CS-TR-3515 August 1995

Fast Sequential and Parallel Algorithms for Association Rule Mining: A Comparison[1]

Andreas Mueller

Department of Computer Science
University of Maryland-College Park
College Park, MD 20742

Abstract

The field of knowledge discovery in databases, or "Data Mining", has received increasing attention during recent years as large organizations have begun to realize the potential value of the information that is stored implicitly in their databases. One specific data mining task is the mining of Association Rules, particularly from retail data. The task is to determine patterns (or rules) that characterize the shopping behavior of customers from a large database of previous consumer transactions. The rules can then be used to focus marketing efforts such as product placement and sales promotions.

Because early algorithms required an unpredictably large number of IO operations, reducing IO cost has been the primary target of the algorithms presented in the literature. One of the most recent proposed algorithms, called PARTITION, uses a new TID-list data representation and a new partitioning technique. The partitioning technique reduces IO cost to a constant amount by processing one database portion at a time in memory. We implemented an algorithm called SPTID that incorporates both TID-lists and partitioning to study their benefits. For comparison, a non-partitioning algorithm called SEAR, which is based on a new prefix-tree data structure, is used. Our experiments with SPTID and SEAR indicate that TID-lists have inherent inefficiencies; furthermore, because all of the algorithms tested tend to be CPU-bound, trading CPU-overhead against I/O operations by partitioning did not lead to better performance.

In order to scale mining algorithms to the huge databases (e.g., multiple Terabytes) that large organizations will manage in the near future, we implemented parallel versions of SEAR and SPEAR (its partitioned counterpart). The performance results show that, while both algorithms parallelize easily and obtain good speedup and scale-up results, the parallel SEAR version performs better than parallel SPEAR, despite the fact that it uses more communication.

[1]This research was funded in part under U.S.Air Force Grant #F19628-94-C-0057 (Syracuse University Subcontract 3531427) and Caltech Subcontract #9503

Table of Contents

Chapter 1

Introduction

While data analysis has long been the object of statistics and several fields of computer science including machine learning, expert systems and knowledge acquisition for expert systems in particular, these techniques have only in recent years been applied to large databases to unveil the wealth of knowledge that is buried there. The growing interest from businesses in *data mining* or *knowledge discovery in data bases* (KDD), as the field is also called, and the appearance of data mining tools in the marketplace show the need for means to handle today's very large and ever growing databases. W.Frawley defined the data mining problem as the " nontrivial extraction of implicit, previously unknown and potentially useful information from data"[15]. Unfortunately, the characteristics of databases, above all their sheer size, often make it impossible to simply apply the tools developed for problems in the other areas mentioned above. New algorithms must be designed to take these peculiarities into account and to provide users with effective and convenient ways to discover the knowledge they need.

Association rule mining (ARM) is one problem in KDD that has received considerable attention during the past year, as demonstrated by the large number of new publications [19, 32, 34, 37]. As with many other problems in KDD, a certain type of database and a special application has lead to specialized algorithms for this problem. In this case, retail data[1], a database of consumer transactions in large retail stores, has to be searched for rules that characterize common consumer behavior. Discovered rules may reveal, for example, that there is a 75% percent chance that people also buy spaghetti sauce if they buy pasta and ground meat. While this example seems rather intuitive, other rules like it may not be so obvious and a means to find them is required. Furthermore, mining tools enable users to quantify their assumptions and refute or confirm them through data mining queries. The relevance of these rules to applications like marketing, product planning, store layout, advertisement and sales promotion can easily be conceived. However, the ARM algorithms can be applied to a much broader scope of problems such as failure correlation in complex systems or even the analysis of medical data. Association rules are introduced more formally in Chapter 2.

Unfortunately, the number of possible association rules grows exponentially with the number of items considered. For 1000 items, for example, more than 2^{1000} rules have to be considered in a naive approach. Several algorithms have been proposed in the literature to make this search more effective, i.e. Apriori [4] and PARTITION[34]. Algorithms differ mainly with respect to the internal data representations used for intermediate results and the IO cost and CPU-overhead they incur. Like all algorithms before it, Apriori needs to scan the entire database several times. To reduce the number of IO operations, PARTITION implements a new divide-and-conquer partitioning strategy. It reads and processes the database one partition after the other and only one second scan is necessary to join the partial results. The second novelty in PARTITION is the use of the new TID-list structure to store a partition in memory (as opposed to the item-list representation used by Apriori). [34] report that PARTITION is superior to Apriori because it needs less IO and less CPU-

[1] sometimes also referred to as basket data

overhead due to these two new features.

In this report, we investigate the effect of data structures and the partitioning concept on the performance of the algorithms. In particular, we present the Sequential Efficient Association Rules algorithm (SEAR) which employs a new *prefix tree* data structure and includes an optimization we call *pass bundling* that has not been investigated in the literature. SEAR, which is based on the item-list representation, is compared to SPTID (short for "Sequential Partitioning with TID-lists") which uses TID-lists.

To examine the effects of partitioning, the non-partitioning SEAR algorithm is contrasted with three partitioning algorithms: SPTID, which uses TID-lists, SPEAR, the partitioned version of SEAR, and SPINC, a new incremental partitioning algorithm that allows less than two scans over the data. As shown in Table 1.1, SEAR, SPEAR and SPINC all use item-lists and prefix-trees to ensure comparability; SPTID is included for completeness.

	non-partitioning	partitioning	incremental
TID-lists	SPTID (1 partition)	SPTID	—
Item-lists (Prefix-trees)	SEAR	SPEAR	SPINC

Table 1.1: Overview of sequential algorithms

Our experiments show that

1. pass bundling reduces both CPU and IO overhead significantly,
2. partitioning introduces constant CPU overhead per partition which exceeds by far the benefits due to reduced IO cost, unless disk IO is extremely expensive,
3. TID-lists are fast in later phases, but highly inefficient in early phases of the algorithm and dissatisfactory performance can only be avoided by bypassing the initial phases with special optimizations (which do not use TID-lists).

Therefore, we contend that partitioning — contrary to common opinion — does not reduce the execution time of ARM algorithms, but leads to worse performance as the number of partitions grows. The efficient use of TID-lists requires partitioning, because large intermediate results would otherwise have to be swapped to disk for databases larger than the available buffer space ; the algorithms that use TID-lists are therefore slowed by the partitioning overhead, even if CPU cost is greatly reduced by means of the bypass optimization. In contrast, item-lists with pass bundling also achieve low IO and CPU cost without the partitioning overhead. Unfortunately, because the bypass optimization was not mentioned in the paper on PARTITION [34], we could not add it to SPTID to compare both concepts directly to determine which performs better, and further research is required to provide a definitive answer to this question.

All sequential algorithms will be described in detail in Chapter 3, the results of our sequential experiments can be found in Chapter 4.

In spite of continued improvements in sequential algorithms, data mining queries in general and the ARM algorithms in particular remain expensive and time-consuming operations that are too costly for convenient use in interactive mining tools. The rapid growth of database sizes also calls for higher mining speeds than are currently available. Sampling is one solution to this problem [28], but looking at only a portion of the database will not produce exact results. Parallel data mining is the second option and the focus of the second part of this report, where parallel SPMD[2] implementations of SEAR and SPEAR on an IBM SP2 message-passing multiprocessor and the results of our speedup and scale-up experiments are presented. We assume the

[2] short for Single Program Multiple Data

shared-nothing paradigm that is commonly used in parallel databases [12, 13, 6] and the nature of the problem suits this assumption well. Our findings confirm our initial expectations that both algorithms parallelize well, require only a comparatively small amount of communication and achieve near-linear speed-up that is only diminished by the sequential portions of the algorithms.

1.1 Overview over KDD Research

To give an overview over the field of KDD we first introduce the major types of knowledge we desire to extract from data. We proceed to list the additional difficulties that arise with databases as data source for pattern discovery algorithms. This section ends with more general issues and a proposed architecture for KDD systems. For more detailed introductions to KDD see [21, 40].

1.1.1 Types of Knowledge

Recalling the aforementioned definition of KDD by W.Frawley – that discovered information is implicit, previously unknown and potentially useful – we can state what "knowledge" means in our context. Being *implicit*, knowledge in the data mining sense goes beyond mere factual knowledge that has been stored and managed successfully by DBMS for years. The fact that Ms X works for company Y and earns \$ 30,000 annually is not the desired output of a data mining algorithm, no matter how revealing and surprising this fact may be to the user. This shows that *previously unknown* is to be understood both from the system's perspective and with respect to the user's current level of knowledge. This kind of knowledge could also be called "meta-knowledge": patterns that characterize the data — hidden laws and structures that need not be strict functional dependencies, but may hold only with a certain probability. Frawley in [15] demands that the statement of the discovered pattern be somewhat simpler than the subset of data objects it describes. What *simpler* means is left vague intentionally. Length of encoding or other information theoretic measures seem reasonable and have been used widely [11, 36, 2]. In short, we are interested in facts about data, and the term knowledge shall be used with this meaning from now on. The last requirement – *potentially useful* – is highly dependent on the application and even on the special focus of the current mining task. This issue will be revisited in later sections when background knowledge and related work are discussed (Section 1.1.3 and Section 7.2.1).

R. Agrawal in [2] identifies the three types of knowledge to be discovered in databases: classification, association and sequences. *Classification* tries to divide the given data set into disjoint classes using supervised or unsupervised learning, the latter also being referred to as *clustering*. The goal is to find a set of predicates[3] that characterize a (in the supervised case predefined) class of data objects and can be applied to unknown objects to predict their class membership. A bank, for example, might want to classify its credit customers to determine whether to give loans or not. In [15] W. Frawley and G. Piatetski-Shapiro subdivide this task into *summarization*, that searches for common characteristic features of one class only, and *discrimination* where the goal is to find features that help distinguish different classes or alternatively one class from all others.

When discovering *sequences*, time is given as an additional attribute and questions concentrate on dynamic patterns. Examples can be found in stock market data or consumer behavior. For special algorithms for sequence discovery that use the Discrete Fourier Transform to speed up the pattern search we refer the reader to [33]. An interesting problem involving sequences is the discovery of *episodes*, frequent partially ordered sets of events that occur within a given time window [29]. The algorithms for finding episodes are very similar to those used in ARM which is why they are described in a comparatively detailed way in Section 7.1.

[3] in machine learning terminology: concept descriptions

The third type of knowledge are *association rules*. Associations can be arbitrary rules of the form $X \rightarrow Y$, X and Y being conjunctions of attribute value restrictions. We have already introduced this area and will treat it in depth in Section 2.1.1.

[2] proposes a unified framework for all three areas, arguing that they all are variations of a basic rule discovery task and can be reduced to a set of standard operations. In fact, classification can be viewed as discovering the antecedent of rules that have the class label as the consequent. A rule like [salary> \$30,000 $\wedge$ average balance >\$1000 $\wedge$ age > 35] [$\rightarrow$ class = low credit risk] may suffice as an example. For performance reasons it is advisable, however, to exploit application characteristics in specialized algorithms for different tasks. This observation is supported by the large number of highly specialized algorithms proposed for ARM alone.

1.1.2 Database Specifics

One might argue that many aspects of data mining are not very different from standard machine learning problems. However, the special situation of using a database as the data source causes additional difficulties.

Size of Data Sets

While training sets in machine learning rarely exceed several thousand items, databases with hundreds of thousands or even millions of items are common today, and the trend is rising. This introduces the problem of handling the data itself and potentially large amounts of intermediate results. Furthermore, most current data mining algorithms view the database as one global relation. This universal-relation assumption aggravates the size problem because databases which are split up into several relations according to some normal form to save disk space have to be joined into one (most likely, very large) relation.

Noise, Missing and Contradicting Data

Databases are typically not generated and maintained for the purpose of data mining; their data is intended to serve the interest of the application, not to facilitate the mining job. In the worst case, attributes that are indispensable for the discovery task at hand may be missing altogether and bad or wrong results will be produced. But even on a smaller scale, the task is made much harder for mining algorithms by NULL values that occur very frequently in relational data. The basic choice in dealing with NULL values is to either replace them by default values determined by conditional probabilities based on the available fields or to ignore the data object altogether and avoid a presumably dangerous wrong choice. Both alternatives are inherently flawed because they may produce wrong results. The last resort is for the system to try to obtain the missing values from the user or other external sources. But this is usually not feasible.

Real world data is often noisy or contains contradictory information due to errors or simply due to the nature of the data. This is not the best possible input to standard learning algorithms. Probabilistic methods are needed to deal with these difficulties.

Redundant Information

While discovery algorithms are supposed to detect patterns in the database, this is undesirable for previously known dependencies. [30] identifies two cases where trivial patterns are wrongly reported as "knowledge" due to redundant information stored in the database. One case is strong functional dependencies when one field is a function of one or several other fields, for example: Profit = Sales - Expenses. The other case occurs when field values are merely constrained by other fields, like BeginDate $\leq$ EndDate. An additional, and unfortunately rather frequent situation is related to concept hierarchies. Whenever a reference to some higher concept is stored with a data object this will be considered a pattern by the algorithm although this knowledge is of no interest to the user. Examples are City $\Rightarrow$ State, Department $\Rightarrow$ Division in a large company, or AIDS$\Rightarrow$ ViralInfection. These difficulties indicate that the mining process has to be directed and supported by *background knowledge* or *domain knowledge*, an issue we will talk about in Section 1.1.3.

Knowledge Representation

The knowledge discovered should be readable by humans, and it should allow easy retrieval of the items in the database that match a discovered rule. This pretty much rules out the use of neural nets, because the learned information is usually hidden in internal layers of the net. This is fine for learning robot motions, but not for when a sales manager wants to know what products to arrange next to each other. The difference is that, to take an example from ARM, symbolic learning produces the actual rules, while a neural net can only decide for a given rule whether it is significant or not. For this reason, neural nets prohibit the use of a database query language for optimized retrieval of items that match a rule. Instead of fast indexed selection, a scan over the entire database is necessary. As argued in [1] symbolic learning outruns neural nets because multiple passes over the training set are necessary in the learning process.

This favors the use of symbolic representations, namely predicates and so called *decision trees* that are predominantly used in classification.

1.1.3 General Issues

Users in the Mine

Mining tools are not yet able and might never be able to determine whether the knowledge they discover is of any interest to the user. A mining tool can easily flood the user with trivial domain knowledge as pointed out in previous sections. It comes as no surprise, for example, that all pregnant patients are female. Therefore interactively directing the discovery process is desired. This requires new means to formulate highly complex queries in a way that is acceptable to the domain expert, and sophisticated means of result representation, in short: high system to user bandwidth and vice versa. An example from association rule discovery that investigates rule templates and visualization of results can be found in [26]. It is not clear just how much user interaction is desired or necessary, and the answer to this question may very well be dependent on the mining domain. Systems with various degrees of independence exist and the tradeoff between versatility and independence of a discovery system was even used in [30] to typify KDD systems.

The "human in the loop" requires fast response times, and execution times of several minutes cannot be tolerated. The demands users make on discovery tools create additional challenges for KDD algorithms and deserve further attention. Users might want fast, but less accurate results for a first quick look at the data and focus on points of interest later demanding more and more accuracy and reliability as they go along. Typical usage patterns like this raise issues of result reuse and result refinement that may be more efficient than rerunning the algorithms from scratch for every user query. Storing discovered knowledge for future reference and consistency of such growing data/knowledge bases in dynamic environments are only examples of some of the problems that have to be addressed in interactive mining systems.

Including Domain Knowledge

This point is strongly related to the previous one in that it tries to contain the amount of insignificant information generated by the mining algorithms. Based on domain knowledge or — in more elaborate systems even common sense reasoning — filters can be designed to prevent trivialities from being presented to the user.

Providing the discovery tool with domain knowledge can also speed up the discovery process because the algorithm can take off from previous knowledge or identify certain paths as useless and abandon them. It is obvious, for example, that the name of a patient is usually not a decisive factor in the diagnosis of his illness. The attribute can therefore be removed entirely before the mining algorithm is invoked. The problem of selecting and exploiting this knowledge is being investigated. Examples of domain knowledge are the use of concept hierarchies, user-defined predicates, and automated selection of relevant attributes to reduce data sizes and running time [16].

1.1.4 KDD System Architecture

Internal and External Miners

A mining tool can be viewed as an application that uses the DBMS as server for its data management requirements. In this case it would be called an external discovery tool, whereas internal miners only read the data from the DBMS once and convert it into their own format, working independently from that point on. External tools are necessarily more tailored to the database interface (SQL or the like) but since DBMS are usually not optimized for data mining, performance may suffer in this approach. We have already mentioned the universal relation assumption that can lead to huge amounts of data or repeated expensive joins. Relations might also have to be inverted to convert them into a format more suitable for mining. Consider, for example, grouping a DebitCredit type history relation by account number.

Internal tools, on the other hand, need storage for the data, but access is more efficient especially because large parts of the data often need not be considered in later passes of the algorithms and internal miners can prune the data set along the way. Hybrid approaches or mining algorithms that rely heavily on database functionality[16] are also possible of course.

Some argue that mining capabilities should not be provided in applications separate from the actual database but be an integral part of a DBMS's capabilities[15]. According to this vision future DBMS should be able to provide the user with more insight on the data than just selection and aggregation. Interesting questions that arise in this context are concerned with the tradeoffs between support for OLTP and the mining algorithm's needs for fast access to the entire database. In other words, the question is how to couple knowledge base and database ? [25]

A Model for KDD

We conclude this chapter by presenting a model which reflects most of the observations made in previous sections. This section follows roughly the idealized model for KDD systems proposed in [30]. Actual systems may more or less map to this scheme, but most major components will be represented.

As depicted in Figure 1.1, input to the system comes from the user who runs mining queries against the database, in the form of domain knowledge and from the database itself. The knowledge base is used to store pre-supplied domain specifics and possibly knowledge acquired in previous mining sessions. Based on this background knowledge, the focus component selects the part of the database that is pertinent to the current task and the DB interface creates the actual database query to retrieve the data. Note that this model precludes neither internal nor external mining. Pattern extraction denotes the set of actual mining algorithms, and the evaluation component filters the discovered patterns according to their interestingness. To the model proposed in [30], we added the user interface and display components to emphasize the need for and the non-triviality of convenient and effective user interaction. Current research is investigating primarily three areas: pattern extraction as the core component of a KDD tool, the inter-working between domain knowledge and the focus–extraction–evaluation process, and finally the user interface aspects.

The subsequent chapters will focus on the pattern extraction problem for association rules. However, many of the issues presented here in a rather general way will surface again under this more narrow perspective.

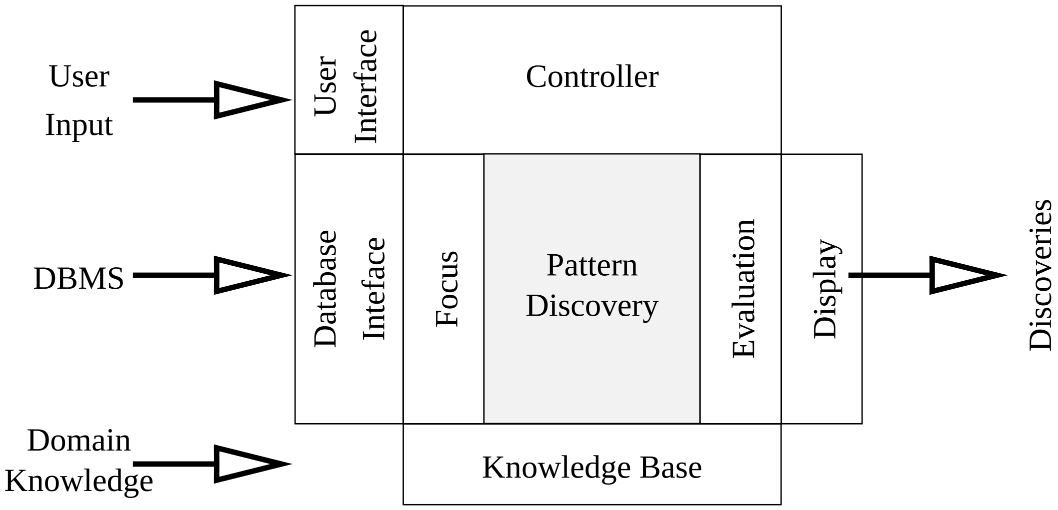


Figure 1.1: General model for KDD systems

1.2 Structure of this Report

The remainder of this report is structured as follows. Chapter 2 introduces association rule mining more formally and provides an overview over previous algorithms. Our own work on sequential algorithms is described in Chapter 3, covering the algorithms SEAR, SPTID, SPEAR and SPINC, in this order. The subsequent chapter reports the results of our experiments with those sequential algorithms, focusing on the effects of different data structures and the benefits of partitioning.

The second part of this report is dedicated to the parallel implementations of the SEAR and SPEAR algorithms. Our major concern is to investigate how these algorithms parallelize; also, contrasting the two algorithms compares a non-partitioning and a partitioning algorithm, providing further insight on (parallel) partitioning. Chapter 5 describes the algorithms PEAR (parallel SEAR) and PPAR (parallel SPEAR), and the results of the parallel experiments with these algorithms can be found in Chapter 6.

A short overview of interesting extensions and other related work is given in Chapter 7, before we conclude this report with a summary of our results and a list of future research topics in Chapter 8.

Chapter 2

Association Rule Mining

This chapter introduces association rule mining in more detail, starting with the formal problem statement and properties of association rules and frequent sets that play a central role in ARM. The second part of the chapter is dedicated to describing and evaluating previous algorithms for the problem. Other related work such as the treatment of concept hierarchies, other parallel KDD projects and episode-discovery in sequences are briefly covered later in Chapter 7.

2.1 Association Rules

In this section we give the formal definition of the association rule mining problem. First, however, we introduce *frequent sets* which form the basis for generating the actual association rules. Except where stated otherwise we follow the initial terminology used by [3].

2.1.1 Definition of Frequent Sets

Given a set $\mathcal{I} = \{I_1, I_2, \ldots, I_m\}$ of items (e.g. a set of items sold by a retail store) a *transaction*[1] T is defined as any subset of items in $\mathcal{I}$ ($\subseteq \mathcal{I}$). Like sets, transactions do not contain duplicates, but we extend the pure notion of a set and assume that the items in transactions and in all other itemsets we may consider are sorted.

Let the database $\mathcal{D}$ be a set of n transactions, and each transaction is labeled with a unique transaction identifier (TID).

A transaction T is said to *support* a set $X \subseteq \mathcal{I}$ if it contains all items of X, i.e. if $X \subseteq T$. Sometimes we need to refer to the set of transactions that support X and use $T(X)$ to denote the set of TIDs of these transactions.

We define the *support* of X, abbreviated *supp*(X), to be the fraction of all transactions in $\mathcal{D}$ that support X. If we have $supp(X) \geq s_{min}$ for a given *minimum support* value s_{min}, the set X is called *frequent*[2]. The motivation behind minimum support is that we want to concern ourselves only with itemsets that occur often enough in $\mathcal{D}$ to be interesting. Infrequent itemsets, i.e. those that do not have minimum support, are not considered interesting. Finally, we will call an itemset X of cardinality $k = |X|$ a *k-itemset*.

Three properties of frequent sets will be helpful later on; Properties 2.2 and 2.3 in particular form the foundation for all ARM algorithms.

[1] Note the special use of the term *transaction* in the ARM context. Different from its usual context in DBMS, the term applies to data about consumer transactions.

[2] [3] in $\mathcal{D}$ and other early work refer to frequent sets as *large* sets, but since this has lead to confusion with the cardinality of the set we chose to adopt the change in terminology proposed recently in [37].

Property 2.1 (*Support for Subsets*)
If $A \subseteq B$ for itemsets A, B, then $supp(A) \geq supp(B)$ because all transactions in $\mathcal{D}$ that support B necessarily support A also.

Property 2.2 (*Supersets of Infrequent Sets are Infrequent*)
If itemset A lacks minimum support in $\mathcal{D}$, i.e. $supp(A) < s_{min}$, every superset B of A will not be frequent either because $supp(B) \leq supp(A) < s_{min}$ according to Property 2.1.

Property 2.3 (*Subsets of Frequent Sets are Frequent*)
If itemset B is frequent in $\mathcal{D}$, i.e. $supp(B) \geq s_{min}$, every subset A of B is also frequent in $\mathcal{D}$ because $supp(A) \geq supp(B) \geq s_{min}$ according to Property 2.1. In particular, if $A = \{i_1, i_2, \ldots, i_k\}$ is frequent, all its k (k-1)-subsets are frequent. Note that the converse does not hold.

2.1.2 Definition of Association Rules

An *association rule* is an implication of the form $R : X \rightarrow Y$, where X and Y are disjoint itemsets: $X, Y \subseteq \mathcal{I}$ and $X \cap Y = \emptyset$. Furthermore, $Y \neq \emptyset$ is required. Such a rule can be understood as the prediction, that if a transaction supports X, it will also support Y with a certain probability, which is called the *confidence* (denoted $conf(R)$) of the rule.

The confidence of R is defined as the conditional probability that given that T supports X, T will also support Y. More formally:

$$conf(R) = p(Y \subseteq T \mid X \subseteq T) = \frac{p(Y \subseteq T \wedge X \subseteq T)}{p(X \subseteq T)} = \frac{supp(X \cup Y)}{supp(X)}.$$

The support for rule R in $\mathcal{D}$ is defined as $supp(X \cup Y)$. The confidence of a rule reveals how often it can be expected to apply, while its support indicates how trustworthy the entire rule is. For a rule to be relevant it needs to have enough support and sufficient confidence. We will therefore say that rule R *holds* with respect to $\mathcal{D}$, some fixed minimum confidence level c_{min} and a fixed minimum support s_{min}, if $conf(R) \geq c_{min}$ and $supp(R) \geq s_{min}$. Note that, as a necessary condition for a rule to hold, both antecedent and consequent of the rule have to be frequent.

To illustrate the importance of both requirements, minimum support and minimum confidence, assume first that we base a rule on just one data object. This rule will have maximum confidence of 100% but it does not describe a common pattern in the database and has to be discarded because it lacks minimum support. Let's assume now that a rule has enough support, but low confidence, for example it says that 2% of all customers who buy soap also purchase tomatoes. Although this fact is well supported by the data in the database, it is not relevant, because it does not express a strong correlation.

Properties of Association Rules

The requirement that antecedent and consequent be disjoint is not absolutely necessary; it does not lead to nonsense rules but only to redundant or insignificant ones. $X \rightarrow X$, for example is trivially true and $X \rightarrow X \cup Y$ is equivalent to $X \rightarrow Y$ and therefore not interesting. The antecedent X of a rule may be empty; every transaction is considered to support the empty itemset and therefore the entire database satisfies the antecedent. The confidence of such a rule is equal to the relative frequency of the consequent itemset. The consequent Y is required to be non-empty for the same reason we demand that antecedent and consequent be disjoint. Recall that for the confidence of a rule we are interested in the conditional probability $c = p(Y \subseteq T \mid X \subseteq T)$ for some transaction T. If $Y = \emptyset$ we have $c = p(\emptyset \subseteq T \mid X \subseteq T) = \frac{supp(X \cup \emptyset)}{supp(X)} = 1$. This shows that, in fact, the rules $X \rightarrow X$ and $X \rightarrow \emptyset$ are equivalent.

As opposed to functional dependencies that require strict satisfaction, association rules are generally not transitive and do not compose. In most cases, whether a rule holds or not cannot be inferred from the confidence of another rule. These observations are stated more formally in the following list:

Property 2.4 (*No Composition of Rules*)
If $X \to Z$ and $Y \to Z$ hold in $\mathcal{D}$, the same is not necessarily true for $X \cup Y \to Z$. Consider the case when $X \cap Y = \emptyset$ and transactions in $\mathcal{D}$ support Z if and only if they support either X or Y. Then the set $X \cup Y$ has support 0, and therefore $X \cup Y \to Z$ has 0% confidence.

A similar argument applies to the composition of rules with the same antecedent:

$$X \to Y \ \wedge \ X \to Z \quad \not\Longrightarrow \ X \to Y \cup Z$$

Property 2.5 (*Decomposition of Rules*)
If $X \cup Y \to Z$ holds, $X \to Z$ and $Y \to Z$ may not hold. This is the case, for example, when Z is present in a transaction only if both X and Y are present also, i.e. $supp(X \cup Y) = supp(Z)$. If the support for X and Y is sufficiently greater than $supp(X \cup Y)$, the two individual rules do not have the required confidence. The situation is depicted in figure 2.1. Circles denote the set of transactions that support the itemset they are labelled with.

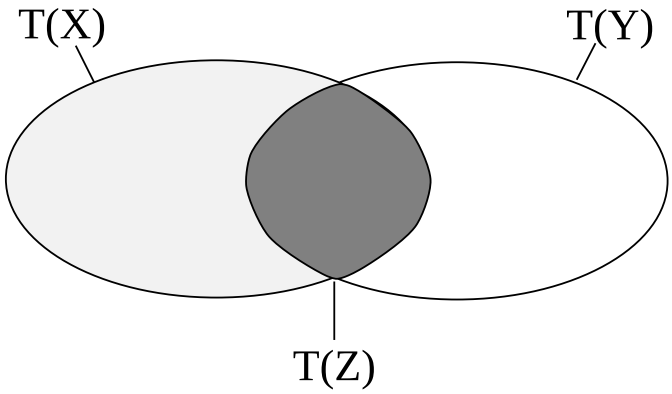


Figure 2.1: Counter-example for composition of rules

However, the converse,

$$X \to Y \cup Z \quad \Longrightarrow \quad X \to Y \ \wedge \ X \to Z,$$

holds because $supp(XY) \geq supp(XYZ)$ and $supp(XZ) \geq supp(XYZ)$. Therefore, both support and confidence of the smaller rules increase compared to the original rule. Unfortunately, this does not help much during the rule construction process, because we would like to build larger rules from smaller ones and not the other way around.

Property 2.6 (*No Transitivity*)
If $X \to Y$ and $Y \to Z$ hold we cannot infer that $X \to Z$. Assume for instance that $T(X) \subset T(Y) \subset T(Z)$ and the minimum confidence level is c_{min}. Let $conf(X \to Y) = conf(Y \to Z) = c_{min}$. Based on relative support values we get $conf(X \to Z) = c_{min}^2 < c_{min}$ since $c_{min} < 1$ which is not enough confidence and the rule does not hold. Note that this computation is based on the set inclusion above. In general, confidence values cannot be multiplied in this manner.

Property 2.7 (*Inferring Whether Rules Hold*)

[2] show that if a rule $A \rightarrow (L - A)$ does not have minimum confidence, neither does $B \rightarrow (L - B)$ for itemsets L, A, B and $B \subseteq A$. Using $supp(B) \geq supp(A)$ (Property 2.1) and the definition of confidence we obtain $conf(B \rightarrow (L - B)) = \frac{supp(L)}{supp(B)} \leq \frac{supp(L)}{supp(A)} < c$.

Likewise, if a rule $(L - C) \rightarrow C$ holds, so does $(L - D) \rightarrow D$ for $D \subseteq C$ and $D \neq \emptyset$, because the consequent is required non-empty.

The last property will be used to speed up the generation of rules once all frequent sets and their support is determined. See Section 2.3.3 for details on the algorithms.

Other Possible Definitions

The preceeding definition is a restricted version of association rules that in their most general form may contain negations and disjunctions also. Disjunctions could be used to express that different variations of a product were sold and the difference is not relevant, as e.g. in [LowFatMilk $\vee$ SkimMilk] $\wedge$ Butter $\rightarrow$ [Jelly $\vee$ Peanutbutter]. But this approach is not likely to be successful because the additional expressivity adds to the complexity of the discovery algorithms. Furthermore, the problem of deciding the interestingness of such rules becomes much harder, because large disjunctions of arbitrary items can easily reach minimum support in a database. Finally, the hierarchical solutions we describe in Section 7.2 are much better suited for treating groups of items.

Rules that contain negation suffer from similar drawbacks. The rule space that has to be searched grows immensely, and discarding the vast amount of irrelevant rules becomes very difficult. It is intuitively obvious that, given the large number of possible itemsets, the number of sets of items not bought together exceeds those actually purchased together by orders of magnitude. Note however that limiting rules to conjuncts in antecedent and consequent actually restricts the patterns we can possibly find. It may very well be an interesting piece of information, that people who buy Coke and Chips display a strong dislike for RootBeer, but this dissociation cannot be expressed in our rules.

2.2 The Basic Algorithmic Scheme for Mining Association Rules

With these definitions we can proceed to describe the structure of the basic rule discovery algorithm. Although the algorithms presented later will be very different from each other, they all use a basic scheme, the components of which may be arranged differently or the entire scheme might be applied repeatedly.

To construct all association rules the support of every frequent itemset in the database has to be computed. Therefore, all algorithms proceed in two stages: First all frequent sets are generated, then a second phase is charged with generating the actual rules and their confidence from those frequent sets.

Finding All Frequent Sets

As noted before, the number of potential frequent sets is equal to the size of the power set of all items, which grows exponentially with the number of items considered. A straightforward algorithm might perform an exhaustive search and test every set in the power set as to whether it is frequent or not.

The basic method every algorithm follows is to create a set of itemsets, called *candidates*, that it believes could be frequent. How many such candidates are created, in which sequence and how often depends on the individual algorithm.

To find out which of these candidate itemsets are actually frequent and what their exact support is, the support for each candidate set has to be counted in a pass over the database.

Since counting the occurrences of a candidate set involves a considerable amount of processing time and memory, the obvious goal is to reduce the number of candidates generated by an algorithm.

Rule Construction

As soon as all support values are available, possible rules can be created and their confidence determined. For every frequent set X, each of its proper subsets is chosen as antecedent of a rule and the remaining items constitute the consequent. Since X itself is frequent, all of its subsets have to be frequent as well according to Property 2.3, and their support is known. The confidence of the rule is computed and the rule is accepted or discarded, depending on the minimum confidence level.

Improvements can be gained from Property 2.7 because once a rule fails, no subsets of its antecedent have to be considered any more. Compared to the task of finding all frequent sets, this step is rather straightforward.

2.3 Previous Algorithms

All ARM algorithms proposed in the literature separate rule construction from finding frequent sets. The latter is the interesting part and the one that is solved differently by the individual algorithms. We therefore focus on finding frequent sets throughout this section. The algorithms use one of three major data representations to store the database: item-lists, candidate-lists and TID-lists. We describe their advantages and disadvantages as they are introduced and argue how the choice of representation influences the performance of the algorithms that use them.

2.3.1 AIS

The problem of association rules was first introduced in [3] along with an algorithm that was later called AIS by the authors [4].

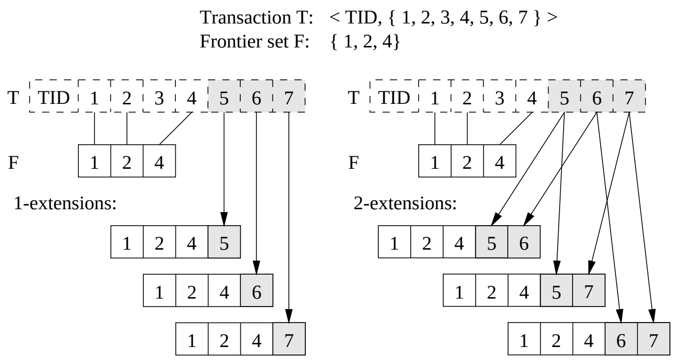


Figure 2.2: Illustration of 1- and 2-extensions for AIS

To find frequent sets, AIS creates candidates on-the-fly while it reads the database. Several passes are necessary, and during one pass, the entire database is read one transaction after the other. A candidate is created by adding items to sets that were found to be frequent in previous passes. Such sets are called *frontier sets*. The candidate that is created by adding an item to a frontier set F is called a *1-extension* of F because one item was added to F. To avoid duplicate candidates, only items that are larger than the largest item in F are considered for 1-extensions. To avoid generating candidates that do not even occur in the database, AIS does not build 1-extensions on blind faith, but only when they are encountered while reading the database. Figure 2.2 illustrates which 1-extensions are created when a transaction is read that supports a frontier set F. This figure illustrates also how this concept can be extended to k-extensions as well, that are formed by adding k items to a frontier set.

Associated with every candidate, a counter is maintained to keep track of the frequency of this candidate in the database. When a candidate is first created, this counter is set to 1, and when it is found subsequently in other transactions, this counter is incremented.

After a complete pass though all transactions, the counts are examined and candidates that meet the minimum support requirement become the new frontier sets[3].

Ideally, all former frontier sets can be discarded, because all their possible extensions have been considered, and keeping them for the next pass would create candidates that were already evaluated. In most cases, however, the storage requirements for candidate sets and frontier sets exceed main memory and frontier sets have to be swapped out to disk. Their extensions, as generated to this point, are abandoned and produced again in the next pass, which is cheaper than swapping them to disk also.

After the new frontier sets have been determined (all candidates with minimum support), the next extension/counting phase begins unless there are no frontier sets left, which means that none of the previous candidates were frequent. Initially, the only frontier set is $\emptyset$, which is extended to all 1-itemsets in the first pass, which in turn are extended to 2-itemsets in the second pass and so on.

Unfortunately, this candidate generation strategy creates a large number of candidates, and sophisticated pruning techniques are necessary to decide whether an extension should be included in the candidate set or not. The methods include a technique called *pruning function optimization* and estimating support for a prospective candidate based on relative frequencies of its subsets. Pruning functions use the fact that a sum of carefully chosen weights per item can rule out certain sets as candidates without actually counting them. An example is the total transaction price. If fewer transactions than the minimum support fraction exceed a price threshold then sets that are more expensive cannot possibly be frequent. These decisions can be fairly costly; moreover, they have to be made repeatedly for many subsets for each transaction. If an unlikely candidate set is rejected, this decision has to be made for every transaction the set appears in.

In this initial paper, rules were restricted to one item in the consequent but allowed any union of items in the antecedent. This limitation was not justified by the algorithm itself because it finds all frequent sets, and this information is enough to produce rules without this limitation.

2.3.2 SETM

SETM [22] is designed to use only standard database operations to find frequent sets. For this reason, it uses its own data representation that stores every itemset supported by a transaction along with the transaction's TID. Figure 2.3 shows part of an example run of SETM on a tiny database and illustrates also, how the database is stored in a table of <TID,itemlist > records. SETM repeatedly modifies the entire database to perform candidate generation, support-counting and the removal of infrequent sets. The above representation is used throughout the algorithm; whether the database contains all candidate itemsets or only all frequent itemsets depends on the current stage.

We use Figure 2.3 to explain the algorithm. We assume that all infrequent items have been deleted already, which is why <1,{5}> is not part of L1, the stage that contains all frequent 1-itemsets. To form C2, L1 is joined with itself on equal TIDs as shown. For each transaction, C2 contains all the 2-candidates it supports each one accompanied by the TID. The next task is to delete infrequent candidate itemsets from C2 to produce a table L2 that contains all frequent 2-itemsets. This is done by first sorting C2 on itemsets,

[3] This is a simplification because determining which expansions to include as candidates becomes more tricky in the presence of k-extensions and support estimation. For k-extensions, for example, only maximal frequent sets become frontier sets. For the details we refer the reader to [3]

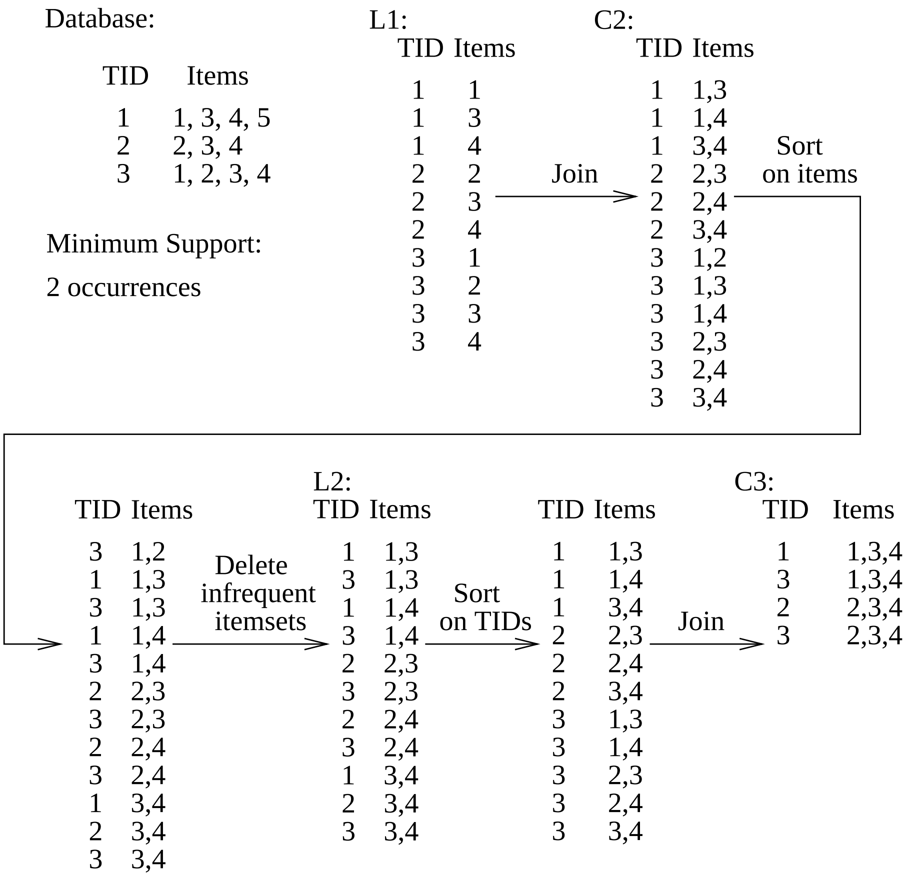


Figure 2.3: Part of a sample run of SETM

counting support by means of aggregation and then deleting infrequent sets. Now, we would like to compute all the 3-candidates supported by each transaction, which is accomplished by another join like the following (k=2):

insert into C(k+1)
select a.TID, a.item$_1$, a.item$_2$,. . ., a.item$_k$, b.item$_k$
from Lk a, Lk b
where a.TID = b.TID, a.item$_1$=b.item$_1$,. . ., a.item$_{k-1}$=b.item$_{k-1}$, a.item$_k$ $<$ b.item$_k$

Unfortunately, L2 is currently sorted on itemsets instead of TIDs which is the order required to carry out the join efficiently. Therefore L2 is sorted on TIDs before the self-join. After C3 is created, it is sorted on itemsets and infrequent sets are removed and so on, until some Lk becomes empty.

The problem with this algorithm is that candidates are replicated for every transaction they occur in, which results in huge sizes of intermediate results. Moreover, the itemsets have to be stored explicitly, i.e. by listing their items in ascending order. Using candidate ids would save space, but then the join could not be carried out as an SQL operation. What is even worse is that these huge relations have to be sorted twice to generate the next larger frequent sets.

The advantage and the novelty of this approach is that, in fact, SETM creates fewer candidates than AIS. Assume AIS reads our small example database in its third pass with all itemsets in L2 as frontier sets. When looking at the first transaction, AIS will notice that it supports the frontier sets $\{1,3\}$, $\{1,4\}$ and $\{3,4\}$ and contains item 5. Nothing stops AIS from generating the candidates $\{1,3,5\}$, $\{1,4,5\}$ and $\{3,4,5\}$, although

Algorithm 2.1 *Apriori-gen*

```
insert into C_{k+1}
select a.item_1, a.item_2,..., a.item_k, b.item_k
from L_k a, L_k b
where a.item_1=b.item_1,..., a.item_{k-1}=b.item_{k-1}, a.item_k < b.item_k

{now prune rules with subsets missing in L_k}
forall itemset c ∈ C_{k+1} do
    forall k-subsets s of c do
        if (s ∉ L_k) then
            delete c from C_{k+1}
end
```

item 5 is not even frequent. But these candidates are never considered by SETM.

In spite of this advantage, as reported in [4], the inefficiencies of SETM outweigh by far those of AIS.

2.3.3 Apriori, AprioriTid and AprioriHybrid

Candidate Generation: Apriori-gen

The vast number of candidates AIS creates caused its authors to develop a new candidate generation strategy called *Apriori-gen*[4] as part of the algorithms Apriori and AprioriTid [4]. Apriori-gen has been so successful in reducing the number candidates that it was used in every algorithm that was proposed since it was published [19, 32, 34, 37].

The underlying principle, based on Property 2.7, is to generate only those candidates for which all subsets have been previously determined to be frequent. In particular, a $(k{+}1)$-candidate will be accepted only if all its k-subsets are frequent.

Assume candidates of size $k{+}1$ are to be created. As shown in the SQL-like code in Algorithm 2.1, Apriori-gen takes the set of frequent k-itemsets L_k as input and searches for pairs of sets that have their k-1 smallest items in common. Taking the k-1 common items and the two different items, these two sets are then joined to form a prospective $(k{+}1)$-candidate. Duplicates are avoided by demanding that the largest item of the second set be greater than the largest item of the first. So far, the frequency of only two subsets has been asserted, because their mere presence in the input set allowed the candidate to be created in the first place. Note that up to this point, Apriori-gen is similar to the candidate generation strategy used by SETM.

The novelty of Apriori-gen is, that the existence of all remaining k-subsets of each candidate is also tested in the second part of Algorithm 2.1.

If, for example, $\{1,3,4,6\}$ and $\{1,3,4,8\}$ are frequent, those two are joined to create the candidate set $\{1,3,4,6,8\}$. The subsets that remain to be checked are $\{3,4,6,8\}$, $\{1,4,6,8\}$ and $\{1,3,6,8\}$, and if any of these are infrequent, the candidate is discarded. Note that it is enough to pair only those frequent sets that differ only in their largest set. Consider frequent sets $\{1,3,4,6\}$ and $\{1,3,5,6\}$, that differ on their third item only. The candidate that could be created from this pair is $\{1,3,4,5,6\}$. But if this set should be a candidate at all, its subsets $\{1,3,4,5\}$ and $\{1,3,4,6\}$ are in L_4 and will be used to generate it.

Since the 1-candidates are simply all the sets that contain only one item, this procedure is not necessary to generate them. Apriori-gen is first used to generate C_2 from L_1.

---

[4] Very much the same idea was suggested independently in [28] where it was called *off-line candidate determination*($\mathcal{OCD}$)

The second novelty of Apriori-gen (and the one leading to its name) is that candidate generation is done prior to and separate from the counting step, and the algorithm is only called once to create the candidates of a given size.

Therefore, the improvements achieved by Apriori-gen over the candidate generation strategy employed by AIS are two-fold: first, fewer candidates are created, and secondly, they are not created repeatedly for every transaction, but only once. SETM also creates more candidates than do algorithms that use Apriori-gen, and it also creates them again and again for every transaction.

Apriori

Apriori is the first algorithm to use Apriori-gen for candidate generation. As mentioned in the previous paragraph, Apriori-gen is separate from the counting step that determines the frequency of each current candidate. This means that each pass of Apriori consists of a call to Apriori-gen to generate all candidates of a given size (size k in pass k) and a counting phase that determines the support for all these candidates. Each counting phase scans the entire database.

Upon reading a transaction T in the counting phase of pass k, Apriori has to determine all the k-candidates supported by T and increment the support counters associated with these candidates.

In order to perform this operation efficiently, Apriori stores candidate sets in a tree. Figure 2.4 illustrates this structure for all 3-candidates that are possible for 5 items. The actual sets are stored in the leaves of the tree, and edges are labeled with items. To find the proper location for a candidate, starting from the root, traverse the edge with the first item in the set. Reaching an internal node, choose the edge labeled with the the next item in the set, until a leaf is reached. The path to locate set $\{1, 3, 4\}$ is marked with thickened arrows in the figure. Note that by virtue of ordering the items, each set has its unique place in the tree. The smallest items in a set that are used along the path to the leaf need not be stored. Inserting sets into the tree can cause a leaf node to overflow, in which case it is split and the tree grows.

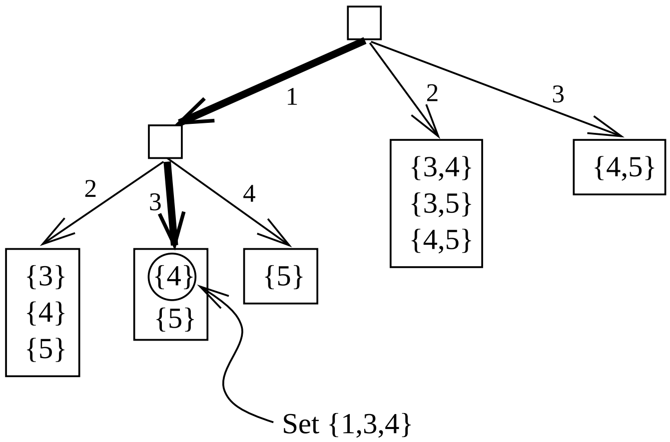


Figure 2.4: Apriori's tree structure for candidates

To count all candidates for transaction T, all leaves that could contain a candidate have to be searched, and to reach all these leaves, Apriori tries all possible combinations of the items in T as paths to a leaf.

Once a leaf with a set of candidates is located in this fashion it remains to be checked which are actually supported by the transaction. As far as the implementation is concerned, this test for set inclusion can be optimized by storing the sets as bitmaps, one bit for each item. As observed in [34], these bitmaps can become quite large for many items (128 Bytes for 1000 items) and cause considerable overhead.

Internal nodes are implemented as hash tables to allow fast selection of the next node. To reach the leaf for a set, start with the root and hash on the first item of the set. Reaching the next internal node, hash on the second item and so on until a leaf is found.

We implemented a different hash-based tree structure for all our algorithms that resembles the one used by Apriori, but avoids the overhead of searching for candidates within leaves. This data structure is explained in Section 3.1.1.

Item-lists

The major problem for Apriori (and for AIS as well) is that it always has to read the entire database in every pass, although many items and many transactions are no longer needed in later passes of the algorithm. In particular, the items that are not frequent and the transactions that contain less items than the current candidates are not necessary. Removing them would obviate the expensive effort to try to count sets that cannot possibly be candidates.

Apriori does not include these optimizations, moreover they would be hard to add to Apriori (and AIS as well). The reason stems from the item-list data representation used by both algorithms. As depicted in Figure 2.5, transactions are stored as a sequence of sorted itemlists in this representation. While item-lists are the most common representation and the one that is usually assumed as input format, they make it difficult to remove unnecessary parts of the data. Let's assume we want to remove all items that are not part of any frequent set. Unfortunately, the knowledge of which items to keep and which to discard is only available and applicable after scanning the database to count the support for the candidates. Therefore, we can eliminate items only in the subsequent pass over the data, that is they have to be read once more, although this is not really necessary. As we will see later, the other two representations remove these items instantly, which leads to much smaller data sizes in later passes; unfortunately this is not the case for early passes, where the volume of intermediate data representations can exceed the original data size. The advantage of item-lists is therefore that the size of the data does not grow in the course of the algorithms.

TID	Items
1	{ 1, 3, 4, 5 }
2	{ 2, 3, 4 }
3	{ 1, 2, 3, 4 }

Figure 2.5: Item-list data representation

AprioriTid

The shortcoming of Apriori, that it could not remove unwanted parts of the database during later passes, has prompted its authors to develop AprioriTid, which uses a different data representation than the item-lists used by Apriori.

AprioriTid[5] [4] can be considered an optimized version of SETM that does not rely on standard database operations and uses Apriori-gen for faster candidate generation. Furthermore, AprioriTid reads the data only once and tries to buffer the data for all other passes. Again, a pass consists of the Apriori-gen call followed by a counting phase to determine the support for the current candidates.

Unlike SETM, AprioriTid stores a transaction as the list of the current candidates it supports. This representation, which we call *candidate-lists* is depicted in Figure 2.6. Note that only the database transactions

[5]The name conflicts with our terminology of the data representations because it suggests the use of TID-lists which is not true. In fact,while the authors use TIDs for illustration, AprioriTid does not need them at all.

are stored in this fashion, all other sets such as candidates and frequent itemsets are still lists of items in ascending order as usual. No initial conversions are necessary to create candidate-lists from the common input structure because an item can be associated with the corresponding 1-candidate-set, thus for the first pass (candidate sets of size 1) there is no difference between this representation and item-lists. After the first pass the data changes, such that for every transaction, all the 2-candidate-sets that it supports are stored; in the second pass, those are replaced by candidates of size 3 and so on. The data from previous passes is not needed any more. Counting support for candidates is easy, because candidates are stored explicitly in each transaction.

Database:

TID	Items
1	1, 3, 4, 5
2	2, 3, 4
3	1, 2, 3, 4

Minimum Support:

2 occurrences

C1:

TID	Candidates
1	{1},{3},{4},{5}
2	{2},{3},{4}
3	{1},{2},{3},{4}

C2:

TID	Candidates
1	{1,3},{1,4},{3,4}
2	{2,3},{2,3},{3,4}
3	{1,2},{1,3},{1,4},{2,3},{2,4},{3,4}

C3:

TID	Candidates
1	{1,3,4}
2	{2,3,4}
3	{1,3,4}{2,3,4}

Figure 2.6: Candidate-list data representation

The interesting part of AprioriTid is how a list of $(k+1)$-candidates C_{k+1}^T supported by some transaction T can be derived from the set of k-candidates C_k^T supported by T. Auxiliary data structures are useful to accomplish this task: When running Apriori-gen to create all $(k+1)$-candidates, we store with each candidate the references to the two k-itemsets that were used in its generation. Furthermore, along with each of the k-itemsets, references to all its candidate extensions are kept. C_{k+1}^T is computed as follows: for every frequent candidate set in C_k^T, consider all its candidate extensions c computed by Apriori-gen and check if both generating itemsets are present in T. If so, add c to C_{k+1}^T.

Compared to SETM, AprioriTid has the advantage that candidate identifiers can be used instead of explicitly listing all the items of a candidate set, which effectively reduces the size of the intermediate results C_k. Furthermore, the auxiliary data structures save work that is wasted by SETM during sorting and joining to create candidate sets.

The importance of AprioriTid stems from the fact that it was the first algorithm to read the database only once and work on a copy in memory from that point on. While this idea is of limited use to this algorithm, because it limits the size of the database, those limits are overcome later by the PARTITION algorithm (Section 2.3.4).

Candidate-Lists

The disadvantage of using candidate lists is that the size of intermediate results, although much smaller than those that have to be handled by SETM, can still be several times the original data size. The reason is that a transaction of k frequent items supports $\binom{k}{2}$ 2-candidates. Therefore $\frac{k(k-1)}{2}$ candidate identifiers have to be stored after the first pass compared to only $m \geq k$ candidates (one for every item) in the original database. In our experience with synthetic datasets[6], between 50% and 95% of all items are typically frequent[7], which means that except for extreme cases we have $0.5m \leq k \leq 0.95m$. For example, for a transaction of 15 items 10 of which are frequent, we have 45 2-candidates as opposed to 15 1-candidates. To make matters worse, the size of intermediate results cannot be known beforehand unless other means (e.g sampling) are used to estimate the characteristics of the data set. Note that this behavior is a feature of candidate lists which is independent of the specifics of AprioriTid.

To handle the cases when intermediate results exceed main memory, AprioriTid uses a buffer manager that swaps part of the data to disk temporarily. This constitutes an inherent limit to the size of the data that can be processed efficiently by AprioriTid.

The advantage of candidate-lists is that useless parts of the data are discarded automatically in the process. A transaction that does not support any of the current candidates will be removed. Of course, only the current candidates supported by a transaction are stored, and this number can be expected to be rather small for candidates with more than 2 items. So, while the data might be very large after the first pass, its size decreases rapidly in subsequent passes.

AprioriHybrid – Combining Apriori and AprioriTid

The performance tests conducted by R.Agrawal et al.[4] show that Apriori and AprioriTid outperform both AIS and SETM.

Comparing Apriori and AprioriTid is more interesting because they both generate the same number of candidates and differ mainly in their underlying data representation. While Apriori avoids swapping data to disk, it does not weed out useless items in later passes and wastes time on futile attempts to count support of sets involving these items. AprioriTid, on the other hand, prunes the data set as described in the previous section and as a result outruns Apriori in later passes. Unfortunately, it is slowed mainly in the second pass if its data does not fit in memory as a consequence of the candidate-list representation and swapping is necessary. In this case Apriori beats AprioriTid.

For this reason another algorithm, AprioriHybrid, is proposed in [4], that uses Apriori for the initial passes and switches to AprioriTid as soon as the data is expected to fit in memory. The switch takes an extra effort to transform one representation into the other that has to be balanced by savings in later passes. The hybrid version led to improvements over Apriori whenever AprioriTid could be used long enough after the switch to offset the extra effort, and performed slightly worse otherwise. Furthermore, the size of the database that can be mined by the hybrid version is not limited any more as it was by AprioriTid.

2.3.4 PARTITION

While all the algorithms presented so far are more or less variations of the same scheme, the PARTITION algorithm [34][8] takes a somewhat different approach. In doing so, PARTITION tries to address two major shortcomings of previous algorithms.

[6] see Section 4.1 for a description of the commonly accepted generation procedure

[7] To be exact, the corresponding 1-itemsets are frequent.

[8] Most of this section is based on this paper.

The first problem with the previous algorithms is that the number of passes over the database is not known beforehand, regardless of which representation is used. Therefore the number of IO operations is not known and is likely to be very large. AprioriTid tries to circumvent this problem by buffering the database, but then the database size is limited by the size of main memory.

The second problem lies with pruning the database in later passes, i.e. removing unnecessary parts of the data. Item-lists as used by AIS and Apriori are not well suited for this optimization, as we said earlier in Section 2.3.3. Candidate-lists do permit pruning the database, but cause problems because of their unpredictably large intermediate results in early passes.

We describe in detail how PARTITION addresses these shortcomings in the paragraphs below. We examine in Chapter 4 how severe these problems really are and how well the counter-measures perform, using our own algorithm SPTID, which uses many of the features of PARTITION. The implementation of SPTID is described in the next chapter.

Partitioning the Database

The approach taken by PARTITION (Algorithm 2.2) to solve the first problem (unpredictably large IO-cost) is to divide the database into equally sized horizontal partitions. An algorithm to determine the frequent sets is run on each subset of transactions independently, producing a set of *local frequent itemsets* for each partition. Let L^i denote the local frequent itemsets for partition i. Note that the minimum support level is a percentage of the number of transactions in the database, or in the partition respectively. 1% minimum support in a partition of 10,000 transactions is equal to 100 transactions, regardless of how large the actual database is. The partition size is chosen such that an entire partition can reside in memory. Hence, only one read is necessary for this step and all passes access only buffered data. The principles of Apriori-gen are applied for candidate generation.

Because the data in each partition is different, the L^i will be different from each other also, to the effect that the support of some sets is not available for all partitions. But all of these counts are necessary to decide whether a set is (globally) frequent or not. To obtain the so-called *global support* for these itemsets, another scan through the database is required, which is called the *counting phase* in the context of the PARTITION algorithm. To know which sets to count during this phase, the *global candidate set* $C^G = \bigcup_i L^i$ is formed as the union of all local frequent sets. All sets in C^G have to be counted, except for the ones that were frequent in every partition, because these counts are already available.

After the counting phase, all global support values are available and sets without minimum support can be discarded, leaving in L^G all and only the frequent sets. Thus, the total IO requirement is only two scans regardless of the database characteristics.

This procedure is correct because C^G contains all possible frequent itemsets. If a set is frequent in the entire database, it has to be frequent in at least one partition, which is expressed in the following property. Needless to say, the converse does not hold, which is why the counting phase is necessary.

Property 2.8 (Support on Partition of $\mathcal{D}$)

Given a horizontal partitioning $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_p\}$ of $\mathcal{D}$ with $\bigcup_i \mathcal{P}_i = \mathcal{D}$ and $\mathcal{P}_i \cap \mathcal{P}_j = \emptyset$ for all $i \neq j$. If itemset X is frequent in $\mathcal{D}$, there exists a partition $\mathcal{P}_x$ such that X is frequent in $\mathcal{P}_x$.

Proof (by contradiction)

Assume X is frequent in $\mathcal{D}$, but not frequent in any $\mathcal{P}_i$, $(1 \leq i \leq p)$. Further, let s_{min} be a fixed minimum support level, and let $n := |\mathcal{D}|$ and $n_i := |\mathcal{P}_i|$ be the number of transactions in $\mathcal{D}$ and $\mathcal{P}_i$ respectively.

Algorithm 2.2 *PARTITION*

$\mathcal{P}$ = compute_partition($\mathcal{D}$)
{*Phase I*}
forall partitions $\mathcal{P}_i$ **do**
  read_in_partition($\mathcal{P}_i$)
  $L^i = \{all frequent 1 - itemsets\}$
  for $(k = 2; L^i_{k-1} \neq \emptyset; k++)$ **do**
    $L^i_k = \emptyset$
    forall candidates c of size k
      $T(c)$ = generate_TID_list(c)
      if $(|T(c)| \geq n_i\, s_{min})$ **then** $L^i_k = L^i_k \cup \{c\}$
      else drop_candidate(c_j)
  $L^i = \bigcup_k L^i_k$
$C^G = \bigcup_i L^i$
{*Phase II*}
forall partitions $\mathcal{P}_i$ **do**
  read_in_partition($\mathcal{P}_i$)
  forall $c \in C^G$ **do**
    $T(c)$ =generate_TID_list(c)
    $c.count += |T(c)|$
return $L^G = \{c \in C^G | c.count \geq n\, s_{min}\}$
end

If X is frequent in $\mathcal{D}$, it follows that $supp_{\mathcal{D}}(X) \geq s_{min}$. But since X is not frequent in any $\mathcal{P}_i$, $supp_{\mathcal{P}_i}(X) < s_{min}$. Therefore

$$n\, supp_{\mathcal{D}}(X) = \sum_i n_i\, supp_{\mathcal{P}_i}(X) < \sum_i n_i s_{min} = n\, s_{min} ,$$

and therefore $supp_{\mathcal{D}}(X) < s_{min}$, which contradicts our assumption that X is frequent in $\mathcal{D}$. □

The TID-List Data Representation

To address the second problem (failure to reduce the database size in later passes), PARTITION uses a new "TID-list" data representation both to determine the frequent itemsets for each partition and to count global support during the counting phase.

TID-lists invert the candidate-list representation by associating with each itemset X a list of all the TIDs for those transactions that support the set; in other words, the database is transformed into a set of $< X$, T(X) $>$ pairs. Figure 2.7 illustrates this representation for initial database and intermediate results on an extended version of our sample database. The minimum support equivalent of 3 transactions is marked by a vertical line. TIDs are required to preserve the association between items of a transaction. The TID-lists for a k-candidate can be computed easily by intersecting the TID-lists of two of its (k-1)-subsets. All TID-lists are sorted so that this intersection can be computed efficiently with a merge-join, which only requires traversing the two lists once.

Like candidate-lists, TID-lists change in every pass and may have to be swapped to disk if there is not enough memory available to store them. Again, the size of intermediate results can be larger than the original

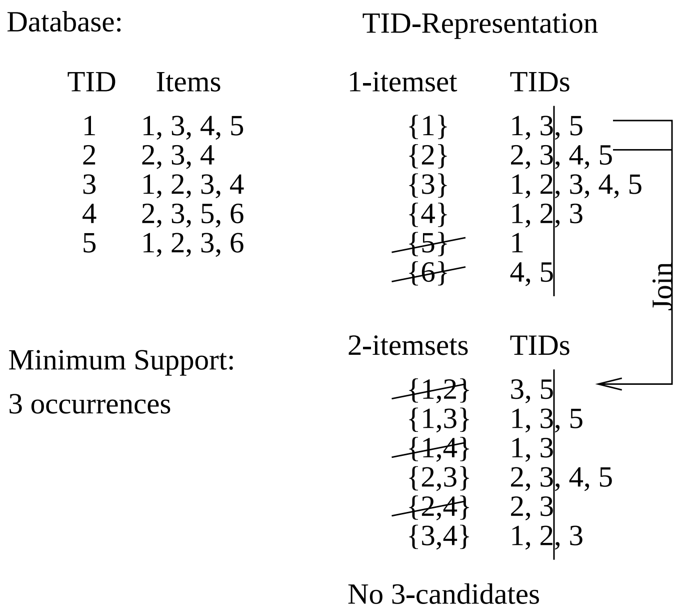


Figure 2.7: TID-list data representation

data size and this figure is not known. The reason is the same as for candidate-lists (Section 2.3.3), with the difference that ins PARTITION, TIDs are replicated for every candidate set instead of replicating candidate identifiers for every transaction. The size problem is less severe for TID-lists because candidate generation and counting are done together in the merge-join operation. If a candidate is found to be infrequent its itemlist can be dropped immediately. In the example shown in Figure 2.7, these lists have been crossed out. Figure 2.8 shows an example how the size of intermediate data structures changes depending on the minimum support. The data originate from runs of SETID, which is our implementation of PARTITION, on a 4.4MBytes database with 1000 different items and 100,000 transactions of average size 10. As can be seen, the data size grows by 20% for the lowest minimum support level, while data sizes decrease continuously for all other minimum support values.

Although increases in data size are not as severe for TID-lists, the problem remains that the amount of the increase cannot easily be estimated beforehand — at least not at the moment.

A second and more important disadvantage, according to our experiments with SETID, is that the merge-join is significantly more costly in early passes of the algorithm than comparable steps in other representations. We will explain this problem in more detail in Section 4.3.2.

TID-lists, on the other hand, permit the removal of all useless data, because TID-lists of infrequent items can be dropped easily and TIDs of transactions that do not support an itemset are omitted automatically in the merge-join. As seen in Figure 2.8, data sizes drop quickly in later passes for all minimum support levels.

The following paragraphs comment on individual aspects of PARTITION, namely the problem of choosing the size of one partition, the details of candidate generation (where Apriori-gen is used in a slightly different way) and how the counting phase is carried out.

Choice of Partition Size

PARTITION is negatively impacted by data skew, which causes the local frequent sets to be very different from each other. In this event, the global candidate set C^G will be very large, which renders the counting phase more expensive. The smaller the partitions, the more likely are negative effects due to data skew, since more itemsets are likely to be frequent either in only a few partitions, or in all but a few partitions.

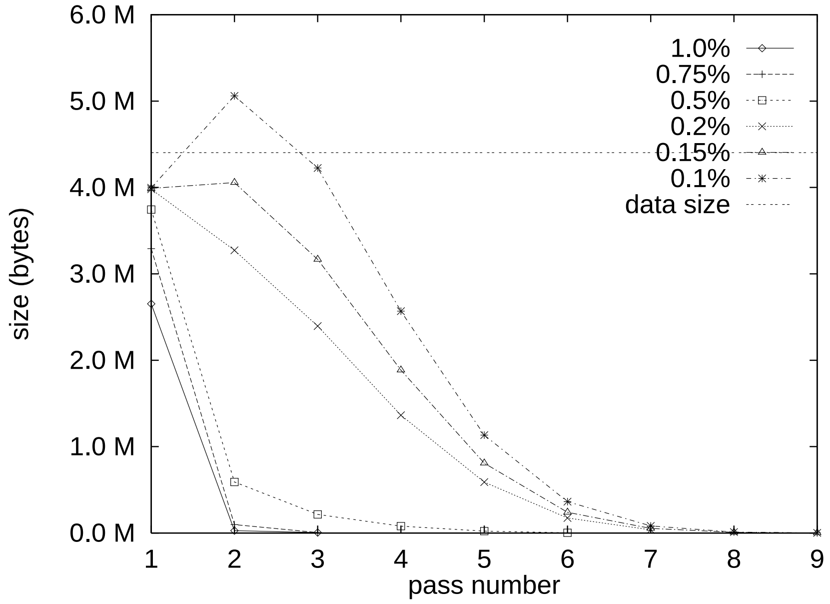


Figure 2.8: Size of intermediate data for various minimum support values

Both cases adversely effect performance because they increase the number of sets that have to be counted in phase II[9].

On the other hand, the partition size should not be chosen too large because the risk increases that intermediate results will exceed the available buffer space and necessitate swapping data to disk, slowing down the algorithm. Unfortunately, the size of intermediate data structures is not known and heuristics have to be applied for choosing the partition size. These heuristics are not detailed in [34].

Any such heuristic must take into account the memory size, the average transaction size, the current minimum support threshold and the number of items m. Both the number of items and minimum support are necessary to estimate the number of frequent sets. To understand this dependency, consider that the expected frequency of 1-itemsets is $\frac{n}{m}$, with n being the total number of transactions. While this is assuming a uniform distribution, which is usually not the actual item distribution, it serves to illustrate that a minimum support of 1% results in many more and larger frequent sets for a database with 100 different items than for one with 1000. These estimates are still very inaccurate because the actual decisive factors are the size and number of frequent sets, which are known only from the result of the algorithm.

One possible improvement is the use of adaptive strategies that can exploit the knowledge gathered in preceeding partitions and adjust the partition size dynamically. We do not pursue this issue further and leave it to further research.

Candidate Generation and Counting Local Support

Candidates are generated in the same way as in Apriori-gen, but counting support for a candidate is done immediately after its creation, as is shown in Algorithm 2.2. If a candidate C has been created as the union of two frequent itemsets X and Y, its support in the current partition is $|T(C)| = |T(X) \cap T(Y)|$ which can be obtained through a merge-join of the sorted TID-lists $T(X)$ and $T(Y)$. If C is frequent, $T(C)$ is stored to be used later in building further larger candidates.

[9] Note that partition sizes for the counting phase can be different from the sizes in phase I and partitions do not even have to be of equal size.

Counting Global Support in Phase II

As soon as the global candidate set is available, global supports can be counted. Candidates that were frequent in every partition need not be counted any more and can be removed from C^G. This phase also uses TID-lists and merge-joins to obtain the support values.

In general, the counting phase is much less expensive than phase I, because only sets, that are already known to be frequent, have to be counted. During phase I, however, TID-lists have to be generated for every candidate, and as mentioned repeatedly, that number is much greater than the number of frequent sets.

This concludes the list of previous algorithms and we continue to describe our own work in the next chapter.

Chapter 3

Sequential Algorithms

In order to investigate the effects of different data representations and the benefits of partitioning, we implemented several algorithms as listed in Table 1.1.

To study the tradeoffs between the item-list and TID-list data representation, we implemented the SEAR algorithm, which uses item-lists in conjunction with a new prefix-tree structure for storage of frequent sets and candidates. SEAR also implements an optimization we call pass bundling[1], the benefits of which have not been examined in the literature previously. SEAR is compared to SPTID, which is based on TID-lists. In fact, SPTID uses partitioning as well, but the number of partitions was set to 1 for these experiments to exclude this influence.

To study the effects of partitioning, we implemented SPEAR, a partitioned version of SEAR, and a new incremental partitioning algorithm we call SPINC, which reduces IO even more than regular partitioning. Both algorithms use the item-list representation and the prefix-tree structure to make them comparable to the non-partitioning SEAR algorithm. We included SETID in this comparison as well.

This chapter describes SEAR and the prefix-tree structure in Section 3.1, followed by SPTID in Section 3.2 and SPEAR in Section 3.2. The explanation of SPINC in Section 3.4 concludes this chapter. Performance results are reported in Chapter 4.

3.1 SEAR : Modifying Apriori

Like Apriori, each pass of SEAR consists of a candidate generation phase followed by a counting phase. An Apriori-gen-like procedure is used to create candidates. In fact, the pseudo-code for SEAR (Algorithm 3.1) is the same as for Apriori. However, SEAR uses the *prefix-tree* data structure for itemsets, which we developed to improve the data structure used by Apriori, and the pass bundling optimization, which is mentioned briefly in the literature [4, 28], but has not been investigated more closely. Prefix-trees and pass bundling are both described below.

3.1.1 Prefix Trees: Storage for Sets and Candidates

Recall how Apriori stores candidates in the tree-like data structure depicted in Figure 2.4. Candidate sets are stored in leaf nodes, each of which accommodates several candidates. The purpose of internal nodes is to direct the search for a candidate to the proper leaf.

In contrast, SEAR employs a prefix-tree structure, which makes is no distinction between internal and leaf nodes. In this structure, nodes do not contain sets, but only information about sets (e.g. counters). Each edge in the tree is labeled with an item, and each node contains the information for the set of items labeling

[1] In [4, 28] this optimization is mentioned, but not given a specific name.

Algorithm 3.1 *SEAR*

$L_0 = \emptyset, \quad k = 1$
$C_1 = \{\{i\} | i \in \mathcal{I}\}$ {*all 1-itemsets*}
while ($C_k \neq \emptyset$) **do**
  {*count support*}
  forall transactions $T \in \mathcal{D}$
    forall k-subsets $t \subseteq T$
      if $(\exists c \in C : c = t)$ **then** $c.count + +$
  $L_k = \{c \in C_k | c.count \geq n\, smin\}$
  C_{k+1} =generate_candidates(L_k)
  k++
return $L = \bigcup_k L_k$
end

the edges of its path to the root. An example is shown in Figure 3.1, where the set $\{1, 3, 4\}$ is marked by a thickened path. The items in a set X can be understood as a path descriptor to reach the node for X. Sorting items is required in order to avoid ambiguity; otherwise several nodes would correspond to the same set.

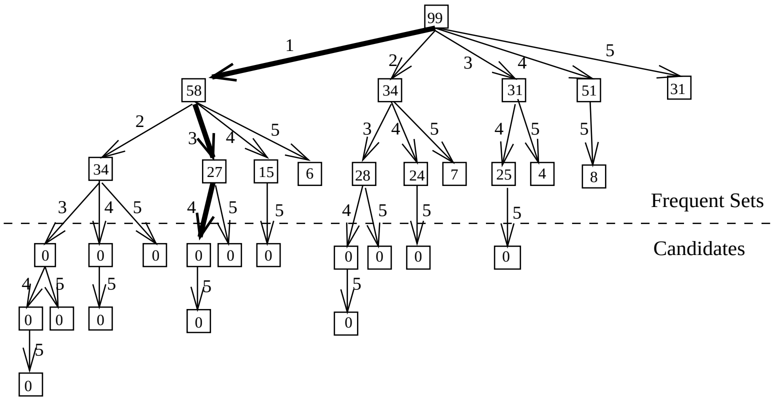


Figure 3.1: Prefix tree storage for sets and candidates

Another difference between the two structures is that prefix-trees store both frequent sets and candidate sets in the same tree. Once candidates are counted and determined to be frequent, they simply remain in their proper position in the tree and become frequent sets. In Figure 3.1 every node in the tree contains the frequency counts for its corresponding set. The root represents the empty set, and so its counter is equal to the number of transactions in the database, since all transactions support the empty set. Candidate sets have a count equal to zero before being processed. Figure 3.1 shows a complete tree, containing all the subsets of $\{1, \ldots, 5\}$, but under normal conditions only frequent sets and candidate sets will be stored in the tree, while infrequent sets are either not created in the first place or deleted immediately. Normally, only one level of candidates would be expanded in an actual run of SEAR. In the figure, all candidates are shown in order to illustrate the unbalanced shape of the tree, which is an immediate result of traversing edges in sorted order of items.

Some properties can be stated about this storage structure when it is used to store frequent sets:

Property 3.1 (Reformulation of Property 2.1)
The counts of nodes along a path are non-increasing.

Property 3.2 (Reformulation of Property 2.3)
If a set is frequent and therefore present in the tree, then *all* its subsets have to be in their proper place in the tree also.

For Property 3.1, consider that the parent of a node has the count for one of the node's subsets, because the parent's path is missing the last edge of the path to the node; and a set cannot occur more frequently in the database than any of its subsets. This property is used in computing the confidence of a rule.

Note that Property 3.2 is not restricted to the sets on the path to the root which naturally have to exist. For example, if set $\{1,2,3\}$ is in the tree, $\{1,2\}$ and $\{1\}$ are clearly present because they are along the path, but by Property 3.2, $\{1,3\}$ and $\{2,3\}$ must also be in the tree.

To illustrate the differences between prefix-trees and the Apriori structure, we use an example which shows how support for candidates is counted. Consider the transaction $T = \{1,4,6,7,9\}$ and assume that the candidates $c_1 = \{1,4,6,7\}$ and $c_2 = \{1,4,7,9\}$ are the only candidates supported by T. Assume further that Apriori has stored c_1 and c_2 in a leaf along with several other 4-candidates ($\{1,4,5,6\}$,$\{1,4,5,9\}$,...) that also have the prefix $\{1,4,\ldots\}$, but are not supported by T. This node can be reached from the root by traversing first the edge labeled with item 1 and then the one for item 4. Apriori tests items 1 and 4 once to reach the leaf, then has to check for all the candidates there if they are supported by T or not. The first two items (1 and 4) do not have to be considered any more, but for all the larger items I in a candidate set we have to check if $I \in T$. If sets are stored as itemlists, this means two comparisons per candidate in our example. If sets are stored as bit-masks, Apriori needs 32 integer comparisons per candidate (in the case of 1000 items and 4 Bytes to an integer) .

SEAR reaches node $\{1,4\}$ like Apriori, but then selects the edge for item 6 directly, then the one for item 7 and increments the count for c_1. The attempt to find edge 9 leaving node $\{1,4,6\}$ fails because $\{1,4,6,9\}$ is not a candidate. SEAR therefore returns to node $\{1,4\}$, selects edge 7, then edge 9, and increments the count for c_2. That amounts to a total of 5 edge selection operations which can be implemented efficiently as hash table look-ups. While more operations are necessary for larger transactions, Apriori will also need additional comparisons to find candidate items in the transaction. Note that in both algorithms, overhead can be reduced by removing items that are not part of any candidate from a transaction.

To summarize, the Apriori subset method uses a tree to reduce the number of candidates that have to be tested against a transaction, while our method uses the tree to reach exactly the candidates that are supported by the transaction.

As mentioned above, prefix-trees contain only frequent itemsets and the current candidates. Figure 3.2 shows the tree from Figure 3.1 for a minimum support of 10% (equivalent to 10 items). Many sets have been removed, and fewer candidates are created during the candidate generation phase. Consider the set $\{1,2,5\}$, for instance. Since $\{1,5\}$ is not frequent, this set is never created as a candidate.

3.1.2 Using Prefix Trees for SEAR

Dead Branches in the Tree

Storing both candidate sets and frequent sets in the same tree can cause severe performance degradation when support for candidates is counted. Consider Figure 3.2 again. The set $\{3,4\}$ does not have any candidate supersets. Keeping it in the tree means that for every transaction that contains the items 3 and 4, we will try to modify the counts for non-existent candidates in that branch. This is clearly a waste of effort. However, the set is frequent, so we want to keep it in the tree in order to use it to generate rules later. Therefore,

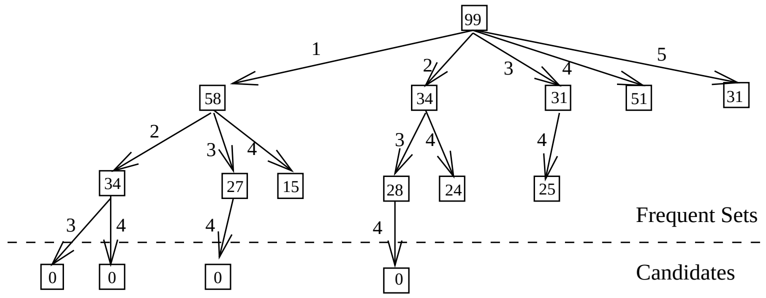


Figure 3.2: Candidates generated in practice

nodes that did not produce any candidates, which we call *dead branches*, are pruned from the tree and stored separately in such a way that they can be "revived" easily when they are needed at a later point of time.

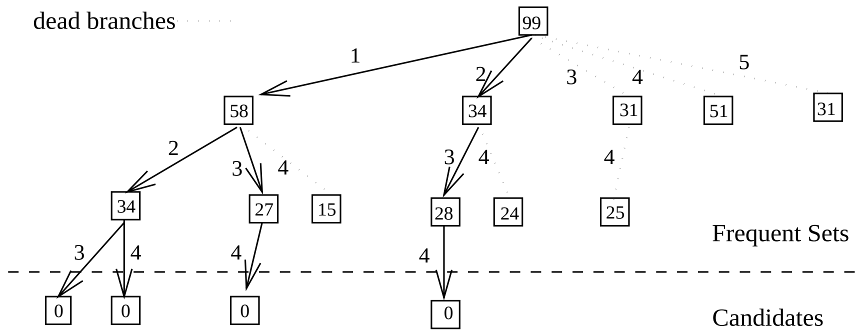


Figure 3.3: Same tree without dead branches

Candidate Generation

Candidate generation is performed as in Apriori-gen. However, the tree structure provides a fast test of whether subsets of prospective candidates are frequent or not. Furthermore, all sets needed to create prospective candidates are readily available; they are exactly all the direct siblings in the tree. To generate the candidate extensions of set $\{1, 2\}$ in Figure 3.2, we only need to consider its sibling items 3 and 4. No sorting of frequent itemsets is necessary as with other algorithms.

The actual implementation optimizes candidate generation further by combining the tests for all candidates possibly produced by a node and its immediate siblings instead of testing a single prospective candidate set at a time. The motivation for this optimization is that all these candidate have similar subsets, and accessing them all in the tree separately would be a repetition of effort. For candidates $\{1, 2, 3, 4, 5\}$ and $\{1, 2, 3, 4, 6\}$, for example, the existence of subsets $\{2, 3, 4, 5\}$ and $\{2, 3, 4, 6\}$ has to be checked (among others). Combining these tests saves having to descend down the same branches repeatedly.

Pass Bundling

In Figure 3.2, if only candidates $\{1, 2, 3\}$ and $\{1, 2, 4\}$ are frequent after the counting pass, a new candidate $\{1, 2, 3, 4\}$ will have to be checked. In fact, this is the only candidate that is left, and an entire pass over the database would be necessary to count just this one set. To avoid this waste of effort, SEAR allows the expansion of several levels of the tree before counting candidates. Thus, all three candidates in our example will be created and counted together. This technique, which we call *pass bundling*, was first mentioned in [28] and [4], but its effects were not examined there.

There is a tradeoff involved between the number of IO operations we save and the additional computation necessary to count the additional candidates. Clearly, pass bundling of several levels is only desirable in later passes when the number of candidates is small. Expanding more than one level early on results in not pruning the candidate set. Therefore pass bundling is implemented by setting a lower limit to the number of candidates that have to be created before a counting step is allowed. By choosing this *pass bundling factor* as some low multiple of the number of items, we ensure that early passes are not effected (because the number of candidates exceeds the the limit by far), while later passes are bundled together.

Implementation of Prefix-Trees

To provide fast selection of an edge while traversing a prefix-tree, a hash-table is associated with each node that has children. As illustrated in Figure 3.4, entries in the hash buckets are records with count, item number and a pointer to the child hash-table. Every hash-table has a *dead*-pointer to store a linked list of dead items that are the roots of dead branches. Crucial in terms of resource usage and speed is the size of the hash-tables. While the root table has to be fairly large, tables deeper in the tree can be much smaller. Since all candidates from one leaf are generated simultaneously, their number is known and we can use it to compute the size of the hash-table. The size of the hash table is a power of 2 to allow fast computation of the hash function using a simple AND operation.

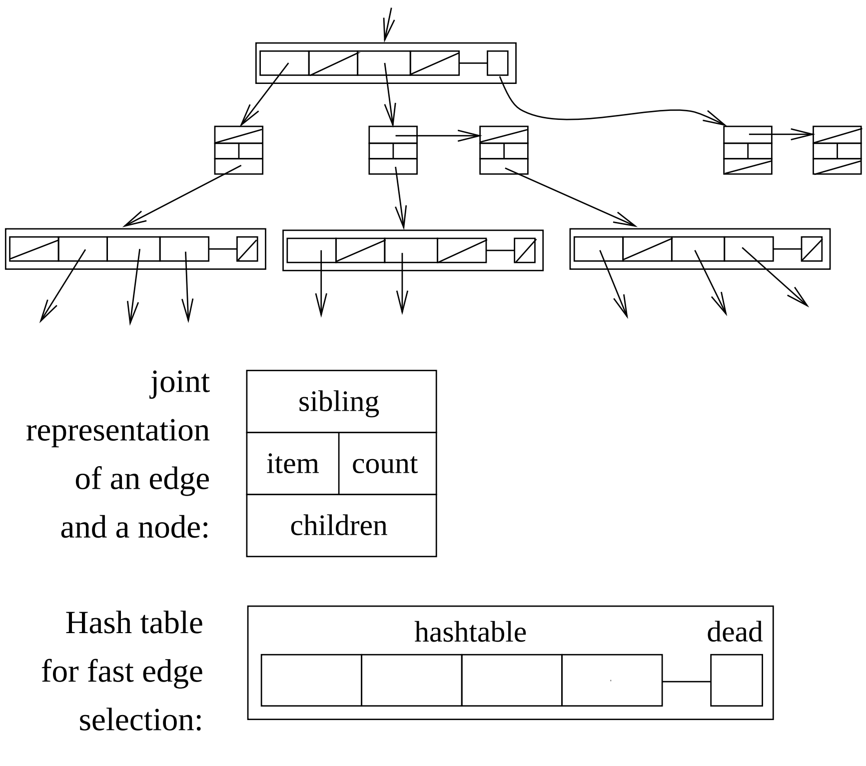


Figure 3.4: Implementation of prefix trees with internal hash-nodes

In addition to the prefix-tree itself, a list of all items that are still in use (i.e. all the items that are still members of candidate sets) is maintained during candidate generation. This list makes it possible to screen out all items of a transaction that are no longer relevant, thus reducing the number of failed hash operations during counting. In our experiments, typically between 30% and 60% of the items were not part of any candidate in later passes.

3.2 SPTID: A Partitioning Algorithm

This short section describes the implementation details of SPTID, which we implemented according to the description of PARTITION in [34]. However, in our experiments, we were unable to obtain results comparable to those reported there. The reason was that the algorithm used to obtain the results in [34] uses important optimization which is not mentioned in the paper. This optimization basically skips the second pass, counting the support for 2-candidates directly. Although we suspected this, we could not confirm it early enough to include it in this report. We will return to this issue in the performance section on TID-lists (Section 4.3.2). To distinguish the two algorithms we refer to our implementation (conforming with the description in [34]) as SPTID and use the name "PARTITION" for the optimized version of the algorithm.

Being similar to PARTITION, SPTID works with the TID-list representation and partitions the data to ensure the database is scanned at most two times. Support for a candidate is determined immediately after it has been generated from two frequent sets. To compute the support, the TID-lists of the two frequent sets are joined using a merge-join.

A minor difference between SPTID and the description in the original paper is that SPTID uses the same prefix-tree[2] structure to store frequent sets and the same candidate generation technique as SEAR (Section 3.1). The prefix-tree is important for SEAR, because fast access to candidates is indispensable for counting their support in the database. This is not the case for SPTID, because support for candidates is generated by means of TID-lists, and fast access to frequent sets is not required. Thus, while prefix-trees are vital to SEAR, they are merely and implementational detail for SPTID. Using the same candidate generation code further simplifies the comparison between TID-lists and item-lists, because our results are not obscured by different candidate generation procedures.

A first implementation stored TID-lists in buffer pages and used a database buffer manager to handle swapping of pages to disk if necessary. While providing a robust storage interface that could easily cope with unexpectedly large intermediate results, the additional layer caused intolerable overhead. For this reason, we switched to a specialized buffer manager that stores TID-lists as dynamic arrays and writes them to disk according to a LRU strategy if the memory used by all TID-lists exceeds the permitted buffer size. This improved the performance significantly and all results reported in Chapter 4 were obtained with this optimized version.

We use just one tree to store all frequent sets, no matter in which partition they are frequent, but work only on the subset relevant to the current partition. The advantage of this approach is that no additional structures have to be maintained, and getting the unused sets out of the way is accomplished easily by declaring those branches "dead".

[2] Actually the nodes in the SPTID prefix-tree are slightly different from the SEAR tree, because they also contain the address of the corresponding TID-list in addition to the frequency counters.

Algorithm 3.2 *SPEAR*

$\mathcal{P}$ = compute_partition($\mathcal{D}$)
{*Phase I*}
forall partitions $\mathcal{P}_i$ **do**
    read_in_partition($\mathcal{P}_i$)
    $k = 1$
    $C_1 = \{\{i\} | i \in \mathcal{I}\}$ {*all 1-itemsets*}
    while $(C_k \neq \emptyset)$ **do**
        forall transactions $T \in \mathcal{P}_i$
            forall k-subsets $t \subseteq T$
                if $(\exists c \in C_k : c = t)$ **then** $c.count++$
        $L_k^i = \{c \in C_k | c.count \geq n_i \, s_{min}\}$
        C_{k+1} =generate_candidates(L_k^i)
        k++
    $L^i = \bigcup_k L_k^i$
$C^G = \bigcup_i L^i$
{*Phase II*}
forall transactions $T \in \mathcal{D}$
    forall $t \subseteq T$
        if $(\exists c \in C^G : c = t)$ **then** $c.count++$
return $L^G = \{c \in C^G | c.count \geq s_{min} \, n\}$
end

3.3 The SPEAR Algorithm

This section describes the SPEAR algorithm, which uses item-lists and prefix-trees like SEAR[3], but also partitions the database like SPTID. This algorithm is based on the realization that the partitioning principle does not require the TID-list representation, and the combination is chosen to avoid the poor performance of TID-lists (Section 2.3.4) but still reduce IO cost due to partitioning. Each partition is loaded into memory and processed completely like SEAR would do with the entire database. As with SPTID, only two scans of the database are required. Algorithm 3.2 shows the pseudo-code for SPEAR.

The advantage of using item-lists is that the data size does not grow during the course of the algorithm, so no buffer management is required. A major disadvantage (as mentioned in Section 2.3.3) is that pruning transactions is not possible to the same extent as in SPTID, leading to higher CPU-costs in later passes.

The counting phase for SPEAR is basically a single counting pass of SEAR during which all active sets in the tree are considered as candidates. Sets which were frequent in all partitions are hidden in dead branches to avoid counting them again.

[3] Actually, the data structure is a little different from that used by SEAR, because nodes have to store counters for local and global support, and for the number of partitions in which a set was frequent. Furthermore, we use just one tree to store candidates, frequent sets and sets that are not frequent in the current partition, and avoiding confusion in the tree is tricky, but these implementation details are not significant enough to be explained here.

3.4 SPINC: An Incremental Algorithm

The fourth algorithm we study is SPINC, a version of SPEAR which uses a new incremental partitioning technique. This incremental method does not process partitions independently like SPEAR, but uses the partial results from preceeding partitions. The basic principle is to count a set in every partition following the one in which it was first found frequent.

Recall that any set that was frequent in at least one partition is part of the global candidate set and has to be counted in phase II of the SPEAR algorithm. Thus it may be a mistake to remove a set from consideration in the current partition, even if it is not frequent. If the set happened to be frequent in some previous partition, it's support will have to be assessed in the current partition anyway during the counting phase.

To be more specific, consider the following case: assume the database is divided into p partitions $\mathcal{P}_1, \ldots, \mathcal{P}_p$ and a set X is frequent in $\mathcal{P}_2$, but not frequent in all other partitions. Then SPTID (and SPEAR as well) will have the support for X only from partition 2 before the counting phase starts. But since $X \in C^G$, it must be counted again in all partitions.

But instead, once X is found frequent in $\mathcal{P}_2$, we could count the support for X during phase I in all succeeding partitions $(\mathcal{P}_3, \ldots, \mathcal{P}_p)$. At the end of phase I, the accumulated support for X is available from all partitions but the first one. Therefore, during phase II, support for X need only be counted in the missing first partition.

SPINC produces immediate savings in that a set will not be counted twice in the partitions in which it was frequent. This directly reduces the amount of computation. Furthermore, the last partition does not have to be counted at all in phase II, which saves the IO operations for this partition and allows the algorithm to use less than two passes over the data. The reason is that if a candidate c is an element of C^G then c has either been frequent in $\mathcal{P}_p$ for the first time, and its support is therefore available, or it was frequent in a previous partition, in which case its support in the last partition has also been counted already.

Generating Frequent Itemsets

Phase I for SPINC, which is outlined in Algorithm 3.3, does not differ greatly from its counterpart for SPEAR. The sets of all previous frequent k-itemsets, L_k, grow from partition to partition as all new frequent sets are added to L_k after pass k of each partition.

Candidates that are newly generated in a partition are labeled with the partition number (stored in *c.first_partition*), so that it is known in phase II when a set was first frequent. The check whether a candidate is in L_k or not is very simple due to the fact that all sets are kept in one tree structure, so before inserting a new candidate into the tree, simply test if it is already there.

While the basic idea of the algorithm is rather simple, it is complicated by the need to distinguish candidates in the current partition ($c \in C_k$) from those sets that are only counted because they were frequent in some earlier partition ($l \in L_k$). Separating those sets is necessary, because candidates for the next pass should be generated only from actual candidates and not from the (possibly much larger) union of C_k and L_k.

Completing Global Support Counts

Candidate sets are removed from C^G prior to counting the partition in which they were first frequent. Thus, the number of sets counted decreases from partition to partition and reaches 0 just before the last partition (possibly earlier). Algorithm 3.4 shows more formally how this is accomplished. Note that, contrary to PARTITION and SPTID, the partitions for the counting phase must be exactly the same as in phase I to avoid missing or duplicate counts for candidates that were not removed from C^G at the right time.

Algorithm 3.3 *SPINC – Phase I*

$\mathcal{P}$ = compute_partition($\mathcal{D}$)
forall $k : L_k = \emptyset$
forall partitions $\mathcal{P}_i$ **do**
  read_in_partition($\mathcal{P}_i$)
  $C_1 = \{\{i\}|i \in \mathcal{I}\}$ {*all 1-itemsets*}
  forall $c \in C_1 \setminus L_1 : \quad c.first\_partition = i$
  while $(C_k \neq \emptyset)$ **do**
    forall transactions $T \in \mathcal{P}_i$
      forall k-subsets $t \subseteq T$
        if $(\exists c \in C_k : c = t)$ **then** $c.count++$
        {*count previously frequent sets also*}
        else if $(\exists l \in L_k : l = t)$ **then** $l.count++$
    $L_k^i = \{c \in C_k | c.count \geq n_i\, s_{min}\}$
    {*add local frequent sets to previously frequent sets*}
    $L_k = L_k \bigcup L_k^i$ {*counts are added for sets* $l \in L_k \bigcap L_k^i$ }
    C_{k+1} =generate_candidates(L_k^i)
    forall $c \in C_{k+1} \setminus L_k : \quad c.first\_partition = i$
    k++

return $C^G = \bigcup_k L_k$
end

If the data distribution is even, which can be expected for large randomized data sets, we could actually hope that no more new sets are found in several of the later partitions. None of those partitions would then have to be considered again in the counting phase. Thus several partitions are read only once.

Lowering Minimum Support

One possible extension to SPINC to increase the savings effect in the counting phase would be to lower the minimum support level slightly for phase I. This would cause more sets to be counted during phase I, but possibly save IO in phase II. We illustrate the reason with an example. Given p partitions, assume a set X is the only set that is only frequent in the last partition. Because of this single set, the entire partition p-1 has to be read in the counting phase; all other sets were already counted for this partition in phase I. Chances are that with a lower minimum support during phase I, X would have been frequent in some partition $\bar{p} < p$. Then reading partitions $\bar{p}, \ldots, p-1$ would not be necessary in phase II.

Note, however, that lowering the minimum support increases the computational cost for phase I, because more locally frequent sets are found. If s_{min} is chosen too low, the savings in IO might easily be offset by the increased cost of counting. Therefore, lowering the minimum support level can be expected to be a delicate trade-off between IO savings and increased CPU-cost.

Still, SPINC can be expected to reduce IO- as well as CPU-overhead compared to SPEAR. Results of performance tests that verify this expectation can be found in Section 4.4.

Algorithm 3.4 *SPINC – Phase II*

```
input $\mathcal{P}$, $C^G$
$L^G = \emptyset, \quad i = 1$
while $(TRUE)$ do
    {prune global candidate set}
    $H = H \bigcup \{c \in C^G | c.first\_partition = i\}$
    $C^G = C^G \setminus H$
    if $(C^G = \emptyset)$ break
    read_in_partition($\mathcal{P}_i$)
    forall transactions $T \in \mathcal{D}$
        forall $t \subseteq T$
            if $(\exists c \in C^G : c = t)$ then $c.count++$
    $i++$  {next partition}

return $L^G = \{c \in H | c.count \geq n\, s_{min}\}$
end
```

Chapter 4

Experiments on Sequential Algorithms

This chapter reports the results of our sequential experiments. After a brief description of the data generation technique used to create all data sets, we show performance results for SEAR, in particular the effects of pass bundling, which significantly reduces both IO and CPU overhead.

TID-lists and item-lists are then compared in the following section, focussing on the algorithms SEAR and SPTID-1 (SPTID for one partition). We will show that the TID-lists in SPTID are inefficient in early passes of the algorithm and SEAR outperforms SPTID-1 for most parameter settings for this reason, although it uses multiple scans over the database.

Section 4.4 investigates the usefulness of partitioning by comparing the algorithms SPTID, SEAR, SPEAR and the incremental SPINC. The main comparison covers the three item-list-based algorithms (SEAR, SPEAR and SPINC), while SPTID is included for completeness; as will be shown, the use of TID-lists remains a handicap for SPTID as the number of partitions increases.

All experiments are run on one SP2 thin node[1], which is equipped with a 66 MHz RS/6000 processor, 64 MB of main memory and three 2 GB SCSI disks, only one of which is used. Our experiments have shown that about 6 MB/sec raw input performance can be obtained from one disk. No other tasks are present in the system.

The timing results show system time obtained by the program itself with the `gettimeofday()` system call. To keep the operating system from buffering the entire database, we flush the system buffers before every scan by writing a 50 MB file. Of course, the clock is turned off for these writes.

4.1 Synthetic Data Generation

All the following experiments were performed on synthetic data generated according to the procedure outlined in [4] which was designed to model the buying behavior of consumers in a retail environment. Note that our notation varies slightly from the one used there.

One of the main characteristics of retail data is that some items are sold much more frequently than others, and some sets of items are often purchased together. To achieve this grouping, items are not put in transactions randomly, but are are taken from a set of *maximal potentially frequent sets*, or just *maximal frequent sets*, $\mathcal{F}$. The basic idea is to create transactions as unions of maximal frequent sets. Thus the ARM algorithms rediscover the maximal frequent sets from the database — or rather combinations and subsets of them.

To generate a transaction, its size is computed first according to a Poisson distribution with mean $|T|$, maximal frequent sets are then chosen from $\mathcal{F}$ based on a fixed probability assigned to each set. This probability is chosen according to an $Exp(1)$ distribution. Once a maximal frequent set is selected, most of

[1] more information on the SP2 can be found in Section 5.1

$\mathcal{F}$	Maximal potentially frequent sets
$n = \lvert\mathcal{D}\rvert$	Number of transactions in $\mathcal{D}$
$\lvert T\rvert$	Average size of transactions
$m = \lvert\mathcal{I}\rvert$	Number of items
$\lvert\mathcal{F}\rvert$	Number of maximal potentially frequent itemsets
$\lvert F\rvert$	Average size of a set in $\mathcal{F}$
$x = 0.4$	Corruption level
$c = 0.5$	Correlation level

Table 4.1: Parameters for synthetic data generation

Name	Transaction size $\lvert T\rvert$	Size of Max. Freq.Sets $\lvert F\rvert$	Transactions $\lvert\mathcal{D}\rvert$	Data Size
T5.I2.100K	5	2	100,000	2.2 MB
T10.I2.100K	10	2	100,000	4.4 MB
T10.I4.100K	10	4	100,000	4.4 MB
T10.I6.100K	10	6	100,000	4.4 MB
T20.I2.100K	20	2	100,000	8.4 MB
T20.I4.100K	20	4	100,000	8.4 MB
T20.I6.100K	20	6	100,000	8.4 MB
likewise for 300,000 transactions, data size triples				

Table 4.2: Synthetic data sets

its items are added to the transaction while some are discarded to introduce noise into the data set. The probability for an item to be dropped is determined by the corruption level x. The corruption level is fixed for each set in $\mathcal{F}$ and chosen according to normal distribution with mean 0.4 and variance 0.1. Maximal frequent sets are selected and their items added to the transaction in this manner until the required transaction size is reached.

The number $|\mathcal{F}|$ of sets in $\mathcal{F}$ is provided as a parameter to the generation algorithm. The size of a set F in $\mathcal{F}$ is is determined by a Poisson distribution with mean $|F|$. To take into account the observation that some frequent sets overlap, the sets for $\mathcal{F}$ are created incrementally, retaining a certain number of items from the previous set and adding the rest randomly. The portion of items that migrate into the next set is controlled by a $Exp(c)$-distributed random variable with $c = 0.5$. Since this distribution determines how similar frequent sets are, the parameter c is called *correlation level.*

The effects of different corruption levels and correlation levels were reported to be rather insignificant in [4], which is why we did not conduct experiments that investigate these issues and used the values proposed there. Table 4.1 lists the different parameters.

A naming convention for data sets is customary for easier reference. According to this notation, T10.I4.100K describes a dataset with 10 items as average transaction size, an average size of maximal frequent itemsets of 4 and 100,000 transactions. Table 4.2 summarizes the parameters for the data sets we used.

4.2 SEAR Pass Bundling

Before comparing SEAR to any of the other algorithms, we examine its behavior more closely in this section. Most importantly, the benefits of pass bundling are presented. We use SEAR without pass bundling, i.e. a pass bundling factor $B = 0$, as reference and call this algorithm *SEAR_plain*.

An experiment with varying minimum support was chosen, because it illustrates best how pass bundling effects mainly later passes. To confirm these results, a size-up experiment for an increasing number of transactions is included in the second part of this section. Further experiments varying the average transaction size and the number of items are provided in Section 4.3 when we compare TID-lists and item-lists.

4.2.1 Varying Minimum Support

As described in Section 3.1, a bundling factor is used by SEAR to allow an additional, but still limited, number of candidates to be created before the database is scanned to count their support. A value of $B = 2$ on a database of m items, for example, has the effect that SEAR adds another level of candidates to the prefix-tree only if less than $2 \times m$ candidates have already been created in this pass. The correct choice of the bundling factor is important, because too small values can have little or even negative effects, and too large values can cause too many candidates to be created, leading to high CPU overhead for the counting step.

Figure 4.1 shows how the choice of this *bundling factor B* effects the running time for several minimum support levels. The range from 0.1% to 2.0% used in all minimum support experiments extends the range of values used in the literature (0.25% to 2.0%) to include even "harder" mining problems with more frequent sets; at the upper limit of 2.0%, only a few 1-candidates are frequent, which do not even allow any rules to be created, so exploring beyond this point is meaningless. $B = 0$ means that pass bundling is turned off; $B = 20$ was chosen as an upper limit, because for this value a minimum of at most 3 passes is reached for all minimum support values. Higher bundling factors lead to performance degradation, because too many useless candidates are created and the effort to count their support exceeds the cost of additional passes. For the same reason, bundling passes 2 and 3 is not useful (the number of 2-candidates is too large). Thus 3 passes constitutes an inherent barrier for the savings that can be achieved by pass bundling as implemented in SEAR[2].

While low bundling factors ($B < 5$) lead to little improvement or even worse performance over SEAR_plain, which corresponds to a bundling factor of 0, higher values for B reduce the running time by almost one half for lower minimum support levels. The highest bundling factor, $B = 20$ showed the greatest improvement, because all possible remaining candidates are generated after pass 2 and counted in a third scan over the database. Table 4.3 lists the number of passes required and the number of candidates generated for each pass bundling factor. Note, how the number of passes is reduced significantly at the expense of comparatively few additional candidates for the lower minimum support level. This saves IO as well as computation time.

Figure 4.1 shows very nicely how pass bundling effects mainly later passes which are only required for small minimum support values. In fact, there are only minor differences between the various bundling factors until more than 3 passes are required.

Note that, although this is not reflected in Figure 4.1, high pass bundling factors can lead to worse performance for higher minimum support values, if too many candidates are created. In our experiment this is the case for $s_{min} = 2.0\%$ and $B = 50$. Here the number of frequent 2-itemsets is so small, that no 3-candidates are created without pass bundling. Therefore no third scan over the database is necessary.

[2] Combining passes 1 and 2 is also possible, but is not implemented in SEAR, because special optimizations are required to count the large number of 2-candidates in this case. As explained in Section 4.3, PARTITION follows this approach to avoid an expensive second pass.

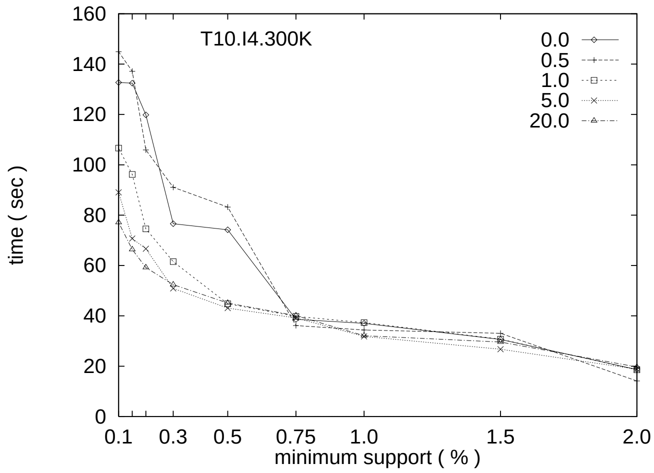


Figure 4.1: SEAR running times for various bundling factors

With a high pass bundling factor, however, a large number of 3-candidates is added to the tree in pass 2 (because the knowledge of which 2-candidates are frequent is not available), resulting in an unnecessary waste of counting time. Table 4.3 shows exactly how the number of candidates is effected in our case. Given the small number of 2-candidates for high minimum support levels, this scenario is highly probable.

Unless stated otherwise, we use a pass bundling factor of 20 for our experiments since it performs best for low minimum support values.

Bundling Factor	s_{min} = 2.0%		s_{min} = 0.15%	
B	Passes	Candidates	Passes	Candidates
0.0	2	2,891	9	482,559
1.0	2	2,891	6	482,595
5.0	2	40,711	4	483,581
20.0	2	40,711	3	485,465
50.0	2	598,556	3	485,465

Table 4.3: Number of passes and number of candidates for different pass bundling factors

4.2.2 Increasing the Number of Transactions

For the T10.I4 database and minimum support values of 0.75% and 0.3% we ran SEAR and SEAR_plain on different database sizes ranging from 250,000 to 10 million transactions. The running times are depicted in Figure 4.2. We expect a linear growth in running times because the work done per transaction remains constant. The results confirm this expectation and our previous observation that SEAR and SEAR_plain do not differ for s_{min} = 0.75%, while the lower minimum support level of s_{min} = 0.3% requires many more passes for SEAR_plain. Here, SEAR runs about twice as fast as its plain counterpart.

4.3 Comparing TID-Lists and Item-Lists

This section investigates hwo TID-lists compare to item-lists used in conjunction with prefix-trees and pass bundling. The algorithms are SPTID-1 (SPTID with only one partition) and SEAR; SEAR_plain is also

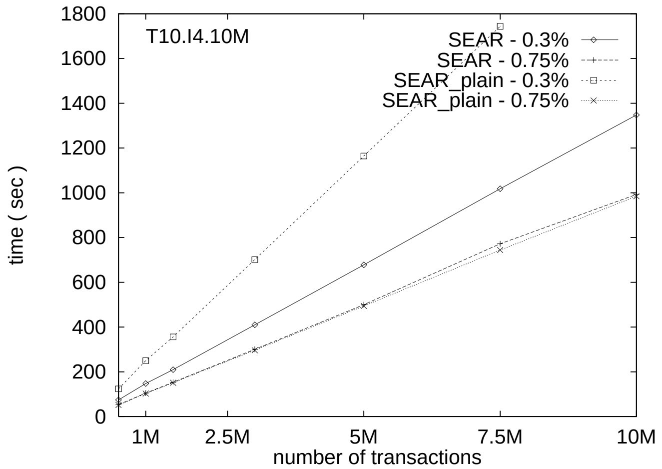


Figure 4.2: SEAR transaction scale-up for T10.I4.xxxK

included to help distinguish the effects caused by item-lists form those caused by the pass bundling optimization.

First, we report the experiments for varying minimum support in Section 4.3.1. These results are explained in the subsequent section. It is then possible to understand the results of experiments that vary the number of items and the average transaction size, which follow in Sections 4.3.3 and 4.3.4. At the end of this section, results of experiments conducted on various other data sets to confirm our results are described briefly and the results of this section are summarized.

The number of partitions is set to 1 for SPTID, because in this section we are not interested the effects of partitioning, but only in comparing the data structures. Using just one partition avoids the overhead of a larger global candidate set and obviates the need for the counting phase. Unless stated otherwise, the database was created as T10.I4.100K with 1000 items and 2000 maximal frequent sets. With a data size of 4.4 MBytes the data and intermediate results fit in memory easily, so distortions of our results by operating system page swapping are precluded. All algorithms generate approximately the same number of candidates, except for SEAR, which creates slightly more due to pass bundling. SEAR_plain and SPTID work with exactly the same number of candidates.

4.3.1 Varying Minimum Support

The first experiment examines the performance for different minimum support levels, the results of which are depicted in Figure 4.3. The running time of SEAR and SEAR_plain is determined by the number of passes, which accounts for the difference in running time between the two algorithms for minimum support less than 0.75%. With decreasing minimum support, the size of the largest frequent set grows, which is the number of passes SEAR_plain has to perform. For SEAR the number of passes increases much slower, because it combines several passes into one as shown in Section 4.2.

For low minimum support levels SPTID is much slower than both SEAR versions, and SEAR is up to 3 times faster than SPTID. Even SEAR_plain retains its edge over SPTID. SPTID is comparable to SEAR for large minimum supports, but for these values only 1-itemsets are frequent and therefore the results rather trivial, because no rules can be generated. The reason is that the number of those frequent 1-itemsets is extremely small, and thus, so is the number of candidates during these runs. For minimum supports of 2.0% and 1.0%, out of 1000 items only 62 and 171 respectively were actually frequent. This reduces the number of TID-lists SPTID has to join, while SEAR does not profit as much from the small number of candidates

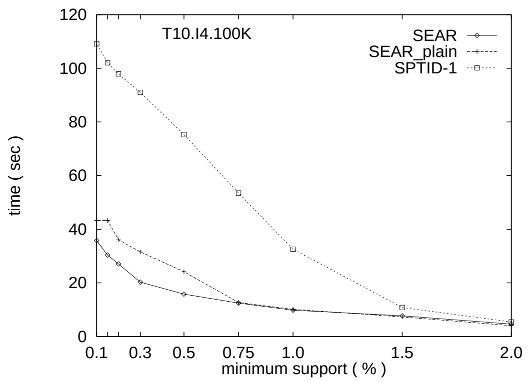


Figure 4.3: Total running times depending on minimum support

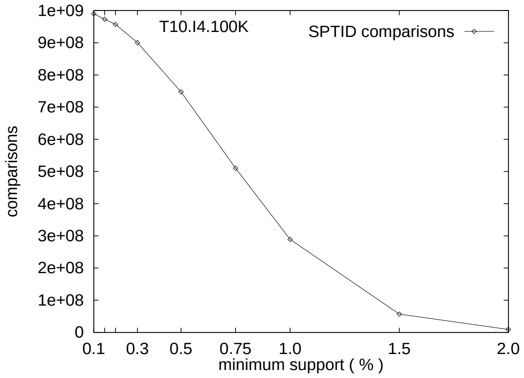


Figure 4.4: SPTID: Number of comparisons depending on minimum support

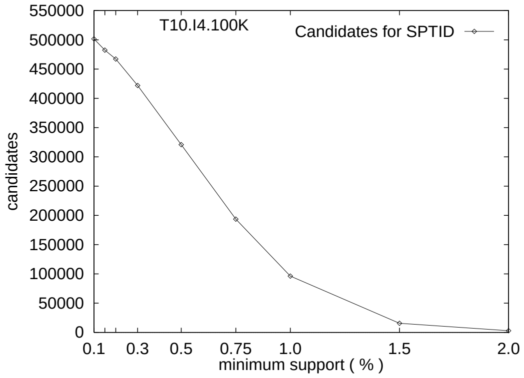


Figure 4.5: SPTID: Candidates depending on minimum support

because it still has to look at every transaction in the database.

Timing experiments have shown that SPTID spends between 80% and 90% of its running time joining TID-lists. The number of elementary integer comparisons (TID-comparisons) is shown in Figure 4.4 as performed by SPTID to join TID-lists in the course of this experiment. The number of candidates that were created for each minimum support level are shown in Figure 4.5 for comparison. The strong similarity of the two graphs suggests that the number of candidates is a decisive factor in the running time of SPTID. The reason is that SPTID has to join two lists of TIDs per candidate. We will explain and support this hypothesis further below.

4.3.2 Explanation of Performance Results

To shed more light on the causes for the differences between SPTID and the SEAR algorithms observed in the last section, Figures 4.6, 4.7 and 4.8 show measurements on a per-pass basis on a logarithmic scale. For the SEAR versions, a "pass" means one database scan, while it denotes one combined candidate generation and counting phase for SPTID. We chose a low minimum support level of $s_{min} = 0.1\%$ because the differences between both algorithms become more apparent for more difficult mining problems. Note further that all graphs use a logarithmic scale. The database (T10.T4.100K) and all other parameters were chosen as in the previous section.

Total	**SEAR**	**SEAR_plain**	**SPTID**
$s = 0.75\%$			
Hash Operations	5,278,990	5,264,380	
Comparisons			509,959,088
Time (sec)	14.5	12.6	53.4
$s = 0.3\%$			
Hash Operations	10,897,962	12,311,939	
Comparisons			900,479,343
Time (sec)	22.7	31.5	90.9
$s = 0.1\%$			
Hash Operations	18,814,116	30,110,862	
Comparisons			990,241,735
Time (sec)	38.2	43.2	109.1

Table 4.4: Total statistics for minimum support experiment

Consider Figure 4.6 which shows the number of candidates created in each pass. As would be expected, most candidates are generated for the second pass. Note that the pass bundling effect for SEAR creates more candidates than SEAR_plain in the third pass, but does not require any passes after that.

Figure 4.7 shows the time each algorithm spends in every pass. The figure in the timing graphs for pass 1 of SPTID is the time it takes to load the database and transform it into TID-list representation, which turns out not to vary significantly from the time taken by SEAR for the first pass.

The most important observation in these graphs is that SPTID spends about 90% of its time in the second pass, while the algorithm is extremely fast in all subsequent passes. In contrast, the amount of time needed by SEAR and SEAR_plain in every pass does not vary greatly. While the decreasing number of candidates in later passes would lead us to expect decreasing times for the SEAR algorithms as well, this is not the case, because the smaller number of candidates is offset by their larger size, which causes more hash operations to reach a candidate at a leaf of the prefix-tree. Again, SEAR is slower in pass 3 due to pass bundling but does not have to perform all subsequent passes.

Figure 4.8 (a) shows the number of integer comparisons for SPTID, while Figure 4.8 (b) depicts the number of hash operations performed by SEAR for every pass. Since SEAR uses powers of 2 for the size of its hash tables, computing the hash function is reduced to an AND operation and is therefore roughly comparable to the integer comparisons performed by SPTID. There is no pass 1 for SPTID because the counting of 1-itemsets is combined with the loading of the database. The graphs confirm our observations from the previous paragraphs that the amount of work done by SPTID is highly concentrated in the second pass, while SEAR spreads this work more evenly across all passes. Table 4.4 contains the total statistics for three different minimum support levels. Note the difference between the number of hash operations for SEAR and the number of comparisons for SPTID.

For SPTID, we find again the correlation between the number of candidates that have to be counted, the number of integer operations and the time spent in each pass. Note that the correlation between candidates and the amount of work cannot be observed for SEAR. This effect can be explained as follows. Consider a database of $m = 1000$ items all of which we assume to be frequent[3]. This means that all 2-combinations

[3] This is a reasonable assumption because almost all items are frequent quite often for small minimum support values as 0.1% in our example.

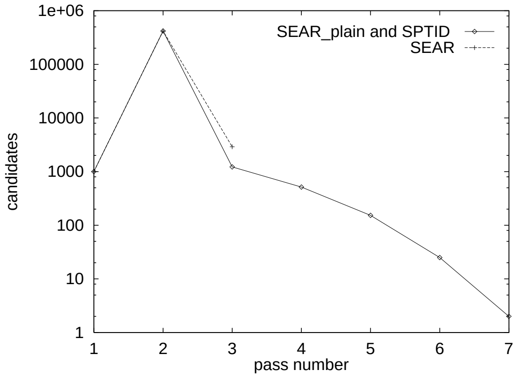


Figure 4.6: Candidates per pass
$s_{min} = 0.1\%$

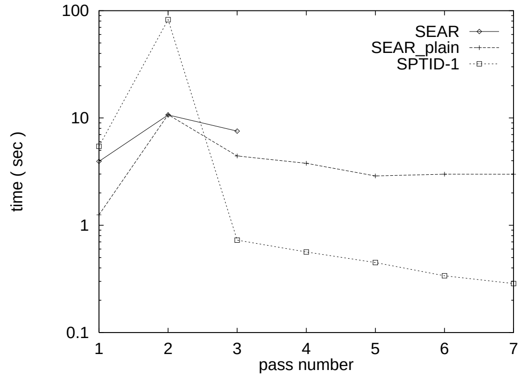
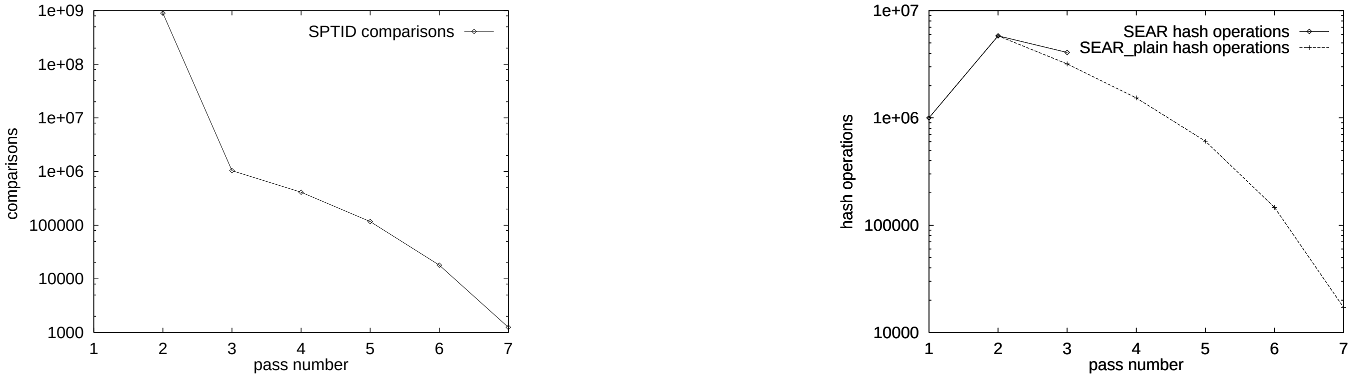


Figure 4.7: Running time per pass
$s_{min} = 0.1\%$



(a) (b)

Figure 4.8: Work per pass $s_{min} = 0.1\%$ (a) integer comparisons for SPTID (b) hash operations for SEAR

of those items, $\frac{m(m-1)}{2} \approx 500,000$ candidates have to be evaluated by SPTID in pass 2. Assume further that there are 100,000 transactions with an average of 10 items. The average length of a TID-list for a 1-itemset is therefore $100,000 \times 10/1000 = 1000$ TIDs. One merge-join to count a candidate requires as many comparisons as there are items in the longer list, thus $500,000 \times 1000 = 500$ million comparisons are necessary during pass 2. This figure is usually even larger, because the lists that are longer than average cause more comparisons than assumed here.

We can estimate the number of hash operations performed by SEAR (regardless of pass bundling) during pass 2. SEAR tries to locate every 2-subset of every transaction in the prefix tree. Again, we assume that all items are frequent, and that the transaction size follows a Poisson distribution[4] with average $\mu = 10$. Let X be the random variable for the transaction size. The average number of subsets per transaction is

$$\int_1^\infty \binom{x}{2} p\,(X = x)\; dx \;=\; \int_1^\infty \frac{x(x-1)}{2}\,\frac{\mu^x e^\mu}{x!}\; dx \;=\; \frac{\mu^2}{2} + \mu \;=\; 60.$$

Therefore, with 2 hash operations per subset, a total of $2 \times 60 \times 100,000 = 12$ million hash operations is necessary.

[4]The actual distribution is not required in the calculation for SPTID, because the error introduced by using the mean is comparatively small.

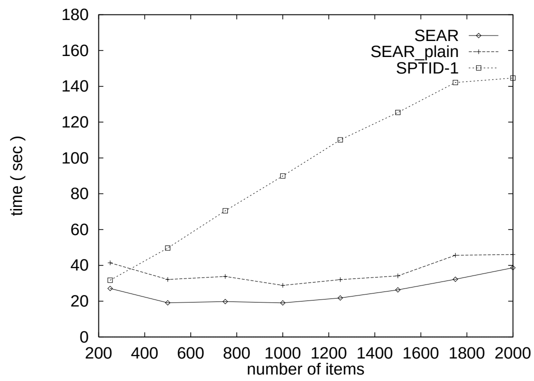


Figure 4.9: Total running times depending on the number of items

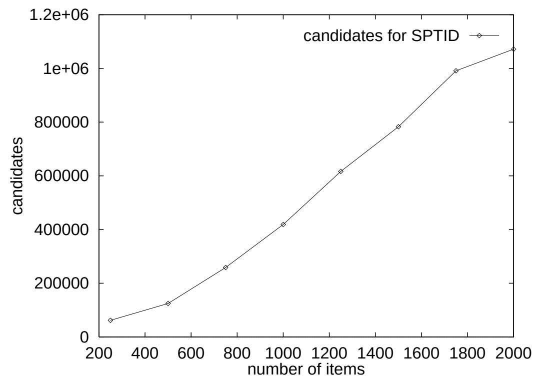


Figure 4.10: SPTID: Candidates depending on the number of items

Since the poor performance of TID-lists is confined to the second pass, TID-list-based algorithms can be optimized by counting the support for 2-candidates directly, using TID-lists only for generating candidates of size 3 and larger. This is in fact the undocumented optimization used by PARTITION. Because of this undocumented optimization we do not know how PARTITION compares to SEAR. While a detailed analysis is left to future research, rough estimates are possible, and we state these expectations at the end of this chapter, after the effects of partitioning have been discussed.

To summarize, we note that for SPTID the number of comparisons is strongly dependent on the number of candidates. This explains the increase in running time for SPTID for decreasing minimum support in Figure 4.3. The work done by SEAR, on the other hand, is not dependent on the number of candidates and therefore not as sensitive to the minimum support level.

4.3.3 Varying the Number of Items

We are now able to explain the effects that occur when the total number of items $|\mathcal{I}|$ is increased. The results of this second experiment are shown in Figure 4.9. The minimum confidence was again 0.3%, and the database size stayed constant because the number of items per transaction remained unchanged. As in the previous experiment, we show the number of candidates generated for these runs in Figure 4.10. As before, a correlation can be observed between the number of candidates and the execution time for SPTID.

The effect of increasing the number of items is that fewer frequent sets are found the larger $\mathcal{I}$ gets, which would lead us to expect the algorithms to run faster for more items. But this effect is outweighed by the increased number of candidates, most of which are created in the second pass. As explained in the last section, SPTID is highly sensitive to the number of candidates, which causes its bad performance for larger numbers of items. In fact, SPTID is only better than SEAR_plain for 200 items and always outperformed by SEAR. For larger sizes of $\mathcal{I}$, SPTID is up to 4 times slower than SEAR, and even the plain SEAR version is considerably faster for these values of $|\mathcal{I}|$.

Furthermore, both SEAS and SEAS_plain are not significantly effected by the number of items, and the slight rise in running time for greater numbers of items can be attributed to the increased effort during candidate generation.

4.3.4 Increasing the Transaction Size

The third experiment compares SPTID and the SEAR algorithms as size of a transaction is varied while the overall data size is kept constant. Figure 4.11 shows the results for this experiment for two different

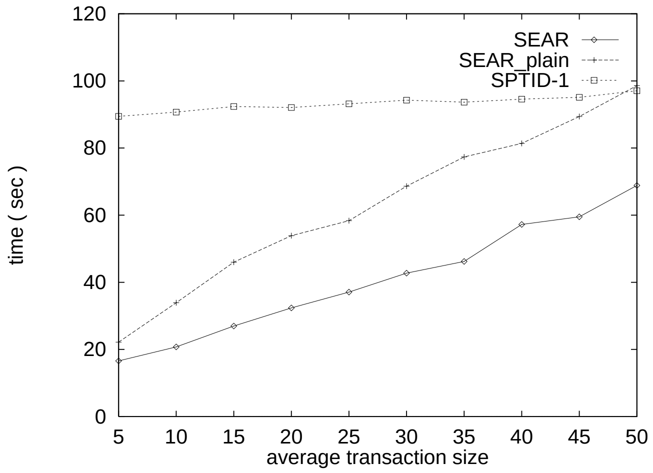


Figure 4.11: Total running times depending on the average transaction size, minimum support: 250 transactions

minimum support levels. The product of transaction size and number of transactions is kept constant at $i_t = 10,000,000$. Thus there are 200,000 transactions of average size 5 down to 20,000 transactions of size 50. This adjustment ensures that each itemset has approximately the same frequency in the database, regardless of the number of transactions. Since our definition of minimum support is based on relative frequency, maintaining a fixed minimum support throughout this experiment would produce distorted results. The reason is that, for a smaller number of transactions, the relative minimum support corresponds to fewer occurrences of a set in the databases. Therefore the number of frequent sets would increase effecting our results. Consequently, we use an absolute minimum support value expressed in the number of transactions that support a set and ensure that both the number of candidates and the number of frequent sets remains constant throughout the experiment. A rather low level of 250 transactions (equivalent to 0.25% for 100K) was chosen, all other parameters staying the same. A higher value of 750 transactions (an equivalent of 0.75% for 100K) leads to similar behavior.

For SEAR, a roughly linear rise in running time is observed as is to be expected according to the following calculation. Since $|\mathcal{D}| \times |T| = i_t$ and the work done for each transaction is approximately $\frac{|T|^2}{2}$, the overall work is described by $|\mathcal{D}| \times \frac{|T|^2}{2} = \frac{i_t}{2} \times |T|$, which is linear in the transaction size. For SEAR_plain, the slope is steeper because the additional cost for more items per transaction has to be paid for each of the additional passes necessary without pass bundling.

The work done by SPTID does not depend on the average transaction size at all, but only on the number of candidates and the average length of the TID-lists. Both values do not change in this experiment, which is why we observe the constant running time regardless of transaction size.

While the running times for SEAR stay well below the times for SPTID in both minimum support levels (250 and 750 transactions), SEAR_plain performs worse than SPTID for large transaction sizes. Clearly, SPTID will eventually perform better than SEAR if the average transaction size grows even further, but such databases are less likely to occur in practice.

4.3.5 Other Data Sets

to complete the comparison between SPTID and SEAR, we have conducted a series of experiments on other data sets (T5.I2.100K, T10.I2.100K, T10.I6.100K, T20.I2.100K, T20.I4.100K and T20.I6.100K). The minimum support level was varied form 0.1% to 2.0% like in the previous minimum support experiment. In general, SPTID performs well if both transaction sizes are small and large minimum support is high, but is clearly outperformed for most other settings by both SEAR and SEAR_plain. A small size of maximal frequent sets puts SPTID at the disadvantage of having to perform the expensive second pass without the opportunity to gain ground again in later passes where SEAR is slower. For large maximal frequent sets, however, both SEAR algorithms have the disadvantage that the prefix tree becomes very deep and all these levels have to be traversed to count the candidates at the leaves. For SEAR_plain these expensive operations are repeated every time the tree is expanded by one level while pass bundling avoids this repeated waste of effort to some extent for SEAR. In spite of this disadvantage, SEAR is at least twice as fast as SPTID for minimum support below 1.5% and all databases listed above.

4.3.6 Summary

The three main experiments conducted in this section (varying minimum support, number of items and the average transaction size) showed that the performance of algorithms using TID-lists is mainly determined by the number of candidates, which leads to inefficiencies in the second pass, where most candidates are created. Therefore, these algorithms are sensitive to the number of items and the minimum support level.

In contrast, item-list-based algorithms are not greatly effected by the number of candidates in the second pass, but rather by the number of passes, which is determined by the maximal size of frequent sets and the number of items per transaction.

The poor performance of TID-lists during the second pass can be avoided by counting 2-candidates directly. In fact, this optimization is necessary to use TID-lists efficiently. How an optimized TID-based algorithm like PARTITION compares to the item-list algorithms presented here will be determined by future research when the optimizations are implemented.

4.4 Partitioning

After investigating item-lists and TID-lists in the last section, this section focuses on the question how partitioning algorithms compare to non-partitioning, but more IO-intensive algorithms. Contrasting SEAR with SPEAR, its partitioned counterpart, SPINC, which uses incremental partitioning, and SPTID, we will see that partitioning does not fulfill the hopes for improved performance, but causes constant CPU overhead for each additional partition instead. The secondary objective is to compare the performance of the partitioning algorithms. Here, the benefits of SPINC over SPEAR were smaller than expected. SPTID remains handicapped by the inefficiency of TID-lists in the second pass (Section 4.3.2), but is included here for completeness.

To find out whether partitioning leads to improvement over the non-partitioning SEAR algorithm a set of experiments was conducted on the same data base with different numbers of partitions. Varying the number of partitions corresponds to different buffer sizes available to the partitioning algorithms to store a partition. These results are compared with the running time of SEAR.

We ran all algorithms on a larger database of 300K transactions to avoid extreme cases for many partitions when the local minimum support in a partition drops to just a few transactions. On the other hand this size can still be processed in memory, so that we could also conduct our tests on just one partition. The databases were T5.I2.300K, T10.I2.300K, T10.I4.300K and T10.I6.300K and minimum support was varied from 0.1% to 2.0% as in the previous experiments; 1000 items and 2000 maximal frequent sets were the other parameters. All partitioning algorithms have comparable memory requirements for buffering the partitions which is why we do not include various buffer sizes in the comparison but focus on the differences that appear when the number of partitions is increased. This means in particular, that that SPTID does not swap intermediate results to disk[5].

Relevance of IO

Since reducing IO cost is one important objective of partitioning algorithms, a few remarks on this issue are in order before we present the performance results. Our experiments show that for SEAR, disk IO accounts for at most 20% of the running time for high minimum support levels with little computation and as little as 5% for low minimum support. The percentages for the partitioned algorithms are even smaller, with a minimum of 2% for the most compute-intensive runs. Recall that all IO operations are sequential scans of the database which are much faster than random accesses. Thus, all algorithms are CPU-bound and IO has comparatively little influence on their performance. For this reason, saving IO at the expense of additional computation does not lead to improvements, as we will see shortly when comparing partitioning and non-partitioning algorithms. Since the amount of computation required per transaction is constant, both IO and CPU cost increase at the same rate as the database size grows, so the fraction of time due to IO does not change.

For SPTID, bypassing the second pass clearly reduces the amount of computation, so one could suspect that IO would become much more important. This is not likely, however, as we can infer from the timing results per pass for SPTID-1 shown in Figure 4.7. Assuming that the optimization reduces the cost of the second pass from 82 to 10 seconds, which is a lower estimate, the total running time is 18 seconds. The database is only read once at an IO rate of 6 MB/sec measured in initial experiments, which means that all 4.4 MB are read in less than 1 second, in other words, in about 6% of the running time. For more than one partition, the database is read twice, but the additional counting phase and the partitioning overhead increase the amount of computation. Hence, we can expect SPTID to be CPU-bound in our environment, even with the bypass optimization.

Performance Comparisons

First, the number of partitions is set to 1 to assess the results of reducing the number of IO operations by buffering the entire database, we then increase the number of partitions to examine the partitioning overhead.

Figure 4.12 shows the typical behavior when only one partition is used, i.e. when the entire database fits in memory. The partitioning algorithms only read the database once and work in memory for later passes, while SEAR is deliberately put at the disadvantage of having to read the entire database from disk in each pass. The object of this experiment is to quantify the effect of this disadvantage. Furthermore, although the database would easily fit into memory, SEAR cannot not expect this for larger databases.

The observation common to all data sets we tested is that SPTID-1 falls far behind for low minimum support values (as was seen previously), while all prefix-tree-based algorithms perform equally well. This is in line with our expectations that IO is of minor relevance for the execution time. Moreover, SEAR requires only 3 passes due to pass bundling, and in terms of computation there are no basic differences between SEAR, SPEAR and SPINC in the 1-partition case.

The results when 2 and 5 partitions are used on the T10.I6.300K database are shown in Figure 4.13. This database was chosen because the differences between the algorithms are greater due to the high CPU cost for large maximal frequent sets — all experiments on other databases produced similar, but less dramatic results. All partitioning algorithms perform worse than the non-partitioning SEAR indicating that additional overhead is associated with partitioning. In particular, this overhead would offset any possible savings resulting from fewer IO operations.

[5] It would have been very difficult to make sure that these accesses actually involve the disk, because of irregular access patterns, and we did not want to complicate the issue any further.

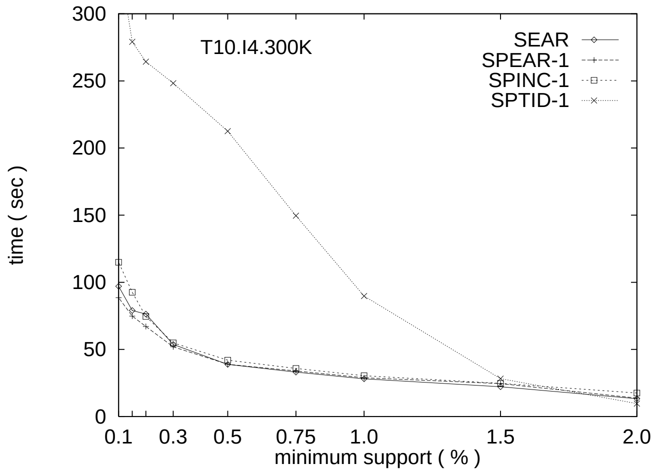
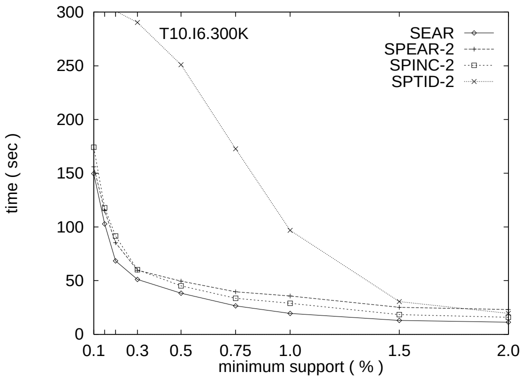
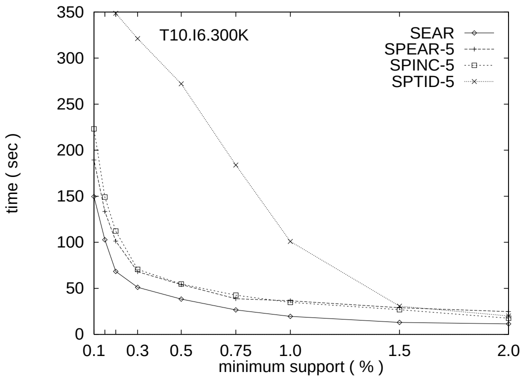


Figure 4.12: Running times of all algorithms for 1 partition



(a)



(b)

Figure 4.13: Running times for SEAR and the partitioning algorithms with (a) 2 partitions and (b) 5 partitions

As the number of partitions grows, the gap between SEAR and the partitioning algorithms widens, which is the result of the additional overhead for partitioning. This additional time is in part due to the additional counting phase, which becomes necessary for more than 1 partition, and in part due to candidate generation which has to be repeated for every partition. Table 4.5 shows the time spent during candidate generation for SEAR and SPEAR as the number of partitions grows. The times for SPINC are comparable to the ones for SPEAR.

For two partitions the incremental version displays noticeable improvements over its non-incremental counterpart SPEAR. The reason is that SPEAR processes the entire database twice, once to generate all possible frequent sets, and the second time to count the global support for all those sets. In contrast, SPINC avoids counting the second partition a second time, thus processing the database only 1.5 times. This puts SPINC in the middle between SPEAR and SEAR that only processes the database once (although this involves several scans of the data).

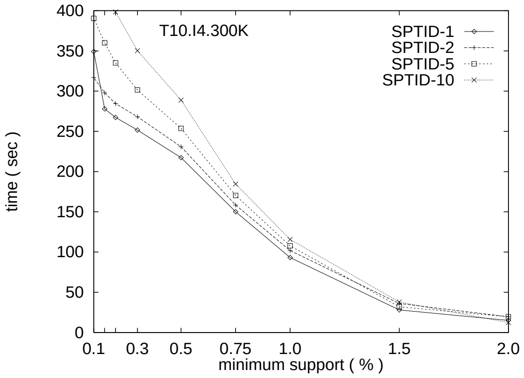
The performance of SPINC for 5 partitions is rather disappointing: while still inbetween SEAR and SPEAR for higher minimum support levels, this improvement over SPEAR is less significant than for 2 partitions, because the partition size gets smaller the more partitions there are. Hence, the savings achieved by not having to process the last partition become less important. As noted when we described SPINC,

	$s_{min} = 0.75\%$	$s_{min} = 0.3\%$
SEAR	1.6	5.9
SPEAR-1	1.6	3.6
SPEAR-2	2.5	6.2
SPEAR-5	5.7	13.9
SPEAR-10	10.8	27.7

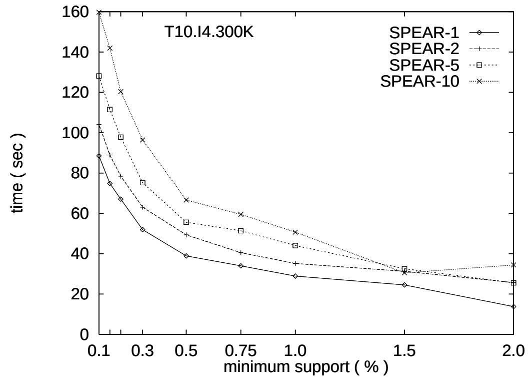
Table 4.5: Time for candidate generation(in sec)

lowering the minimum support for the first phase might improve the performance of the algorithm, but we do not pursue this idea here. Furthermore, SPINC is always outperformed for more demanding problems. This effect, which was also present (although not as dramatically) for 2 partitions, is caused by the increased overhead required to manage old frequent sets and candidate sets in the tree.

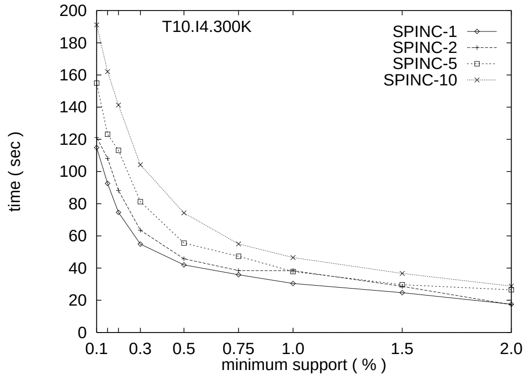
The slowing effect of an increased number of partitions for all partitioning algorithms is shown in Figure 4.14. The graphs show that all algorithms are adversely effected.



(SPTID)



(SPEAR)



(SPINC)

Figure 4.14: Times for different numbers of partitions

In summary SPTID is comparable to the prefix-tree algorithms only for very high minimum support parameters, regardless of the number of partitions. Furthermore, partitioning does not show the expected improvements due to reduced IO, but rather, the overhead due to several partitions outweighs any IO savings. Finally, the incremental SPINC achieves savings over the simple partitioned version for high minimum support values, but the gains become less noticeable the more partitions are processed.

4.5 Summary

Comparing TID-lists and item-lists, our results indicate that SEAR is superior to SPTID with the exception of a few marginal cases that rarely occur in practice. Even SEAR_plain without the pass bundling optimization was found to be better in most cases. We explained the influence of parameters on the performance of both algorithms, most importantly SPTID's sensitivity to large numbers of candidates that causes it to be twice as slow as our SEAR algorithm on the average. The weaknesses of the TID-list representation in the second pass was identified as the major cause for the poor performance of SPTID. SEAR, on the other hand, depends much more on the average transaction size than SPTID.

All partitioning algorithms performed worse than SEAR; in particular, SPEAR and SPINC, which should be comparable to SEAR because they all make use of the prefix-tree structure, did not show the expected improvements due to reduced IO. This is in keeping with our observation that IO accounts for only between 2% and 20% of the running time, and thus all algorithms are CPU-bound. The overhead due to additional partitions slowed down all partitioned algorithms even more. While partitioning algorithms might be justified in environments with extremely high IO cost or with a high priority to keep IO loads low, we did not find them useful in our (admittedly high-end) environment.

We showed that pass bundling reduces both IO and CPU overhead significantly for item-list based algorithms. Therefore pass bundling is an alternative to partitioning. However, the question remains how item-lists with prefix-trees and pass bundling compare to TID-lists, if the expensive second pass is avoided by counting support for 2-candidates directly. Since TID-lists are superior to the item-list algorithms in later passes, it seems likely that SEAR is outperformed by this TID-list algorithm. On the other hand, pass bundling greatly reduces the number of passes, while TID-lists require partitioning large databases and incur the overhead associated with it. Furthermore, short-cutting the second pass causes some additional overhead. Therefore, item-lists may very well outperform TID-lists, but definitive answers to this question require future research.

Finally, the performance of the incremental SPINC was disappointing in that the savings achieved over SPEAR are not significant for greater numbers of partitions. For more difficult problems, SPEAR was found even to outperform SPINC, which is slowed by the overhead of managing different sets in the prefix-tree.

Chapter 5

Parallel Algorithms

5.1 Programming Environment and Data Distribution

We assume a shared-nothing parallel database architecture where each processor is equipped with its own main memory and its private disks. Remote data can only be accessed through the processing node that owns it. The SP2[1]([38, 24]) distributed memory/message-passing architecture suits this paradigm well because of the clear distinction between local and remote data. Communication has to be included explicitly by the programmer and does not occur automatically as the result of a memory access to a page that happens to be located on another processor, as is the case with shared memory architectures. Message passing allows us to control nature and extent of communication, which is beneficial in our case, as the results in Chapter 6 demonstrate. The SPMD programming model provided on the SP2 requires the same program to run on all processor nodes, but, in contrast to SIMD, different instructions can be executed concurrently. In fact, the programs run independently unless communication is desired that may act as a barrier for the processors that participate in it. Figure 5.1 shows a 16-node SP2 switch.

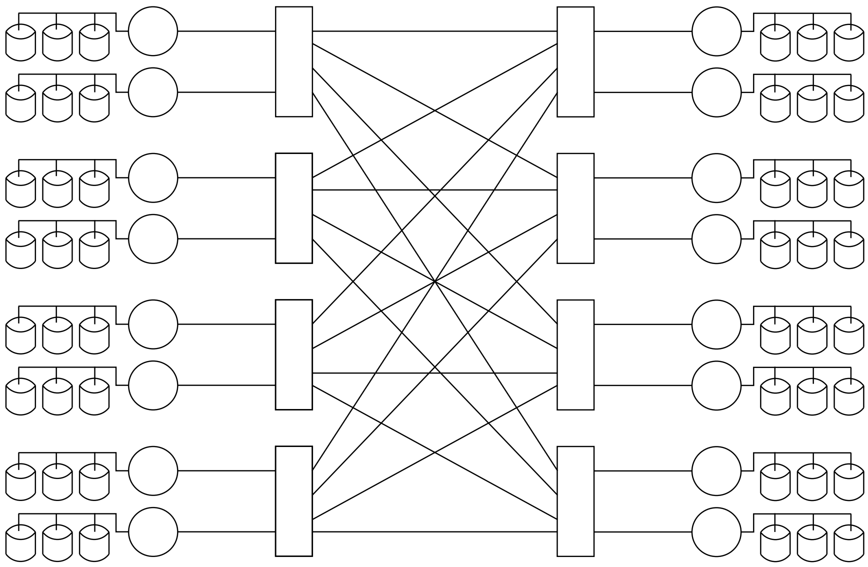


Figure 5.1: SP2 High Performance Switch

The transaction database is assumed to be partitioned horizontally between a set of p processing nodes,

[1]Scalable POWERparallel System, Trademark by IBM.

Algorithm 5.1 *PEAR*

$C_1 = \{\{i\} | i \in \mathcal{I}\}$ $\{$*all 1-itemsets*$\}$
$k = 1$
while $(C_k \neq \emptyset)$ **do**
  foreach processor $i = 1, \ldots, p$ **do in parallel**
    $\{$*count local support for candidates*$\}$
    forall transactions $T \in \mathcal{D}^{(i)}$
      forall k-subsets $t \subseteq T$
        if $(\exists c \in C_k : c = t)$ **then** $c.count^{(i)}++$
    $\{$*compute global support for all candidates*$\}$
    foreach $c \in C_k$
      $c.count = \sum_{i=1}^{p} c.count^{(i)}$ $\{$*parallel combine*$\}$
  end parallel
  $L_k = \{c \in C_k | c.count \geq n\, s_{min}\}$
  C_{k+1} =generate_candidates(L_k)
  k++
return $L = \bigcup_k L_k$
end

and all partitions have approximately the same number of transactions to avoid load imbalances. The data should also be assigned to processors randomly to couter possible load imbalances, for example by hashing on TID.

Apart from these restrictions, the data to be processed in parallel are stored on disk as in the sequential case. No special data structures are used. We assume that every processor knows the number of distinct items in the database, the number of transactions assigned to it and the average transaction size that is needed to determine the partition size for partitioning algorithms.

5.2 PEAR: The Parallel SEAR Algorithm

5.2.1 Algorithm Description

Recall that SEAR alternates between candidate generation and support counting steps, a pair of which we call one pass. Counting support for candidates can easily be done in parallel, each processor evaluating its private data. In contrast, the decision of which candidates to accept as frequent and which to discard cannot be done correctly without knowledge of the global support for these candidates, which has to be computed by adding all the local support counts from every processor. Consider the code for PEAR, the parallel version of SEAR, in Algorithm 5.1 that shows how SEAR can be augmented with communication to run in parallel. Statements that are not within the *in parallel ... end parallel* block are the "sequential" parts of the algorithm which means that all processors perform exactly the same operations on identical data. The data on node i is referred to as $\mathcal{D}^{(i)}$ and should not be confused with partitions $\mathcal{P}_i$ in the sequential case.

The algorithm starts with the set of 1-candidates (which can easily be created locally without any communication from the number of items in the database). Then each processor counts the support for these candidates individually in its portion of the database. The superscript (i) for count-variable $c.count^{(i)}$ refers to the processor number and indicates that the variable contains different values for each processor.

After the counting step, a global *combine* operation that adds all the local counts is used to determine the global support of all candidates. The result is distributed to all processors, so that the global count for every candidate is available to every processor. Then each node can continue to compute L_1, the set of all frequent 1-itemsets, by removing all infrequent candidate sets, which completes the first pass. Note that every processor eliminates exactly the same candidates and comes up with the same set L_1. Therefore all nodes also create the same candidate set C_2 in the next pass. This procedure continues until an empty set L_k indicates that no more frequent sets are found.

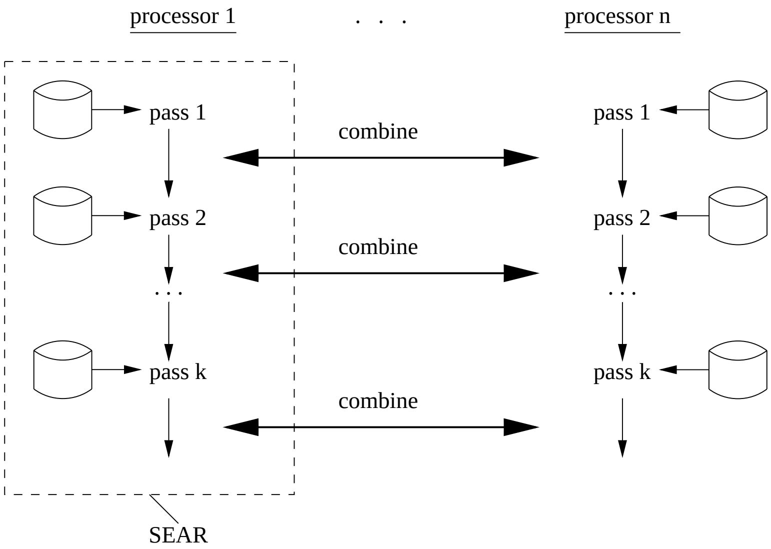


Figure 5.2: PEAR algorithm scheme

Essentially, the program running on the individual processors is the same as SEAR, with the exception that after every pass (candidate generation and local support counting) the counts from each processor are added to be able to proceed correctly to the next pass. This scheme is depicted in Figure 5.2.

The *combine* is the only communication that is necessary to compute all frequent sets. It has the effect of a global barrier, because no processor can continue with the next pass until it has the counts to generate the next set of candidates. Therefore, the amount of work each node has to perform during one pass has to be roughly balanced to make sure that excessive idle times are avoided.

The actual association rules are not computed in parallel but can be determined sequentially from all frequent sets on any one of the nodes.

5.2.2 Implementation

The Message Passing Library (MPL)[23] *combine* operation, which is used to compute the global counts, accepts vectors for addition also, in which case the corresponding elements are added and their sum returned to every processor as shown in Figure 5.3. To make efficient use of this facility, we keep all local counts in an array and store an index into this array with each candidate-node in the prefix-tree. The *combine* is performed on this integer-array.

If a candidate is found to be frequent after the *combine*, the index in the tree is replaced by the actual support count, and the memory used for the array can be freed. It is important to release this memory, because the large number of candidates can lead to huge buffer requirements. For a (quite reasonable) number of 400,000 candidates and 4 Bytes to one integer, the array size is 1.6 MB. Furthermore, the *combine* does not work in-place, but requires separate input and output buffers. Therefore, a huge *combine* is split into several smaller *combine* operations to allow the use of a smaller output buffer.

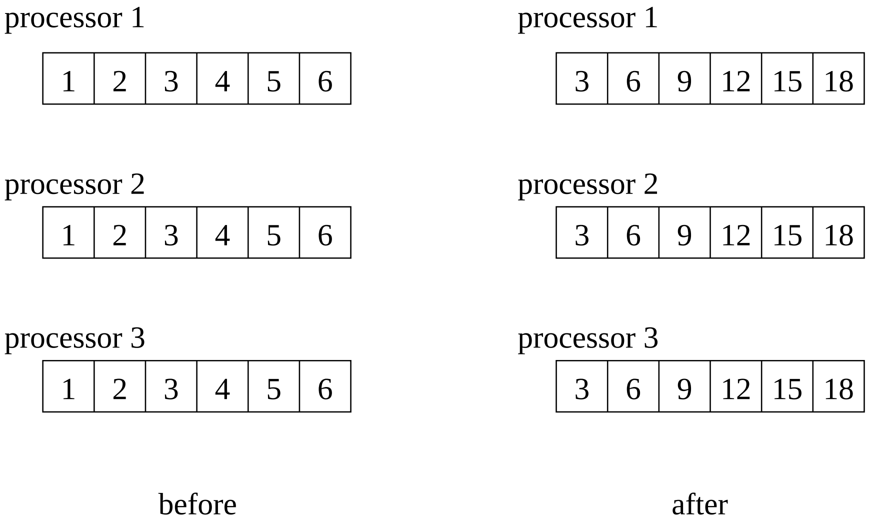


Figure 5.3: Example of an add-combine operation for an array of 6 integers

5.3 PPAR: The Partitioned Parallel Algorithm

A potential problem with PEAR is that it needs large *combine* operations on many count variables. These *combines* function as global barriers that reduce efficiency because processors with a smaller load have to wait idle.

The partitioning approach allows a reduction in both the overall amount of communication and the number of synchronizations, as we describe in this section on PPAR, the parallel version of SPEAR. This paralel version of a partitioned algorithm was briefly outlined along with the PARTITION algorithm in [34].

5.3.1 Algorithm Description

SPEAR processes each partition independently; their order is not relevant, and results from one partition are not needed in another. Therefore, the straightforward approach, and also the one we choose here, is to consider the data given to a processor as one partition. All local frequent sets are determined in the first phase, and "local" means "local to each processor" in this case. Let $L^{(i)}$ be this set on processor i. No communication is required at all during this phase. Then, to form the global candidate set C^G, the union of all $L^{(i)}$ has to be computed across processors in a second phase. The result is broadcast to all processors. The third phase corresponds to phase II of the sequential algorithm where each processor has to determine the local support for all sets in C^G. As in SPEAR, sets do not have to be counted again, for which global counts are available from Phase I already. As in phase I, no communication is needed in this phase. Finally, during phase IV, the global support counts for all sets in C^G are coalesced with a *combine* operation, just as the candidate counts were combined by PEAR. After discarding all globally infrequent sets, the final result is available. The algorithm is outlined in Figure 5.4

The explanation so far has assumed that all of the data local to one processor fits into its memory and can be processed as one partition. This is usually not the case, and data on each processor has to be partitioned again, which we call *local partitioning* (as opposed to *global partitioning* amon processors). These local partitions are processed in sequence, and phase I in the parallel case is in fact identical to phase I in SPEAR, with the only difference that it runs on each processor independently. See Algorithm 5.2[2] for the

[2]Note that for clarity this code does not include the removal of candidates in phase III that don't have to be counted any

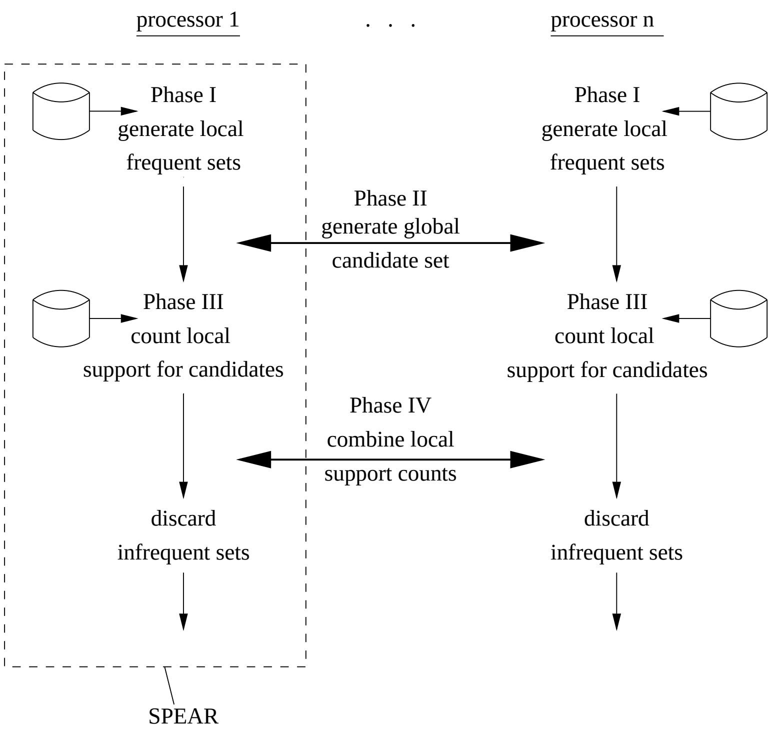


Figure 5.4: PPAR algorithm scheme

pseudo-code for this general case.

PPAR has only two phases that involve communication, and thus only two barriers that may cause performance degradation. Furthermore the size of the messages sent during both phases is smaller than the *combines* used by PEAR. The reason is that PEAR needs to exchange counts for candidates, while PPAR only exchanges counts for locally frequent sets, the number of which is only a fraction of the number of candidates.

Phase I and phase III are implemented in a manner similar to the corresponding phases in SPEAR and we don't recount the details of the implementation. While the *combine* in phase IV is straightforward and identical to the *combines* used in PEAR, the union of locally frequent sets in phase III is more interesting, which is why we use the following section to explain our solution.

5.3.2 Computing the Union of Locally Frequent Sets

Combine operations can be used as long as all processors know which array position belongs to which set. Then adding corresponding array elements adds the correct local support values. This is the case for PEAR, because all processors always create the same number of candidates in the same sequence. This is also the case for phase IV in PPAR, because every processor knows C^G and can determine the rank of every set in C^G.

This is not the case, however, for PPAR in phase II. Each processor has a different set of locally large sets that is determined directly by its share of the data and the frequencies of itemsets therein. In fact, the whole purpose of this phase is to establish a common understanding among the processors as to which sets have to be counted eventually.

---

more.

Algorithm 5.2 *PPAR*

{*Phase I — generate locally frequent itemsets*}
foreach processor $i = 1, \ldots, p$ **do in parallel**
    $\mathcal{P}$ = compute_partition($\mathcal{D}^{(i)}$)
    forall partitions $\mathcal{P}_j \in \mathcal{P}$ **do**
        $L_j^{(i)} = find\_frequent\_sets(\mathcal{P}_j)$
    $L^{(i)} = \bigcup_j L_j^{(i)}$
end parallel

{*Phase II — compute global candidate set*}
$C^G = \bigcup_{i=1}^p L^{(i)}$
Broadcast C^G to every processor

{*Phase III — local support for global candidates* }
foreach processor $i = 1, \ldots, p$ **do in parallel**
    foreach $c \in C^G$
        $c.count^{(i)}$ =count_support(c)
end parallel

{*Phase IV — global support for global candidates*}
foreach $c \in C_k$
    $c.count = \sum_{i=1}^p c.count^{(i)}$ {*combine counts*}

return $L^G = \{c \in C^G | c.count \geq n\, s_{min}\}$
end

Recall further that all sets of sets are stored in prefix trees, and therefore the $L^{(i)}$ are also represented as trees on each processor. Our solution is to linearize the tree according to a scheme that uses exactly two integers per frequent set (PEAR needs 1 integer per candidate). To form the union of two sets of sets, one processor linearizes its tree and sends it to the other node that compares the input with its own set and updates it accordingly. If more than two nodes are involved, the recipient creates the linearization of its updated tree on-the-fly and sends it to the third node (and so on). Thus every node receives the union of the sets on all its predecessors, and the last node has the complete union.

This procedure is linear in the number of nodes involved and would be rather inefficient if a node could only start once its predecessor is finished. Therefore we adjust the message size such that processing can be overlapped in pipeline fashion. Figure 5.5 illustrates the technique for 4 nodes. Communicating the result to all nodes can be done easily with a series of broadcasts as shown in the Figure.

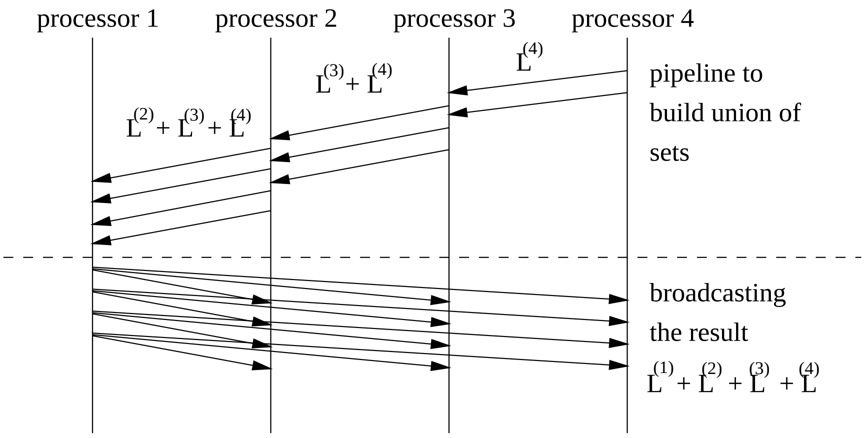


Figure 5.5: Union of sets of sets across processors

Chapter 6

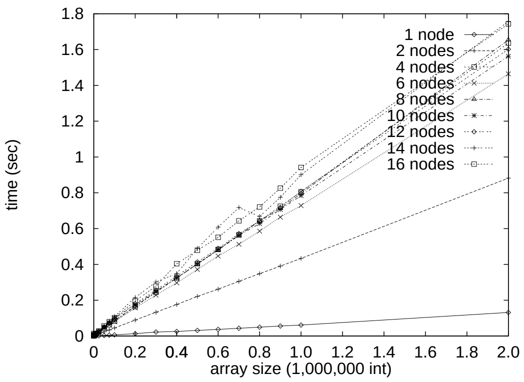
Parallel Experiments

In this chapter we report the results of the experiments with our parallel algorithms PEAR and PPAR. We show that both algorithms display near-optimal speed-up, scale-up and size-up behavior. These results are due largely to the fact that communication overhead is small for both algorithms, moreover, it is constant with respect to the number of transactions and varies only insignificantly with the number of processors involved. Therefore it is not surprising that the partitioned PPAR algorithm performs worse than PEAR although it needs less communication.

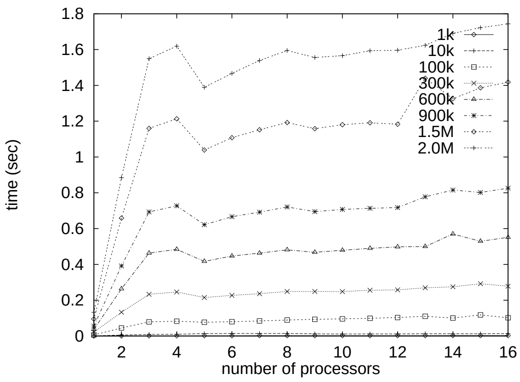
Before we present the results in detail, we show the results of initial experiments with the *combine* operation which is the most important communication operation for PEAR. All experiments were conducted on a 16-node SP2 multiprocessor equipped with a 66 MHz RS/6000 processor, 64 MB of main memory and 2 GB SCSI disks. The data came from only one of the disks.

6.1 Cost of the Combine Operation

Figure 6.1 shows the results of tests we conducted to asses the efficiency of the MPL *combine* operation for integer vector addition. Recall that this operation can be performed on arrays of arbitrary size, as long as sufficient buffer space is provided. The number of processor of processors was varied from 1 to 16, the number of integers in the array ranged from 1,000 to 2 million. Figure 6.1 (a) shows the timing results depending on the array size, Figure 6.1 (b) shows the same numbers depending on the number of processors. The times were obtained as an averages of 5 runs; the variance of the individual times was not significant.



(a)



(b)

Figure 6.1: Times for parallel combine operation (a) depending on array size (b) depending on the number processing nodes

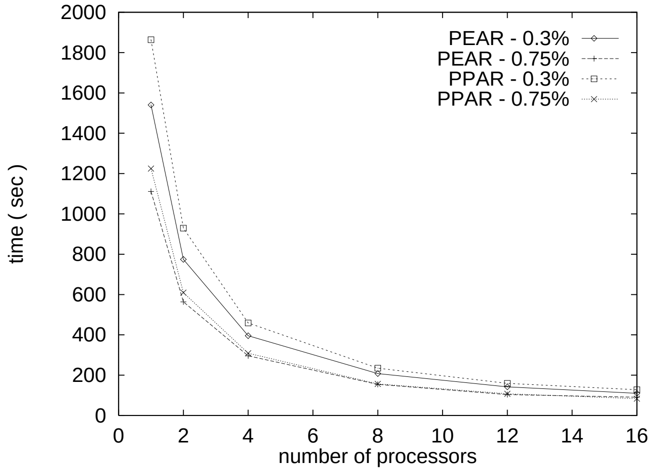


Figure 6.2: Running times for various numbers of processors on T10.I4.10M

Three observations are important in our context. First of all, the operations are extremely fast, considering that the values from each processor have to be added and the result needs to be distributed again. Secondly, Figure 6.1 (a) shows nicely that the *combine* is linear in the number of array elements, and a rate of 1.1 million elements can be processed per second for more than 3 processing nodes, at about twice the rate for 2 processors. Note that the time for 1 processor is basically the time needed to copy the data from the input buffer to the output buffer. Finally, as shown in Figure 6.1 (b), the time required for a combine does not vary significantly with the number of processors, as long as this number is more than 2.

The importance of these results for our algorithms is that we can expect low communication times that are unlikely to vary with the number of processors involved.

6.2 Speed-Up Experiments

We ran both algorithms on the T10.I4.10M data set, increasing the number of processors. Minimum support was set to 0.75% and 0.3%; the size of the data set was 440 MB. As for all other experiments in this chapter, the buffer size for PPAR was set to a constant amount of 10 MB per node even when the number of processors varies. Figure 6.2 shows the resulting running times, indicating that PEAR is superior to PPAR for more demanding low minimum support values. PPAR is slowed by the local partitioning overhead, which offsets the benefits of reduced communication due to global partitioning. Of course, the times for 0.75% are less than for the lower minimum support level of 0.3% for both algorithms. This does not change as the computation is distributed among an increasing number of nodes, indicating that both algorithms scale well to larger numbers of processors. The speed-ups obtained by both algorithms in this experiment are shown in Figure 6.3. Both algorithms show almost linear speed-up. The exact values can be found in Table 6.1.

	s_{min}	4 Nodes	12 Nodes	16 Nodes
PEAR	0.75%	3.7	10.7	12.2
PEAR	0.3%	3.8	10.8	14.0
PPAR	0.75%	3.9	11.3	14.6
PPAR	0.3%	4.0	11.7	14.5

Table 6.1: Exact speed-up numbers on T10.I4.10M

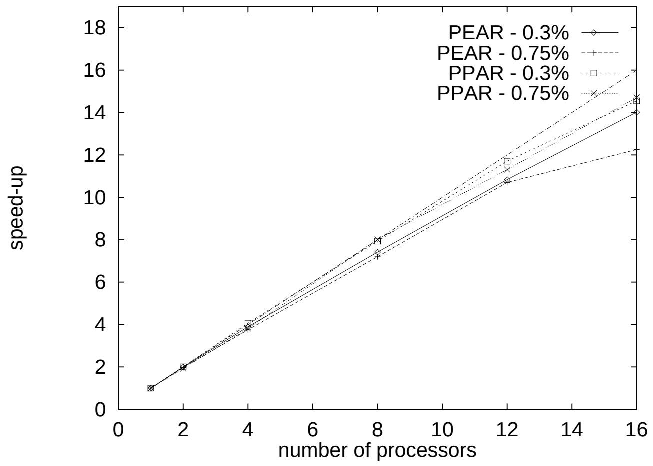


Figure 6.3: Speed-up for PEAR and PPAR on T10.I4.10M

The deviation from the optimal linear speed-up curve is due to the both the "sequential" portions of the algorithm and the communication overhead. The sequential parts of the algorithm are those that are performed by each processor on exactly the same data, (i.e., the candidate generation step for PEAR). These parts of the algorithm are not done in parallel, and their relative portion of the running time grows as the number of processors increases. So does the time wasted due to communication. It is not so much the actual communication that slows the algorithm but the time spent waiting because the communication acts as a barrier.

A first observation from Figure 6.3 is that PPAR obtains better speed-ups than PEAR. One reason is that PEAR spends more time on communication (an average of 10 seconds as opposed to PPAR's 7 seconds for 0.3% minimum support and less than 5 seconds for 0.75%). The different times for communication are only in part caused by the reduced amount of communication as listed in Table 6.2 (recall that PPAR only needs to communicate support for frequent sets while PEAR does the same for all candidates). The difference in communication time is mainly caused by the fact that PPAR has only 2 communication operations, as opposed to 3 for PEAR, and spends less time waiting at these barriers.

s_{min}	**PEAR**	**PPAR**
0.75%	769,088	10,880
0.3%	1,681,140	82,080

Table 6.2: Communication sizes for PEAR and PPAR (in bytes)

Another reason for the better speed-ups achieved by PPAR lies with the fact that it repeats candidate generation for every local partition. Candidate generation is therefore dependent on the number of local partitions (and thus on the size of the local data), which causes the time for this task to decrease as the number of processors grows, contributing to higher speed-ups. In contrast, the time spent for candidate generation by PEAR remains constant throughout the experiments. Ironically, the repeated effort for candidate generation due to local partitioning is also the reason why PPAR is outperformed by PEAR for low minimum support.

The second observation is that higher minimum support values achieve worse speed-ups. The speed-ups in Table 6.1 are mostly lower for 0.75% than for 0.3% because the amount of computation is smaller, while communication overhead stays approximately the same regardless of the minimum support level. This is also the reason for the comparatively low speed-up of PEAR for 0.75% on 16 nodes.

Note that for PEAR both communication overhead and the sequential parts are constant with respect to

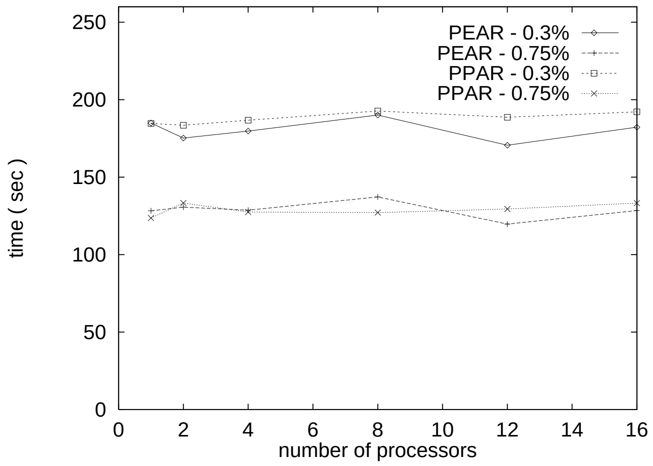


Figure 6.4: Scale-up for PEAR and PPAR T10.I4, 1M transactions per node

the number of transactions. Running larger problems will therefore raise the speed-up for this algorithm.

To summarize, we note that both algorithms obtain good speed-ups that improve as the size of the data base and the problem difficulty increases. PPAR obtains better speed-ups, but performs worse than PEAR.

6.3 Scale-Up Experiments

The next experiment examines the behavior of the algorithms when the data size is increased along with the number of processing nodes. Minimum support levels were 0.75% and 0.3% again, the database was T10.I4 with 1 million transactions (44.0 MB) on each node. Figure 6.4 shows the running times for 1 to 16 processors.

PPAR is given 10 MB of buffer space, which causes the database to be processed in 5 local partitions and leads to slightly worse performance than PEAR.

Both algorithms display near-optimal scale-up behavior as the number for processors increases. Running times are even slightly less for greater numbers of nodes, which we attribute to characteristics of this particular data set.

6.4 Size-Up Experiments

Our last experiment, shown in Figure 6.5, investigates the effects of increasing the number of transactions while keeping constant the number of processors. This experiment was conducted on all 16 processors at minimum support levels of 0.75% and 0.3%. We mentioned earlier in Section 6.2 that growing problem sizes reduce the portion of the running time due to communication, which leads to the expectation that our algorithms "size up"[1] well, i.e. running times are linear in the problem size. Figure 6.5 confirms this expectation for a range from 500,000 to 40 million transactions. All other parameters were chosen as above.

[1] In addition to speed-up and scale-up, this term is used in [12] as third criteria to evaluate parallel database performance for the case when the number of processors remains constant while the problem size is increased.

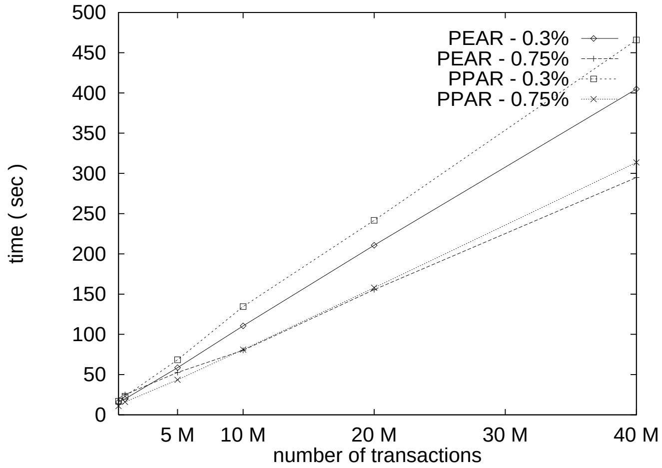


Figure 6.5: Size-up for PEAR and PPAR   500K to 40M transactions on 16 nodes

Chapter 7

Extensions and Other Related Work

The following sections outline the current work on concept hierarchies and parallel classification, which is not directly related to the subject of this report but belongs in its context. Especially the first section on Episode Discovery is interesting in that it demonstrates how ARM algorithms, given some modifications, can be applied to a completely different problem.

7.1 Discovering Episodes in Sequences

7.1.1 Episodes in Sequences

Surprisingly enough, discovering events in sequences that frequently occur together, i.e. within a given time interval, can easily be mapped to the ARM problem. Let $\mathcal{I}$ be the set of event types and let a "transaction" contain all the events that occurred within a time window of fixed length. In Figure 7.1, the transaction corresponding to the window starting at 2 would be $\{A, B, C\}$. The window has to slide over the sequence to create the set of transactions. After this simple preprocessing step, all ARM algorithms could be used to solve this problem without further modification.

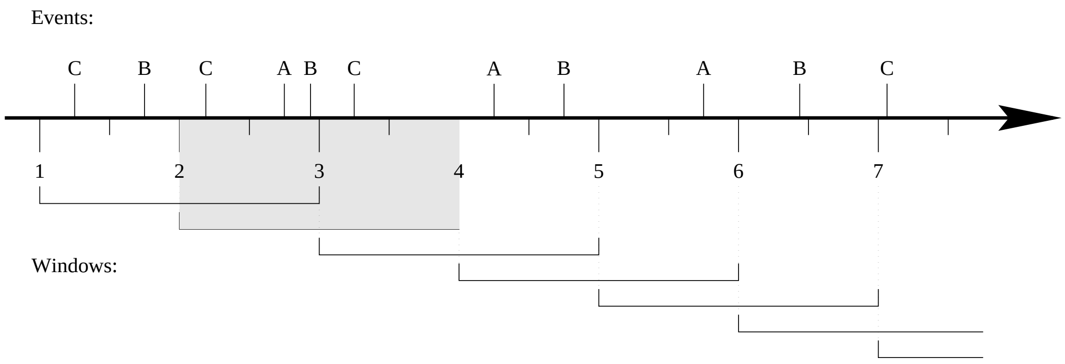


Figure 7.1: Events and Windows in Episode Discovery

Unfortunately, as simple as this solution may be, it does not take into account the most important characteristic of sequences: the partial ordering of events, i.e. that events may occur before or after others or in parallel. Also, several occurrences of the same event within one window are not considered, as for example event C in our example window.

For these reasons, [29] define an *episode* to be a partially ordered multi-set of events to capture the time relation between events. Events that are not ordered with respect to each other are considered parallel. The episodes in our example window are $\{A\}$, $\{B\}$, $\{C\}$, $\{C \| A\}$, $\{C \| B\}$, $\{C \| C\}$, $\{C < A\}$, $\{C < B\}$,

$\{C < C\}$, $\{A \| B\}$, $\{A < B\}$, $\{A < C\}$, $\{B < C\}$ and numerous sequential and parallel compositions of these like $\{A < (B \| C)\}$ or $\{A < B < C\}$. To model the recursive nature of episodes, episodes can consist of other sub-episodes which can also be partially ordered with respect to each other.

An episode is *frequent*, if it can be found more often than a predefined threshold. Similar properties hold for frequent episodes and frequent itemsets, for example,that an episode can only be frequent if all its sub-episodes are frequent.

Similar to association rules, prediction rules $E \rightarrow F$ can be constructed that predict the occurrence of an episode F if E took place. The confidence of such rules is defined in a manner similar to association rules.

7.1.2 Algorithm

Like the ARM algorithm, the algorithm presented in [29] alternates between candidate generation and counting steps. Candidate episodes are initially built from elementary events; the support is then counted, the frequent episodes are used to create new candidate episodes, and so on.

Candidate episodes are created in the same way candidate sets are built by Apriori-gen. Two episodes of size k that share $k-1$ sub-episodes (atomic or compound) are joined to form a $(k+1)$-candidate. The frequency of all other k-sub-episodes is then tested as a mandatory prerequisite.

Counting support for episodes is more tricky, because episodes can be present in the window more than once, because the detection of sequential episodes requires finite state automata and because of the recursive structure of episodes. Furthermore, the task is complicated by optimizations based on the fact that adjacent windows do not differ greatly and the episodes from the previous window are likely to be part of the next.

Results show that a sufficiently small number of candidates is generated for the algorithm to be effective, and savings of up to a factor of 20 can be obtained by using incremental episode counting.

7.2 Concept Hierarchies

Concept hierarchies, or taxonomies, have been mentioned before as one way to use domain knowledge to either speed up the mining process or improve the quality of its results. Usually a taxonomy is provided by domain experts or may even be determined in a very straightforward manner as in the case of geographical data. If neither is feasible, methods are available to create concept hierarchies from data automatically and to modify existing hierarchies to suit the current mining task if for example only a part of the data is being examined [18]. This is necessary when the current structure is too general, specific or unbalanced and thus causes distorted results.

Concept hierarchies have been used in classification mining before, the most prominent example of which is attribute oriented induction that is realized in the DBLEARN system [7, 17]. Here aggregate relations are built successively by replacing values by their ancestors in the hierarchy.

In the context of association mining, concept hierarchies can be viewed as directed acyclic graphs, the nodes of which are labeled with the literals from $\mathcal{I}$. Compared to the definition given in the last Section 2.1.1, $\mathcal{I}$ is augmented with literals for higher level concepts; for example, in addition to basic items like SkimMilk and LowFatMilk, Milk and Beverage are part of the universe of “items”. An edge between two nodes in the graph represents the *is-a* relationship between them. The notion of ancestors and descendants of items is defined in terms of edges in the transitive closure of the taxonomy graph. Sets and rules in which one or more items were replaced by their ancestors are called antcestors of the original set or rule respectively.

In the presence of multiple levels of a concept hierarchy, association rules may contain literals from several levels in the hierarchy or may be restricted to one level only. These generalized association rules are defined like one-level rules, but require additionally that no literal in the consequent be an ancestor of any literal in the antecedent to avoid trivial rules.

Further insight can be gained by the use of multiple taxonomies that may be based on price, per-item-profit for the store or based on brand/producer.

Reasons for the use of taxonomies include the following:

- *Rules at the leaf level may not have minimum support.*
  There may be too many different brands of one good, like the confusing variety of cereals, and while Cereal → Milk is a likely association, KelloggsSmacks → Dairyland2%Milk might not be, because KelloggsSmacks does not have minimum support.

- *Pruning of redundant rules is possible.*
  Continuing the example, we can expect to find many rules that link individual cereal brands to milk when mining is restricted to the leaf level. None of these brands need to be especially popular, in which case all those rules only express the fact that some cereal is often bought together with milk. Clearly, they could easily be subsumed under one rule Cereal → Milk.

- *Fast aggregate analysis is possible.*
  If the discovery process can be conducted on different levels independently, users can start the discovery at a higher level and focus on confined areas of interest on lower levels. Looking only for a certain group of items in a database reduces the amount of work significantly and speeds up the mining process, especially in interactive mining sessions.

- *More items are possible*
  Removing all items on entire branches of the taxonomy from consideration if a higher-level concept is not frequent can reduce the number of literals in lower levels. Therefore larger numbers of items become tractable.

The content of this chapter is based on [19, 37]where different algorithms to include concept hierarchies[1]. After general remarks we provide a short sketch of these algorithms in Section 7.2.2

7.2.1 New Definitions for Interestingness

Based on concept hierarchies new criteria for the interestingness of a rule can be applied. The general idea is that a rule is interesting only if it reveals information that could not be expected based on the ancestors of this rule. Since the ancestor rule is more general, it should be preferred. Recalling the cereal example, rules involving individual cereal brands can be considered redundant unless their confidence or support deviate significantly from the ancestor rule Cereal → Milk. Details can be found in [37].

[19] introduce the notion of *strong rules* to focus the search in several hierarchy levels. A rule is strong if every ancestor of every item in its antecedent and consequent is frequent on its level in the taxonomy. This can be strengthened even more to require the antecedent and consequent sets (i.e. their equivalent after replacing each item by its antecedent) to be frequent on the previous level. This is possible because different minimum support and confidence levels are permitted on the various levels.

7.2.2 Algorithms

The basic approach in [37] is to add ancestor literals to each transaction and run the algorithms used for mining single level rules. This can dramatically increase the size of transactions and the number of possible

[1][37] refer to their treatment of taxonomies as *generalized association rules* while [19] use the term *multiple-level association rules* are proposed. The two are used synonymously here, along with *hierarchical association rules.*

frequent sets. Therefore several methods including sampling are proposed to prune the candidate set and avoid unnecessary passes over the database due to the large number of candidates.

The algorithms in [19] encode the hierarchy in the representation of the leaf literal and work on this encoded transaction table after the initial translation step. In its original version the basic algorithm scheme from Section 2.2 is applied repeatedly to mine rules of one level of the hierarchy after the other, starting with the most general level. Modifications of the algorithms are provided to allow rules with items from different levels and multiple concept hierarchies. One major drawback of this approach seems to be the large number of scans required due to the separation of hierarchy levels. The advantage is that different minimum support and confidence values are possible on each level.

7.3 Parallel Discovery of Classification Rules

Although the contents of this Section are not directly applicable to the problem of parallel ARM, they belong into the general context of parallel KDD.

7.3.1 Classification Rule Mining

This section attempts only a brief introduction to the work done on classification in Machine Learning and KDD. The reader is referred to [1, 2, 14, 31] for more information.

A basic familiarity with decision trees as constructed by classification algorithms and general AI search techniques is assumed. The supervised learning case is presupposed because both approaches presented here solve this problem. Each data object is labeled with a class identifier, usually as an additional attribute to the relation.

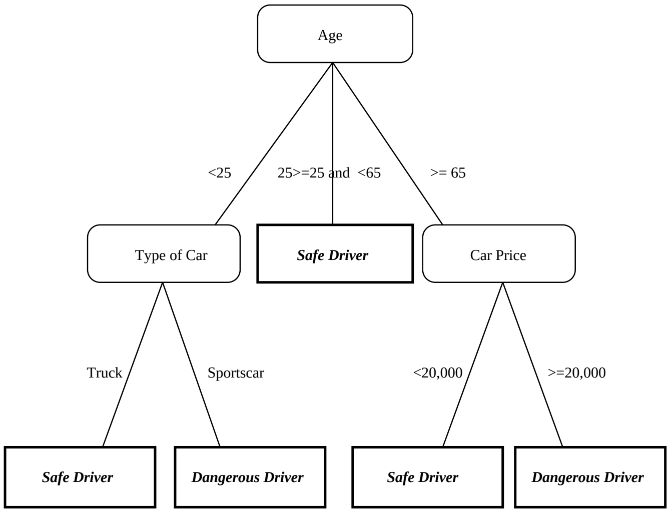


Figure 7.2: Example of Decision Trees Used in Classification

The dominant operation in building decision trees such as the one in Figure 7.2 is the gathering of histograms on attribute values. The conjunction of attribute restrictions along the path to a leaf selects one portion of the database, and all paths partition the relation horizontally in disjoint subsets. Histograms have to be built for each subset, on each attribute and for every class individually.

Based on this information and information theoretic measures, nodes are added to leaves of the tree to the effect of splitting the partition associated with the parent node. Then histograms have to be built anew for all new sub-partitions. The process continues until the classification performance of the tree or its depth reach predefined thresholds. It is an open question as to which methods are optimal for deciding which leaves to expand and when to stop expanding the tree altogether to avoid poor performance due to over-fitting on the one hand or incomplete trees on the other [5, 27, 35, 39].

Two approaches are presented here: one parallel system features a sequential external mining tool that uses a parallel databases server, and the other approach, called *meta-learning*, allows the actual mining to be done in parallel.

7.3.2 A Parallel Mining Engine

The classification tool developed at CWI [20] merely directs the discovery process; the expensive histogram building procedures are incorporated into the extendable parallel DBMS *Monet*, which is designed to exploit large amounts of main memory on shared-memory multi-processors. Experiments were run on a 6-node SGI machine with 256 MBytes of main memory.

The database is completely partitioned vertically into Binary Association Tables in accordance to the assumed column-oriented nature of the access patterns. Hence, selection to create partitions and the construction of histograms can be done in parallel. Also the successive subdivision of horizontal partitions suggests assigning individual processors to different partitions.

Results in [20] show "considerable", although not linear speed-up (factor 2.5 on 4 nodes on a 25K database) , but the authors are rather brief on this issue. In particular, issues like scalability, skew and load balancing and the impact of communication overhead are not addressed.

Since we assume a shared-nothing architecture and the ARM algorithms are more row- than column-oriented, vertical partitioning of the database is unlikely to yield satisfactory results for association mining.

7.3.3 Meta-Learning and Multi-Strategy-Learning

Unlike the previous approach, meta-learning [8, 9, 10][2] does not parallelize the mining algorithm itself but involves horizontally dividing the database into subsets and applying sequential learning algorithms to each subset. The result is a set of different decision trees each depending on the data it was trained on. To avoid losing classification reliability due to the smaller size of the subsets, an arbiter combines the votes of each classifier into one single result. This is where the actual meta-learning part begins: The arbiter is trained to correctly decide between conflicting votes from the individual classifiers. This concept is extended to use not just one arbiter but a binary tree of arbiters (as shown in Figure 7.3) to resolve conflicting results. Adjusting the arbiter tree to the set of classifiers involves further learning from the data, i.e. further passes over various parts of the database. Therefore, while building the individual classifiers in parallel is done efficiently, a considerable amount of time has to be spent to build a suitable arbitrator.

Theoretical speedups of $O(p/\lg p)$ can be obtained on p processors and results reported in [10] reach this boundary. Apart from the rather moderate speedups, we see two shortcomings of this approach: One is that using the classifier for unseen data is slow, because all classifiers have to be applied to the data item, and these results have to be processed by the arbiter tree. While this can be done in parallel, it is certainly not efficient. Secondly, the discovered knowledge is hidden inside the classifier-arbitrator set much in the way it is hidden in neural nets. This makes database retrieval of class members rather inefficient because the use of indexes is precluded.

Nevertheless, the strength of this approach lies in the possibility to incorporate different classification methods as "leaf"-classifiers to improve classification performance by means of diversity. This version of meta-learning is then referred to as *multi-strategy-learning*.

[2] *Sequence analysis* in [8] refers to discovery in DNA sequences, but classification algorithms are used.

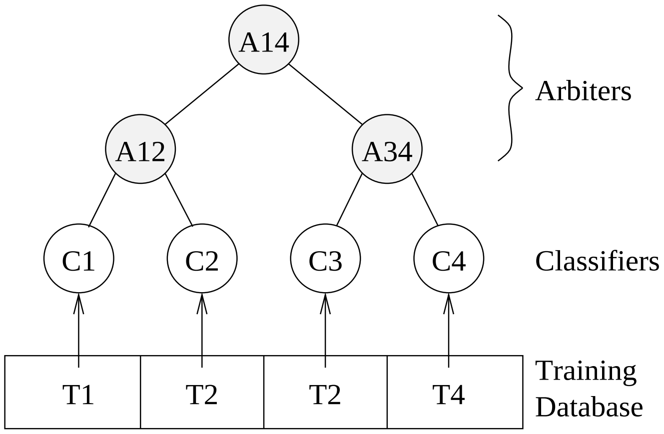


Figure 7.3: Example of Arbiter Tree in Meta-Learning

To our knowledge, no attempts have been made to apply the meta-learning principle to ARM, and although the partitioning principle is used in PARTITION (see Section 2.3.4), intermediate and not final results are coalesced here which constitutes a fundamental difference between the two principles.

Chapter 8

Conclusion and Future Work

We have studied the association rule mining problem and have made the following contributions related to sequential algorithms:

- We have shown that partitioning, although it reduces IO cost, does not lead to performance improvements but introduces constant CPU overhead per partition instead. Therefore, algorithms that use it are slowed considerably, unless IO cost is extremely high. All algorithms are CPU-bound in our environment, even for very large databases, and trying to save IO operations at the expense of additional computation is not bound for success.
- This report provides the first performance analysis of the item-list based pass bundling technique and shows that it reduces both CPU and IO cost significantly without the overhead caused by partitioning.
- We have shown that TID-lists perform very well in later phases of the algorithms, but are highly inefficient in early phases, resulting in poor performance overall, which can only be avoided by special optimizations that do not use TID-lists for the early phases.
- We describe a new incremental partitioning algorithm, which produced slight improvements over the standard partitioning technique, but incurs the same overhead due to partitioning. Therefore incremental partitioning, like its standard counterpart, performed worse than algorithms with pass bundling.

The conclusions from our studies are therefore, that partitioning, which is currently considered the best solution to the ARM problem, does not reduce the execution time of ARM algorithms, but, in fact, can lead to worse performance as the number of partitions grows. Partitioning may be beneficial in conjunction with TID-lists, which greatly reduce CPU cost (provided the optimization of bypassing early phases is used), but even these benefits are lessened by the partitioning overhead. In contrast, item-lists with pass bundling also achieve low IO and CPU cost without the partitioning overhead. Because the bypassing optimization was not mentioned in the paper on PARTITION [34], both techniques could not be compared directly to find a definitive answer to the question which performs better. However, we plan to conduct future research on this issue, which requires implementing the bypass optimization for SPTID and SEAR.

In the second part of this report we described parallel versions of the SEAR and SPEAR algorithms that differ in the amount of communication they require. Although PPAR, SPEAR's parallel counterpart uses about 1/10 of the communication necessary for parallel SEAR due to a global partitioning scheme, it performs worse due to local partitioning.

Nevertheless, we show that both algorithms display near-optimal scale-up and size-up behavior; speed-ups are almost linear, and a speed-up factor of 14 was obtained on a 16-node SP2 multi-processor for most problems tested. Moreover, speed-ups improve further as the database size is increased and the mining problem is made more difficult.

Unfortunately, we were limited to experiments on synthetic data sets (like many other previous studies[19, 32, 34]), so the next task is to evaluate the performance of our algorithms against a real-life data set. Integrating these algorithms into existing DBMS is also likely to lead to additional problems related to generating the proper input format for the mining algorithms and query optimization for queries involving data mining operations, to mention only a few of theses difficulties.

Other issues that have to be investigated more closely in the future are the re-use of results for subsequent runs of the algorithms on the same database with different minimum support values, a situation which is likely to occur in interactive mining.

Acknowledgements

This report is a version of my Master Thesis, and I want to thank Dr. Mike Franklin for his guidance, encouragement and support that made this this study possible.

Bibliography

[1] Rakesh Agrawal, Sakti Ghosh, Tomasz Imielinski, and Arun Swami. An interval classifier for database mining applications. In *18th Int'l Conf. on Very Large Databases (VLDB)*, Vancouver, Canada, pages 560–573, 1992.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: a performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 6, December 1993:914 – 925, 1993.

[3] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, Washington D.C., pages 207–216, May 1993.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *20th Int'l Conf. on Very Large Databases (VLDB)*, Santiago, Chile, Sept. 1994. Expanded version available as IBM Research Report RJ9839, June 1994.

[5] Marko Bohanec and Ivan Bratko. Trading accuracy for simplicity in decision trees. *Machine Learning*, 15:223–250, 1994.

[6] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[7] Yandong Cai, Nick Cercone, and Jaiwei Han. Attribute-oriented induction in relational databases. In Gregory Piatetsky-Shapiro and William J. Frawley, editors, *Knowledge Discovery in Databases*, pages 213–228. AAAI/MIT, 1991.

[8] Philip K. Chan and Salvatore J.Stolfo. Toward multi-strategy parallel and distributed learning in sequence analysis. In *Proc. First Intl. Conf. Intel. Sys. Molecular Biology*, pages 65–73, 1993.

[9] Philip K. Chan and Salvatore J.Stolfo. Toward parallel and distributed learning by meta-learning. In *Working Notes AAAI Work. Knowledge Discovery in Databases*, pages 227–24, 1993.

[10] Philip K. Chan and Salvatore J. Stolfo. Toward scalable and parallel learning: A case study in splice junction prediction. Technical Report CUCS-032-94, Columbia University, 1994.

[11] David K. Chiu, Andrew K.C. Wong, and Benny Cheung. Information discovery through hierarchical maximum enthropy discretization and synthesis. In Gregory Piatetsky-Shapiro and William J. Frawley, editors, *Knowledge Discovery in Databases*, pages 125–140. AAAI/MIT, 1991.

[12] David DeWitt and J. Gray. Parallel database systems: The future of database processing or a passing fad? *sigmod*, 19(4):104–112, December 1990.

[13] David J. DeWitt, Shaharm Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui i Hsiao, and Rick Rasmusen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[14] Tapio Elomaa. In defense of c4.5: Notes on learning one-level decision trees. In W.Cohen and H. Hirsh, editors, *Machine Learning: Proc. 11th Int'l Conference*, pages 62–60. Morgan Kaufmann, July 1994.

[15] William J. Frawley, Gregory Piatetsky-Shapiro, and Christopher J. Matheus. Knowledge discovery in databases: An overview. In Gregory Piatetsky-Shapiro and William J. Frawley, editors, *Knowledge Discovery in Databases*, pages 1–30. AAAI/MIT, 1991.

[16] Jiawei Han, Yandong Cai, and Nick Cercone. Knowledge discovery in databases: An attribute-oriented approach. In *18th Int'l Conf. on Very Large Databases (VLDB)*, Vancouver, Canada, 1992.

[17] Jiawei Han, Yandong Cai, and Nick Cercone. Data-driven discovery of quantitative rules in relational databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(1), Feb. 1993.

[18] Jiawei Han and Yongjian Fu. Dynamic generation and refinement of concept hierarchies for knowledge discovery in databases. In *AAAI'94 Workshop on Knowledge Discovery in Databases (KDD'94)*, Seattle, WA, July 1994.

[19] Jiawei Han and Yongjian Fu. Discovery of multiple-level association rules from large databases. In *21st Int'l Conf. on Very Large Databases (VLDB)*, Zürich, Switzerland, Sept. 1995. To appear.

[20] Marcel Holsheimer and Arno P.J.M. Siebes. Architectural support for data mining. Technical Report CS-R9429, CWI, 1994.

[21] Marcel Holsheimer and Arno P.J.M. Siebes. Data mining: the search for knowledge in databases. Technical Report CS-R9406, CWI, January 1994.

[22] Houtsma and Arun Swami. Set-oriented mining of association rules. Technical Report RJ 9567, IBM Research Report, Oct. 1993.

[23] IBM Corporation, Kingston, NJ 12401-1099. *IBM AIX Parallel Environment, Release 2.0*, 1994.

[24] IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598. *The SP2 Communication Subsystem*, Aug. 1994. Also available via http://ibm.tc.cornell.edu/bm/pps/doc/css/css.ps.

[25] Kenneth Kaufman, Ryszard S. Michalski, and Larry Kerschberg. Mining for knowledge in databases: Goals and general description of the inlen system. In Gregory Piatetsky-Shapiro and William J. Frawley, editors, *Knowledge Discovery in Databases*, pages 449–464. AAAI/MIT, 1991.

[26] Mika Klemettinen, Heikki Mannila, Pirjo Ronkainen, Hannu Toivonen, and A. Inkeri Verkamo. Finding interesting rules from large sets of discovered association rules. In 3^{rd} *Internat. Conf. on Information and Knowledge Management*, Maryland. ACM Press, Nov. 1994.

[27] Wei Zhong Liu and Allan P. White. The importance of attribute selection measures in decision tree induction. *Machine Learning*, 15:25–41, 1994.

[28] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Improved methods for finding association rules. In *AAAI Worshop on Knowledge Discovery*, Seattle, Washington, pages 181–192, July 1994.

[29] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovering frequent episodes in sequences. In *Proc. Knowledge Discovery and Data Mining (KDD'95)*, (to appear), 1995. Also: Technical Report C-1995-10, University of Helsinki, Department of Computer Science, Finland, March 1995.

[30] Christopher J. Matheus, Philip K.Chan, and Gregory Piatetsky-Shapiro. Systems for knowledge discovery in databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(6), Dec. 1993.

[31] Gür-Ali Özden and William A. Wallace. Induction of rules subject to a quality constraint: Probabilistic inductive learning. *IEEE Transactions on Knowledge and Data Engineering*, 5(6), Dec. 1993.

[32] Jong Soo Park, Mink-Syan Chen, and Philip S. Yu. An effective hash-based algorithm for mining association rules. In *SIGMOD*, San Jose, CA, pages 175–186. ACM, 1995.

[33] Arun Swami Rakesh Agrawal, Christos Faloutsos. Efficient similarity search in sequence databases. In *4th Int'l Conf. on Foundations of Data Organization and Algorithms*, Chicago, Oct. 1993. Also in Lecture Notes in Computer Science 730, Springer Verlag, 1993, 69-84.

[34] Ashok Sarasere, Edward Omiecinsky, and Shamkant Navathe. An efficient algorithm for mining association rules in large databases. In *21st Int'l Conf. on Very Large Databases (VLDB)*, Zürich, Switzerland, Sept. 1995. Also Gatech Technical Report No. GIT-CC-95-04.

[35] Cullen Schaffer. Overfitting avoidance as bias. *Machine Learning*, 10:153–178, 1993.

[36] Padhraic Smyth and Rodney M. Goodman. Rule induction using information theory. In Gregory Piatetsky-Shapiro and William J. Frawley, editors, *Knowledge Discovery in Databases*, pages 159–176. AAAI/MIT, 1991.

[37] Ramakrishnan Srikant and Rakesh Agrawal. Mining generalized association rules. In *21st Int'l Conf. on Very Large Databases (VLDB)*, Zürich, Switzerland, Sept. 1995. To appear.

[38] Craig B. Stunkel, Denis G. Shea, Don G. Grice, Peter H. Hochschild, and Michael Tsao. The sp1 high-performance switch. In *Proc. 1994 Scalable High-Performance Computing Conference*, pages 150–157, May 1994.

[39] Allan P. White and Wei Zhong Liu. Bias in information-based measures in decision tree induction. *Machine Learning*, 15:321–329, 1994.

[40] Beat Wüthrich. Knowledge discovery in databases. Draft course manuscript, Hong Kong University of Science and Technology, May 1994.