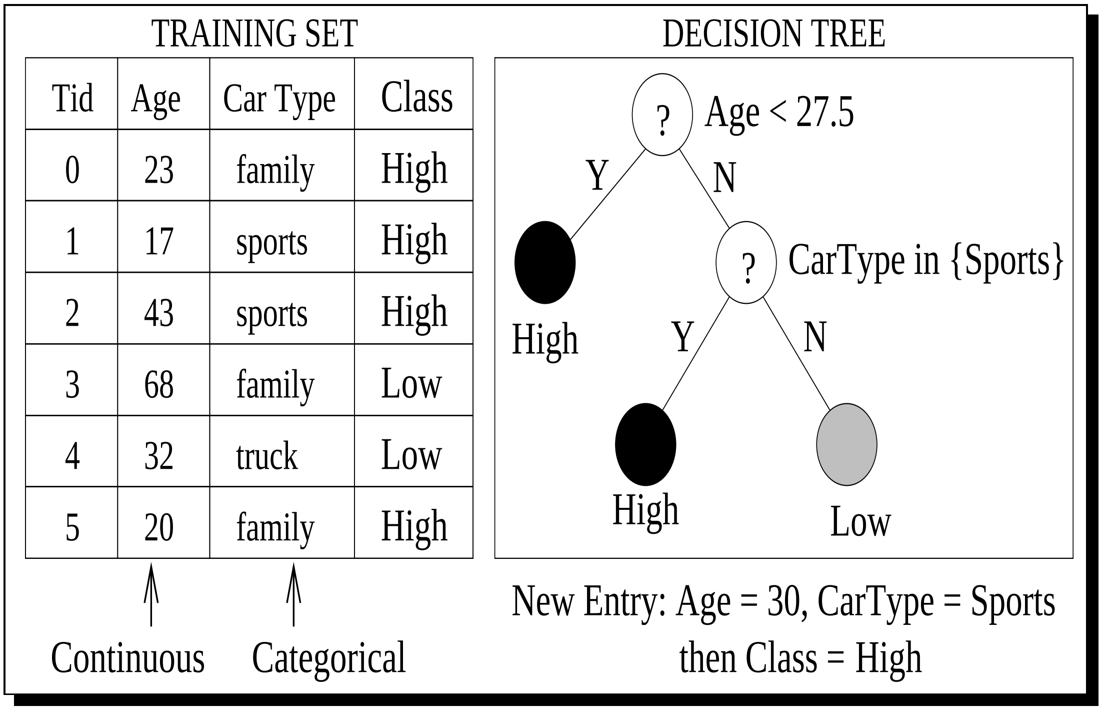
Random sampling is often used to handle large datasets when building a classifier. Previous work on building tree-classifiers from large datasets includes Catlett's study of two methods [Catlett, 1991; Wirth and Catlett, 1988] for improving the time taken to develop a classifier. The first method used data sampling at each node of the decision tree, and the second discretized continuous attributes. However, Catlett only considered datasets that could fit in memory; the largest training data had only 32,000 examples. Chan and Stolfo [Chan and Stolfo, 1993a; Chan and Stolfo, 1993b] considered partitioning the data into subsets that fit in memory and then developing a classifier on each subset in parallel. The output of multiple classifiers is combined using various algorithms to reach the final classification. Their studies showed that although this approach reduces running time significantly, the multiple classifiers did not achieve the accuracy of a single classifier built using all the data. Incremental learning methods, where the data is classified in batches, have also been studied [Quinlan, 1979; Wirth and Catlett, 1988]. However, the cumulative cost of classifying data incrementally can sometimes exceed the cost of classifying the entire training set once. In [Agrawal *et al.*, 1992], a classifier built with database considerations, the size of the training set was overlooked. Instead, the focus was on building a classifier that could use database indices to improve the retrieval efficiency while classifying test data.

Work by Fifield in [Fifield, 1992] examined parallelizing the ID3 [Quinlan, 1986] decision-tree classifier, but it assumes that the entire dataset can fit in memory and does not address issues such as disk I/O. The algorithms presented there also require processor communication to evaluate any given split point, limiting the number of possible partitioning schemes the algorithms can efficiently consider for each leaf. The Darwin toolkit from Thinking Machines also contained a parallel implementation of the decision-tree classifier CART [Breiman *et al.*, 1984]; however, details of this parallelization are not available in published literature.

The recently proposed SLIQ classification algorithm [Mehta *et al.*, 1996] addressed several issues in building a fast scalable classifier. SLIQ gracefully handles disk-resident data that is too large to fit in memory. It does not use small memory-sized datasets obtained via sampling or partitioning, but builds a single decision tree using the *entire* training set. However, SLIQ does require that some data per record stay memory-resident all the time. Since the size of this in-memory data structure grows in direct proportion to the number of input records, this limits the amount of data that can be classified by SLIQ. Its successor, the SPRINT classifier [Shafer *et al.*, 1996] removed all memory restrictions and was fast and scalable. It was also parallelized on the IBM SP2 distributed-memory system. As noted earlier, our goal in this chapter is to study the efficient implementation of SPRINT on shared-memory systems. These machines represent a popular parallel programming architecture and paradigm, and have very different characteristics.

6.3 Serial Classification

A decision tree contains tree-structured nodes. Each node is either a *leaf*, indicating a class, or a *decision node*, specifying some test on one or more attributes, with one branch or subtree for each of the possible outcomes of the split test. Decision trees successively divide the set of training examples until all the subsets consist of data belonging entirely, or predominantly, to a single class. Figure 6.1 shows a decision-tree classifier developed from the example training set. ($Age < 27.5$) and ($CarType \in \{sports\}$) are two split points that partition the records into *High* and *Low* risk classes. The decision tree can be used to screen future insurance applicants by classifying them into the High or Low risk categories.



TRAINING SET

Tid	Age	Car Type	Class
0	23	family	High
1	17	sports	High
2	43	sports	High
3	68	family	Low
4	32	truck	Low
5	20	family	High

Figure 6.1: Car Insurance Example

A decision tree classifier is usually built in two phases [Breiman *et al.*, 1984; Quinlan, 1993]: a growth phase and a prune phase. The tree is grown using an elegantly simple divide and conquer approach. It takes as input a set of training examples S. There are basically two cases to be considered. If all examples in S entirely or predominantly (a user specified parameter) belong to a single class, then S becomes a leaf in the tree. If on the other hand it contains a mixture of examples from different classes, we need to further partition the input into subsets that tend towards a single class. The data is partitioned based on a test on the attributes, and can take the form of a binary or k-ary split. We will consider only binary splits because they usually lead to more accurate trees; however, our techniques can be extended to handle multi-way splits. Based on the outcomes, the data is partitioned into two subsets S_1 and S_2, which in turn serve as inputs to the recursive process. This tree growth phase is shown in Figure 6.2.

```
Partition(Data $S$)
    if (all points in $S$ are of the same class) then
        return;
    for each attribute $A$ do
        evaluate splits on attribute $A$;
    Use best split found to partition $S$ into $S_1$ and $S_2$;
    Partition($S_1$);
    Partition($S_2$);

Initial call: Partition(TrainingData)
```

Figure 6.2: General Tree-growth Algorithm

The tree built using the recursive partitioning approach can become very complex, and as such can be thought of as being an "overfit" of the data. Remember that the goal of classification is to predict new unseen cases. The tree pruning phase tries to generalize the tree by removing dependence on statistical noise or variation that may be particular only to the training set. This step requires access only to the fully grown tree, while the tree growth phase usually requires multiple passes over the training data, and as such is much more expensive. Previous studies from SLIQ suggest that usually less than 1% of the total time needed to build a classifier was spent in the pruning phase. In this chapter we will therefore only concentrate on the computation and I/O intensive tree growth phase. We use a Minimum Description Length [Rissanen, 1989] based algorithm for the tree pruning phase. (see [Mehta *et al.*, 1996] for additional details). There are two major issues that have critical performance implications in the tree-growth phase.

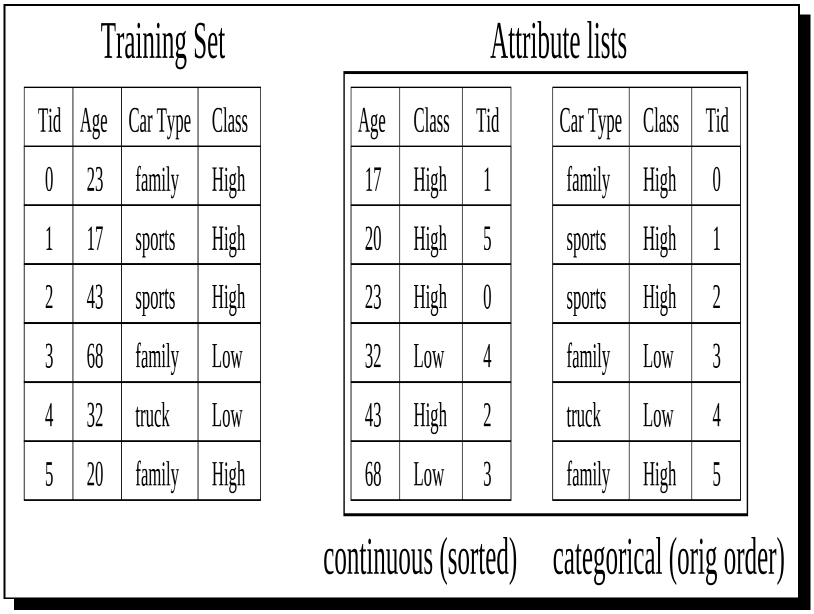
1. How to find split points that define node tests.

2. Having chosen a split point, how to partition the data.

We will first look at the data structures used in SPRINT, and then describe how it handles the two steps above.

6.3.1 Data Structures

The SPRINT classifier was designed to be disk-based. It builds the tree breadth-first and uses a one-time pre-sorting technique to reduce the cost of continuous attribute evaluation. In contrast to this, the well-known CART [Breiman *et al.*, 1984] and C4.5 [Quinlan, 1993] classifiers, grow trees depth-first and repeatedly sort the data at every node of the tree to arrive at the best splits for continuous attributes.



Training Set

Tid	Age	Car Type	Class
0	23	family	High
1	17	sports	High
2	43	sports	High
3	68	family	Low
4	32	truck	Low
5	20	family	High

Attribute lists

Age	Class	Tid
17	High	1
20	High	5
23	High	0
32	Low	4
43	High	2
68	Low	3

continuous (sorted)

Car Type	Class	Tid
family	High	0
sports	High	1
sports	High	2
family	Low	3
truck	Low	4
family	High	5

categorical (orig order)

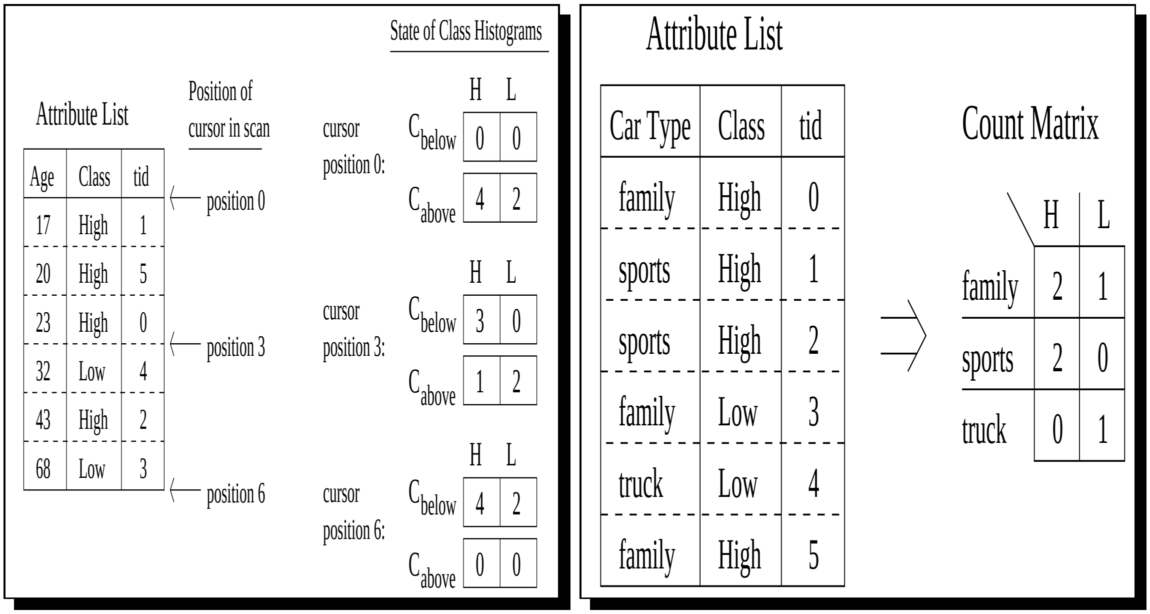
Figure 6.3: Attribute Lists

Attribute lists SPRINT initially creates an disk-based *attribute list* for each attribute in the data. Each entry in the list is called an *attribute record*, and consists of an attribute value, a class label, and a record identifier or *tid*. Initial lists for continuous attributes are sorted by attribute value when first created. The lists for categorical attributes remain in unsorted order. Figure 6.3 shows the initial attribute lists for our example training set. The initial lists created from the training set are associated with the root of the classification tree. As the tree is grown and is split into two subtrees, the attribute lists are also split at the same time. By simply preserving the order of records in the partitioned lists, they don't require resorting. Figure 6.5 shows an example of the initial sorted attribute lists associated with the root of the tree and also the resulting partitioned lists for the two children.

Histograms SPRINT uses histograms tabulating the class distributions of the input records. For continuous attributes, two histograms are maintained – C_{below} keeps the counts for examples that have already been processed, and C_{above} for those that have not been seen yet. For categorical attributes one histogram, called the *count matrix*, is sufficient for the class distribution of a given attribute. There is one set of histograms for each node in the tree. However, since the attribute lists are processed one after the other, only one set of histograms need be kept in memory at any given time. Example histograms are shown in Figure 6.4.

6.3.2 Finding good split points

The form of the split used to partition the data depends on the type of the attribute used in the split. Splits for a continuous attribute A are of the form $value(A) < x$ where



Age	Class	tid
17	High	1
20	High	5
23	High	0
32	Low	4
43	High	2
68	Low	3

	H	L
cursor position 0: C_{below}	0	0
C_{above}	4	2
cursor position 3: C_{below}	3	0
C_{above}	1	2
cursor position 6: C_{below}	4	2
C_{above}	0	0

Car Type	Class	tid
family	High	0
sports	High	1
sports	High	2
family	Low	3
truck	Low	4
family	High	5

	H	L
family	2	1
sports	2	0
truck	0	1

Figure 6.4: Evaluating a) Continuous and b) Categorical Split Points

x is a value in the domain of A. Splits for a categorical attribute A are of the form $value(A) \in X$ where $X \subset domain(A)$.

To build compact trees modeling the training set, one approach would be to explore the space of all possible trees and select the best one. This process is unfortunately NP-Complete. Most tree construction methods therefore use a *greedy* approach, the idea being to determine the split point that "best" divides the training records belonging to that leaf. The "goodness" of the split obviously depends on how well it separates the classes. Several splitting indices have been proposed in the past to evaluate the goodness of the split. SPRINT uses the *gini* index [Breiman *et al.*, 1984] for this task. For a data set S containing examples from n classes, $gini(S)$ is defined as

$$gini(S) = 1 - \sum p_j^2$$

where p_j is the relative frequency of class j in S. If a split divides S into two subsets S_1 and S_2, with n_1 and n_2 classes, respectively, the index of the divided data $gini_{split}(S)$ is given by

$$gini_{split}(S) = \frac{n_1}{n} gini(S_1) + \frac{n_2}{n} gini(S_2)$$

The computation of the gini index only requires the class distributions on each side of a candidate partition point. This information is kept in the class histograms described above. Once this information is known, the best split can be found using a simple approach. Each node's attribute lists are scanned, the histograms are updated, and the gini value for different split points is calculated. The attribute with the minimum gini value and the associated split point is used to partition the data.

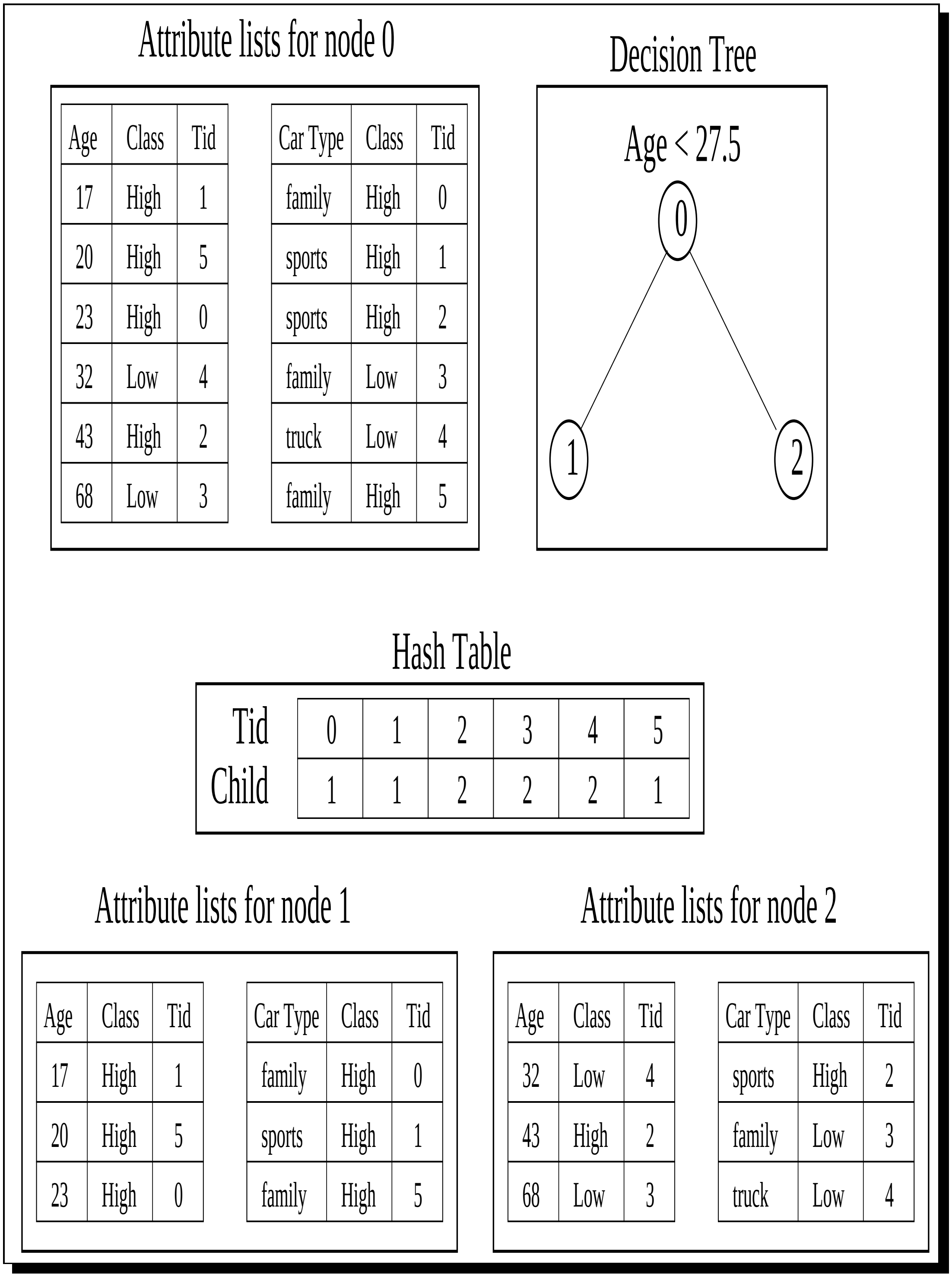
Continuous attributes For continuous attributes, the candidate split points are mid-points between every two consecutive attribute values in the training data. Initially C_{below} has all zeros, while C_{above} has the class distributions of the current node under consideration. For each attribute record, the histograms are updated and the gini index is calculated. The current minimum gini value, also called the *winning* split point is saved during this process. Since all the continuous values are sorted one scan over the attribute lists is sufficient. Figure 6.4a illustrates the histogram update for continuous attributes.

Categorical attributes For categorical attributes, all possible subsets of the attribute values are considered as potential split points. If the cardinality is too large a greedy subsetting algorithm (initially used in IND [NASA, 1992]) is used. The histogram updation is shown in Figure 6.4b.

6.3.3 Splitting the data

Once the winning split point has been found for a node, the node is split into two children, along with the division of the node's attribute lists into two. Figure 6.5 shows this process. The attribute list splitting for the winning attribute (Age in our example) is quite straightforward. We simply scan the attribute records, and apply the split test. Those records satisfying the test go to the left child, and those that fail the test go to the right child. For the remaining "losing" attributes ($CarType$ in our example) more work is needed. While dividing the winning attribute SPRINT also constructs a probe structure (bit mask or hash table) on the *tids*, noting the child where a particular record belongs. To split the other attributes now only requires a scan of each record and a probe to determine the child where this record should be placed. This probe structure need not be memory-resident. If it occupies too much memory the splitting takes multiple steps. In each step only a portion of the attribute lists are partitioned. At the same time the split happens, SPRINT also collects the new class distribution histograms for the children nodes.

Avoiding multiple attribute lists Recall that the attribute lists of each attribute are stored in disk files. As the tree is split, we would need to create new files for the children, and delete the parent's files. File creation is usually an expensive operation, and this process can add significant overhead. For example, Figure 6.6 shows that if we create new files at each level, then if the tree has N levels, we would potentially require 2^N files. Rather than creating a separate attribute list for each attribute for each node, SPRINT actually uses only four physical files per attribute. Since we are dealing with binary splits, we have one attribute file for all leaves that are "left" children (file $L0$) and one file for all leaves that are "right" children (file $R0$). We also have two more list files per attribute that serve as alternates (files $L1$ and $R1$). All the attribute records for a node are kept in a contiguous section within one of the two primary files (files $L0$ and $R0$). When reading records for a particular node, we read the corresponding portion of the attribute's left or right file. When splitting a node's attribute records,



Attribute lists for node 0

Age	Class	Tid
17	High	1
20	High	5
23	High	0
32	Low	4
43	High	2
68	Low	3

Car Type	Class	Tid
family	High	0
sports	High	1
sports	High	2
family	Low	3
truck	Low	4
family	High	5

Hash Table

Tid	0	1	2	3	4	5
Child	1	1	2	2	2	1

Attribute lists for node 1

Age	Class	Tid
17	High	1
20	High	5
23	High	0

Car Type	Class	Tid
family	High	0
sports	High	1
family	High	5

Attribute lists for node 2

Age	Class	Tid
32	Low	4
43	High	2
68	Low	3

Car Type	Class	Tid
sports	High	2
family	Low	3
truck	Low	4

Figure 6.5: Splitting a Node's Attribute Lists

we append the left and right child's records to the end of the alternate left and right files (files $L1$ and $R1$).

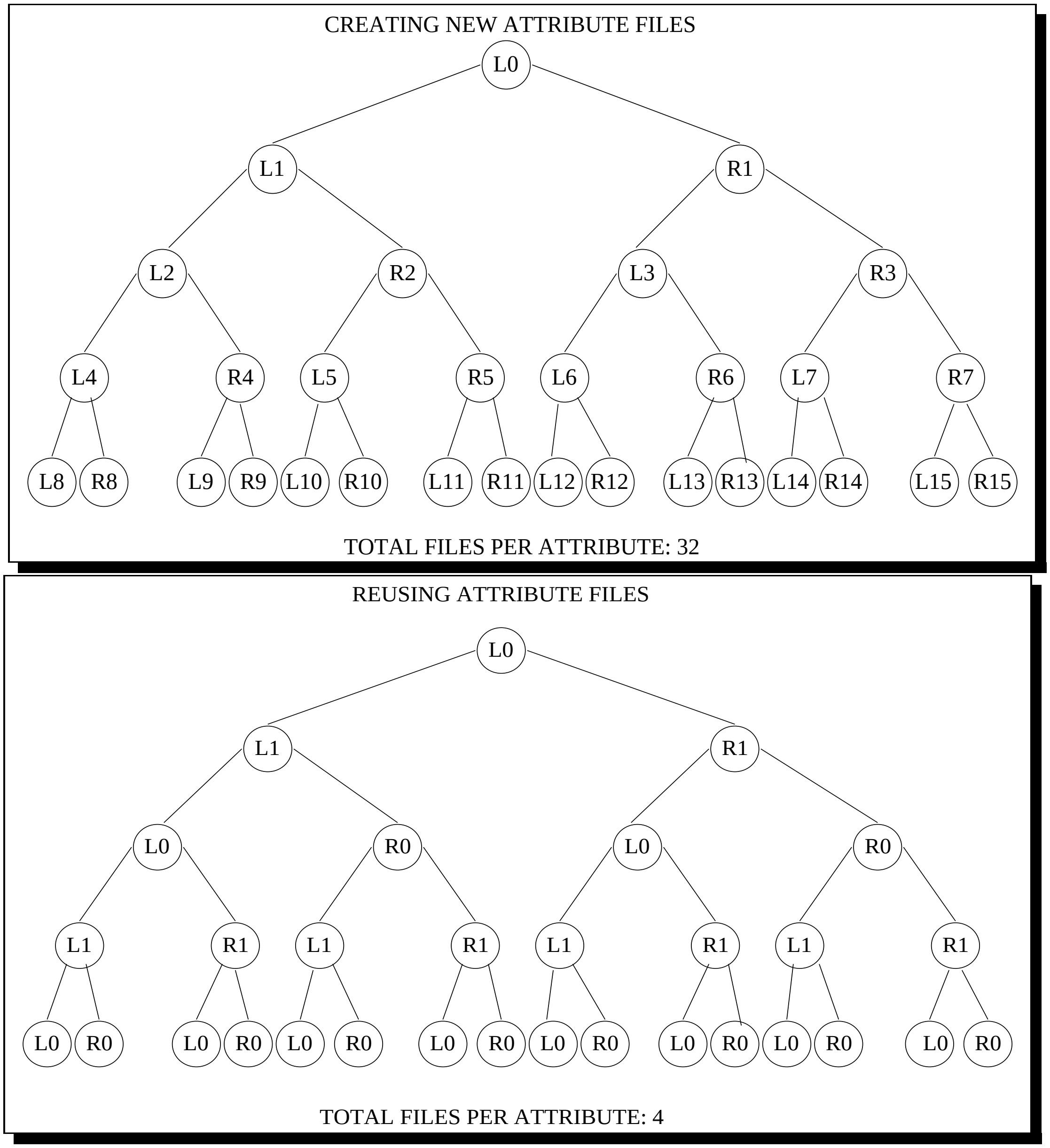


Figure 6.6: Avoiding Multiple Attribute Files

After splitting each node, all the training data will reside in the alternate files. These become the primary files for the next level. The old primary lists are cleared and they become new alternate files. Thus, we never have to pay the penalty of creating new attribute lists for new leaves; we can simply reuse the ones we already have. By processing tree nodes in the order they appear in the attribute files, this approach also avoids any random seeks within a file to find a node's records — reading and writing remain sequential operations. This optimization can be done with virtually no overhead and with no modifications to the SPRINT algorithm. It also has important implications for the parallelization strategies presented below.

6.4 Parallel Classification on Shared-memory Systems

We now turn our attention to the problem of building classification trees in parallel on SMP systems. We will only discuss the tree growth phase due to its compute and data-intensive nature. Tree pruning is relatively inexpensive [Mehta *et al.*, 1996], as it requires access to only the decision-tree grown in the training phase.

6.4.1 SMP Schemes

While building a decision-tree, there are three main steps that must be performed for each node at each level of the tree:

1. Evaluate split points for each attribute (denoted as step $\mathcal{E}$).
2. Find the winning split-point and construct a probe structure using the attribute list of the winning attribute (denoted as step $\mathcal{W}$).
3. Split all the attribute lists into two parts, one for each child, using the probe structure (denoted as step $\mathcal{S}$).

Our parallel schemes will be described in terms of these steps. Our prototype implementation of these schemes uses the POSIX threads (pthread) standard [Lewis and Berg, 1996]. A thread is a light weight process. It is a *schedulable* entity that has only those properties that are required to ensure its independent flow of control, including the stack, scheduling properties, set of pending and blocking signals, and some thread-specific data. To keep the exposition simple, we will not differentiate between threads and processes and pretend as if there is only one process per processor. We propose two approaches to building a tree classifier in parallel: a *data parallel* approach and a *task parallel* approach.

Data parallel In data parallelism the P processors work on distinct portions of the datasets and synchronously construct the global decision tree. It essentially exploits the intra-node parallelism, i.e. that available within a decision tree node. There are two kinds of data parallelism possible in classification. In the first case, which can be called *record parallelism*, we split the attribute lists evenly among all the processors. Each processor is responsible for $1/P$ records from each attribute list. The implementation of this scheme on the IBM SP2 was presented in [Shafer *et al.*, 1996]. In the second case, called *attribute parallelism*, the attributes are divided equally among the different processors so that each element is responsible for $1/P$ attributes. The record parallelism approach doesn't look very promising on an SMP system. Some of the reason are: 1) It would require a lot of data structure and attribute file replication. For example, each input file will have to be pre-split into P files, or it will have to split during the attribute list creation phase, resulting in P files per attribute. The hash probe, count matrix and class histograms will also have to be replicated on each processor, so that they can independently process their local attribute lists. 2) If the attribute lists are partitioned in a simple blocked manner, there is potential of load imbalance, especially if the record values are skewed. Furthermore, performing any kind of dynamic load balancing is likely to be quite complex and will introduce extra overhead. In the following discussion we will present three schemes of varying complexity based on the attribute parallelism approach. As part of future work we plan to experimentally verify our hypothesis for the record parallelism approach.

Task Parallelism The other major kind of parallelism is provided by the task parallel approach. It exploits the inter-node parallelism, i.e. different portions of the decision tree can be built in parallel among the processors.

6.4.2 Attribute Data Parallelism

We first describe the Moving-Window-K algorithm (MWK) based on attribute data parallelism. For pedagogical reasons, we will introduce two intermediate schemes called BASIC and Fixed-Window-K (FWK) and then evolve them to the more sophisticated MWK algorithm. MWK and the two intermediate schemes utilize dynamic attribute scheduling. In a static attribute scheduling, each process gets d/P attributes where d denotes the number of attributes. However, this static partitioning is not particularly suited for classification. Different attributes may have different processing costs because of two reasons. First, there are two kinds of attributes – continuous and categorical, and they use different techniques to arrive at split tests. Second, even for attributes of the same type, the computation depends on the distribution of the record values. For example, the cardinality of the value set of a categorical attribute determines the cost of gini index evaluation. These factors warrant a dynamic attribute scheduling approach.

The Basic Scheme (BASIC)

Figure 6.7 shows the pseudo-code for the BASIC scheme. A barrier represents a point of synchronization. While a full tree is shown here, the tree generally may have a sparse irregular structure. At each level a processor evaluates the assigned attributes, which is followed by a barrier.

Attribute scheduling Attributes are scheduled dynamically by using an attribute counter and locking. A processor acquires the lock, grabs an attribute, increments the counter, and releases the lock. This method achieves the same effect as self-scheduling [Tang and Yew, 1986], i.e., there is lock synchronization per attribute. For typical classification problems with up to a few hundred attributes, this approach works fine. For thousands of attributes self-scheduling can generate too much unnecessary

```
// Starting with the root node execute the
// following code for each new tree level
forall attributes in parallel (dynamic scheduling)
    for each leaf
        evaluate attributes (E)

barrier
if (master) then
    for each leaf
        get winning attribute; form hash-probe (W)

barrier
forall attributes in parallel (dynamic scheduling)
    for each leaf
        split attributes (S)
```

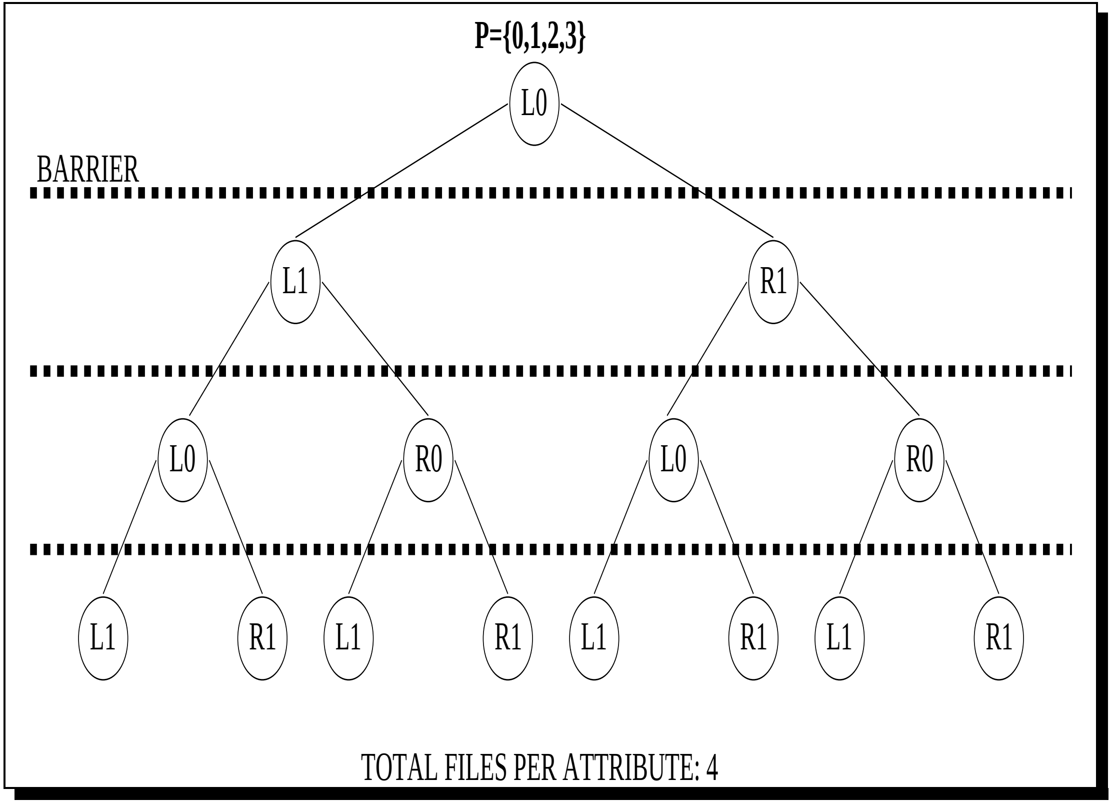


Figure 6.7: The BASIC Algorithm

synchronization. The latter can be addressed by using guided self-scheduling [Polychronopoulos and Kuck, 1987] or its extensions, where a processor grabs a dynamically shrinking chunk of remaining attributes, thus minimizing the synchronization. Another possibility would be to use affinity scheduling [Markatos and LeBlanc, 1994], where attention is paid to the location of the attribute lists so that accesses to local attribute lists are maximized.

Finding split points ($\mathcal{E}$) Since each attribute has its own set of four reusable attribute files, as long as no two processors work on the same attribute at the same time, there is no need for file access synchronization. To minimize barrier synchronization the tree is built in a breadth-first manner. The advantage is that once a processor has been assigned an attribute, it can evaluate the split points for that attribute for all the leaves in the current level. This way, each attribute list is accessed only once sequentially during the evaluation for a level. Once all attributes have been processed in this fashion, a single barrier ensures that all processors have reached the end of the attribute evaluation phase. In contrast, depth-first tree growth would require a barrier synchronization once per leaf, which could become a significant source of overhead in large trees.

As each processor works independently on the entire attribute list, they can independently carry out gini index evaluation to determine the best split point for each attribute assigned to it.

Hash probe construction ($\mathcal{W}$) Once all the attributes of a leaf have been processed, each processor will have what it considers to be the best split for that leaf. We now need to find the best split point from among each processor's locally best split. We can then proceed to scan the winning attribute's records and form the hash probe.

The breadth-first tree construction imposes some constraints on the hash probe construction. We could keep separate hash tables for each leaf. If there is insufficient memory to hold these hash tables in memory, they would have to be written to disk. The size of each leaf's hash table can be reduced by keeping only the smaller child's *tids*, since the other records must necessarily belong to the other child. Another option is to maintain a global bit probe for all the current leaves. It has as many bits as there are tuples in the training set. As the records for each leaf's winning attribute are processed, the corresponding bit is set to reflect whether the record should be written to a *left* or *right* file. A third approach is to maintain an index of valid *tids* of a leaf, and relabel them starting from zero. Then each leaf can keep a separate bit probe.

BASIC uses the second approach, that maintains a global bit vector, due to its simplicity. Both the tasks of finding the minimum split value and bit probe construction are performed serially by a pre-designated master processor. This step thus represents a potential bottleneck in this BASIC scheme, which we will eliminate later in MWK. During the time the master computes the hash probe, the other processors enter a barrier and go to sleep. Once the master finishes, it also enters the barrier and wakes up the sleeping processors, setting the stage for the splitting phase.

Attribute list splitting ($\mathcal{S}$) The attribute list splitting phase proceeds in the same manner as the evaluation. A processor dynamically grabs an attribute, scans its records, hashes on the *tid* for the child node, and performs the split. Since the files for each attribute are distinct there is no read/write conflict among the different processors.

The Fixed-Window-K Scheme (FWK)

We noted above that the winning attribute hash probe construction phase $\mathcal{W}$ in BASIC is a potential sequential bottleneck. The Fixed-Window-K (FWK) scheme shown in Figure 6.8 addresses this problem. The basic idea is to overlap the $\mathcal{W}$-phase with the $\mathcal{E}$-phase of the next leaf at the current level, a technique called *task pipelining*. The degree of overlap can be controlled by a parameter K denoting the window of current overlapped leaves. Let $\mathcal{E}_i$, $\mathcal{W}_i$, and $\mathcal{S}_i$ denote the evaluation, winning hash construction, and partition steps for leaf i at a given level. Then for $K = 2$, we get the overlap of $\mathcal{W}_0$ with $\mathcal{E}_1$. For $K = 3$, we get an overlap of $\mathcal{W}_0$ with $\{\mathcal{E}_1, \mathcal{E}_2\}$, and an overlap of $\mathcal{W}_1$ with $\mathcal{E}_2$. For a general K, we get an overlap of $\mathcal{W}_i$ with $\{\mathcal{E}_{i+1}, \cdots, \mathcal{E}_{K-1}\}$, for all $1 \leq i \leq K - 1$.

The attribute scheduling, split finding, and partitioning remain the same. The difference is that depending on the window size K, we group K leaves together. For each leaf within the K-block (i.e., K leaves of the same group), we first evaluate all attributes. At the last leaf in each block we perform a barrier synchronization to ensure that all evaluations for the current block have completed. The hash probe for a leaf is constructed by the last processor to exit the evaluation for that leaf. This ensures that no two processors access the hash probe at the same time.

Managing attribute files There are four reusable files per attribute in the BASIC scheme. However, if we are to allow overlapping of the hash probe construction step with the evaluation step, which uses dynamic attribute scheduling within each leaf, we would require K distinct files for the current level, and K files for the parent's attribute lists, that is $2K$ files per attribute. This way all K leaves in a group have separate files for each attribute and there is no read/write conflict. Another complication arises from the fact that some children may turn out to be pure (i.e., all records belong to the same class) at the next level. Since these children will not be processed after the next stage, we have to be careful in the file assignment for these children. A simple file assignment, without considering the child purity, where children are assigned files from $0, \cdots, K - 1$, will not work well, as it may introduce "holes" in the schedule (see Figure 6.9). However, if we knew the pure children of the next level, we can do better.

The class histograms gathered while splitting the children are adequate to determine purity. We add a pre-test for child purity at this stage. If the child will become pure at the next level, it is removed from the list of valid children, and the files are assigned consecutively among the remaining children. This insures that there are no holes in the K block, and we get perfect scheduling. The two approaches are contrasted in Figure 6.9. The bold circles show the valid children for the current and next level. With the simple labeling scheme the file labels for the valid children are $L0, L0, R0, R0, R0$. With a

```
// Starting with the root node execute the
// following code for each new tree level
forall attributes in parallel (dynamic scheduling)
    for each block of K leaves
        for each leaf i
            evaluate attributes (E)
        if (last leaf of block) then
            barrier
        if (last processor finishing on leaf i) then
            get winning attribute; form hash-probe (W)

barrier
forall attributes in parallel (dynamic scheduling)
    for each leaf
        split attributes (S)
```

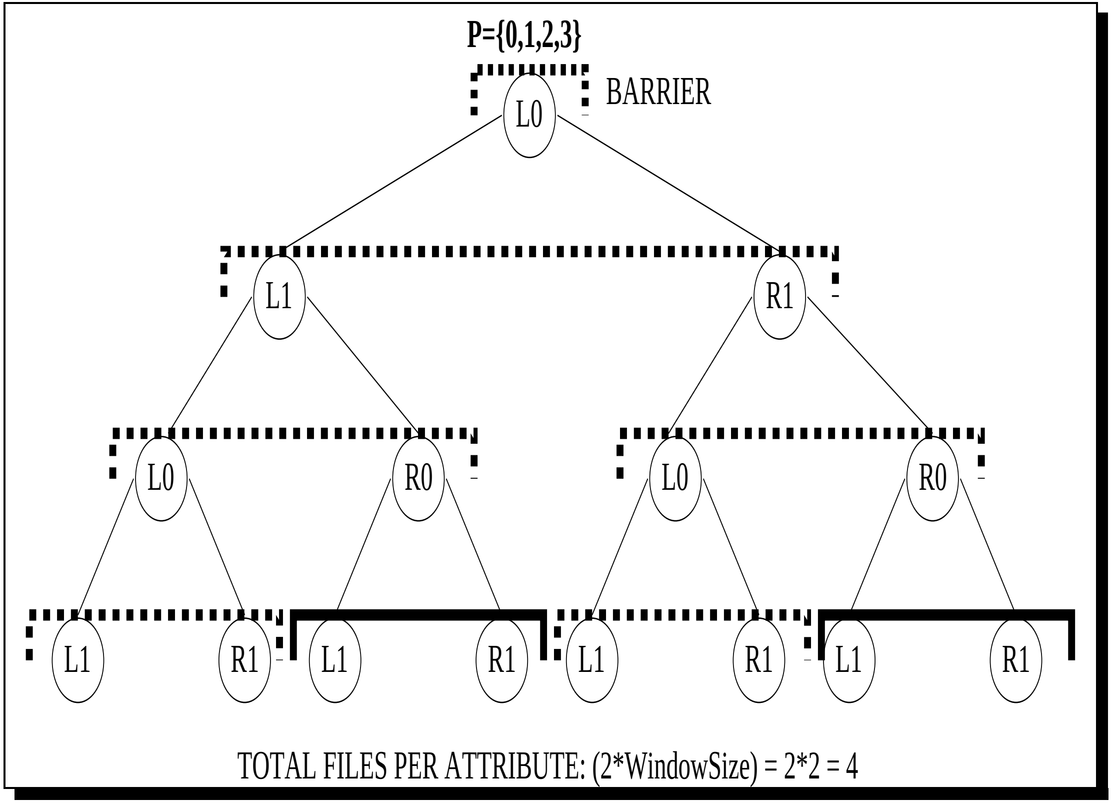


Figure 6.8: The FWK Algorithm

window of size $K = 2$, there is only one instance where work can overlap, i.e., when going from $L0$ to $R0$. However, if we relabel the valid children's files then we obtain the perfectly schedulable sequence $L0, R0, L0, R0, L0$.

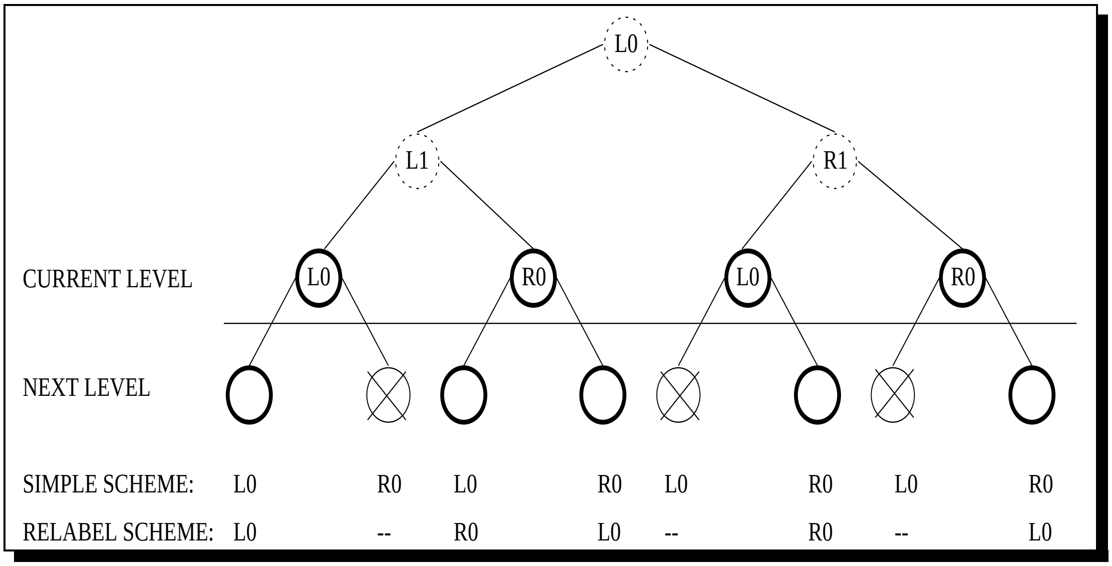

L0
L1
R1
CURRENT LEVEL
L0
R0
L0
R0
NEXT LEVEL
SIMPLE SCHEME: L0 R0 L0 R0 L0 R0 L0 R0
RELABEL SCHEME: L0 -- R0 L0 -- R0 -- L0

Figure 6.9: Scheduling Attribute Files

Note that the overlapping of work is achieved at the cost of increased barrier synchronization, one per each K-block. A large window size not only increases the overlap but also minimizes the number of synchronizations. However, a larger window size requires more temporary files, which incurs greater file creation overhead and tends to have less locality. The ideal window size is a trade-off between the above conflicting goals.

The Moving-Window-K Algorithm (MWK)

We now describe the Moving-Window-K (MWK) algorithm which eliminates the serial bottleneck of BASIC and exploits greater parallelism than FWK. Figures 6.10 shows the pseudo-code for MWK.

Consider a current leaf frontier: $\{L0_1, R0_1, L0_2, R0_2\}$. With a window size of $K = 2$, not only is there parallelism available for fixed blocks $\{L0_1, R0_1\}$ and $\{L0_2, R0_2\}$ (used in FWK), but also between these two blocks, $\{R0_1, L0_2\}$. The MWK algorithm makes use of this additional parallelism.

This scheme is implemented by replacing the barrier per block of K leaves with a wait on a *conditional variable.* Before evaluating leaf i, a check is made whether the i-th leaf of the previous block has been processed. If not, the processor goes to sleep on the conditional variable. Otherwise, it proceeds with the current leaf. The last processor to finish the evaluation of leaf i from the previous block constructs the hash probe, and then signals the conditional variable, so that any sleeping processors are woken up.

```
// Starting with the root node execute the
// following code for each new tree level
forall attributes in parallel (dynamic scheduling)
    for each block of K leaves
        for each leaf i
            if  (last block's i-th leaf not done) then
                wait
            evaluate attributes (E)
            if (last processor finishing on leaf i) then
                get winning attribute; form hash-probe (W)
                signal that i-th leaf is done

barrier
forall attributes in parallel (dynamic scheduling)
    for each leaf
        split attributes (S)
```

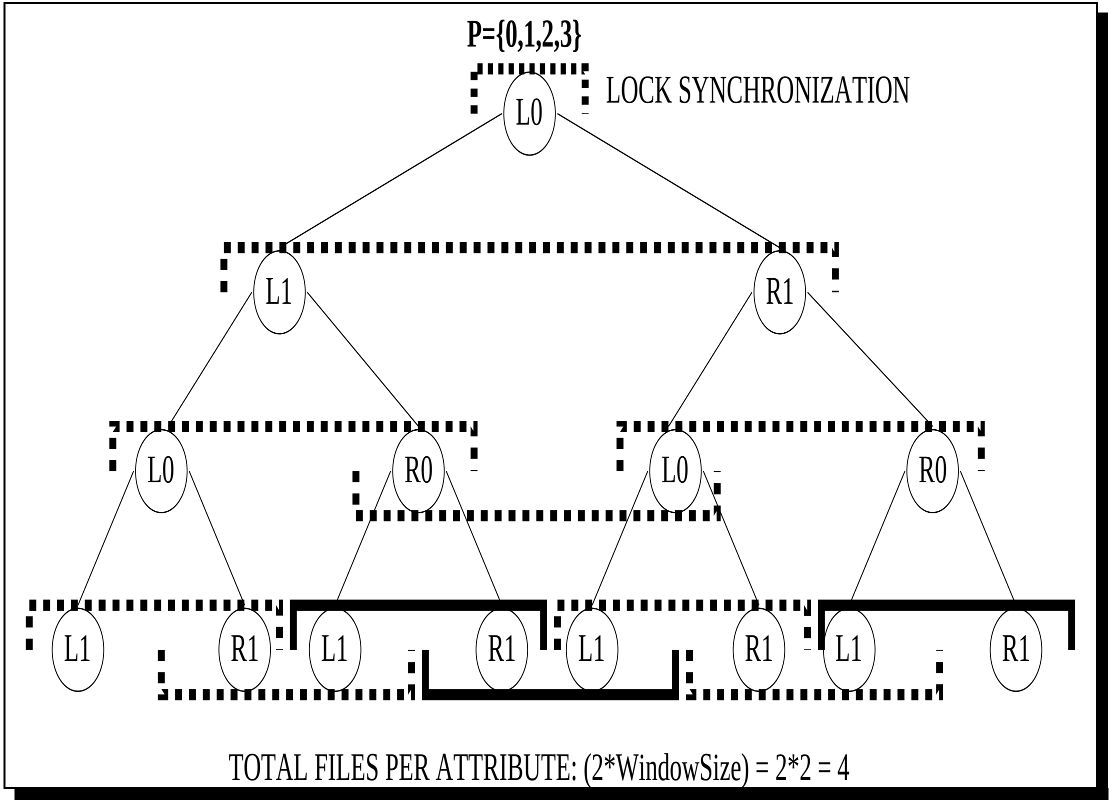


Figure 6.10: The MWK Algorithm

It should be observed that the gain in available parallelism comes at the cost of increased lock synchronization per leaf (however, there is no barrier anymore). As in the FWK approach, the files are relabeled by eliminating the pure children. A larger K value would increase parallelism, and while the number of synchronizations remain about the same, it will reduce the average waiting time on the conditional variable. Like FWK, this scheme requires $2K$ files per attribute, so that each of the K leaves has separate files for each attribute and there is no read/write conflict.

6.4.3 Task Parallelism — The Subtree Algorithm (SUBTREE)

The data parallel approaches target the parallelism available among the different attributes. On the other hand the task parallel approach is based on the parallelism that exists in different sub-trees. Once the attribute lists are partitioned, each child can be processed in parallel. One implementation of this idea would be to initially assign all the processors to the tree root, and recursively partition the processor sets along with the attribute lists. Once a processor gains control of a subtree, it will work only on that portion of the tree. This approach would work fine if we have a full tree. In general, the decision trees are imbalanced and this static partitioning scheme can suffer from large load imbalances. We therefore use a dynamic subtree task parallelism scheme.

The pseudo-code and illustration for the dynamic SUBTREE algorithm is shown in Figure 6.11. To implement dynamic processor assignment to different subtrees, we maintain a queue of currently idle processors, called the *FREE* queue. Initially this queue is empty, and all processors are assigned to the root of the decision tree, and belong to a single group. One processor within the group is made the group master (we chose the processor with the smallest identifier as the master). The master is responsible for partitioning the processor set.

At any given point in the algorithm, there may be multiple processor groups working on distinct subtrees. Each group independently executes the following steps once the BASIC algorithm has been applied to the current subtree level. First, the new subtree leaf frontier is constructed. If there are no children remaining, then each processor inserts itself in the *FREE* queue, ensuring mutually exclusive access via locking. If there is more work to be done, then all processors except the master go to sleep on a conditional variable. The group master checks if there are any new arrivals in the *FREE* queue and grabs all free processors in the queue. This forms the new processor set.

There are three possible cases at this juncture. If there is only one leaf remaining, then all processors are assigned to that leaf. If there is only one processor in the previous group and there is no processor in the *FREE* queue, then it forms a group on its own and works on the current leaf frontier. Lastly, if there are multiple leaves and multiple processors, the group master splits the processor set into two parts, and also splits the leaves into two parts. The two newly formed processor sets become the new groups, and work on the corresponding leaf sets.

Finally, the master wakes up the all the relevant processors—those in the original group and those acquired from the *FREE* queue. Since there are P processors, there

```
SubTree (Processor Group P = {p1, p2, ..., px},
            Leaf Frontier L = {l1, l2, ..., ly})

    apply SIMPLE algorithm on L with P processors
    NewL = {l1, l2, ..., lm}   //new leaf frontier

    if (NewL is empty) then
         put self in FREE queue

    elseif (group master) then
         get FREE processors; NewP = {p1, p2, ..., pn}
         if (only one leaf remaining) then
             SubTree (NewP, l1)
         elseif (only one processor in group) then
             SubTree (p1, NewL)
         else   //multiple leaves and processors
             split NewL into L1 and L2
             split NewP into P1 and P2
             SubTree (P1, L1)
             SubTree (P2, L2)

        wakeup processors in NewP

    else //not the group master
        go to sleep
```

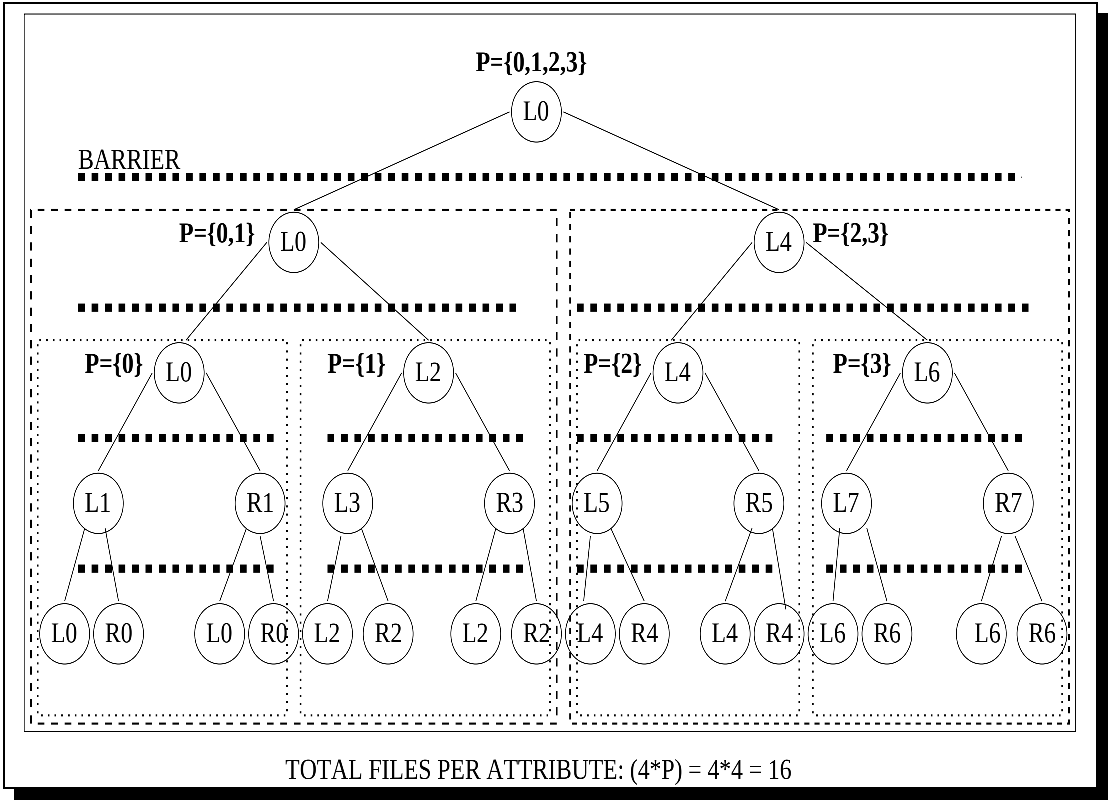


Figure 6.11: The SUBTREE Algorithm

can be at most P groups, and since the attribute files for all of these must be distinct, this scheme requires up to $4P$ files per attribute.

6.4.4 Discussion

We now qualitatively discuss the relative merits of each of the proposed algorithms. The MWK scheme eliminates the hash-probe construction bottleneck of BASIC via task pipelining. Furthermore, it fully exploits the available parallelism via the moving window mechanism, instead of using the fixed window approach of FWK. It also eliminates barrier synchronization completely. However, it introduces a lock synchronization per leaf per level. If the tree is bushy, then the increased synchronization could nullify the other benefits. A feature of MWK and FWK is that they exploit parallelism at a finer grain. The attributes in a K-block may be scheduled dynamically on any processor. This can have the effect of better load balancing compared to the coarser grained BASIC approach where a processor works on all the leaves for a given attribute. While MWK is essentially a data parallel approach, it utilizes some elements of task parallelism in the pipelining of the evaluation and hash probe construction stages.

The SUBTREE approach is also a hybrid approach in that it uses the BASIC scheme within each group. In fact we can also use FWK or MWK as the subroutine. The pros of SUBTREE are that it has only one barrier synchronization per level within each group and it has good processor utilization. As soon as a processor becomes idle it is likely to be grabbed by some active group. Some of the cons are that it is sensitive to the tree structure and may lead to excessive synchronization for the *FREE* queue, due to rapidly changing groups. Another disadvantage is that it requires more memory, because we need a separate hash probe per group.

As described above, our SMP algorithms ostensibly can create a large number of temporary files ($2Kd$ for MWK and $4dP$ for SUBTREE). However, it is possible to have a little more complex design so that lists for different attributes are combined into the same physical file. Such a design will reduce the number of temporary files to $2K$ for MWK and $4P$ for SUBTREE. The essential idea is to associate physical files for writing attribute lists with a processor (rather than with an attribute). In the split phase, a processor now writes all attribute lists to the same two physical files (for the left and right children). Additional bookkeeping data structures keep track of the start and end of different attribute lists in the file. These data structures are shared at the next tree level by all processors to locate the input attribute list for each dynamically assigned attribute. Note that this scheme does not incur additional synchronization overhead because a processor starts processing a new attribute list only after completely processing the one on hand.

6.5 Experimental Results

The primary metric for evaluating classifier performance is *classification accuracy* — the percentage of *test* examples (different from training examples used for building the classifier) that are correctly classified. The other important metrics are time to build the classifier and the *size* of the decision tree. The ideal goal for a decision tree classifier is to produce compact, accurate trees in a short time.

The accuracy and tree size characteristics of our SMP classifier are identical to SLIQ and SPRINT since they consider the same splits and use the same pruning algorithm. SLIQ's accuracy, execution time, and tree size have been compared with other classifiers such as CART [Breiman *et al.*, 1984] and C4 (a predecessor of C4.5 [Quinlan, 1993]). This performance evaluation, available in [Mehta *et al.*, 1996], shows that compared to other classifiers SLIQ achieves comparable or better classification accuracy, but produces small decision trees and has small execution times. We, therefore, focus only on the classifier build time in our performance evaluation.

6.5.1 Experimental Setup

Machine Configuration Experiments were performed on two SMP machines with different configurations shown in Table 6.1. On both machines, each processor is a PowerPC-604 processor running at 112 MHz with a 16 KB instruction cache, a 16 KB data cache, and a 1 MB L2-Cache. These two machines represent two possible scenarios. With Machine A, the amount of memory is insufficient for training data, temporary files, and data structures to fit in memory. Therefore, the data will have to be read from and written to the disk for almost every new level of the tree. Machine B has a large memory relative to the size of the data. Therefore, all the temporary files created during the run are cached in memory. The first case is of greater interest to the database community and we present a detailed set of experiments for this configuration. However, due to the decreasing cost of RAM, the second configuration is also increasingly realizable in practice. We present this case to study the impact of large memories on the performance of our algorithms.

Machine Name	Number Processors	Main Memory	Disk Space Available	Access Type	Operating System
Machine A	4	128 MB	300 MB	local disk	AIX 4.1.4
Machine B	8	1 GB	2 GB	main-memory(cached)	AIX 4.1.5

Table 6.1: Machine Configurations

Datasets An often used classification benchmark is STATLOG[Michie *et al.*, 1994]. Its largest dataset contains about 57,000 records. In our performance study we are interested in evaluating the SMP algorithms on large out-of-core data. We therefore use the synthetic benchmark proposed in [Agrawal *et al.*, 1992] and used in several past studies. Example tuples in this benchmark have both continuous and categorical attributes. The benchmark gives several classification functions of varying complexity to generate synthetic databases. We present results for two of these functions, which are at the two ends of the complexity spectrum. Function 2 is a simple function to learn and results in fairly small decision trees, while Function 7 is the most complex function

Function 2 (F2) - Group A:

$((\textbf{age} < 40) \ \wedge \ (50K \le \textbf{salary} \le 100K)) \vee$
$((40 \le \textbf{age} < 60) \ \wedge \ (75K \le \textbf{salary} \ge 125K)) \vee$
$((\textbf{age} \ge 60) \ \wedge \ (25K \le \textbf{salary} \le 75K))$

Function 7 (F7) - Group A:

$\textbf{disposable} > 0$
where $\textbf{disposable} = (0.67 \times (\textbf{salary} + \textbf{commission}))$
$- (0.2 \times \textbf{loan} \ - \ 20K)$

Figure 6.12: Classification Functions for Synthetic Data

and produces large trees (see Table 6.2). Both these functions divide the database into two classes: Group A and Group B. Figure 6.12 shows the predicates for Group A for each function. For each of Functions 2 and 7, we try 3 different databases: 8 attributes with 1 million records, 32 attributes with 250K records, and 64 attributes with 125K records. The database parameters are shown in Table 6.2. The notation *Fx-Ay-DzK* is used to denote the dataset with function x, y attributes and $z \cdot 1000$ example records. The above choices allow us to investigate the effect of different data characteristics such as number of tuples and number of attributes. Also note that while the initial input size of the ASCII databases is around 60MB, the final input size after the creation of attribute lists is roughly 4 times more, i.e., around 240MB. Since Machine A has only 128MB main memory, the databases will be disk resident.

Dataset						Corresponding Tree	
Dataset Notation	Func.	No. Attr.	No. Tuple	Initial Size	Final Size	No. Levels	Max. No. Leaves/Level
F2-A8-D1000K	F2	8	1000K	61 MB	240MB	4	2
F2-A32-D250K	F2	32	250K	57.3 MB	225MB	4	2
F2-A64-D125K	F2	64	125K	56.6 MB	225MB	4	2
F7-A8-D1000K	F7	8	1000K	61 MB	240MB	60	4662
F7-A32-D250K	F7	32	250K	57.3 MB	225MB	59	802
F7-A64-D125K	F7	64	125K	56.6 MB	225MB	55	384

Table 6.2: Dataset Characteristics

Algorithms Our initial experiments confirmed that MWK was indeed better than BASIC as expected, and that it performs as well or better than FWK. Thus, we will only present the performance of MWK and SUBTREE.

We experimented with window sizes of 2, 4 and 8 for MWK. A larger window size implies more overhead on the file creation and managing related data structures. On the other head, a smaller window size may not have enough parallelism, especially when there are many processors and relatively few attributes. We found for our experiments a window size of 4 to be a good overall choice unless the ratio of the number of attributes to the number of processors is small (less than 2) and in that case we use a window size of 8 (which performs better than a window size of 4 by as much as 9%). In general, a simple rule of thumb for the window size is that if the number of attributes is at least twice the number of processors (which is typically the case for a real world run), a window size of 4 should be chosen. In the rare case when $d/P < 2$, we choose the smallest W such that $W * d/P \geq 8$.

6.5.2 Initial Setup and Sort Time

Table 6.3 shows the uniprocessor time spent in the initial attribute list creation phase (*setup* phase), as wells as the time spent in one-time sort of the attribute lists for the continuous attributes (*sort* phase). The time spent in these two phases as a fraction of total time to build a classifier tree depends on the complexity of the input data for which we are building the classification model. For simple datasets such as F2, it can be significant, whereas it is negligible for complex datasets such as F7.

We have not focussed on parallelizing these phases, concentrating instead on the more challenging build phase. There is much existing research in parallel sorting on SMP machines [Bitton *et al.*, 1984]. The creation of attribute lists can be speeded up by essentially using multiple input streams and merging this phase with the sort phase. In our implementation, the data scan to create attribute lists is sequential, although we write attribute lists in parallel. Attribute lists are sorted in parallel by assigning them to different processors. When we present the speedup graphs in the next section, we will show the speedups separately for the build phase as well as for the total time (including initial setup and sort times). There is obvious scope for improving the speedups for the total time.

Dataset	Setup Time (seconds)	Sort Time (seconds)	Total Time (seconds)	Setup %	Sort %
F2-A8-D1000K	721	633	3597	20.0%	17.6%
F2-A32-D250K	685	598	3584	19.1%	16.6%
F2-A64-D125K	705	626	3665	19.2%	17.1%
F7-A8-D1000K	989	817	23360	4.2%	3.5%
F7-A32-D250K	838	780	24706	3.4%	3.2%
F7-A64-D125K	672	636	22664	3.0%	2.8%

Table 6.3: Sequential Setup and Sorting Times

6.5.3 Parallel Build Performance: Local Disk Access

We consider four main parameters for performance comparison: 1) number of processors, 2) number of attributes, 3) number of example tuples, and 4) classification function (Function 2 or Function 7). We first study the effect of varying these parameters on the

MWK and SUBTREE algorithms on Machine A, which has less main memory than the database size, so that disk I/O is required at each level while building the tree.

Figure 6.13 shows the parallel performance and speedup of the two algorithms as we vary the number of processors for the two classification functions F2 and F7, and using the dataset with eight attributes and one million records (*A8-D1000K*). Figures 6.14 and 6.15 show similar results for datasets *A32-D250K* and *A64-D125K*, respectively. The speedup chart in the bottom right part of each figure shows the speedup of the total time (including setup and sort time), while the speedup chart to the left of it and the two top bar charts show only the build time (excluding setup and sort time).

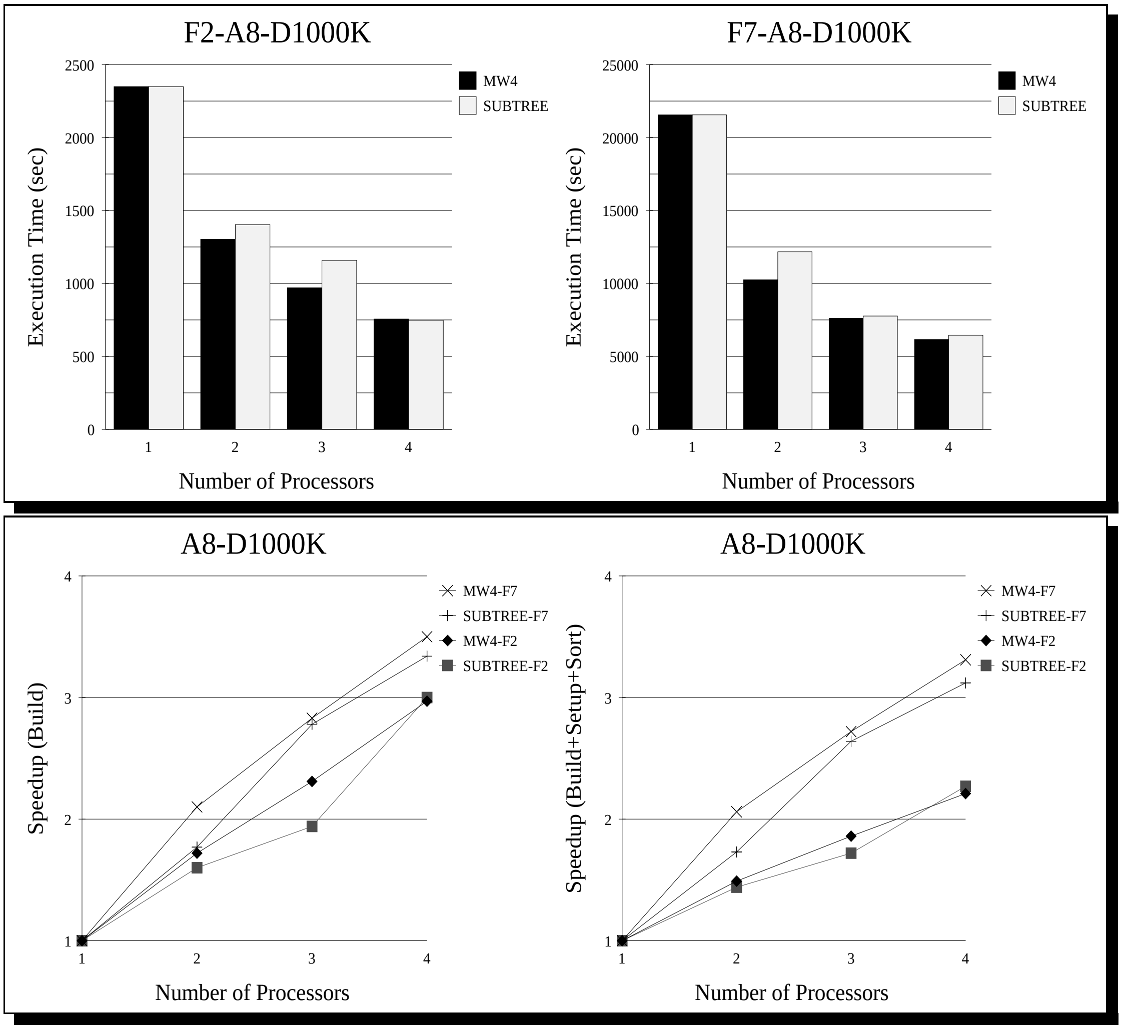


Figure 6.13: Local Disk Access: Functions 2 and 7; 8 Attributes; 1000K Records

Considering the build time only, the speedups for both algorithms on 4 processors range from 2.97 to 3.32 for function F2 and from 3.25 to 3.86 for function F7. For function F7, the speedups of total time for both algorithms on 4 processors range from 3.12 to 3.67. The important observation from these figures is that both algorithms

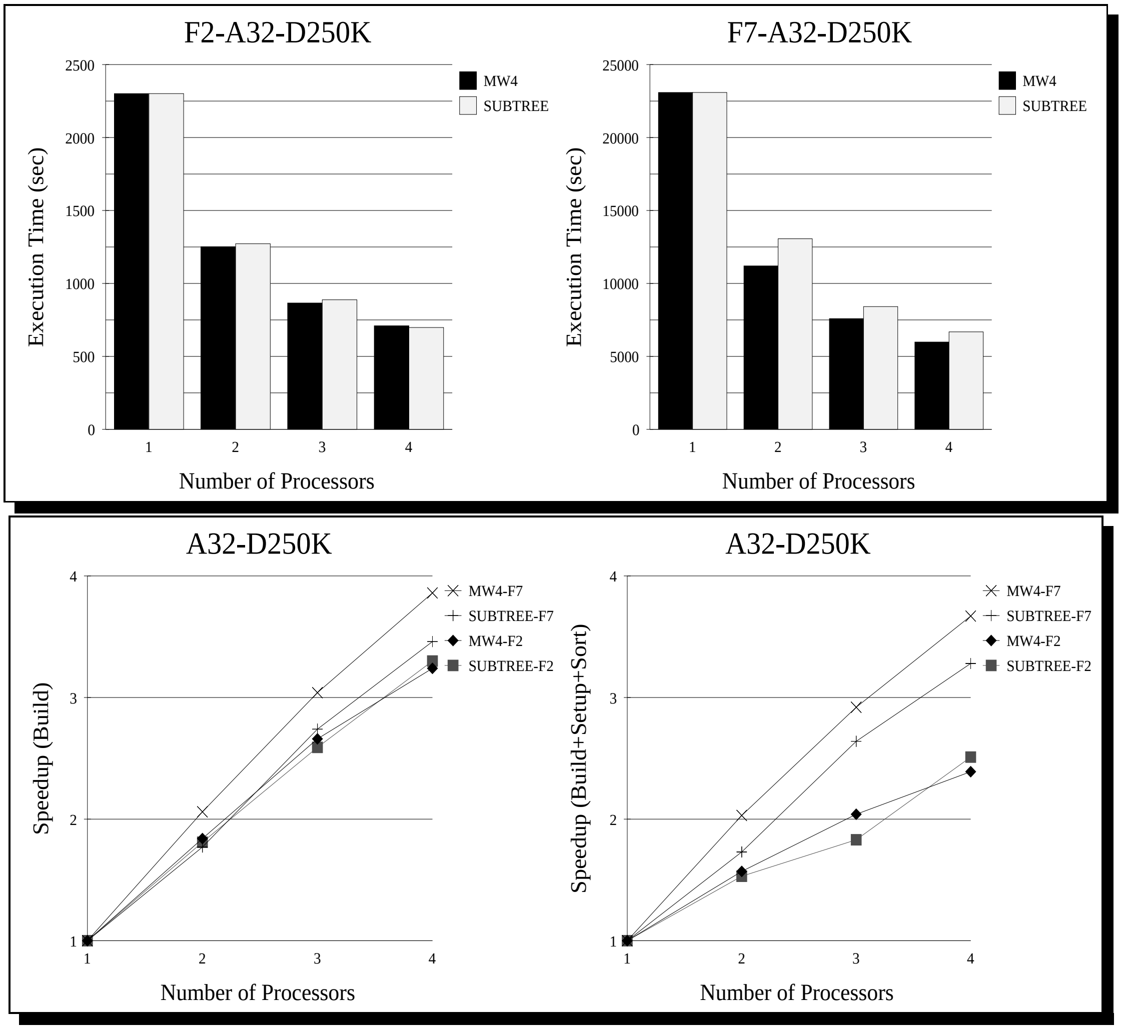


Figure 6.14: Local Disk Access: Functions 2 and 7; 32 Attributes; 250K Records

perform quite well for various datasets. Even the overall speedups are good for complex datasets generated with function F7. As expected, the overall speedups for simple datasets generated by function F2, in which build time is a smaller fraction of total time, are relatively not as good (around 2.2 to 2.5 on 4 processors). These speedups can be improved by parallelizing the setup phase more aggressively.

MWK's performance is mostly comparable or better than SUBTREE. The difference ranges from 8% worse than SUBTREE to 22% better than SUBTREE. Most of the MWK times are within 10% better than SUBTREE.

An observable trend is that having greater number of processors tends to favor SUBTREE. In other words, the advantage of MWK over SUBTREE tends to decrease as the number of processors increases. This is can be seen from figures for both $F2$ and $F7$ by comparing the build times for the two algorithms first with 2 processors, then with 4 processors. This is because after about $\log P$ levels of the tree growth, the only

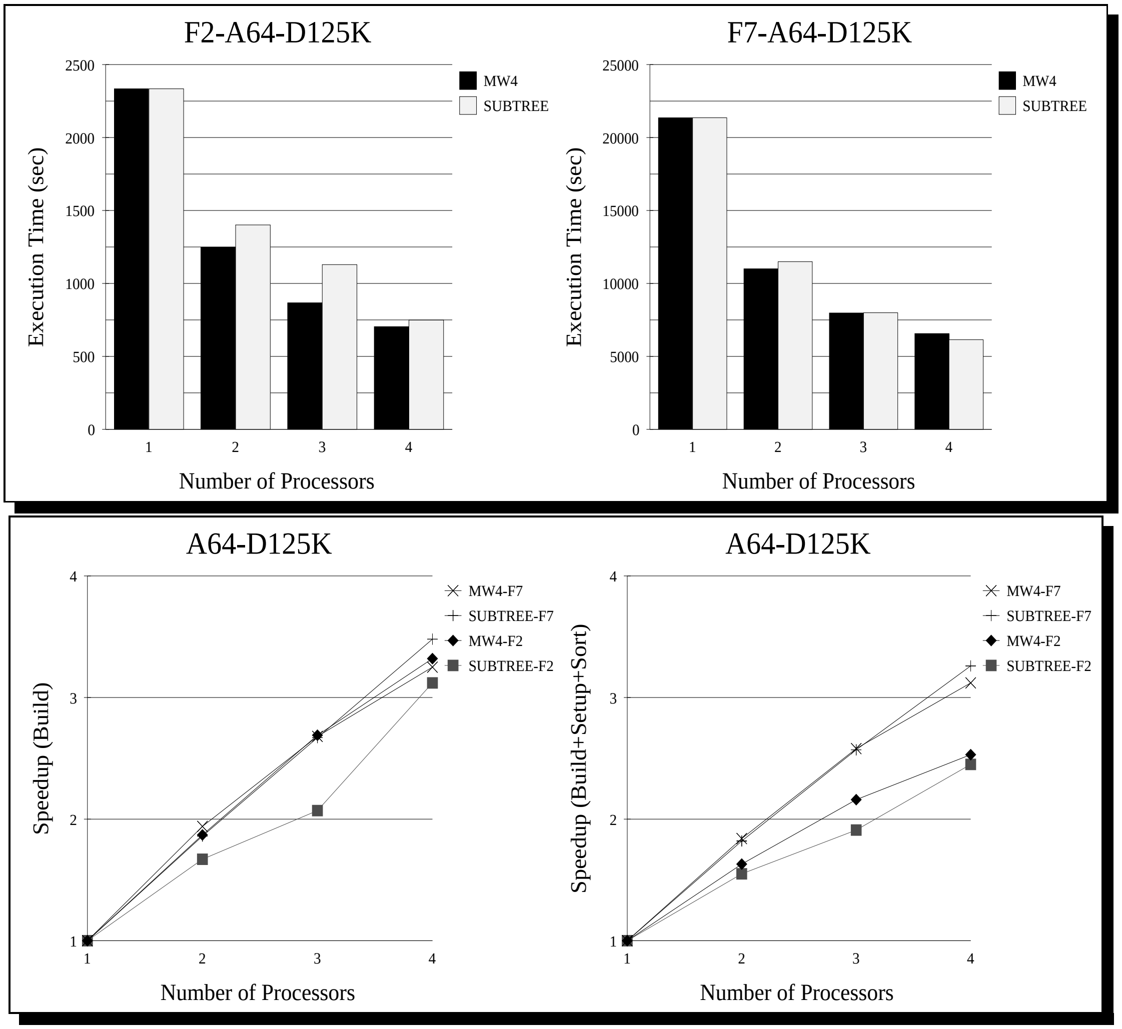


Figure 6.15: Local Disk Access: Functions 2 and 7; 64 Attributes; 125K Records

synchronization overhead for SUBTREE, before any processor becomes free, is that each processor checks the FREE queue once per level. On the other hand, for MWK, there will be relatively more processor synchronization overhead, as the number of processors increases, which includes acquiring attributes, checking on conditional variables, and waiting on barriers. As part of future work we plan to compare these algorithms on larger SMP configurations.

6.5.4 Parallel Build Performance: Main-Memory (Cached) Access

We next compare the parallel performance and speedups of the algorithms on Machine B. This configuration has 1 GB of main-memory available. Thus after the very first access the data will be cached in main-memory, leading to fast access times. Machine B has 8 processors. Figures 6.16, 6.17, and 6.18 show three sets of timing and speedup charts for the *A8-D1000K*, *A32-D250K* and *A64-D125K* datasets, on Functions 2 and 7, and on 1, 2, 4 and 8 processors.

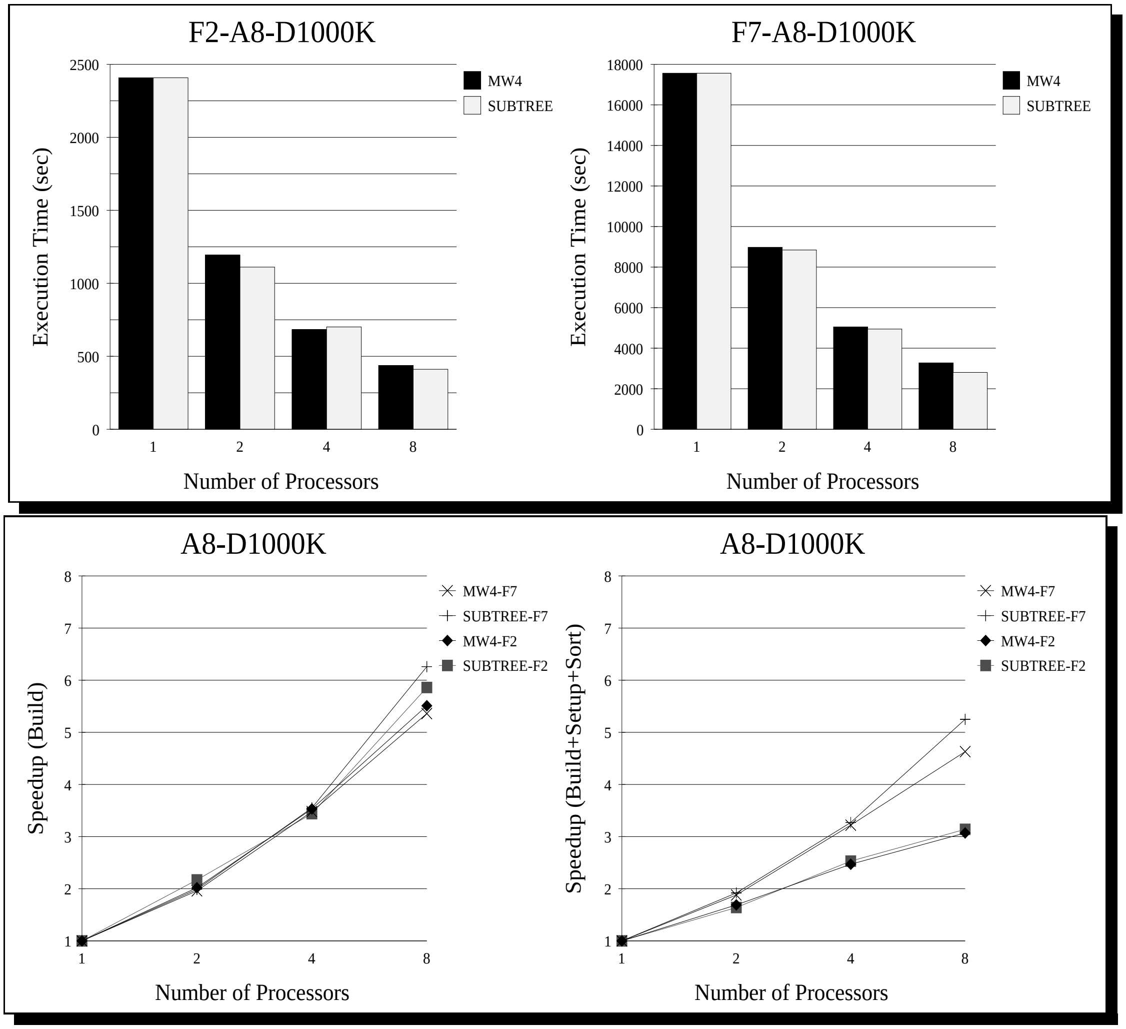


Figure 6.16: Main-memory Access: Functions 2 and 7; 8 Attributes; 1000K Records

Considering the build time only, the speedups for both algorithms on 8 processors range from 5.46 to 6.37 for function F2 and from 5.36 to 6.67 (and at least 6.22 with 32 or 64 attribute) for function F7. For function F7, the speedups of total time for both algorithms on 8 processors range from 4.63 to 5.77 (and at least 5.25 for 32 or 64 attributes). Again, the important observation from these figures is that both algorithms perform very well for various datasets even up to 8 processors. The advantage of MWK over SUBTREE is more visible for the simple function F2. The reason is that F2 generates very small trees with 4 levels and a maximum of 2 leaves in any new leaf frontier. Around 40% of the total time is spent in the root node, where SUBTREE has only one process group. Thus on this dataset SUBTREE is unable to fully exploit the inter-node parallelism successfully. MWK is the winner because it not only overlaps the

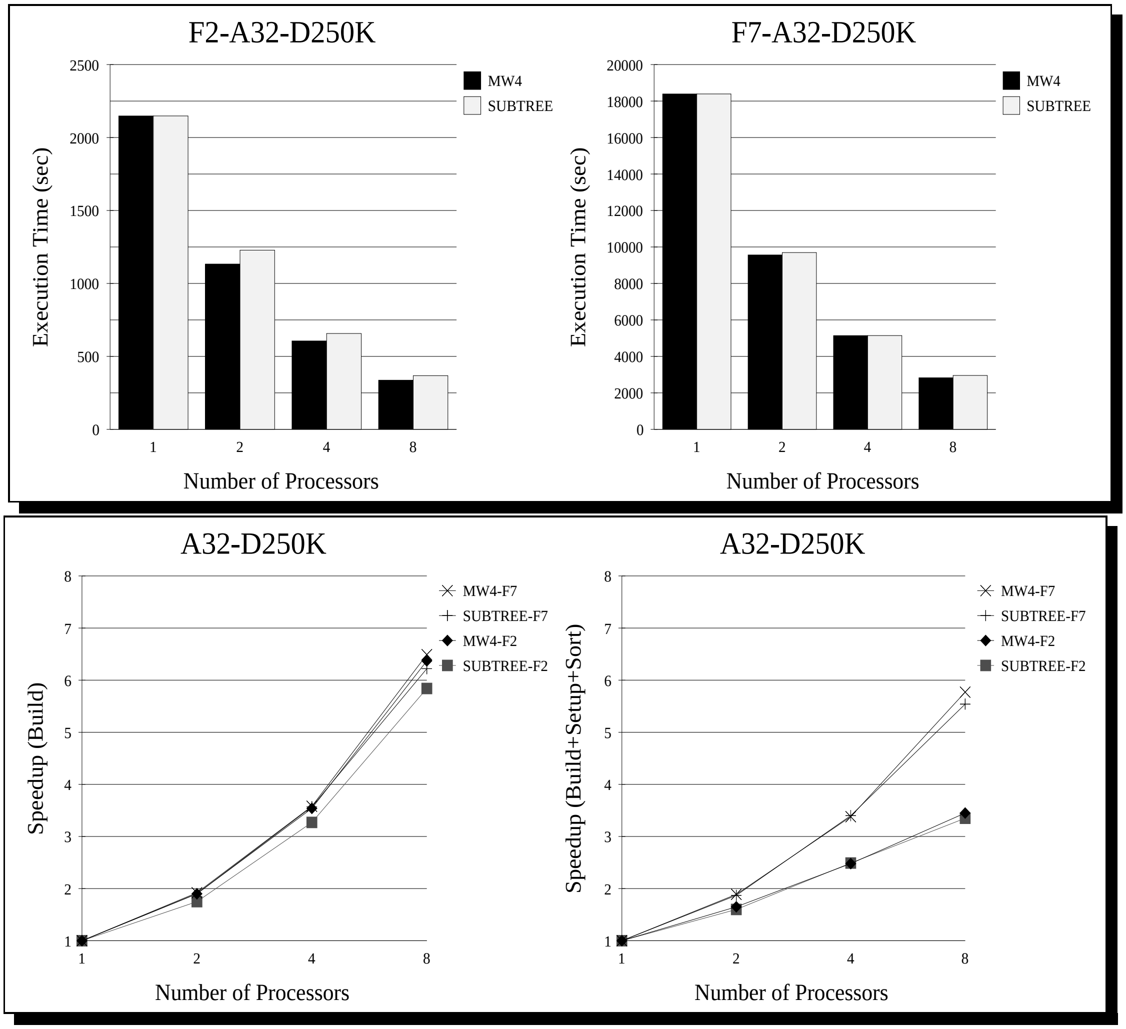


Figure 6.17: Main-memory Access: Functions 2 and 7; 32 Attributes; 250K Records

$\mathcal{E}$ and $\mathcal{W}$ phases, but also manages to reduce the load imbalance.

The overall trends observable from these figures are similar to those for the disk configuration. First, shallow trees (e.g., generated by F2) tend to hurt SUBTREE, 2) Greater number of processors tends to favor SUBTREE more, 3) Having a small number of attributes tends to hurt MWK.

6.6 Conclusions

We presented parallel algorithms for building decision-tree classifiers on SMP systems. The proposed algorithms span the gamut of data and task parallelism. The MWK algorithm uses data parallelism from multiple attributes, but also uses task pipelining to overlap different computing phases within a tree node, thus avoiding potential a sequential bottleneck for the hash-probe construction in the split phase. The MWK al-

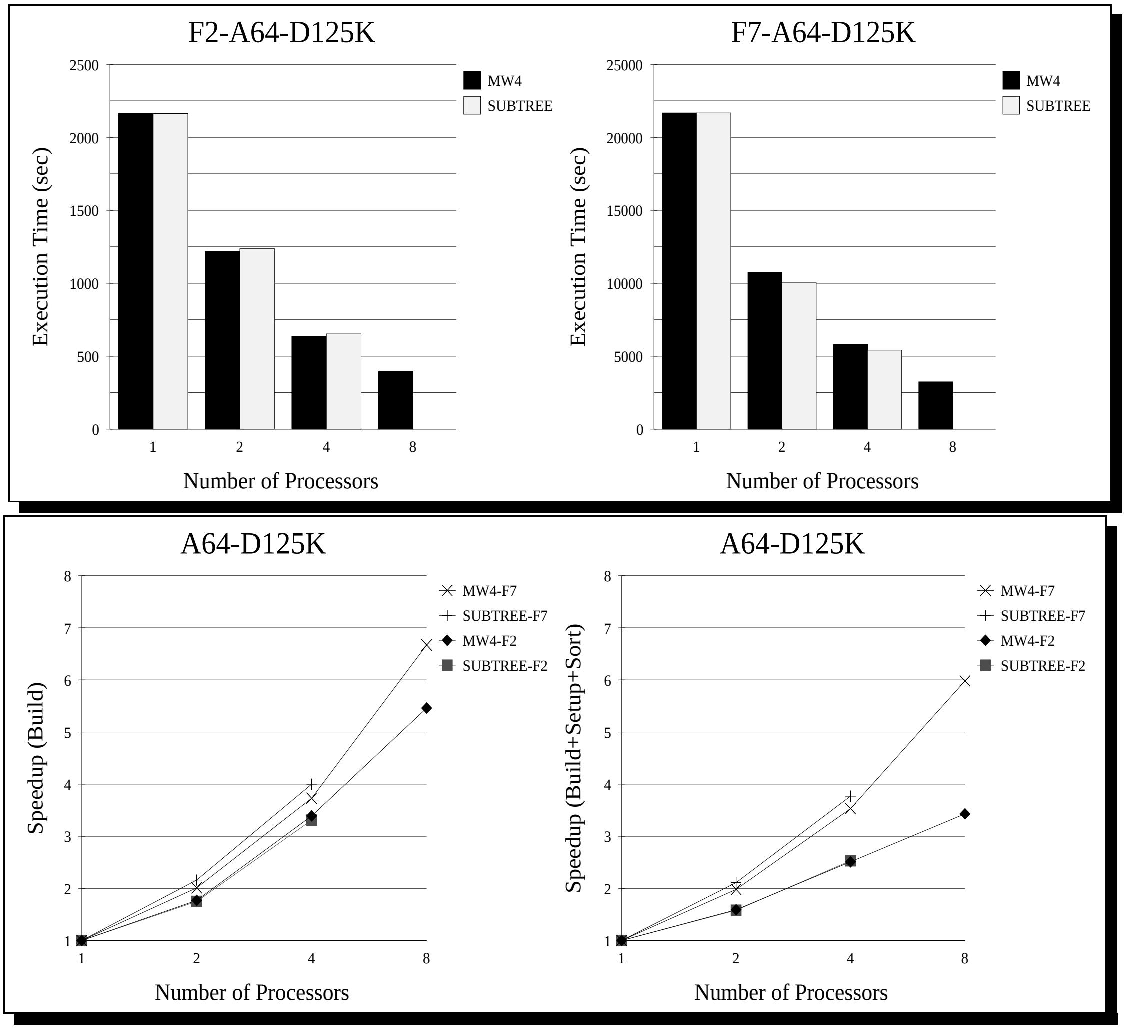


Figure 6.18: Main-memory Access: Functions 2 and 7; 64 Attributes; 125K Records

gorithm employs a conditional variable, instead of a barrier, among leaf nodes to avoid unnecessary processor blocking time. It also exploits dynamic assignment of attribute files to a fixed set of physical files, which maximizes the number of concurrent accesses to disk without file interference. The SUBTREE algorithm uses recursive divide-and-conquer to minimize processor interaction, and assigns free processors dynamically to busy groups to achieve load balancing.

Experiments show that both algorithms achieve good speedups in building the classifier on a 4-processor SMP with disk configuration and on an 8-processor SMP with memory configuration, for various numbers of attributes, various numbers of example tuples of input databases, and various complexities of data models. The performance of both algorithms are comparable, but MWK has a slight edge on the SMP configurations we looked at. These experiments demonstrate that the important data mining task of classification can be effectively parallelized on SMP machines.

7 Summary and Future Work

In this chapter we will summarize the research contributions of this thesis, and point out directions for future work.

7.1 Thesis Summary

The main contributions of this thesis are: 1) We study three key rule discovery techniques that include Association Rules, Sequence Discovery, and Decision Tree Classification. 2) Our algorithms scale to large disk-resident databases 3) Our techniques are based on a sound lattice-theoretic framework. 4) We experimentally evaluate our approach on major parallel platforms which include shared-memory multiprocessors, and hierarchical clusters of SMP workstations. We now present a summary of each chapter in the thesis.

7.1.1 Mining Association Rules

We began this thesis by presenting new algorithms for fast mining of association rules. The main contributions of this work are:

1. *Combinatorial Decomposition*: We used lattice-theoretic combinatorial properties to decompose the original search space into smaller pieces that can be solved independently. Two techniques were presented: prefix-based and maximal-clique-based decomposition. The prefix-based approach returns coarser partitions, while the maximal-clique-based returns a more refined set, but at a greater computational cost.

2. *Efficient Search Techniques*: Once the problem is partitioned, each sub-problem is solved for frequent patterns using one of the three lattice search techniques: bottom-up search, top-down search and hybrid search. The bottom-up search enumerates all the frequent itemsets. The top-down search lists only the maximal ones. The hybrid technique enumerates a few long maximal frequent patterns and some non-maximal ones, and it was found to work best in practice.

3. *Efficient Counting*: The vertical database format using tid-lists enables us to compute itemset support via simple intersection operations. Coupled with the problem decomposition and efficient search methods, this format allows us to enumerate all patterns in usually a single database scan after a pre-processing step or in three scans without the preprocessing.

Six new algorithms were proposed and tested: *Eclat*, *MaxEclat*, *Clique*, *MaxClique*, *Top-Down*, and *AprClique*. We showed experimentally that the best new algorithm *MaxClique* outperforms the best previous algorithm by more than an order of magnitude. Future work will concentrate on the development of new algorithms for mining generalized and quantitative association rules [Srikant and Agrawal, 1995; Srikant and Agrawal, 1996a], and on the direct implementation of these algorithms on DBMS systems using SQL.

7.1.2 Parallel Association Mining

The parallel formulation is based on the sequential association mining algorithms using the techniques proposed above. The main features of the parallel algorithms are:

1. *Set-up Phase*: We decouple processors up-front. The independent sub-problems are scheduled among the available processors, and the database is selectively replicated, so that all information needed for computing the associations is local to each processor.

2. *Asynchronous Phase*: Since each sub-problem is independent, and all data is locally available, each processor can enumerate the frequent itemsets asynchronously. There is no communication or synchronization beyond the set-up phase.

3. *Good Locality and Memory Usage*: The vertical tid-lists enable us to compute support via simple intersections, which involve only a linear scan of two lists. This results in very good data locality (unlike previous approaches that use hash trees for computing support, which have poor locality). Furthermore, the aggregate memory of the whole system is used effectively, since each processor solves disjoint sub-problems (unlike previous algorithms that replicate the hash trees on all processors). Finally, usually only two database scans are needed to compute all frequent itemsets: one pass for replicating the database, and another for enumerating the patterns.

Four new algorithms, *Par-Eclat*, *Par-MaxEclat*, *Par-Clique*, and *Par-MaxClique*, were proposed, and the best new algorithm *Par-MaxClique* was shown to outperform earlier approaches by upto an order of magnitude. Future work will involve applying these techniques for developing new algorithms on SMP systems, and for implementing new parallel algorithms for the task of mining generalized and quantitative associations rules.

7.1.3 Theoretical Foundations of Association Rules

We presented some complexity results of mining associations based on the connection between frequent itemsets and bipartite cliques. We also placed association rule mining within the lattice-theoretic framework of formal concept analysis. We showed that all frequent itemsets are uniquely determined by the set of frequent concepts. We then tackled the problem of constructing a rule generating set and a base, a minimal rule set, from which all the other association rules can be inferred. We showed that while there exists a characterization of a base for rules with 100% confidence, the problem of constructing a base for all associations is still open, which provides an opportunity for future research. Nevertheless, we showed how to construct a generating set for all associations rules, which is a good approximation for a base, and which can drastically reduce the number of rules presented to the user.

7.1.4 Mining Sequence Rules

We presented SPADE, a new algorithm for fast mining of sequential patterns in large databases. It has the same features as the association mining algorithm, even though the search space is larger and more complex, and the intersection is also more complicated. The new algorithm decomposes the original problem into smaller sub-problems using equivalence classes on frequent sequences. Not only can each equivalence class be solved independently, but it is also very likely that it can be processed in main-memory. The new approach makes only three database scans and uses only simple intersection operations. An extensive set of experiments was conducted to show that SPADE outperforms the best previous algorithm by a factor of two, and by an order of magnitude with a preprocessing step. This work opens several research opportunities, which we plan to address in the future:

1. Direct integration with a DBMS system.
2. Discovery of *quantitative* sequences – where the quantity of items bought is also considered.
3. Efficient solutions to the problem of *generalized* sequences using the SPADE approach – introducing minimum and maximum time gap constraints on sequence elements, relaxing the definition of a transaction to encompass all transaction within a sliding window, and imposing a taxonomy on the items.

7.1.5 Mining Classification Rules

We developed fast scalable decision-tree-based classification algorithms targeting shared-memory systems, the first such study. The algorithms were based on the sequential SPRINT classifier, and spanned the gamut of data and task parallelism. The data parallelism was based on attribute scheduling among processors. This was extended with task pipelining and dynamic load balancing to yield more complex schemes. The task parallel approach used dynamic subtree partitioning among processors. Two main algorithms were presented: MWK and SUBTREE. These algorithms were evaluated on two SMP configurations: one in which data is too large to fit in memory and must be paged from a local disk as needed and the other in which memory was sufficiently large to hold the whole input data and all temporary files. Experiments confirmed that the new algorithms achieved excellent speedup and sizeup, making them ideal for mining large datasets. Future work will study the scaleup of these algorithms on larger SMP configurations.

7.2 Future Work

Data Mining or KDD refers to the overall process of discovering new, useful, and understandable patterns in data. Developing fast algorithms is just one of the steps in the process. The other steps include data selection, cleaning and preprocessing, transformation, data-mining task and algorithm selection, and finally post-processing. This KDD process tends to be highly interactive and iterative. Future research needs to target the interaction of the overall KDD process and parallel computing, in addition to developing faster parallel solutions for the core data mining tasks. Some of the problems that need to be addressed are:

- *Scalability:* Databases will continue to increase in size, in both the number of records and the number of attributes. Parallel processing is ideally suited for addressing issues of scalability. The ultimate goal is to handle giga/tera-bytes of data efficiently.

- *Incremental Techniques:* In many domains the data changes over time. Old extracted knowledge may need to be updated or may even become invalid. Parallel incremental algorithms can provide fast modification, deletion or augmentation of discovered knowledge.

- *Interaction:* The KDD process is highly interactive. The human is involved in almost all the steps. For example, the user is heavily involved in the initial data understanding, selection, cleaning, and transformation phases. These steps in fact consume more time than data mining *per se*. Moreover, the output of data mining may itself be large, requiring additional refinement or generalization to produce understandable patterns. Parallel methods can be successful in providing the desired rapid response in all of the above steps.

- *Data Type and Locality:* Distributed collaboration is becoming increasingly common, with need for data analysis at widely dispersed sites. Furthermore, the rapid growth of resources on the Internet, which is inherently distributed in nature, and has data in a variety of non-standard and unstructured formats, poses new challenges in KDD. Techniques from parallel and distributed computing will lie at the heart of any proposed solutions.

- *Parallel DBMS Integration:* The various steps of the KDD process, along with the core data mining methods, need to be integrated with a DBMS to provide common representation, storage, and retrieval, to build general purpose KDD systems. Moreover, enormous gains are possible when combined with parallel DBMS or parallel file servers.

- *Applications:* KDD is ultimately motivated by the need to analyze data from a variety of practical applications, be it in business domains such as finance, marketing, telecommunications and manufacturing, or in scientific fields such as biology, geology, astronomy and medicine. Future research must identify new application domains that can benefit from data mining. This will lead to the refinement of existing techniques, and also to the development of new methods where current tools are inadequate.

In conclusion, the phenomenal growth of data has spurred the need for KDD. While the existing techniques are usually *ad hoc*, as the field matures, solutions are being proposed for the problems outlined above along with other crucial problems of incorporation of prior knowledge, avoiding over-fitting, handling missing data, adding visualization, and improving understandability. Furthermore, as the data continue to increase in complexity (which includes size, type, and location of data), parallel computing will be essential in delivering fast interactive solutions for the overall KDD process. This thesis is a step in this direction.

Bibliography

[Aggarwal and Yu, 1998] C. Aggarwal and P. Yu, "Online Generation of Association Rules," In *14th Intl. Conf. on Data Engineering*, February 1998.

[Agrawal *et al.*, 1992] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami, "An Interval Classifier for Database Mining Applications," In *18th VLDB Conference*, August 1992.

[Agrawal *et al.*, 1993a] R. Agrawal, T. Imielinski, and A. Swami, "Database Mining: A Performance Perspective," *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993.

[Agrawal *et al.*, 1993b] 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.

[Agrawal *et al.*, 1996] 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.

[Agrawal and Shafer, 1996] R. Agrawal and J. Shafer, "Parallel Mining of Association Rules," *IEEE Trans. on Knowledge and Data Engg.*, 8(6):962–969, December 1996.

[Agrawal and Srikant, 1994] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," In *20th VLDB Conference*, September 1994.

[Agrawal and Srikant, 1995] R. Agrawal and R. Srikant, "Mining Sequential Patterns," In *11th Intl. Conf. on Data Engg.*, 1995.

[Agrawal *et al.*, 1993c] Rakesh Agrawal, Christos Faloutsos, and Arun Swami, "Efficient Similarity Search in Sequence Databases," In *Proc. of the Fourth Int'l Conference on Foundations of Data Organization and Algorithms*, October 1993.

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

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

[Bitton *et al.*, 1984] Dina Bitton, David DeWitt, David K. Hsiao, and Jaishankar Menon, “A Taxonomy of Parallel Sorting,” *ACM Computing Surveys*, 16(3):287–318, September 1984.

[Breiman *et al.*, 1984] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*, Wadsworth, Belmont, 1984.

[Brin *et al.*, 1997] 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.

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

[Carpineto and Romano, 1993] C. Carpineto and G. Romano, “GALOIS: An order-theoretic approach to conceptual clustering,” In *10th Intl. Conf. on Machine Learning*, June 1993.

[Carpineto and Romano, 1996] C. Carpineto and G. Romano, “A lattice conceptual clustering system and its application to browsing retrieval,” *Machine Learning*, 24:95–112, 1996.

[Catlett, 1991] Jason Catlett, *Megainduction: Machine Learning on Very Large Databases*, PhD thesis, University of Sydney, 1991.

[Chan and Stolfo, 1993a] Philip K. Chan and Salvatore J. Stolfo, “Experiments on Multistrategy Learning by Meta-Learning,” In *Proc. Second Intl. Conference on Info. and Knowledge Mgmt.*, pages 314–323, 1993.

[Chan and Stolfo, 1993b] Philip K. Chan and Salvatore J. Stolfo, “Meta-learning for Multistrategy and Parallel Learning,” In *Proc. Second Intl. Workshop on Multistrategy Learning*, pages 150–165, 1993.

[Cheeseman *et al.*, 1988] P. Cheeseman, James Kelly, Matthew Self, *et al.*, “AutoClass: A Bayesian Classification System,” In *5th Int'l Conference on Machine Learning*. Morgan Kaufman, June 1988.

[Cheung *et al.*, 1996a] 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.

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

[Cheung *et al.*, 1996b] D. Cheung, V. Ng, A. Fu, and Y. Fu, “Efficient Mining of Association Rules in Distributed Databases,” In *IEEE Trans. on Knowledge and Data Engg.*, pages 8(6):911–922, 1996.

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

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

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

[Faloutsos *et al.*, 1994] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos, "Fast Subsequence Matching in Time-Series Databases," In *Proc. of the ACM SIGMOD Conference on Management of Data*, May 1994.

[Fayyad *et al.*, 1996a] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "From Data Mining to Knowledge Discovery: An Overview," In *Advances in Knowledge Discovery and Data Mining.* AAAI Press, Menlo Park, CA, 1996.

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

[Fifield, 1992] D. J. Fifield, "Distributed Tree Construction from Large Data-sets," Bachelor's Honours Thesis, Australian National University, 1992.

[Fisher, 1987] Douglas H. Fisher, "Knowledge Acquisition Via Incremental Conceptual Clustering," *Machine Learning*, 2(2), 1987.

[Ganter, 1987] B. Ganter, "Algorithmen zur formalen begriffsanalyse," *Beiträge zur Begriffsanalyse (Ganter, Wille, Wolff, eds.)*, Wissenschaft-Verlag, 1987.

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

[Gillett, 1996] R. Gillett, "Memory Channel: An Optimized Cluster Interconnect," In *IEEE Micro, 16(2)*, February 1996.

[Godin *et al.*, 1991] R. Godin, R. Missaoui, and H. Alaoui, "Incremental concept formation algorithms based on Galois (concept) lattices," *Computational Intelligence*, 11(2):246–267, 1991.

[Goldberg, 1989] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Morgan Kaufmann, 1989.

[Guenoche, 1990] A. Guenoche, "Construction du treillis de Galois d'une relation binaire," *Math. Sci. Hum.*, 109:41–53, 1990.

[Guigues and Duquenne, 1986] J. L. Guigues and V. Duquenne, "Familles minimales d'implications informatives resultant d'un tableau de donnees binaires," *Math. Sci. hum.*, 24(95):5–18, 1986.

[Gunopulos *et al.*, 1997a] 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.

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

[Han *et al.*, 1997] E-H. Han, G. Karypis, and V. Kumar, "Scalable Parallel Data Mining for Association Rules," In *ACM SIGMOD Conf. Management of Data*, May 1997.

[Harary, 1969] F. Harary, *Graph Theory*, Addison-Wesley Pub. Co., 1969.

[Hatonen *et al.*, 1996] K. Hatonen, M. Klemettinen, H. Mannila, P. Ronkainen, and H. Toivonen, "Knowledge Discovery from Telecommunication Network Alarm Databases," In *12th Intl. Conf. Data Engineering*, February 1996.

[Holsheimer *et al.*, 1995] 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.

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

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

[IBMb] IBM, *http://www.rs6000.ibm.com/hardware/largescale/index.html, IBM RS/6000 SP System*, IBM Corp.

[Jain and Dubes, 1988] A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data*, Prentice Hall, 1988.

[James, 1985] M. James, *Classificaton Algorithms*, Wiley, 1985.

[Kashiwabara *et al.*, 1992] 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.

[Kuznetsov, 1989] 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.

[Kuznetsov, 1993] S. O. Kuznetsov, "Fast algorithm for construction of all intersections of the objects from a finite semi-lattice," *Nauchn. Tekh. Inf., Ser. 2 (Automatic Document. Math. Linguist.)*, 9:11–21, 1993.

[Lewis and Berg, 1996] Bill Lewis and Daniel J. Berg, *Threads Primer*, Prentice Hall, New Jersey, 1996.

[Lin and Kedem, 1998] 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.

[Lin and Dunham, 1998] J-L. Lin and M. H. Dunham, "Mining Association Rules: Anti-Skew Algorithms," In *14th Intl. Conf. on Data Engineering*, February 1998.

[Lippmann, 1987] R. Lippmann, "An Introduction to Computing with Neural Nets," *IEEE ASSP Magazine*, 4(22), April 1987.

[Luxenburger, 1991] M. Luxenburger, "Implications partielles dans un contexte," *Math. Inf. Sci. hum.*, 29(113):35–55, 1991.

[Maier, 1983] D. Maier, *The Theory of Relational Databases*, Computer Science Press, 1983.

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

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

[Markatos and LeBlanc, 1994] E.P. Markatos and T.J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, 5(4), April 1994.

[Mehta *et al.*, 1996] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen, "SLIQ: A Fast Scalable Classifier for Data Mining," In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996.

[Michalski and Stepp, 1983] Ryszard S. Michalski and Robert E. Stepp, "Learning from Observation: Conceptual Clustering," In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, volume I, pages 331–363. Morgan Kaufmann, 1983.

[Michie *et al.*, 1994] D. Michie, D. J. Spiegelhalter, and C. C. Taylor, *Machine Learning, Neural and Statistical Classification*, Ellis Horwood, 1994.

[Mueller, 1995] 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.

[Mulligan and Corneil, 1972] G. D. Mulligan and D. G. Corneil, "Corrections to Bierstone's Algorithm for Generating Cliques," *J. of the ACM*, 19(2):244–247, 1972.

[NASA, 1992] NASA, *Introduction to IND Version 2.1*, NASA Ames Research Center, GA23-2475-02 edition, 1992.

[Oates *et al.*, 1997] T. Oates, M. D. Schmill, D. Jensen, and P. R. Cohen, "A family of algorithms for finding temporal structure in data," In *6th Intl. Workshop on AI and Statistics*, March 1997.

[Papadimitriou, 1994] C. H. Papadimitriou, *Computational Complexity*, Addison-Wesley Pub. Co., 1994.

[Park *et al.*, 1995a] 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.

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

[Parthasarathy *et al.*, 1998] 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.

[Polychronopoulos and Kuck, 1987] C. D. Polychronopoulos and D. J. Kuck, "Guided Self-Scheduling: a Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. on Computers*, C-36(12), December 1987.

[Quinlan, 1979] J. R. Quinlan, "Induction over large databases," Technical Report STAN-CS-739, Stanford University, 1979.

[Quinlan, 1986] J. Ross Quinlan, "Induction of Decision Trees," *Machine Learning*, 1:81–106, 1986.

[Quinlan, 1993] J. Ross Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufman, 1993.

[Rissanen, 1989] J. Rissanen, *Stochastic Complexity in Statistical Inquiry*, World Scientific Publ. Co., 1989.

[Rota, 1964] G.-C. Rota, "On the foundations of combinatorial theory: I. theory of Möbius functions," *Z. Wahrscheinlichkeitstheorie*, 2:340–368, 1964.

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

[Shafer *et al.*, 1996] J. Shafer, R. Agrawal, and M. Mehta, "SPRINT: A Scalable Parallel Classifier for Data Mining," In *22nd VLDB Conference*, March 1996.

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

[Simoudis, 1996] E. Simoudis, "Reality Check for Data Mining," In *IEEE Expert: Intelligent Systems and Their Applications*, pages 11(5):26–33, October 1996.

[Srikant and Agrawal, 1995] R. Srikant and R. Agrawal, "Mining Generalized Association Rules," In *21st VLDB Conf.*, 1995.

[Srikant and Agrawal, 1996a] R. Srikant and R. Agrawal, "Mining Quantitative Association Rules in Large Relational Tables," In *ACM SIGMOD Conf. Management of Data*, June 1996.

[Srikant and Agrawal, 1996b] R. Srikant and R. Agrawal, "Mining Sequential Patterns: Generalizations and performance improvements," In *5th Intl. Conf. Extending Database Technology*, March 1996.

[Tang and Yew, 1986] P. Tang and P.-C. Yew, "Processor Self-Scheduling for Multiple Nested Parallel Loops," In *International Conference On Parallel Processing*, August 1986.

[Toivonen, 1996] H. Toivonen, "Sampling Large Databases for Association Rules," In *22nd VLDB Conf.*, 1996.

[Toivonen *et al.*, 1995] H. Toivonen, M. Klemettinen, P. Ronkainen, K. Hätönen, and H. Mannila, "Pruning and grouping discovered association rules," In *MLnet Wkshp. on Statistics, Machine Learning, and Discovery in Databases*, April 1995.

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

[Valiant, 1979] L. G. Valiant, "The complexity of computing the permanent," *Theoretical Computer Science*, 8:189–201, 1979.

[Weiss and Kulikowski, 1991] Sholom M. Weiss and Casimir A. Kulikowski, *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems*, Morgan Kaufman, 1991.

[Wild, 1991] M. Wild, "Implicational bases for finite closure systems," *Arbeitstagung, Begriffsanalyse und Künstliche Intelligenz (W. Lex, ed.)*, pages 147–169, 1991.

[Wille, 1982] R. Wille, "Restructuring lattice theory: an approach based on hierarchies of concepts," *Ordered Sets (I. Rival, ed.)*, pages 445–470, Reidel, Dordrecht-boston, 1982.

[Wirth and Catlett, 1988] J. Wirth and J. Catlett, "Experiments on the Costs and Benefits of Windowing in ID3," In *5th Int'l Conference on Machine Learning*, 1988.

[Yen and Chen, 1996] 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.

[Zaki, 1998] M. J. Zaki, "Efficient Enumeration of Frequent Sequences," In *7th Intl. Conf. on Information and Knowledge Management*, November 1998.

[Zaki *et al.*, 1998a] M. J. Zaki, C.-T. Ho, and R. Agrawal, "Parallel Classification on SMP Systems," In *1st Workshop on High Performance Data Mining*, March 1998.

[Zaki *et al.*, 1999] M. J. Zaki, C.-T. Ho, and R. Agrawal, "Parallel Classification for Data Mining on Shared-Memory Multiprocessors," In *15th IEEE Intl. Conf. on Data Engineering*, March 1999.

[Zaki *et al.*, 1998b] M. J. Zaki, N. Lesh, and M. Ogihara, "PLANMINE: Sequence mining for plan failures," In *4th Intl. Conf. Knowledge Discovery and Data Mining*, August 1998.

[Zaki and Ogihara, 1998] M. J. Zaki and M. Ogihara, "Theoretical Foundations of Association Rules," In *3rd ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, June 1998.

[Zaki *et al.*, 1996] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li, "Parallel Data Mining for Association Rules on Shared-memory Multi-processors," In *Supercomputing'96*, November 1996.

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

[Zaki *et al.*, 1997b] 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.

[Zaki *et al.*, 1997c] 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.

[Zaki *et al.*, 1997d] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New algorithms for fast discovery of association rules," Technical Report URCS TR 651, University of Rochester, April 1997.

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