FAST ALGORITHMS FOR MINING ASSOCIATION RULES AND SEQUENTIAL PATTERNS

By

Ramakrishnan Srikant

A dissertation submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1996

Abstract

This dissertation presents fast algorithms for mining associations in large datasets. An example of an association rule may be "30% of customers who buy jackets and gloves also buy hiking boots." The problem is to find all such rules whose frequency is greater than some user-specified minimum. We first present a new algorithm, Apriori, for mining associations between boolean attributes (called *items*). Empirical evaluation shows that Apriori is typically 3 to 10 times faster than previous algorithms, with the performance gap increasing with the problem size. In many domains, taxonomies (*isa* hierarchies) on the items are common. For example, a taxonomy may say that jackets *isa* outerwear *isa* clothes. We extend the Apriori algorithm to find associations between items at any level of the taxonomy.

Next, we consider associations between quantitative and categorical attributes, not just boolean attributes. We deal with quantitative attributes by fine-partitioning the values of the attribute and then combining adjacent partitions as necessary. We also introduce measures of partial completeness which quantify the information loss due to partitioning. These measures can be used to determine the number of partitions for each attribute. We enhance the Apriori algorithm to efficiently find quantitative associations.

Finally, we consider associations over time, called *sequential patterns*. An example of such a sequential pattern may be "2% of customers bought 'Ringworld' in one transaction, followed by 'Foundation' and 'Ringworld Engineers' in a later transaction". We allow time constraints between elements of the sequential patterns, and also allow all the items in an element to be present in a sliding time window rather than at a single point in time. We present GSP, an algorithm for finding such patterns, based on intuitions

similar to those for the Apriori algorithm.

All the above algorithms scale linearly with the size of the data (for constant data characteristics). These algorithms have been used in a variety of domains, including market basket analysis, attached mailing, fraud detection and medical research. We conclude the dissertation with directions for future work.

Acknowledgements

I would like to thank Rakesh Agrawal and Jeff Naughton, who were terrific advisors. I also wish to thank the other members of my dissertation committee: Mike Carey, Jude Shavlik, Raghu Ramakrishnan and E.W. Frees. I also thank the faculty at Wisconsin, including Yannis Ioannidis.

I would like to thank all my friends and colleagues at both Wisconsin and Almaden, including Manish Mehta, Jignesh Patel, John Shafer, Kyuseok Shim, V. Shivakumar, Vinita Subramanian, Prakash Sundaresan, Sunita Sarawagi and T.N. Vijaykumar. I would undoubtedly have finished faster if I hadn't spent so much time chatting with some of them! I would like to specially thank Jignesh for helping me submit the thesis remotely.

Finally, I would like to thank my parents and my sister for their support and encouragement.

Contents

Chapter 1

Introduction

1.1 Background

Data mining [AIS93a] [ABN92] [HS94] [MKKR92] [SAD+93] [Tsu90], also known as knowledge discovery in databases [FPSSU95] [HCC92] [PSF91], has been recognized as a promising new area for database research. This area can be defined as efficiently discovering interesting patterns from large databases [AIS93a]. The motivation for data mining is that most organizations have collected massive amounts of data, and would like to discover useful or interesting patterns in their data. For example, insurance companies would like to discover patterns that indicate possible fraudulent activity, mail order companies would like to discover patterns that let them cut their mailing costs by only targeting customers likely to respond, and hospitals would like to use their data for medical research. Data mining problems include:

- **Associations:** The problem of mining association rules over basket data was introduced in [AIS93b]. Given a set of transactions, where each transaction is a set of literals (called items), an association rule is an expression of the form $X \Rightarrow Y$, where X and Y are sets of items. The intuitive meaning of such a rule is that transactions of the database which contain X tend to contain Y. An example of an association rule is: "30% of transactions that contain beer also contain diapers; 2% of all transactions contain both of these items". Here 30% is called the *confidence* of the rule, and 2% the *support* of the rule. The problem is to find all association rules that satisfy user-specified minimum support and minimum confidence constraints.

In this dissertation, we examine the problem of mining association rules. We first present fast algorithms for this problem, then generalize the problem to include taxonomies (*isa* hierarchies) and quantitative attributes, and finally describe how these ideas can be applied to the problem of mining **sequential patterns**. Sequential patterns are inter-transaction associations, unlike the intra-transaction association rules. An example of a sequential pattern might be "5% of customers bought 'Foundation' and 'Ringworld' in one transaction, followed by 'Second Foundation' in a later transaction". We describe the contributions of this dissertation in Section 1.2, and its applications in Section 1.3.

- **Classification** [BFOS84] [Cat91] [FWD93] [HCC92] [Qui93]:

  The input data for classification, also called the training set, consists of multiple examples (records), each having multiple attributes or features. Additionally, each example is tagged with a special class label. The objective of classification is to analyze the input data and to develop an accurate description or model for each class using the features present in the data. The class descriptions are used to classify future test data for which the class labels are unknown. They can also be used to develop a better understanding of each class in the data.

  For instance, consider a credit card company with data about its cardholders. Assume that the cardholders have been divided into two classes, *good* and *bad* customers, based on their credit history. The company wants to develop a profile for each customer class that can be used to accept/reject future credit card applicants. This problem can be solved using classification. First, a classifier is given the customer data along with the assigned classes as input. The output of the classifier is a description of each class (good/bad) which can then be used to process future

card applicants. Similar applications of classification include target marketing, medical diagnosis, treatment effectiveness and store location.

- **Clustering** [ANB92] [CKS+88] [Fis87]:

  The input data for clustering looks quite similar to that for classification, except that there are no class labels. The objective of clustering is to group the records such that "similar" records are in the same group. Applications of clustering include customer segmentation, and targeted marketing. A classifier is often run on the results of clustering to understand the clusters.

- **Similar Time Sequences** [AFS93] [FRM94] [ALSS95]:

  Time-series data constitute a large portion of data stored in computers. The capability to find time-series (or portions thereof) that are "similar" to a given time-series or to be able to find groups of similar time-series has several applications. Examples include identifying companies with similar pattern of growth, finding products with similar selling patterns, discovering stocks with similar price movements, determining portions of seismic waves that are not similar to spot geological irregularities, etc.

- **Data Summarization:**

  [HCC92] use attribute-oriented induction to summarize a database relation. They use taxonomies on the attributes to generalize records, and merge duplicates (while maintaining their count). If an attribute has a large number of values that cannot be generalized, the attribute is dropped. The result of this process is a small number of records that describe the data.

Some data mining problems may require several techniques. For example, a classification algorithm may be run on the results of clustering to understand the results. Visualization is often useful both before the mining (to look at the data) and after the mining (to understand the output).

The need for a human in the loop and providing tools to allow human guidance of the rule discovery process has been articulated, for example, in [B+93] [KI91] [Tsu90]. We do not discuss these issues further, except to point out that these are necessary features of a rule discovery system that may use our algorithms as the engine of the discovery process.

1.2 Dissertation Contributions

Association Rules We present two new algorithms for mining association rules. Experiments with synthetic as well as real-life data show that these algorithms outperform earlier algorithms by factors ranging from three for small problems to more than an order of magnitude for large problems. We also show how the best features of the two proposed algorithms can be combined into a hybrid algorithm, called AprioriHybrid.

Generalized Association Rules In many cases, users have a taxonomy (*isa* hierarchy) on the items, and want to find *generalized associations* between items at any level of the taxonomy. For example, given a taxonomy that says that jackets *isa* outerwear *isa* clothes, we may infer a rule that "people who buy outerwear tend to buy shoes". This rule may hold even if rules that "people who buy jackets tend to buy shoes", and "people who buy clothes tend to buy shoes" do not hold.

An obvious solution to the problem is to add all ancestors of each item in a transaction to the transaction, and then run any of the algorithms for mining association rules on

these "extended transactions". However, this "Basic" algorithm is not very fast; we present two algorithms, Cumulate and EstMerge, which run 2 to 5 times faster than Basic (and more than 100 times faster on one real-life dataset).

We also present a "greater-than-expected-value" interest-measure for rules which uses the information in the taxonomy. Given a user-specified "minimum-interest-level", this measure prunes a large number of redundant rules; 40% to 60% of all the rules were pruned on two real-life datasets.

Quantitative Association Rules So far, we considered associations between boolean variables (whether or not an item is present in a transaction). We now introduce the problem of mining association rules in large relational tables containing both quantitative and categorical attributes. An example of such an association might be "10% of married people between age 50 and 60 have at least 2 cars". We deal with quantitative attributes by fine-partitioning the values of the attribute and then combining adjacent partitions as necessary. We introduce measures of partial completeness which quantify the information loss due to partitioning. A direct application of this technique can generate too many similar rules. We extend the "greater-than-expected-value" interest measure to identify the interesting rules in the output. Finally, we give an algorithm for mining such quantitative association rules, and describe the results of using this approach on a real-life dataset.

Sequential Patterns We introduce the problem of mining sequential patterns. Given a database of sequences, where each sequence is a list of transactions ordered by transaction time, and each transaction is a set of items, the problem is to discover all sequential patterns with a user-specified minimum support, where the support of a pattern is the

number of data-sequences that contain the pattern. Thus sequential patterns are inter-transaction patterns, while association rules are intra-transaction patterns. An example of a sequential pattern is "2% of customers bought 'Ringworld' in one transaction, followed by 'Foundation' and 'Ringworld Engineers' in a later transaction".

We generalize the above problem formulation as follows. First, we add time constraints that specify a minimum and/or maximum time period between adjacent elements in a pattern. Second, we relax the restriction that the items in an element of a sequential pattern must come from the same transaction, instead allowing the items to be present in a set of transactions whose transaction-times are within a user-specified time window. Third, given a user-defined taxonomy (*isa* hierarchy) on items, we allow sequential patterns to include items across all levels of the taxonomy.

We present GSP, an algorithm that discovers these generalized sequential patterns. GSP scales linearly with the number of data-sequences, and has very good scale-up properties with respect to the average data-sequence size.

1.3 Dissertation Applications

The problem of mining association rules was originally motivated by the decision support problem faced by most large retail organizations [SAD+93]. Progress in bar-code technology has made it possible for retail organizations to collect and store massive amounts of sales data, referred to as the *basket* data. A record in such data typically consists of the transaction date and the items bought in the transaction. Successful organizations view such databases as important pieces of the marketing infrastructure [Ass92]. They are interested in instituting information-driven marketing processes, managed by database technology, that enable marketers to develop and implement customized marketing programs and strategies [Ass90]. There are now a variety of applications for association rules

and sequential patterns in multiple domains. We first describe *market basket analysis*, the original motivation for mining association rules, followed by other applications.

Market Basket Analysis By mining *basket data* (transaction data), a retail store can find associations between the sale of items. This information could be used in several ways. For example, rules with "Maintenance Agreement" in the consequent may be helpful for boosting Maintenance Agreement sales. Rules with "Home Electronics" may indicate other products the store should stock up on if the store has a sale on Home Electronics.

A related application is **loss-leader analysis**. Stores often sell some items at a loss during a promotion, in the hope that customers would buy other items along with the loss-leader. However, many customers may "cherry-pick" the item on sale. By mining associations over the time period of the promotion as well as before the promotion, and looking at the changes in the support and confidence of rules involving the promotional items , the store can determine whether or not "cherry-picking" occurred.

Item Placement Knowledge about what items are sold together is an useful input for determining where to place items in a store. A closely related application is **catalog placement**. Mail-order companies can use associations to help determine what items should be placed on the same page of a catalog.

Attached Mailing Rather than sending the same catalog to everyone, direct marketing retailers can use associations and sequential patterns to customize the catalog based on the items a person has bought. These customized catalogs may be much smaller, or may be mailed less frequently, reducing mailing costs.

Fraud Detection Insurance companies are interested in finding groups of medical service providers (doctors or clinics) who "ping-pong" patients between each other for unnecessary tests. Given medical claims data, each patient can be mapped to a transaction, and each doctor/clinic visited by a patient to an item in the transaction. The items in an association rule now correspond to a set of providers, and the support of the rule corresponds to the number of patients these providers have in common. The insurance company can now investigate the claim records for sets of providers who have a large number of common patients to determine if any fraudulent activity actually occurred.

Another application is detecting the use of wrong medical payment codes. For example, insurance companies are interested in detecting "unbundling", where a set of payment codes corresponding to the components of a medical procedure are used to claim payment, rather than the code for the whole procedure. (The motivation is that the sum of the payments for the component codes may be greater than the normal payment for the procedure.) Associations between medical payment codes can also be used to find sets of payment codes which are used frequently. See [NRV96] for a similar application.

Medical Research A data-sequence may correspond to the symptoms or diseases of a patient, with a transaction corresponding to the symptoms exhibited or diseases diagnosed during a visit to the doctor. The patterns discovered using this data could be used in disease research to help identify symptoms/diseases that precede certain diseases.

1.4 Dissertation Outline

Chapter 2 describes fast algorithms for the problem of mining association rules. Chapter 3 generalizes the problem to incorporate taxonomies (*isa* hierarchies) on the data.

Chapter 4 looks at associations over quantitative and categorical attributes. Chapter 5 introduces the problem of sequential patterns, and applies some of the earlier ideas to this problem. Finally, Chapter 6 summarizes the dissertation and presents suggestions for future work.

Chapter 2

Fast Algorithms for Mining Association Rules

2.1 Introduction

The problem of mining association rules over basket data was introduced in [AIS93b]. The input data consists of a set of transactions, where each transaction is a set of items. An example of an association rule might be that 98% of customers that purchase tires and auto accessories also get automotive services done. Finding all such rules is valuable for cross-marketing and attached mailing applications. Other applications include catalog design, add-on sales, store layout, and customer segmentation based on buying patterns. The databases involved in these applications are very large. It is imperative, therefore, to have fast algorithms for this task.

The following is a formal statement of the problem [AIS93b]: Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called items. Let $\mathcal{D}$ be a set of transactions, where each transaction T is a set of items such that $T \subseteq \mathcal{I}$. Associated with each transaction is a unique identifier, called its *TID*. We say that a transaction T *contains* X, a set of some items in $\mathcal{I}$, if $X \subseteq T$. An *association rule* is an implication of the form $X \Rightarrow Y$, where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. The rule $X \Rightarrow Y$ holds in the transaction set $\mathcal{D}$ with *confidence* c if $c\%$ of transactions in $\mathcal{D}$ that contain X also contain Y. The rule $X \Rightarrow Y$ has *support* s in the transaction set $\mathcal{D}$ if $s\%$ of transactions in $\mathcal{D}$ contain $X \cup Y$. Our rules are somewhat more general than in [AIS93b] in that we allow a consequent to have more than one item.

Given a set of transactions $\mathcal{D}$, the problem of mining association rules is to generate

all association rules that have support and confidence greater than the user-specified minimum support (called *minsup*) and minimum confidence (called *minconf*) respectively. Our discussion is neutral with respect to the representation of $\mathcal{D}$. For example, $\mathcal{D}$ could be a data file, a relational table, or the result of a relational expression.

An algorithm for finding all association rules, henceforth referred to as the *AIS* algorithm, was presented in [AIS93b]. Another algorithm for this task, called the *SETM* algorithm, has been proposed in [HS95]. In this chapter, we present two algorithms, *Apriori* and *AprioriTid*, that differ fundamentally from these algorithms. We present experimental results, using both synthetic and real-life data, showing that the proposed algorithms always outperform the earlier algorithms. The performance gap is shown to increase with problem size, and ranges from a factor of three for small problems to more than an order of magnitude for large problems. We then discuss how the best features of Apriori and AprioriTid can be combined into a hybrid algorithm, called *AprioriHybrid*. Experiments show that the AprioriHybrid has excellent scale-up properties, opening up the feasibility of mining association rules over very large databases.

Problem Decomposition

The problem of discovering all association rules can be decomposed into two subproblems [AIS93b]:

1. Find all sets of items (*itemsets*) that have transaction support above minimum support. The *support* for an itemset is the number of transactions that contain the itemset. Itemsets with minimum support are called *frequent* itemsets.[1] In Section 2.2, we give new algorithms, Apriori and AprioriTid, for solving this problem.

[1] In [AIS93b] itemsets with minimum support were called *large* itemsets. However, some readers associated "large" with the number of items in the itemset, rather than its support. So we are switching the terminology to *frequent* itemsets.

2. Use the frequent itemsets to generate the desired rules. We give algorithms for this problem in Section 2.3. The general idea is that if, say, $ABCD$ and AB are frequent itemsets, then we can determine if the rule $AB \Rightarrow CD$ holds by computing the ratio $conf = \text{support}(ABCD)/\text{support}(AB)$. If $conf \geq minconf$, then the rule holds. (The rule will surely have minimum support because $ABCD$ is frequent.)

   Unlike [AIS93b], where rules were limited to only one item in the consequent, we allow multiple items in the consequent. An example of such a rule might be that in 58% of the cases, a person who orders a comforter also orders a flat sheet, a fitted sheet, a pillow case, and a ruffle. The algorithms in Section 2.3 generate such multi-consequent rules.

In Section 2.4, we show the relative performance of the proposed Apriori and AprioriTid algorithms against the AIS [AIS93b] and SETM [HS95] algorithms. To make the chapter self-contained, we include an overview of the AIS and SETM algorithms in this section. We also describe how the Apriori and AprioriTid algorithms can be combined into a hybrid algorithm, AprioriHybrid, and demonstrate the scale-up properties of this algorithm. We conclude by pointing out some related open problems in Section 2.6.

Related Work

The closest work in the machine learning literature is the KID3 algorithm presented in [PS91]. If used for finding all association rules, this algorithm will make as many passes over the data as the number of combinations of items in the antecedent, which is exponentially large. Related work in the database literature is the work on inferring functional dependencies from data [Bit92] [MR87]. Functional dependencies are rules requiring strict satisfaction. Consequently, having determined a dependency $X \rightarrow A$, the algorithms in [Bit92] [MR87] consider any other dependency of the form $X + Y \rightarrow A$

redundant and do not generate it. The association rules we consider are probabilistic in nature. The presence of a rule $X \rightarrow A$ does not necessarily mean that $X + Y \rightarrow A$ also holds because the latter may not have minimum support. Similarly, the presence of rules $X \rightarrow Y$ and $Y \rightarrow Z$ does not necessarily mean that $X \rightarrow Z$ holds because the latter may not have minimum confidence.

Mannila et al. [MTV94] independently came up with an idea similar to apriori candidate generation. However, they counted support for the candidates by checking every candidate against every transaction, rather than using the hash-tree (described later in this chapter).

2.2 Discovering Frequent Itemsets

Algorithms for discovering frequent itemsets make multiple passes over the data. In the first pass, we count the support of individual items and determine which of them are *frequent*, i.e. have minimum support. In each subsequent pass, we start with a seed set of itemsets found to be frequent in the previous pass. We use this seed set for generating new potentially frequent itemsets, called *candidate* itemsets, and count the actual support for these candidate itemsets during the pass over the data. At the end of the pass, we determine which of the candidate itemsets are actually frequent, and they become the seed for the next pass. This process continues until no new frequent itemsets are found.

The Apriori and AprioriTid algorithms we propose differ fundamentally from the AIS [AIS93b] and SETM [HS95] algorithms in terms of which candidate itemsets are counted in a pass and in the way that those candidates are generated. In both the AIS and SETM algorithms (see Sections 2.4.1 and 2.4.2 for a review), candidate itemsets are generated on-the-fly during the pass as data is being read. Specifically, after reading a transaction,

it is determined which of the itemsets found frequent in the previous pass are present in the transaction. New candidate itemsets are generated by extending these frequent itemsets with other items in the transaction. However, as we will see, the disadvantage is that this results in unnecessarily generating and counting too many candidate itemsets that turn out to not have minimum support.

The Apriori and AprioriTid algorithms generate the candidate itemsets to be counted in a pass by using only the itemsets found frequent in the previous pass – without considering the transactions in the database. The basic intuition is that any subset of a frequent itemset must be frequent. Therefore, the candidate itemsets having k items can be generated by joining frequent itemsets having $k-1$ items, and deleting those that contain any subset that is not frequent. This procedure results in generation of a much smaller number of candidate itemsets.

The AprioriTid algorithm has the additional property that the database is not used at all for counting the support of candidate itemsets after the first pass. Rather, an encoding of the candidate itemsets used in the previous pass is employed for this purpose. In later passes, the size of this encoding can become much smaller than the database, thus saving much reading effort. We will explain these points in more detail when we describe the algorithms.

Notation We assume that items in each transaction are kept sorted in their lexicographic order. It is straightforward to adapt these algorithms to the case where the database $\mathcal{D}$ is kept normalized and each database record is a <TID, item> pair, where TID is the identifier of the corresponding transaction.

We call the number of items in an itemset its *size*, and call an itemset of size k a k-itemset. Items within an itemset are kept in lexicographic order. We use the notation

k-itemset	An itemset having k items.
L_k	Set of frequent k-itemsets (those with minimum support). Each member of this set has two fields: i) itemset and ii) support count.
C_k	Set of candidate k-itemsets (potentially frequent itemsets). Each member of this set has two fields: i) itemset and ii) support count.
$\overline{C}_k$	Set of candidate k-itemsets when the TIDs of the generating transactions are kept associated with the candidates.

Table 1: Notation for Algorithms

$c[1] \cdot c[2] \cdot \ldots \cdot c[k]$ to represent a k-itemset c consisting of items $c[1], c[2], \ldots c[k]$, where $c[1] < c[2] < \ldots < c[k]$. If $c = X \cdot Y$ and Y is an m-itemset, we also call Y an *m-extension* of X. Associated with each itemset is a count field to store the support for this itemset. The count field is initialized to zero when the itemset is first created.

We summarize in Table 1 the notation used in the algorithms. The set $\overline{C}_k$ is used by AprioriTid and will be further discussed when we describe this algorithm.

2.2.1 Apriori Algorithm

Figure 1 gives the Apriori algorithm. The first pass of the algorithm simply counts item occurrences to determine the frequent 1-itemsets. A subsequent pass, say pass k, consists of two phases. First, the frequent itemsets L_{k-1} found in the $(k-1)$th pass are used to generate the candidate itemsets C_k, using the apriori candidate generation function described below. Next, the database is scanned and the support of candidates in C_k is counted. For fast counting, we need to efficiently determine the candidates in C_k that are contained in a given transaction T. We now describe candidate generation, finding the candidates which are subsets of a given transaction, and then discuss buffer management.

L_1 := {frequent 1-itemsets};
$k := 2$; // k represents the pass number
while ($L_{k-1} \neq \emptyset$) **do begin**
  C_k := New candidates of size k generated from L_{k-1};
  forall transactions $T \in \mathcal{D}$ **do begin**
    Increment the count of all candidates in C_k that are contained in T.
  end
  L_k := All candidates in C_k with minimum support.
  $k := k + 1$;
end
Answer := $\bigcup_k L_k$;

Figure 1: Apriori Algorithm

Apriori Candidate Generation

Given L_{k-1}, the set of all frequent $(k-1)$-itemsets, the algorithm returns a superset of the set of all frequent k-itemsets. The function works as follows. First, in the *join* step, we join L_{k-1} with L_{k-1}:

insert into C_k
select p.item$_1$, p.item$_2$, ..., p.item$_{k-1}$, q.item$_{k-1}$
from L_{k-1} p, L_{k-1} q
where p.item$_1 = q$.item$_1$, ..., p.item$_{k-2} = q$.item$_{k-2}$, p.item$_{k-1} < q$.item$_{k-1}$;

Next, in the *prune* step, we delete all itemsets $c \in C_k$ such that some $(k-1)$-subset of c is not in L_{k-1}:

forall itemsets $c \in C_k$ **do**
  forall $(k-1)$-subsets s of c **do**
    if $(s \notin L_{k-1})$ **then**
      delete c from C_k;

Example Let L_3 be {{1 2 3}, {1 2 4}, {1 3 4}, {1 3 5}, {2 3 4}}. After the join step, C_4 will be {{1 2 3 4}, {1 3 4 5} }. The prune step will delete the itemset {1 3 4 5} because the itemset {1 4 5} is not in L_3. We will then be left with only {1 2 3 4} in C_4.

Contrast this candidate generation with the one used in the AIS and SETM algorithms. In pass k of these algorithms (see Section 2.4 for details), a database transaction

T is read and it is determined which of the frequent itemsets in L_{k-1} are present in T. Each of these frequent itemsets l is then extended with all those frequent items that are present in T and occur later in the lexicographic ordering than any of the items in l. Continuing with the previous example, consider a transaction {1 2 3 4 5}. In the fourth pass, AIS and SETM will generate two candidates, {1 2 3 4} and {1 2 3 5}, by extending the frequent itemset {1 2 3}. Similarly, an additional three candidate itemsets will be generated by extending the other frequent itemsets in L_3, leading to a total of 5 candidates for consideration in the fourth pass. Apriori, on the other hand, generates and counts only one itemset, {1 3 4 5}, because it concludes *a priori* that the other combinations cannot possibly have minimum support.

Correctness We need to show that $C_k \supseteq L_k$. Clearly, any subset of a frequent itemset must also have minimum support. Hence, if we extended each itemset in L_{k-1} with all possible items and then deleted all those whose (k–1)-subsets were not in L_{k-1}, we would be left with a superset of the itemsets in L_k.

The join is equivalent to extending L_{k-1} with each item in the database and then deleting those itemsets for which the (k–1)-itemset obtained by deleting the (k–1)th item is not in L_{k-1}. The condition $p.\text{item}_{k-1} < q.\text{item}_{k-1}$ simply ensures that no duplicates are generated. Thus, after the join step, $C_k \supseteq L_k$. By similar reasoning, the prune step, where we delete from C_k all itemsets whose ($k-1$)-subsets are not in L_{k-1}, also does not delete any itemset that could be in L_k.

Variation: Counting Candidates of Multiple Sizes in One Pass Rather than counting only candidates of size k in the kth pass, we can generate C'_{k+1}, candidates of size $k+1$, from C_k, and count them simultaneously. We use C'_{k+1} to denote candidates generated from C_k and C_{k+1} to denote candidates generated from L_k. Since $C_k \supseteq L_k$,

completeness is maintained. However, C'_{k+1} will typically contain more candidates than C_{k+1} since C_k typically has more itemsets than L_k. This variation can pay off in the later passes when the cost of counting and keeping in memory additional $C'_{k+1} - C_{k+1}$ candidates becomes less than the cost of scanning the database.

Membership Test The prune step requires testing that all $(k-1)$-subsets of a newly generated k-candidate-itemset are present in L_{k-1}. To make this membership test fast, frequent itemsets are stored in a hash table.

Subset Function

Given a set of candidates C_k and a transaction T, we need to find all candidates that are contained in T. We now describe data structures to make this operation efficient.

Candidate itemsets C_k are stored in a *hash-tree*. A node of the hash-tree either contains a list of itemsets (a *leaf* node) or a hash table (an *interior* node). In an interior node, each bucket of the hash table points to another node. The root of the hash-tree is defined to be at depth 1. An interior node at depth d points to nodes at depth $d+1$. Itemsets are stored in the leaves. When we add an itemset c, we start from the root and go down the tree until we reach a leaf. At an interior node at depth d, we decide which branch to follow by applying a hash function to the dth item of the itemset. (As mentioned earlier, we assume the items in an itemset are in lexicographic order. Hence we can treat the itemset as a list rather than a set.) All nodes are initially created as leaf nodes. When the number of itemsets in a leaf node exceeds a specified threshold, the leaf node is converted to an interior node.

Starting from the root node, the subset function finds all the candidates contained in a transaction T as follows. If we are at a leaf, we find which of the itemsets in the leaf are contained in T and add references to them to the answer set. If we are at an

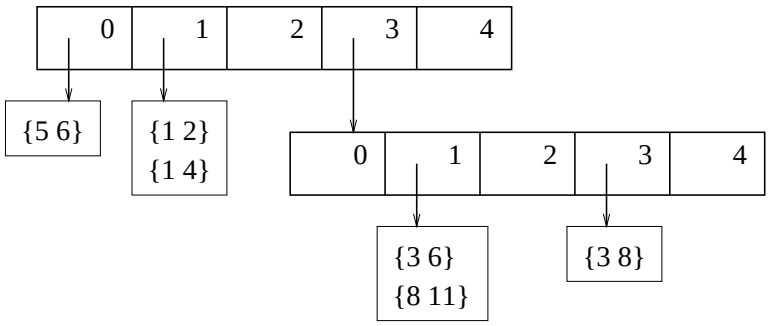


Figure 2: Example of Hash Tree

interior node and we have reached it by hashing the item i, we hash on each item that comes after i in T and recursively apply this procedure to the node in the corresponding bucket. For the root node, we hash on every item in T.

Figure 2 shows an example of the hash tree, where the hash function is "mod 5", the size of the hash table is 5, and the threshold for converting a list to an interior node is 3 itemsets. For a transaction $\{1\ 4\}$, buckets 1 and 4 will be checked at the first level. Hence only the candidates $\{1\ 2\}$ and $\{1\ 4\}$ will be checked for this transaction.

To see why the subset function returns the desired set of references, consider what happens at the root node. For any itemset c contained in transaction T, the first item of c must be in T. At the root, by hashing on every item in T, we ensure that we only ignore itemsets that start with an item not in T. Similar arguments apply at lower depths. The only additional factor is that, since the items in any itemset are ordered, if we reach the current node by hashing the item i, we only need to consider the items in T that occur after i.

In the second-pass, we use a specialized implementation of the hash-tree. Since C_2 is $L_1 \times L_1$, we first generate a mapping from items to integers, such that large items are mapped to contiguous integers and non-large items to 0. We now allocate an array of $|L_1|$ pointers, where each element points to another array of up to $|L_1|$ elements. Each

element of the latter array corresponds to a candidate in C_2, and will contain the support count for that candidate. The first array corresponds to the hash-table in the first level of the hash-tree, and the set of second-level arrays to the hash-tables in the second level of the hash-tree. This specialized structure has two advantages. First, it uses only 4 bytes of memory per candidate, since only the support count for the candidate is stored. (We know that count[i][j] corresponds to candidate (i, j), and so we need not explicitly store the items i and j with the count.) Second, we avoid the overhead of function calls (apart from the mapping) since we can just do a two-level for-loop over each transaction. If there isn't enough memory to generate this structure for all candidates, we generate part of the structure and make multiple passes over the data.

If k is the size of a candidate itemset in the hash-tree, we can find in O(k) time whether the itemset is contained in a transaction by using a temporary bitmap to represent the transaction. Each bit of the bitmap corresponds to an item, and the status of the bit denotes whether or not the transaction contains that item. Thus we simply check the bit corresponding to each item in the candidate to check whether the candidate is contained in the transaction. This bitmap is created once for the hash-tree, and initialized and reset for each transaction. This initialization takes O(size(transaction)) time for each transaction.

Buffer Management

In the candidate generation phase of pass k, we need storage for frequent itemsets L_{k-1} and the candidate itemsets C_k. In the counting phase, we need storage for C_k and at least one page to buffer the database transactions.

First, assume that L_{k-1} fits in memory but that the set of candidates C_k does not. The candidate generation function is modified to generate as many candidates of C_k as

will fit in the buffer and the database is scanned to count the support of these candidates. Frequent itemsets resulting from these candidates are written to disk, while those candidates without minimum support are deleted. This procedure is repeated until all of C_k has been counted.

If L_{k-1} does not fit in memory either, we externally sort L_{k-1}. We bring into memory a block of L_{k-1} in which the first $k-2$ items are the same. We now generate candidates using this block. We keep reading blocks of L_{k-1} and generating candidates until the memory fills up, and then make a pass over the data. This procedure is repeated until all of C_k has been counted. Unfortunately, we can no longer prune those candidates whose subsets are not in L_{k-1}, as the whole of L_{k-1} is not available in memory.

2.2.2 AprioriTid Algorithm

The AprioriTid algorithm, shown in Figure 3, also uses the apriori candidate generation function (given in Section 2.2.1) to determine the candidate itemsets before the pass begins. The interesting feature of this algorithm is that the database $\mathcal{D}$ is not used for counting support after the first pass. Rather, the set $\overline{C}_k$ is used for this purpose. Each member of the set $\overline{C}_k$ is of the form $< TID, \{X_k\} >$, where each X_k is a potentially frequent k-itemset present in the transaction with identifier TID. For $k = 1$, $\overline{C}_1$ corresponds to the database $\mathcal{D}$, although conceptually each item i is replaced by the itemset $\{i\}$. For $k > 1$, $\overline{C}_k$ is generated by the algorithm (step 10). The member of $\overline{C}_k$ corresponding to transaction T is $< t.TID, \{c \in C_k \mid c \text{ contained in } t\}>$. If a transaction does not contain any candidate k-itemset, then $\overline{C}_k$ will not have an entry for this transaction. Thus, the number of entries in $\overline{C}_k$ may be smaller than the number of transactions in the database, especially for large values of k. In addition, for large values of k, each entry may be smaller than the corresponding transaction because very few candidates

1) L_1 := {frequent 1-itemsets};
2) $\overline{C}_1$:= database $\mathcal{D}$;
2a) $k := 2$; // k represents the pass number
3) **while** ($L_{k-1} \neq \emptyset$) **do begin**
4) C_k := New candidates of size k generated from L_{k-1};
5) $\overline{C}_k := \emptyset$;
6) **forall** entries $T \in \overline{C}_{k-1}$ **do begin**
7) // determine candidate itemsets in C_k contained in
// the transaction with identifier T.TID
$C_t := \{c \in C_k \mid (c - c[k]) \in t.\text{set-of-itemsets} \wedge (c - c[k-1]) \in t.\text{set-of-itemsets}\}$;
8) Increment the count of all candidates in C_t.
10) **if** $(C_t \neq \emptyset)$ **then** $\overline{C}_k$ += $\langle t.\text{TID}, C_t \rangle$;
11) **end**
12) L_k := All candidates in C_k with minimum support.
12a) $k := k + 1$;
13) **end**
14) Answer = $\bigcup_k L_k$;

Figure 3: AprioriTid Algorithm

may be contained in the transaction. However, for small values for k, each entry may be larger than the corresponding transaction because an entry in C_k includes all candidate k-itemsets contained in the transaction. We further explore this trade-off in Section 2.4.

In the rest of this section, we establish the correctness of the algorithm, give the data structures used to implement the algorithm, and finally discuss buffer management.

Example Consider the database in Figure 4 and assume that minimum support is 2 transactions. Calling the apriori candidate generation function with L_1 at step 4 gives the candidate itemsets C_2. In steps 6 through 10, we count the support of candidates in C_2 by iterating over the entries in $\overline{C}_1$ and generating $\overline{C}_2$. The first entry in $\overline{C}_1$ is { {1} {3} {4} }, corresponding to transaction 100. The C_t at step 7 corresponding to this entry T is { {1 3} }, because {1 3} is a member of C_2 and both ({1 3} - {1}) and ({1 3} - {3}) are members of T.set-of-itemsets.

Database

TID	Items
100	1 3 4
200	2 3 5
300	1 2 3 5
400	2 5

$\overline{C}_1$

TID	Set-of-Itemsets
100	{ {1}, {3}, {4} }
200	{ {2}, {3}, {5} }
300	{ {1}, {2}, {3}, {5} }
400	{ {2}, {5} }

L_1

Itemset	Support
{1}	2
{2}	3
{3}	3
{5}	3

C_2

Itemset
{1 2}
{1 3}
{1 5}
{2 3}
{2 5}
{3 5}

$\overline{C}_2$

TID	Set-of-Itemsets
100	{ {1 3} }
200	{ {2 3}, {2 5}, {3 5} }
300	{ {1 2}, {1 3}, {1 5}, {2 3}, {2 5}, {3 5} }
400	{ {2 5} }

L_2

Itemset	Support
{1 3}	2
{2 3}	2
{2 5}	3
{3 5}	2

C_3

Itemset
{2 3 5}

$\overline{C}_3$

TID	Set-of-Itemsets
200	{ {2 3 5} }
300	{ {2 3 5} }

L_3

Itemset	Support
{2 3 5}	2

Figure 4: Example for AprioriTid Algorithm

Calling the apriori candidate generation function with L_2 gives C_3. Making a pass over the data with $\overline{C}_2$ and C_3 generates $\overline{C}_3$. Note that there is no entry in $\overline{C}_3$ for the transactions with TIDs 100 and 400, since they do not contain any of the itemsets in C_3. The candidate {2 3 5} in C_3 turns out to be frequent and is the only member of L_3. When we generate C_4 using L_3, it turns out to be empty, and we terminate.

Correctness

Rather than using the database transactions, AprioriTid uses the entries in $\overline{C}_k$ to count the support of candidates in C_k. To simplify the proof, we assume that in step 10 of AprioriTid, we always add $< t.\text{TID}, C_t >$ to $\overline{C}_k$, rather than adding an entry only when C_t is non-empty. For correctness, we need to establish that the set C_t generated in step 7 in the kth pass is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier $T.\text{TID}$.

We say that the set $\overline{C}_k$ is *complete* if $\forall t \in \overline{C}_k$, T.set-of-itemsets includes all frequent k-itemsets contained in the transaction with identifier T.TID. We say that the set $\overline{C}_k$ is *correct* if $\forall t \in \overline{C}_k$, T.set-of-itemsets does not include any k-itemset not contained in the transaction with identifier T.TID. The set L_k is *correct* if it is the same as the set of all frequent k-itemsets. We say that the set C_t generated in step 7 in the kth pass is *correct* if it is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier T.TID.

Lemma 1 *$\forall k > 1$, if $\overline{C}_{k-1}$ is correct and complete and L_{k-1} is correct, then the set C_t generated in step 7 in the kth pass is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier $T.TID$.*

Proof: By simple rewriting, a candidate itemset $c = c[1] \cdot c[2] \cdot \ldots \cdot c[k]$ is present in transaction T.TID if and only if both $c^1 = (c - c[k])$ and $c^2 = (c - c[k-1])$ are in transaction T.TID. Since C_k was obtained by calling the apriori candidate generation function with L_{k-1}, all subsets of $c \in C_k$ must be frequent. So, c^1 and c^2 must be frequent itemsets. Thus, if $c \in C_k$ is contained in transaction T.TID, c^1 and c^2 must be members of T.set-of-itemsets since $\overline{C}_{k-1}$ is complete. Hence c will be a member of C_t. Since $\overline{C}_{k-1}$ is correct, if c^1 (c^2) is not present in transaction T.TID then c^1 (c^2) is not contained in T.set-of-itemsets. Hence, if $c \in C_k$ is not contained in transaction T.TID, c will not be a member of C_t. □

Lemma 2 *$\forall k > 1$, if L_{k-1} is correct and the set C_t generated in step 7 in the kth pass is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier $T.TID$, then the set $\overline{C}_k$ is correct and complete.*

Proof: Since the apriori candidate generation function guarantees that $C_k \supseteq L_k$, the set C_t includes all frequent k-itemsets contained in T.TID. These are added in step 10 to $\overline{C}_k$

and hence $\overline{C}_k$ is complete. Since C_t only includes itemsets contained in the transaction T.TID, and only itemsets in C_t are added to $\overline{C}_k$, it follows that $\overline{C}_k$ is correct. □

Theorem 1 *$\forall k > 1$, the set C_t generated in step 7 in the kth pass is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier $T.TID$.*

Proof: We first prove by induction on k that the set $\overline{C}_k$ is correct and complete and that L_k is correct for all $k \geq 1$. For $k = 1$, this is trivially true since $\overline{C}_1$ corresponds to the database $\mathcal{D}$. By definition, L_1 is also correct. Assume this holds for $k = n$. From Lemma 1, the set C_t generated in step 7 in the $(n+1)$th pass will consist of exactly those itemsets in C_{n+1} contained in the transaction with identifier T.TID. Since the apriori candidate generation function guarantees that $C_{n+1} \supseteq L_{n+1}$ and C_t is correct, L_{n+1} will be correct. From Lemma 2, the set $\overline{C}_{n+1}$ will be correct and complete.

Since $\overline{C}_k$ is correct and complete and L_k correct for all $k \geq 1$, the theorem follows directly from Lemma 1. □

Data Structures

We assign each candidate itemset a unique number, called its ID, at the time the candidate is generated. (We simply increment a counter to get the ID.) Each set of candidate itemsets C_k is kept in an array indexed by the IDs of the itemsets in C_k. A member of $\overline{C}_k$ is now of the form $< \text{TID}, \{\text{ID}\} >$. Each $\overline{C}_k$ is stored in a sequential structure.

The apriori candidate generation function generates a candidate k-itemset c_k by joining two frequent $(k-1)$-itemsets. We maintain two additional fields for each candidate itemset: i) *generators* and ii) *extensions*. The generators field of a candidate itemset c_k stores the IDs of the two frequent $(k-1)$-itemsets whose join generated c_k. The extensions field of an itemset c_k stores the IDs of all the $(k+1)$-candidates that are extensions of c_k. Thus, when a candidate c_k is generated by joining l^1_{k-1} and l^2_{k-1}, we save the IDs of

l^1_{k-1} and l^2_{k-1} in the generators field for c_k. At the same time, the ID of c_k is added to the extensions field of l^1_{k-1}.

We now describe how Step 7 of Figure 3 is implemented using the above data structures. Recall that the T.set-of-itemsets field of an entry T in $\overline{C}_{k-1}$ gives the IDs of all $(k-1)$-candidates contained in transaction T.TID. For each such candidate c_{k-1} the extensions field gives T_k, the set of IDs of all the candidate k-itemsets that are extensions of c_{k-1}. For each c_k in T_k, the generators field gives the IDs of the two itemsets that generated c_k. If these itemsets are present in the entry for T.set-of-itemsets, we can conclude that c_k is present in transaction T.TID, and add c_k to C_t.

We actually need to store only l^2_{k-1} in the generators field, since we reached c_k starting from the ID of l^1_{k-1} in T. We omitted this optimization in the above description to simplify exposition. Given an ID and the data structures above, we can find the associated candidate itemset in constant time. We can also find in constant time whether or not an ID is present in the T.set-of-itemsets field by using a temporary bitmap. Each bit of the bitmap corresponds to an ID in C_k. This bitmap is created once at the beginning of the pass and is reinitialized for each entry T of $\overline{C}_k$.

Buffer Management

In the kth pass, AprioriTid needs memory for L_{k-1} and C_k during candidate generation. During the counting phase, it needs memory for C_{k-1}, C_k, and a page each for $\overline{C}_{k-1}$ and $\overline{C}_k$. Note that the entries in $\overline{C}_{k-1}$ are needed sequentially and that the entries in $\overline{C}_k$ can be written to disk as they are generated.

At the time of candidate generation, when we join L_{k-1} with itself, we fill up roughly half the buffer with candidates. This allows us to keep the relevant portions of both C_k and C_{k-1} in memory during the counting phase. In addition, we ensure that all

candidates with the same first $(k-1)$ items are generated at the same time.

The computation is now effectively partitioned because none of the candidates in memory that turn out to frequent at the end of the pass will join with any of the candidates not yet generated to derive potentially frequent itemsets. Hence we can assume that the candidates in memory are the only candidates in C_k and find all frequent itemsets that are extensions of candidates in C_k by running the algorithm to completion. This may cause further partitioning of the computation downstream. Having thus run the algorithm to completion, we return to L_{k-1}, generate some more candidates in C_k, count them, and so on. Note that the prune step of the apriori candidate generation function cannot be applied after partitioning because we do not know all the frequent k-itemsets.

When L_k does not fit in memory, we need to externally sort L_k, as in the buffer management scheme used for Apriori.

2.3 Discovering Rules

The association rules that we consider here are somewhat more general than in [AIS93b] in that we allow a consequent to have more than one item; rules in [AIS93b] were limited to single item consequents. We first give a straightforward generalization of the algorithm in [AIS93b] and then present a faster algorithm.

Basic Algorithm To generate rules, for every frequent itemset l, we find all non-empty subsets of l. For every such subset a, we output a rule of the form $a \Rightarrow (l-a)$ if the ratio of support(l) to support(a) is at least *minconf*. We consider all subsets of l to generate rules with multiple consequents. Since the frequent itemsets are stored in hash tables, the support counts for the subset itemsets can be found efficiently.

```
// Simple Algorithm
forall frequent itemsets l_k, k ≥ 2 do
    call genrules(l_k, l_k);

// The genrules generates all valid rules ã ⇒ (l_k − ã), for all ã ⊂ a_m
procedure genrules(l_k: frequent k-itemset, a_m: frequent m-itemset)
1)  A := {(m−1)-itemsets a_{m−1}  |  a_{m−1} ⊂ a_m};
2)  forall a_{m−1} ∈ A do begin
3)      conf := support(l_k)/support(a_{m−1});
4)      if (conf ≥ minconf) then begin
7)          output the rule a_{m−1} ⇒ (l_k − a_{m−1}), with confidence = conf
                and support = support(l_k);
8)          if (m − 1 > 1) then
9)              call genrules(l_k, a_{m−1});  // to generate rules with subsets
                                              // of a_{m−1} as the antecedents
10)     end
11) end
```

Figure 5: Generating Rules: Simple Algorithm

We can improve the above procedure by generating the subsets of a frequent itemset in a recursive depth-first fashion. For example, given an itemset $ABCD$, we first consider the subset ABC, then AB, etc. Then if a subset a of a frequent itemset l does not generate a rule, the subsets of a need not be considered for generating rules using l. For example, if $ABC \Rightarrow D$ does not have enough confidence, we need not check whether $AB \Rightarrow CD$ holds. We do not miss any rules because the support of any subset $\tilde{a}$ of a must be as great as the support of a. Therefore, the confidence of the rule $\tilde{a} \Rightarrow (l - \tilde{a})$ cannot be more than the confidence of $a \Rightarrow (l - a)$. Hence, if a did not yield a rule involving all the items in l with a as the antecedent, neither will $\tilde{a}$. Figure 5 shows an algorithm that embodies these ideas.

A Faster Algorithm We showed earlier that if $a \Rightarrow (l - a)$ does not hold, neither does $\tilde{a} \Rightarrow (l - \tilde{a})$ for any $\tilde{a} \subset a$. By rewriting, it follows that for a rule $(l - c) \Rightarrow c$ to

hold, all rules of the form $(l - \tilde{c}) \Rightarrow \tilde{c}$ must also hold, where $\tilde{c}$ is a non-empty subset of c. For example, if the rule $AB \Rightarrow CD$ holds, then the rules $ABC \Rightarrow D$ and $ABD \Rightarrow C$ must also hold.

Consider the above property, which states that for a given frequent itemset, if a rule with consequent c holds then so do rules with consequents that are subsets of c. This is similar to the property that if an itemset is frequent then so are all its subsets. From a frequent itemset l, therefore, we first generate all rules with one item in the consequent. We then use the consequents of these rules and the apriori candidate generation function in Section 2.2.1 to generate all possible consequents with two items that can appear in a rule generated from l, etc. An algorithm using this idea is given in Figure 6. The rules having one-item consequents in step 2 of this algorithm can be found by using a modified version of the preceding genrules function in which steps 8 and 9 are deleted to avoid the recursive call.

Example Consider a frequent itemset $ABCDE$. Assume that $ACDE \Rightarrow B$ and $ABCE \Rightarrow D$ are the only one-item consequent rules derived from this itemset that have the minimum confidence. If we use the simple algorithm, the recursive call genrules($ABCDE$, $ACDE$) will test if the two-item consequent rules $ACD \Rightarrow BE$, $ADE \Rightarrow BC$, $CDE \Rightarrow BA$, and $ACE \Rightarrow BD$ hold. The first of these rules cannot hold, because $E \subset BE$, and $ABCD \Rightarrow E$ does not have minimum confidence. The second and third rules cannot hold for similar reasons. The call genrules($ABCDE$, $ABCE$) will test if the rules $ABC \Rightarrow DE$, $ABE \Rightarrow DC$, $BCE \Rightarrow DA$ and $ACE \Rightarrow BD$ hold, and will find that the first three of these rules do not hold. In fact, the only two-item consequent rule that can possibly hold is $ACE \Rightarrow BD$, where B and D are the consequents in the valid one-item consequent rules. This is the only rule that will be tested by the faster

```
// Faster Algorithm
1) forall frequent k-itemsets l_k, k ≥ 2 do begin
2)     H_1 = { consequents of rules derived from l_k with one item in the consequent };
3)     call ap-genrules(l_k, H_1);
4) end

procedure ap-genrules(l_k: frequent k-itemset, H_m: set of m-item consequents)
   if (k > m + 1) then begin
      H_{m+1} := result of calling the apriori candidate generation function with H_m;
      forall h_{m+1} ∈ H_{m+1} do begin
         conf := support(l_k)/support(l_k − h_{m+1});
         if (conf ≥ minconf) then
            output the rule (l_k − h_{m+1}) ⇒ h_{m+1} with confidence = conf
               and support = support(l_k);
         else
            delete h_{m+1} from H_{m+1};
      end
      call ap-genrules(l_k, H_{m+1});
   end
```

Figure 6: Generating Rules: Faster Algorithm

algorithm.

2.4 Performance Evaluation

To assess the relative performance of the algorithms for discovering frequent itemsets, we performed several experiments on an IBM RS/6000 530H workstation with a CPU clock rate of 33 MHz, 64 MB of main memory, and running AIX 3.2. The data resided in the AIX file system and was stored on a 2GB SCSI 3.5" drive, with a measured sequential throughput of about 2 MB/second.

We first give an overview of the AIS [AIS93b] and SETM [HS95] algorithms against which we compare the performance of the Apriori and AprioriTid algorithms. We then describe the synthetic datasets used in the performance evaluation and show the performance results. Next, we show the performance results for three real-life datasets obtained

from a retail and a direct mail company. Finally, we describe how the best performance features of Apriori and AprioriTid can be combined into an AprioriHybrid algorithm and demonstrate its scale-up properties.

2.4.1 Overview of the AIS Algorithm

Figure 7 summarizes the essence of the AIS algorithm (see [AIS93b] for further details). Candidate itemsets are generated and counted on-the-fly as the database is scanned. After reading a transaction, it is determined which of the itemsets that were found to be frequent in the previous pass are contained in this transaction (step 5). New candidate itemsets are generated by extending these frequent itemsets with other items in the transaction (step 7). A frequent itemset l is extended with only those items that are frequent and occur later than any of the items in l in the lexicographic ordering of items. The candidates generated from a transaction are added to the set of candidate itemsets maintained for the pass, or the counts of the corresponding entries are increased if they were created by an earlier transaction (step 9).

Data Structures The data structures required for maintaining frequent and candidate itemsets were not specified in [AIS93b]. We store the frequent itemsets in a dynamic multi-level hash table to make the subset operation in step 5 fast, using the algorithm described in Section 2.2.1. Candidate itemsets are kept in a hash table associated with the respective frequent itemsets from which they originate in order to make the membership test in step 9 fast.

Buffer Management When a newly generated candidate itemset causes the buffer to overflow, we discard from memory the corresponding frequent itemset and all candidate itemsets generated from it. This reclamation procedure is executed as often as necessary

```
1)   L_1 := {frequent 1-itemsets};
1a)  k := 2;  // k represents the pass number
2)   while ( L_{k-1} ≠ ∅ ) do begin
3)       C_k := ∅;
4)       forall transactions T ∈ 𝒟 do begin
5)           L_t := Members of L_{k-1} that are contained in T;
6)           forall frequent itemsets l_t ∈ L_t do begin
7)               C_t := 1-extensions of l_t contained in T;  // Candidates contained in T
8)               forall candidates c ∈ C_t do
9)                   if (c ∈ C_k) then
                         add 1 to the count of c in the corresponding entry in C_k
                     else
                         add c to C_k with a count of 1;
10)          end
11)      L_k := All candidates in C_k with minimum support.
11a)     k := k + 1;
12)  end
13)  Answer := ⋃_k L_k;
```

Figure 7: AIS Algorithm

during a pass. The frequent itemsets discarded in a pass are extended in the next pass. This technique is a simplified version of the buffer management scheme presented in [AIS93b].

2.4.2 Overview of the SETM Algorithm

The SETM algorithm [HS95] was motivated by the desire to use SQL to compute frequent itemsets. Our description of this algorithm in Figure 8 uses the same notation as used for the other algorithms, but is functionally identical to the SETM algorithm presented in [HS95]. $\overline{C}_k$ ($\overline{L}_k$) in Figure 8 represents the set of candidate (frequent) itemsets in which the TIDs of the generating transactions have been associated with the itemsets. Each member of these sets is of the form $< TID, \text{itemset} >$.

Like AIS, the SETM algorithm also generates candidates on-the-fly based on transactions read from the database. It thus generates and counts every candidate itemset

that the AIS algorithm generates. However, to use the standard SQL join operation for candidate generation, SETM separates candidate generation from counting. It saves a copy of the candidate itemset together with the TID of the generating transaction in a sequential structure (step 9). At the end of the pass, the support count of candidate itemsets is determined by sorting (step 12) and aggregating this sequential structure (step 13).

SETM remembers the TIDs of the generating transactions with the candidate itemsets. To avoid needing a subset operation, it uses this information to determine the frequent itemsets contained in the transaction read (step 6). $\overline{L}_k \subseteq \overline{C}_k$ and is obtained by deleting those candidates that do not have minimum support (step 13). Assuming that the database is sorted in TID order, SETM can easily find the frequent itemsets contained in a transaction in the next pass by sorting $\overline{L}_k$ on TID (step 15). In fact, it needs to visit every member of $\overline{L}_k$ only once in the TID order, and the candidate generation in steps 5 through 11 can be performed using the relational merge-join operation [HS95].

The disadvantage of this approach is mainly due to the size of candidate sets $\overline{C}_k$. For each candidate itemset, the candidate set now has as many entries as the number of transactions in which the candidate itemset is present. Moreover, when we are ready to count the support for candidate itemsets at the end of the pass, $\overline{C}_k$ is in the wrong order and needs to be sorted on itemsets (step 12). After counting and pruning out candidate itemsets that do not have minimum support, the resulting set $\overline{L}_k$ needs another sort on TID (step 15) before it can be used for generating candidates in the next pass.

Buffer Management The performance of the SETM algorithm critically depends on the size of the set $\overline{C}_k$ relative to the size of memory. If $\overline{C}_k$ fits in memory, the two sorting

1) L_1 := {frequent 1-itemsets};
2) $\overline{L}_1$:= {Frequent 1-itemsets together with the TIDs in which they appear, sorted on TID};
2a) $k := 2$; // k represents the pass number
3) **while** ($L_{k-1} \neq \emptyset$) **do begin**
4) $\overline{C}_k := \emptyset$;
5) **forall** transactions $T \in \mathcal{D}$ **do begin**
6) $L_t := \{l \in \overline{L}_{k-1} \mid l.\text{TID} = t.\text{TID}\}$; // Frequent $(k-1)$-itemsets contained in T
7) **forall** frequent itemsets $l_t \in L_t$ **do begin**
8) C_t := 1-extensions of l_t contained in T; // Candidates in T
9) $\overline{C}_k$ += $\{< t.\text{TID}, c > \mid c \in C_t\}$;
10) **end**
11) **end**
12) **sort** $\overline{C}_k$ on itemsets;
13) **delete** all itemsets $c \in \overline{C}_k$ for which c.count $<$ minsup giving $\overline{L}_k$;
14) $L_k := \{< l.\text{itemset}, \text{count of } l \text{ in } \overline{L}_k > \mid l \in \overline{L}_k\}$; // Combined with step 13
15) **sort** $\overline{L}_k$ on TID;
15a) $k := k + 1$;
16) **end**
17) Answer := $\bigcup_k L_k$;

Figure 8: SETM Algorithm

.

steps can be performed using an in-memory sort. In [HS95], $\overline{C}_k$ was assumed to fit in main memory and buffer management was not discussed.

If $\overline{C}_k$ is too large to fit in memory, we write the entries in $\overline{C}_k$ to disk in FIFO order when the buffer allocated to the candidate itemsets fills up, as these entries are not required until the end of the pass. However, $\overline{C}_k$ now requires two external sorts.

2.4.3 Synthetic Data Generation

We generated synthetic transactions to evaluate the performance of the algorithms over a large range of data characteristics. These transactions mimic the transactions in the retailing environment. Our model of the "real" world is that people tend to buy sets of items together. Each such set is potentially a maximal frequent itemset. An example of such a set might be sheets, pillow case, comforter, and ruffles. However, some people may buy only some of the items from such a maximal set. For instance, some people might buy only sheets and pillow case, and some might buy only sheets. A transaction may contain more than one frequent itemset. For example, a customer might place an order for a dress and jacket when ordering sheets and pillow cases, where the dress and jacket together form another frequent itemset. Transaction sizes are typically clustered around a mean and a few transactions have many items. Typical sizes of maximal frequent itemsets are also clustered around a mean, with a few maximal frequent itemsets having a large number of items.

To create a dataset, our synthetic data generation program takes the parameters shown in Table 2.

We first determine the size of the next transaction. The size is picked from a Poisson distribution with mean μ equal to $|T|$. Note that if each item is chosen with the same probability p, and there are N items, the expected size of a transaction is given by

$\|\mathcal{D}\|$	Number of transactions
$\|T\|$	Average size of the transactions
$\|I\|$	Average size of the maximal potentially frequent itemsets
$\|\mathcal{I}\|$	Number of maximal potentially frequent itemsets
N	Number of items

Table 2: Parameters

a binomial distribution with parameters N and p, and is approximated by a Poisson distribution with mean Np.

We then assign items to the transaction. Each transaction is assigned a series of potentially frequent itemsets. If the frequent itemset on hand does not fit in the transaction, the itemset is put in the transaction anyway in half the cases, and the itemset is moved to the next transaction the rest of the cases.

Frequent itemsets are chosen from a set $\mathcal{I}$ of such itemsets. The number of itemsets in $\mathcal{I}$ is set to $|\mathcal{I}|$. There is an inverse relationship between $|\mathcal{I}|$ and the average support for potentially frequent itemsets. An itemset in $\mathcal{I}$ is generated by first picking the size of the itemset from a Poisson distribution with mean μ equal to $|I|$. Items in the first itemset are chosen randomly. To model the phenomenon that frequent itemsets often have common items, some fraction of items in subsequent itemsets are chosen from the previous itemset generated. We use an exponentially distributed random variable with mean equal to the *correlation level* to decide this fraction for each itemset. The remaining items are picked at random. In the datasets used in the experiments, the correlation level was set to 0.5. We ran some experiments with the correlation level set to 0.25 and 0.75 but did not find much difference in the nature of our performance results.

Each itemset in $\mathcal{I}$ has a weight associated with it, which corresponds to the probability that this itemset will be picked. This weight is picked from an exponential distribution with unit mean, and is then normalized so that the sum of the weights for all the itemsets in $\mathcal{I}$ is 1. The next itemset to be put in the transaction is chosen from $\mathcal{I}$ by tossing an

Name	$\|T\|$	$\|I\|$	$\|\mathcal{D}\|$	Size in Megabytes
T5.I2.D100K	5	2	100K	2.4
T10.I2.D100K	10	2	100K	4.4
T10.I4.D100K	10	4	100K	
T20.I2.D100K	20	2	100K	8.4
T20.I4.D100K	20	4	100K	
T20.I6.D100K	20	6	100K	

Table 3: Parameter settings (Synthetic datasets)

$|\mathcal{I}|$-sided weighted coin, where the weight for a side is the probability of picking the associated itemset.

To model the phenomenon that all the items in a frequent itemset are not always bought together, we assign each itemset in $\mathcal{I}$ a *corruption level* c. When adding an itemset to a transaction, we keep dropping an item from the itemset as long as a uniformly distributed random number between 0 and 1 is less than c. Thus for an itemset of size l, we will add l items to the transaction $1 - c$ of the time, $l - 1$ items $c(1 - c)$ of the time, $l - 2$ items $c^2(1 - c)$ of the time, etc. The corruption level for an itemset is fixed and is obtained from a normal distribution with mean 0.5 and variance 0.1.

We generated datasets by setting $N = 1000$ and $|\mathcal{I}| = 2000$. We chose 3 values for $|T|$: 5, 10, and 20. We also chose 3 values for $|I|$: 2, 4, and 6. The number of transactions was to set to 100,000 because, as we will see in Section 2.4.4, SETM could not be run for larger values. However, for our scale-up experiments, we generated datasets with up to 10 million transactions (838MB for $|T| = 20$). Table 3 summarizes the dataset parameter settings. For the same $|T|$ and $|\mathcal{D}|$ values, the size of datasets in megabytes were roughly equal for the different values of $|I|$.

2.4.4 Experiments with Synthetic Data

Figure 9 shows the execution times for the six synthetic datasets given in Table 3 for decreasing values of minimum support. As the minimum support decreases, the execution

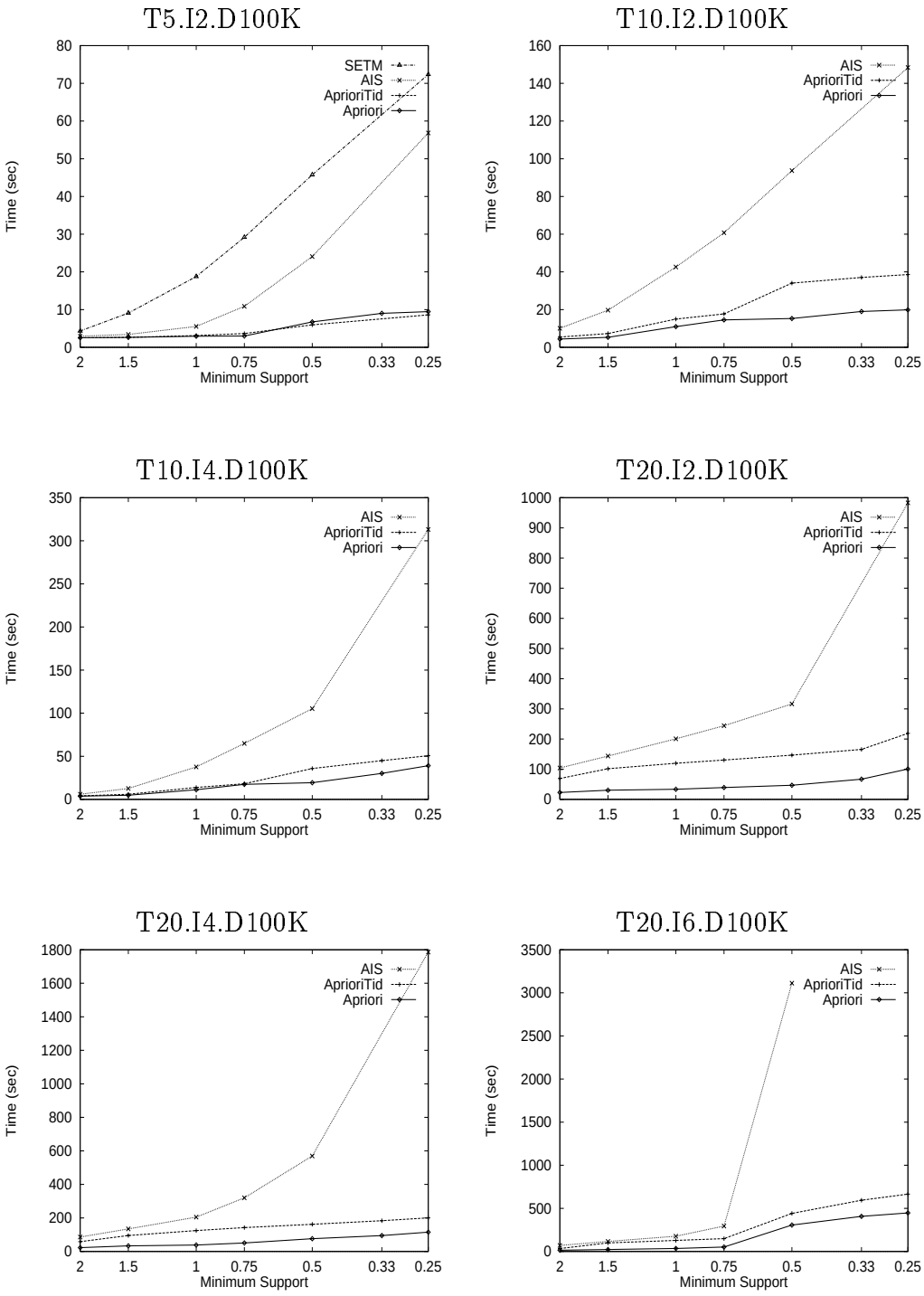


Figure 9: Execution times: Synthetic Data

Dataset	Algorithm	Minimum Support				
		2.0%	1.5%	1.0%	0.75%	0.5%
T10.I2.D100K	SETM	74	161	838	1262	1878
	Apriori	4.4	5.3	11.0	14.5	15.3
T10.I4.D100K	SETM	41	91	659	929	1639
	Apriori	3.8	4.8	11.2	17.4	19.3

Table 4: Execution times in seconds for SETM

times of all the algorithms increase because of increases in the total number of candidate and frequent itemsets.

For SETM, we have only plotted the execution times for the dataset T5.I2.D100K in Figure 9. The execution times for SETM for the two datasets with an average transaction size of 10 are given in Table 4. We did not plot the execution times in Table 4 on the corresponding graphs because they are too large compared to the execution times of the other algorithms. For the three datasets with transaction sizes of 20, SETM took too long to execute and we aborted those runs as the trends were clear. Clearly, Apriori beats SETM by more than an order of magnitude for large datasets.

Apriori beats AIS for all problem sizes, by factors ranging from 2 for high minimum support to more than an order of magnitude for low levels of support. AIS always did considerably better than SETM. For small problems, AprioriTid did about as well as Apriori, but performance degraded to about twice as slow for large problems.

Apriori and AIS were always CPU-bound, since they did sequential scans of the data and the Unix file system was able to prefetch pages. AprioriTid was CPU-bound for small problems, but IO-bound for large problems.

2.4.5 Explanation of the Relative Performance

To explain these performance trends, we show in Figure 10 the sizes of the frequent and candidate sets in different passes for the T10.I4.D100K dataset for the minimum support

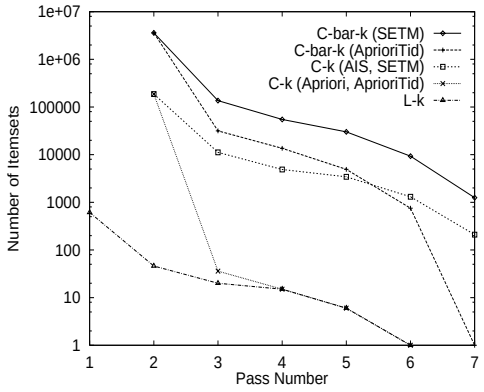


Figure 10: Sizes of the frequent and candidate sets (T10.I4.D100K, minsup = 0.75%)

of 0.75%. Note that the Y-axis in this graph has a log scale.

Apriori candidate generation generates dramatically fewer candidates than on-the-fly candidate generation after the second pass. For the second pass, the number of candidates generated for both is nearly identical. For Apriori, C_2 is really a cross-product of L_1 with L_1. For the older algorithms, we added the optimization that a frequent itemsets are only extended with frequent item to generate a candidate. (Without this optimization, the number of candidates would be much higher for these algorithms even during the second pass). Since the on-the-fly methods only count pairs that actually appear in the data, the number of candidates with this optimization should in fact be slightly less than for Apriori during the second pass. However, nearly all combinations of frequent items appear in large datasets.

The fundamental problem with the SETM algorithm is the size of its $\overline{C}_k$ sets. Recall that the size of the set $\overline{C}_k$ is given by $\sum_{\text{candidate itemsets } c} \text{support-count}(c)$. Thus, the sets $\overline{C}_k$ are roughly S times bigger than the corresponding C_k sets, where S is the average support count of the candidate itemsets. Unless the problem size is very small, the $\overline{C}_k$ sets have to be written to disk, and externally sorted twice, causing the SETM algorithm

to perform poorly.[2] This explains the jump in time for SETM in Table 4 when going from 1.5% support to 1.0% support for datasets with transaction size 10. The largest dataset in the scale-up experiments for SETM in [HS95] was still small enough that $\overline{C}_k$ could fit in memory; hence they did not encounter this jump in execution time. Note that for the same minimum support, the support count for candidate itemsets increases linearly with the number of transactions. Thus, as we increase the number of transactions for the same values of $|T|$ and $|I|$, though the size of C_k does not change, the size of $\overline{C}_k$ goes up linearly. Thus, for datasets with more transactions, the performance gap between SETM and the other algorithms will become even larger.

The problem with AIS is that it generates too many candidates that later turn out not to have minimum support, causing it to waste too much effort. Apriori also counts too many sets without minimum support in the second pass. However, this wastage decreases dramatically from the third pass onward. Note that for the example in Figure 10, after pass 3, almost every candidate itemset counted by Apriori turns out to be a frequent set.

AprioriTid also has the problem of SETM that $\overline{C}_k$ tends to be large. However, the apriori candidate generation used by AprioriTid generates significantly fewer candidates than the transaction-based candidate generation used by SETM. As a result, the $\overline{C}_k$ of AprioriTid has fewer entries than that of SETM. AprioriTid is also able to use a single word (ID) to store a candidate rather than requiring as many words as the number of items in the candidate.[3] In addition, unlike SETM, AprioriTid does not have to sort $\overline{C}_k$.

[2] The cost of external sorting in SETM can be reduced somewhat as follows. Before writing out entries in $\overline{C}_k$ to disk, we can sort them on itemsets using an internal sorting procedure, and write them as sorted runs. These sorted runs can then be merged to obtain support counts. However, given the poor performance of SETM, we do not expect this optimization to affect the algorithm choice.

[3] For SETM to use IDs, it would have to maintain two additional in-memory data structures: a hash table to find out whether a candidate has been generated previously, and a mapping from the IDs to candidates. However, this would destroy the set-oriented nature of the algorithm. Also, once we have the hash table which gives us the IDs of candidates, we might as well count them at the same time and avoid the two external sorts. We experimented with this variant of SETM as well and found that, while it did better than SETM, it still performed much worse than Apriori or AprioriTid.

Thus, AprioriTid does not suffer as much as SETM from maintaining $\overline{C}_k$.

AprioriTid has the nice feature that it replaces a pass over the original dataset by a pass over the set $\overline{C}_k$. Hence, AprioriTid is very effective in later passes when the size of $\overline{C}_k$ becomes small compared to the size of the database. Thus, we find that AprioriTid beats Apriori when its $\overline{C}_k$ sets can fit in memory and the distribution of the frequent itemsets has a long tail. When $\overline{C}_k$ doesn't fit in memory, there is a jump in the execution time for AprioriTid, such as when going from 0.75% to 0.5% for datasets with transaction size 10 in Figure 9 (T10.I4.D100K). In this region, Apriori starts beating AprioriTid.

2.4.6 Reality Check

To confirm the relative performance trends we observed using synthetic data, we experimented with three real-life datasets: a sales transactions dataset obtained from a retail chain and two customer-order datasets obtained from a mail order company. We present the results of these experiments below.

Retail Sales Data The data from the retail chain consists of the sales transactions from one store over a short period of time. A transaction contains the names of the departments from which a customer bought a product in a visit to the store. There are a total of 63 items, representing departments. There are 46,873 transactions with an average size of 2.47. The size of the dataset is very small, only 0.65MB. Some performance results for this dataset were reported in [HS95].

Figure 11 shows the execution times of the four algorithms.[4] The $\overline{C}_k$ sets for both SETM and AprioriTid fit in memory for this dataset. Apriori and AprioriTid are roughly

[4]The execution times for SETM in this figure are a little higher compared to those reported in [HS95]. The timings in [HS95] were obtained on a RS/6000 350 processor, whereas our experiments have been run on a slower RS/6000 530H processor. The execution time for 1% support for AIS is lower than that reported in [AIS93b] because of improvements in the data structures for storing frequent and candidate itemsets.

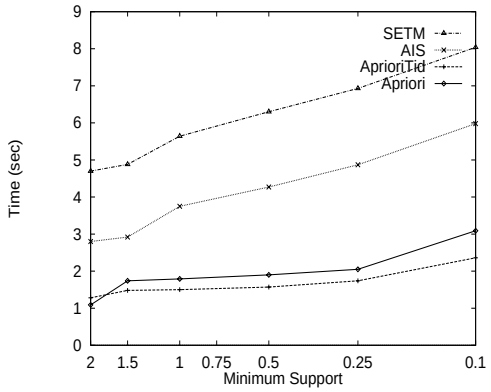


Figure 11: Execution times: Retail sales data

three times as fast as AIS and four times faster than SETM.

Mail Order Data A transaction in the first dataset from the mail order company consists of items ordered by a customer in a single mail order. There are a total of 15836 items. The average size of a transaction is 2.62 items and there are a total of 2.9 million transactions. The size of this dataset is 42 MB. A transaction in the second dataset consists of all the items ordered by a customer from the company in all orders together. Again, there are a total of 15836 items, but the average size of a transaction is now 31 items and there are a total of 213,972 transactions. The size of this dataset is 27 MB. We will refer to these datasets as M.order and M.cust respectively.

The execution times for these two datasets are shown in Figures 12 and 13 respectively. For both datasets, AprioriTid is initially comparable to Apriori but becomes up to twice as slow for lower supports. For M.order, Apriori outperforms AIS by a factor of 2 to 6 and beats SETM by a factor of about 15. For M.cust, Apriori beats AIS by a factor of 3 to 30. SETM had to be aborted (after taking 20 times the time Apriori took to complete) because, even for 2% support, the set $\overline{C}_2$ became larger than the disk capacity.

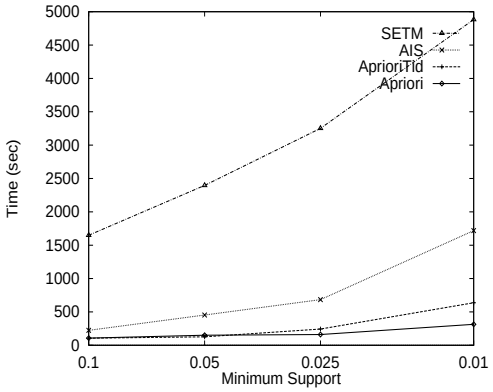


Figure 12: Execution times: M.order

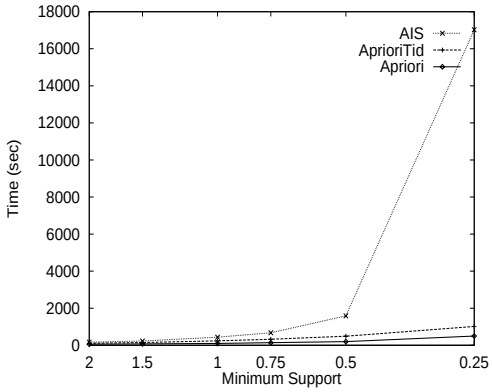


Figure 13: Execution times: M.cust

2.4.7 AprioriHybrid Algorithm

It is not necessary to use the same algorithm in all the passes over data. Figure 14 shows the execution times for Apriori and AprioriTid for different passes over the dataset T10.I4.D100K. In the earlier passes, Apriori does better than AprioriTid. However, AprioriTid beats Apriori in later passes. We observed similar relative behavior for the other datasets, the reason for which is as follows. Apriori and AprioriTid use the same candidate generation procedure and therefore count the same itemsets. In the later passes, the number of candidate itemsets reduces (see the size of C_k for Apriori and AprioriTid in Figure 10). However, Apriori still examines every transaction in the database. On the other hand, rather than scanning the database, AprioriTid scans $\overline{C}_k$ for obtaining support counts, and the size of $\overline{C}_k$ has become smaller than the size of the database. When the $\overline{C}_k$ sets can fit in memory, we do not even incur the cost of writing them to disk.

Based on these observations, we can design a hybrid algorithm, which we call AprioriHybrid, that uses Apriori in the initial passes and switches to AprioriTid when it expects that the set $\overline{C}_k$ at the end of the pass will fit in memory. We use the following heuristic to estimate if $\overline{C}_k$ would fit in memory in the next pass. At the end of

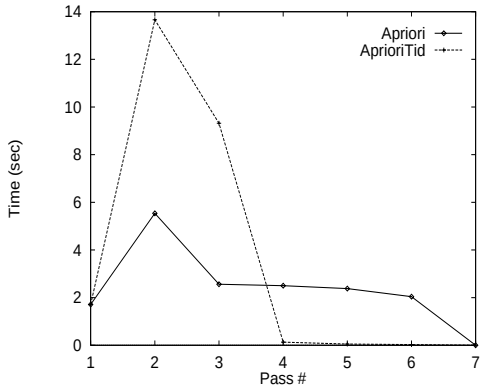


Figure 14: Per pass execution times of Apriori and AprioriTid (T10.I4.D100K, minsup = 0.75%)

the current pass, we have the counts of the candidates in C_k. From this, we estimate what the size of $\overline{C}_k$ would have been if it had been generated. This size, in words, is ($\sum_{\text{candidates } c \in C_k} \text{support}(c) + \text{number of transactions}$). If $\overline{C}_k$ in this pass was small enough to fit in memory, and there were fewer frequent candidates in the current pass than the previous pass, we switch to AprioriTid. The latter condition is added to avoid switching when $\overline{C}_k$ in the current pass fits in memory but $\overline{C}_k$ in the next pass may not.

Switching from Apriori to AprioriTid does involve a cost. Assume that we decide to switch from Apriori to AprioriTid at the end of the kth pass. In the $(k+1)$th pass, after finding the candidate itemsets contained in a transaction, we will also have to add their IDs to $\overline{C}_{k+1}$ (see the description of AprioriTid in Section 2.2.2). Thus there is an extra cost incurred in this pass relative to just running Apriori. It is only in the $(k+2)$th pass that we actually start running AprioriTid. Thus, if there are no frequent $(k+1)$-itemsets, or no $(k+2)$-candidates, we will incur the cost of switching without getting any of the savings of using AprioriTid.

Figure 15 shows the performance of AprioriHybrid relative to Apriori and AprioriTid for large datasets. AprioriHybrid performs better than Apriori in almost all cases. For T10.I2.D100K with 1.5% support, AprioriHybrid does a little worse than Apriori since the

pass in which the switch occurred was the last pass; AprioriHybrid thus incurred the cost of switching without realizing the benefits. In general, the advantage of AprioriHybrid over Apriori depends on how the size of the $\overline{C}_k$ set decline in the later passes. If $\overline{C}_k$ remains large until nearly the end and then has an abrupt drop, we will not gain much by using AprioriHybrid since we can use AprioriTid only for a short period of time after the switch. This is what happened with the M.cust and T20.I6.D100K datasets. On the other hand, if there is a gradual decline in the size of $\overline{C}_k$, AprioriTid can be used for a while after the switch, and a significant improvement can be obtained in the execution time.

The current heuristic used to switch from Apriori to AprioriTid will result in AprioriHybrid degenerating to Apriori if the database size is very large compared to the memory available. If disks with high throughput (for example, RAID disks) are available, the performance of AprioriTid would improve since it would not be IO bound. In this case, heuristics which switch earlier, say, when the average number of candidates contained in a transaction is small, would maintain the performance advantage of AprioriHybrid.

2.4.8 Scale-up

Figure 16 shows how AprioriHybrid scales up as the number of transactions is increased from 100,000 to 10 million transactions. We used the combinations (T5.I2), (T10.I4), and (T20.I6) for the average sizes of transactions and itemsets respectively. All other parameters were the same as for the data in Table 3. The sizes of these datasets for 10 million transactions were 239MB, 439MB and 838MB respectively. The minimum support level was set to 0.75%. The execution times are normalized with respect to the times for the 100,000 transaction datasets in the first graph and with respect to the 1 million transaction dataset in the second. As shown, the execution times scale quite

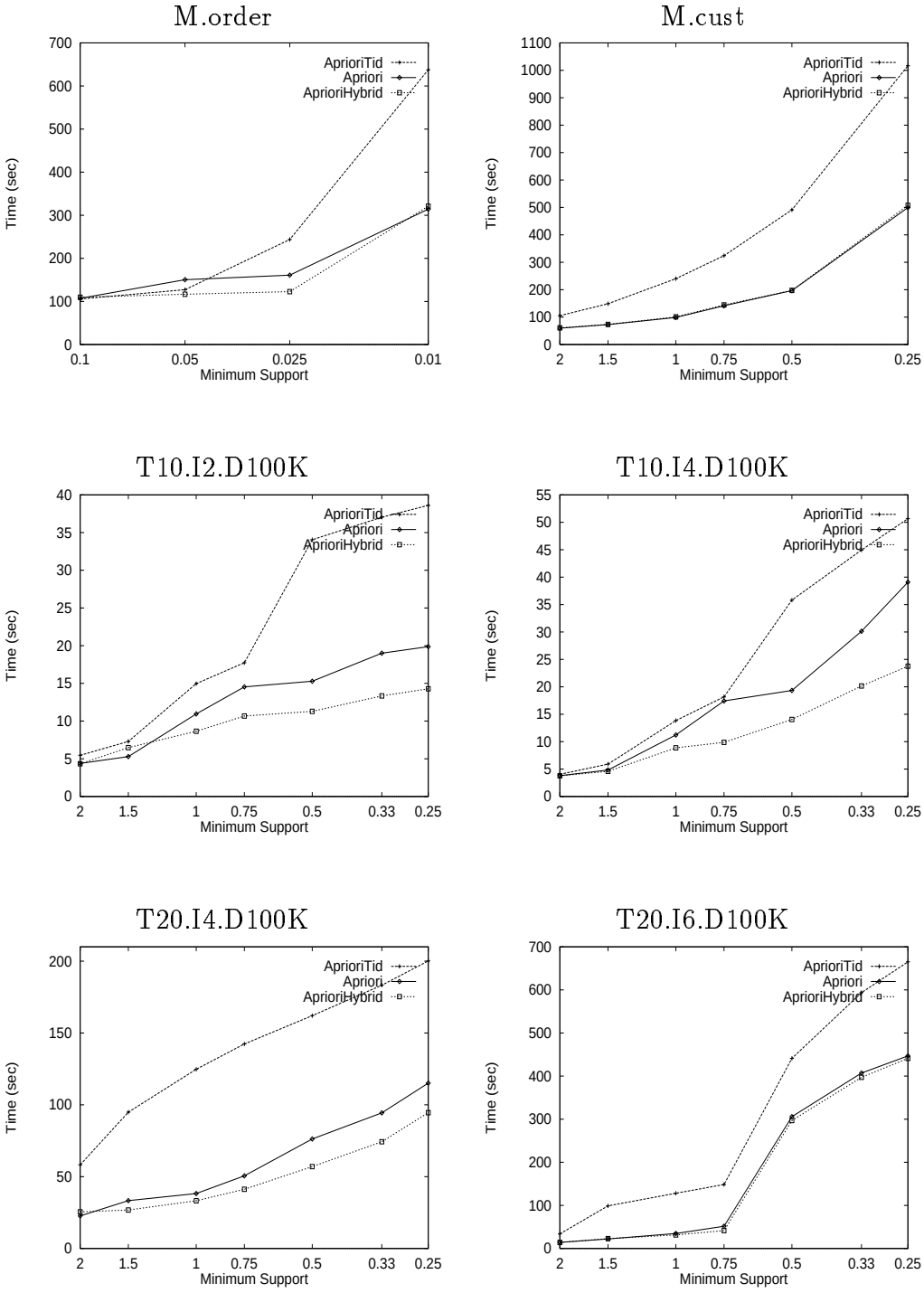
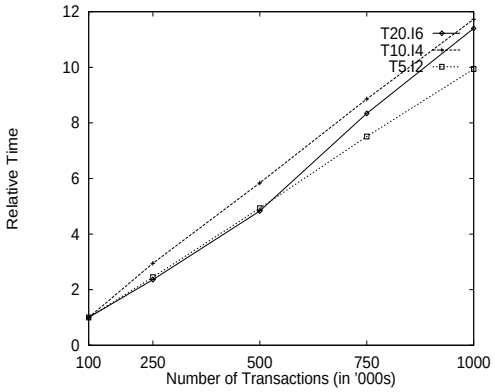


Figure 15: Execution times: AprioriHybrid Algorithm



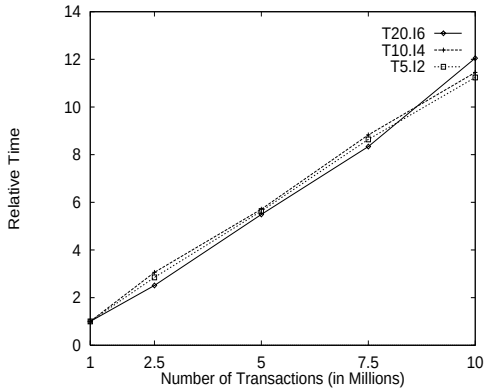


Figure 16: Number of transactions scale-up

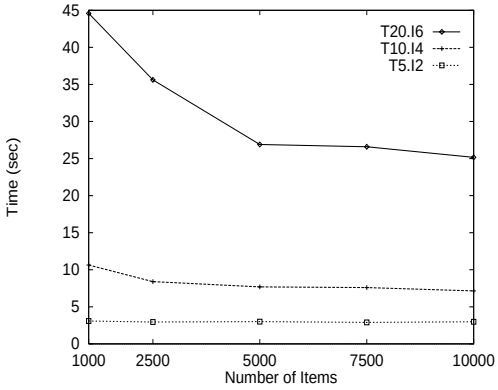


Figure 17: Number of items scale-up

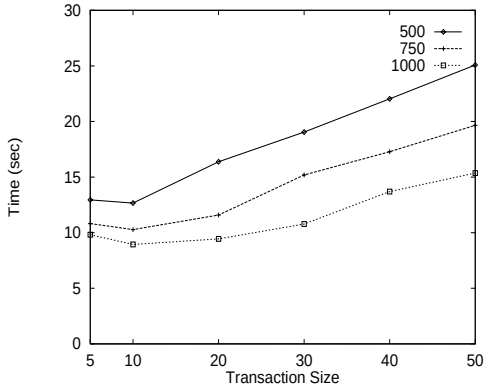


Figure 18: Transaction size scale-up

linearly.

Next, we examined how AprioriHybrid scaled up with the number of items. We increased the number of items from 1000 to 10,000 for the three parameter settings T5.I2.D100K, T10.I4.D100K and T20.I6.D100K. All other parameters were the same as for the data in Table 3. We ran experiments for a minimum support at 0.75%, and obtained the results shown in Figure 17. The execution times decreased a little since the average support for an item decreased as we increased the number of items. This resulted in fewer frequent itemsets and, hence, faster execution times.

Finally, we investigated the scale-up as we increased the average transaction size. The aim of this experiment was to see how our data structures scaled with the transaction

size, independent of other factors like the physical database size and the number of frequent itemsets. We kept the physical size of the database roughly constant by keeping the product of the average transaction size and the number of transactions constant. The number of transactions ranged from 200,000 for the database with an average transaction size of 5 to 20,000 for the database with an average transaction size 50. Fixing the minimum support as a percentage would have led to large increases in the number of frequent itemsets as the transaction size increased, since the probability of a itemset being present in a transaction is roughly proportional to the transaction size. We therefore set the minimum support level as a fixed number of transactions. Hence the minimum support, as a percentage, actually increases as the number of transactions decreases. The results are shown in Figure 18. The numbers in the key (e.g. 500) refer to this minimum support. As shown, the execution times increase with the transaction size, but only gradually. The main reason for the increase was that in spite of setting the minimum support in terms of the number of transactions, the number of frequent itemsets increased with increasing transaction length. A secondary reason was that finding the candidates present in a transaction took a little more time.

Although we only presented scale-up results for AprioriHybrid, we observed nearly identical results for the Apriori algorithm.

2.5 Overview of Follow-on Work

Subsequent to this work, [PCY95] came up with a variation on the Apriori algorithm. Their extension was to hash candidates of size 2 into a hash-table during the first pass, in addition to counting support of items. However, as mentioned in Section 2.2.1, the hash-tree can be implemented as a two-dimensional array during the second pass, since the candidates are generated by essentially taking a cross-product of the set of all frequent

items. There are two advantages to an array implementation. First, there is no function-call overhead, since incrementing the support of candidates contained in the transaction can be done by a 2-level for-loop over the frequent items in the transaction. Second, the memory utilization per candidate is just 4 bytes. The amount of work done per transaction for counting candidates of size 2 with this approach is O(average-number-of-frequent-items-per-transaction^2), while their approach does O(average-number-of-items-per-transaction^2) work. [PCY95] did not use this more sophisticated implementation of the hash-tree for the second pass. so, their experiments showed their approach being faster (due to the function-call overhead for the hash-tree). Their results indicate the importance of this optimization.

Subsequent to this work, [SON95] proposed the Partition algorithm to minimize the number of passes over the data. The idea was to partition the data into small chunks, find the locally frequent itemsets in each chunk, and then make one more pass to find the actual support for the union of all the locally frequent itemsets. The intuition behind this procedure is that any frequent itemset must be locally frequent in at least one of the partitions. While the Apriori algorithm can be used to find the locally frequent itemsets, [SON95] used an algorithm similar to AprioriTid. Their algorithm keeps a list of the TIDs for each frequent itemset, and the support for a candidate (generated using apriori candidate generation) is found by scanning the TID lists of the two frequent itemsets from which this candidate was generated. This algorithm tends to do better than Apriori in later passes, while Apriori tends to do better in earlier passes. In fact, [SON95] used the Apriori algorithm for the second pass. [SON95] assumed that the number of items would be small enough that the first and second passes could be done simultaneously. This is rarely the case for real-life datasets. Thus, in many cases, Partition will require a total of three passes over the data, rather than two. Partition is also susceptible to the

fact that, unless the data is scanned in random order, there may be many locally frequent itemsets which are not globally frequent. However, random I/O would incur disk seeks, and be much slower than sequential I/O. When reading data from files, the bottleneck is currently CPU, not I/O.[5] Hence Partition is probably a good idea if the data is being read from a repository with expensive I/O. In other scenarios, Apriori would probably be a better choice.

2.6 Summary

In this chapter, we presented two new algorithms, Apriori and AprioriTid, for discovering all significant association rules between items in a large database of transactions. We compared these algorithms to the previously known algorithms, the AIS [AIS93b] and SETM [HS95] algorithms. We presented experimental results, using both synthetic and real-life data, showing that the proposed algorithms always outperform AIS and SETM. The performance gap increased with the problem size, and ranged from a factor of three for small problems to more than an order of magnitude for large problems.

We showed how the best features of the two proposed algorithms can be combined into a hybrid algorithm, called AprioriHybrid, which then becomes the algorithm of choice for this problem. Scale-up experiments showed that AprioriHybrid scales linearly with the number of transactions. In addition, the execution time decreases a little as the number of items in the database increases. As the average transaction size increases (while keeping the database size constant), the execution time increases only gradually. These experiments demonstrate the feasibility of using AprioriHybrid in real applications involving very large databases.

Despite its performance advantages, the implementation of AprioriHybrid is more

[5] This will probably change in the future, since CPU speeds are rising faster than disk speeds. However, parallel disks can be used to improve I/O bandwidth.

complex than Apriori. Hence the somewhat worse performance of Apriori, which has the same scale-up characteristics as AprioriHybrid, may be an acceptable tradeoff in many situations.

Chapter 3

Mining Generalized Association Rules

3.1 Introduction

In Chapter 2, we gave fast algorithms for the problem of mining association rules. In many cases, taxonomies (*isa* hierarchies) over the items are available. An example of a taxonomy is shown in Figure 19: this taxonomy says that Jacket *isa* Outerwear, Ski Pants *isa* Outerwear, Outerwear *isa* Clothes, etc. Users are interested in generating rules that span different levels of the taxonomy. For example, we may infer a rule that people who buy Outerwear tend to buy Hiking Boots from the fact that people bought Jackets with Hiking Boots and and Ski Pants with Hiking Boots. However, the support for the rule "Outerwear $\Rightarrow$ Hiking Boots" may not be the sum of the supports for the rules "Jackets $\Rightarrow$ Hiking Boots" and "Ski Pants $\Rightarrow$ Hiking Boots" since some people may have bought Jackets, Ski Pants and Hiking Boots in the same transaction. Also, "Outerwear $\Rightarrow$ Hiking Boots" may be a valid rule, while "Jackets $\Rightarrow$ Hiking Boots" and "Clothes $\Rightarrow$ Hiking Boots" may not. The former may not have minimum support, and the latter may not have minimum confidence.

Finding rules across different levels of the taxonomy is valuable since:

- Rules at lower levels may not have minimum support. Few people may buy Jackets with Hiking Boots, but many people may buy Outerwear with Hiking Boots. Thus many significant associations may not be discovered if we restrict rules to items

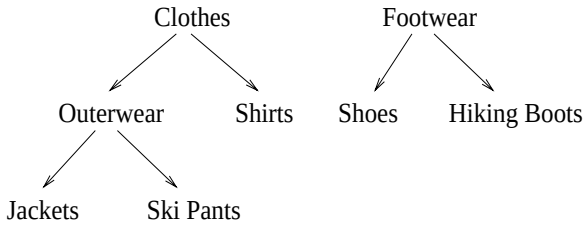


Figure 19: Example of a Taxonomy

at the leaves of the taxonomy. Since department stores or supermarkets typically have hundreds of thousands of items, the support for rules involving only leaf items (typically UPC or SKU codes[1]) tends to be extremely small.

- Taxonomies can be used to prune uninteresting or redundant rules. We will discuss this further in Section 3.2.1.

In general, multiple taxonomies may be present. For example, there could be a taxonomy for the price of items (cheap, expensive, etc.), and another for the category. Multiple taxonomies may be modeled as a single taxonomy which is a DAG (directed acyclic graph). A common application that uses multiple taxonomies is loss-leader analysis. In addition to the usual taxonomy which classifies items into brands, categories, product groups, etc., there is a second taxonomy where items which are on sale are considered to be children of a "items-on-sale" category, and users look for rules containing the "items-on-sale" item.

In this chapter, we introduce the problem of mining *generalized* association rules. Informally, given a set of transactions and a taxonomy, we want to find association rules where the items may be from any level of the taxonomy. We give a formal problem description in Section 3.2. One drawback users experience in applying association rules to real problems is that they tend to get a lot of uninteresting or redundant rules along with the interesting rules. We introduce an interest-measure that uses the taxonomy to

[1]Universal Product Code, Stock Keeping Unit

prune redundant rules.

An obvious solution to the problem is to replace each transaction T with an "extended transaction" T', where T' contains all the items in T as well as all the ancestors of each items in T. For example, if the transaction contained Jackets, we would add Outerwear and Clothes to get the extended-transaction. We can then run any of the algorithms for mining association rules given in Chapter 2 (e.g., Apriori) on the extended transactions to get generalized association rules. However, this "Basic" algorithm is not very fast; two more sophisticated algorithms that we propose in this Chapter will be shown to run 2 to 5 times faster than Basic (and more than 100 times faster on one real-life dataset).

We describe the Basic algorithm and our two algorithms in Section 3.3, and evaluate their performance on both synthetic and real-life data in Section 3.4. Finally, we summarize our work and conclude in Section 3.5.

Related Work

Han and Fu [HF95] concurrently came up with an algorithm similar to our upcoming Cumulate algorithm, based on the Apriori algorithm in Chapter 2. However, the focus of [HF95] is on finding association rules at the same level of the taxonomy, rather than on both rules at the same level and rules across different levels. (Finding rules across different levels is mentioned in a "Discussion" section, without an implementation or a performance evaluation.) Further, their paper focuses on the tradeoffs involved in creating multiple copies of the transaction data (one for each taxonomy level), rather than adding the items in the taxonomy on-the-fly. In many cases, however, there may not be enough disk space to create multiple copies of the data.

3.2 Problem Formulation

Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called items. Let $\mathcal{T}$ be a directed acyclic graph on the literals. An edge in $\mathcal{T}$ represents an *isa* relationship, and $\mathcal{T}$ represents a set of taxonomies. If there is an edge in $\mathcal{T}$ from p to c, we call p a *parent* of c and c a *child* of p (p represents a generalization of c.) We model the taxonomy as a DAG rather than a forest to allow for multiple taxonomies.

We use lower case letters to denote items and upper case letters for sets of items (itemsets). We call $\hat{x}$ an *ancestor* of x (and x a *descendant* of $\hat{x}$) if there is an edge from $\hat{x}$ to x in the transitive-closure of $\mathcal{T}$. Note that a node is not an ancestor of itself, since the graph is acyclic.

Let $\mathcal{D}$ be a set of transactions, where each transaction T is a set of items such that $T \subseteq \mathcal{I}$. (While we expect the items in T to be leaves in $\mathcal{T}$, we do not require this.) We say that a transaction T *supports* an item $x \in \mathcal{I}$ if x is in T or x is an ancestor of some item in T. We say that a transaction T *supports* an itemset $X \subseteq \mathcal{I}$ if T supports every item in the set X.

A *generalized association rule* is an implication of the form $X \Rightarrow Y$, where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, $X \cap Y = \emptyset$, and no item in Y is an ancestor of any item in X. The rule $X \Rightarrow Y$ holds in the transaction set $\mathcal{D}$ with *confidence* c if $c\%$ of transactions in $\mathcal{D}$ that support X also support Y. The rule $X \Rightarrow Y$ has *support* s in the transaction set $\mathcal{D}$ if $s\%$ of transactions in $\mathcal{D}$ support $X \cup Y$. The reason for the condition that no item in Y should be an ancestor of any item in X is that a rule of the form "x $\Rightarrow$ ancestor(x)" is trivially true with 100% confidence, and hence redundant. We call rules that satisfy our definition *generalized* association rules because both X and Y can contain items from any level of the taxonomy $\mathcal{T}$, a possibility not entertained by the formalism introduced in [AIS93b].

We first introduce a problem statement without the interest measure. This definition may lead to many "redundant" rules may be found. We will formalize the notion of redundancy and modify the problem statement accordingly in Section 3.2.1. (We introduce this version of the problem statement here in order to explain redundancy.)

Problem Statement (without interest measure) Given a set of transactions $\mathcal{D}$ and a set of taxonomies $\mathcal{T}$, the problem of mining generalized association rules is to discover all rules that have support and confidence greater than the user-specified minimum support (called *minsup*) and minimum confidence (called *minconf*) respectively.

Example Let $\mathcal{I}$ = {Footwear, Shoes, Hiking Boots, Clothes, Outerwear, Jackets, Ski Pants, Shirts} and $\mathcal{T}$ the taxonomy shown in Figure 19. Consider the database shown in Figure 20. Let minsup be 30% (that is, two transactions) and minconf 60%. Then the sets of items with minimum support (*frequent* itemsets), and the rules corresponding to the these itemsets are shown in Figure 20. Note that the rules "Ski Pants $\Rightarrow$ Hiking Boots" and "Jackets $\Rightarrow$ Hiking Boots" do not have minimum support, but the rule "Outerwear $\Rightarrow$ Hiking Boots" does.

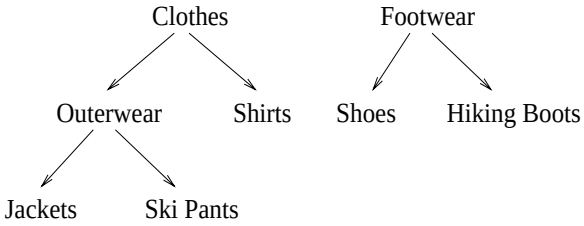
Observation Let $\Pr(X)$ denote the probability that *all* the items in X are contained in a transaction. Then support$(X \Rightarrow Y) = \Pr(X \cup Y)$ and confidence$(X \Rightarrow Y) = \Pr(Y \mid X)$ (since $\Pr(Y \mid X) = \Pr(X \cup Y) / \Pr(X)$). Note that $\Pr(X \cup Y)$ is the probability that all the items in $X \cup Y$ are present in the transaction.

If a set $\{x,y\}$ has minimum support, so do $\{x,\widehat{y}\}$, $\{\widehat{x},y\}$ and $\{\widehat{x},\widehat{y}\}$. ($\widehat{x}$ denotes an ancestor of x). However if the rule $x \Rightarrow y$ has minimum support and confidence, only the rule $x \Rightarrow \widehat{y}$ is guaranteed to have both minimum support and confidence. While the rules $\widehat{x} \Rightarrow y$ and $\widehat{x} \Rightarrow \widehat{y}$ will have minimum support, they may not have minimum

Database $\mathcal{D}$

Transaction	Items Bought
1	Shirt
2	Jacket, Hiking Boots
3	Ski Pants, Hiking Boots
4	Shoes
5	Shoes
6	Jacket

Taxonomy $\mathcal{T}$



Frequent Itemsets

Itemset	Support
{ Jacket }	2
{ Outerwear }	3
{ Clothes }	4
{ Shoes }	2
{ Hiking Boots }	2
{ Footwear }	4
{ Outerwear, Hiking Boots }	2
{ Clothes, Hiking Boots }	2
{ Outerwear, Footwear }	2
{ Clothes, Footwear }	2

Rules

Rule	Support	Conf.
Outerwear $\Rightarrow$ Hiking Boots	33%	66.6%
Outerwear $\Rightarrow$ Footwear	33%	66.6%
Hiking Boots $\Rightarrow$ Outerwear	33%	100%
Hiking Boots $\Rightarrow$ Clothes	33%	100%

Figure 20: Example of associations with taxonomies

confidence. For example, in Figure 20, the confidence of "Outerwear $\Rightarrow$ Hiking boots" is 66.6% while that of "Clothes $\Rightarrow$ Hiking Boots" is 50%, less than minimum support.

It is important to note that the support for an item in the taxonomy is *not* equal to the sum of the supports of its children, since several of the children could be present in a single transaction. Hence we cannot directly infer rules about items at higher levels of the taxonomy from rules about the leaves.

3.2.1 Interest Measure

Since the output of the associations algorithm can be quite large, we would like to identify the interesting or useful rules. [ST95] looks at subjective measures of interestingness and suggests that a pattern is interesting if it is unexpected (surprising to the user) and/or actionable (the user can do something with it). [ST95] also distinguishes between subjective and objective interest measures. Previous work on quantifying the "usefulness" or "interest" of a rule has focused on by how much the support of a rule exceeds its expected support based on independence between the antecedent and consequent. For example, Piatetsky-Shapiro [PS91] argues that a rule $X \Rightarrow Y$ is not interesting if $\text{support}(X \Rightarrow Y) \approx \text{support}(X) \times \text{support}(Y)$. We implemented this idea and used the chi-square value to check if the rule was statistically significant. (Intuitively, the test looks at whether the support was different enough from the expected support for the correlation between the antecedent and consequent to be statistically significant.) However, this measure did not prune many rules; on two real-life datasets (described in Section 3.4.5), less than 1% of the generalized association rules were found to be redundant (not statistically significant). In this section, we use the information in taxonomies to derive a new interest measure that prunes out 40% to 60% of the rules as "redundant" rules.

To motivate our approach, consider the rule

Milk $\Rightarrow$ Cereal (8% support, 70% confidence)

If "Milk" is a parent of "Skim Milk", and about a quarter of sales of "Milk" are "Skim Milk", we would expect the rule

Skim Milk $\Rightarrow$ Cereal

to have 2% support and 70% confidence. If the actual support and confidence for "Skim Milk $\Rightarrow$ Cereal" are around 2% and 70% respectively, the rule can be considered redundant since it does not convey any additional information and is less general than the first rule. We capture this notion of "interest" by saying that we only want to find rules whose support is more than R times the expected value or whose confidence is more than R times the expected value, for some user-specified constant R.[2] We formalize the above intuition below.

We call $\widehat{Z}$ an *ancestor* of Z (where $Z, \widehat{Z}$ are sets of items such that $Z, \widehat{Z} \subseteq \mathcal{I}$) if we can get $\widehat{Z}$ from Z by replacing one or more items in Z with their ancestors and Z and $\widehat{Z}$ have the same number of items. (The reason for the latter condition is that it is not meaningful to compute the expected support of Z from $\widehat{Z}$ unless they have the same number of items. For instance, the support for {Clothes} does give any clue about the expected support for {Outerwear, Shirts}.) We call the rules $\widehat{X} \Rightarrow Y$, $\widehat{X} \Rightarrow \widehat{Y}$ or $X \Rightarrow \widehat{Y}$ ancestors of the rule $X \Rightarrow Y$. Given a set of rules, we call $\widehat{X} \Rightarrow \widehat{Y}$ a *close ancestor* of

[2] We can easily enhance this definition to say that we want to find rules with minimum support whose support (or confidence) is either more or less than the expected value. However, many rules whose support is less than expected will not have minimum support. As a result, the most interesting negative rules, with the largest deviation from expected support, are unlikely to have minimum support. Hence to find all rules whose support is less than their expected support by at least some user-specified minimum difference, we should check candidates without minimum support to see if their support differs from their expected support by this amount before discarding them. We defer implementation of this idea to future work.

$X \Rightarrow Y$ if there is no rule $X' \Rightarrow Y'$ such that $X' \Rightarrow Y'$ is an ancestor of $X \Rightarrow Y$ and $\widehat{X} \Rightarrow \widehat{Y}$ is an ancestor of $X' \Rightarrow Y'$. (Similar definitions apply for $X \Rightarrow \widehat{Y}$ and $\widehat{X} \Rightarrow Y$.)

Consider a rule $X \Rightarrow Y$, and let $Z = X \cup Y$. The support of Z will be the same as the support of the rule $X \Rightarrow Y$. Let $E_{\Pr(\widehat{Z})}[\Pr(Z)]$ denote the "expected" value of $\Pr(Z)$ given $\Pr(\widehat{Z})$, where $\widehat{Z}$ is an ancestor of Z. Let $Z = \{z_1, \ldots, z_n\}$ and $\widehat{Z} = \{\widehat{z_1}, \ldots, \widehat{z_j}, z_{j+1}, \ldots, z_n\}$, $1 \leq j \leq n$, where $\widehat{z_i}$ is an ancestor of z_i. Then we define

$$E_{\Pr(\widehat{Z})}[\Pr(Z)] = \frac{\Pr(z_1)}{\Pr(\widehat{z_1})} \times \cdots \times \frac{\Pr(z_j)}{\Pr(\widehat{z_j})} \times \Pr(\widehat{Z}).$$

to be the expected value of $\Pr(Z)$ given the itemset $\widehat{Z}$.[3]

Similarly, let $E_{\Pr(\widehat{Y} \mid \widehat{X})}[\Pr(Y \mid X)]$ denote the "expected" confidence of the rule $X \Rightarrow Y$ given the rule $\widehat{X} \Rightarrow \widehat{Y}$. Let $Y = \{y_1, \ldots, y_n\}$ and $\widehat{Y} = \{\widehat{y_1}, \ldots, \widehat{y_j}, y_{j+1}, \ldots, y_n\}$, $1 \leq j \leq n$, where $\widehat{y_i}$ is an ancestor of y_i. Then we define

$$E_{\Pr(\widehat{Y} \mid \widehat{X})}[\Pr(Y \mid X)] = \frac{\Pr(y_1)}{\Pr(\widehat{y_1})} \times \cdots \times \frac{\Pr(y_j)}{\Pr(\widehat{y_j})} \times \Pr(\widehat{Y} \mid \widehat{X})$$

Note that $E_{\widehat{X} \Rightarrow Y}[\Pr(Y \mid X)] = \Pr(Y \mid \widehat{X})]$.

Definition of Interesting Rules Given a user-specified minimum interest R, we call a rule $X \Rightarrow Y$ *R-interesting* w.r.t an ancestor $\widehat{X} \Rightarrow \widehat{Y}$ if the support of the rule $X \Rightarrow Y$ is R times the expected support based on $\widehat{X} \Rightarrow \widehat{Y}$, or the confidence is R times the expected confidence based on $\widehat{X} \Rightarrow \widehat{Y}$.

Given a set of rules S and a minimum interest R, a rule $X \Rightarrow Y$ is *interesting* (in S) if it has no ancestors or it is R-interesting with respect to its close ancestors among its interesting ancestors. We say that an rule $X \Rightarrow Y$ is *partially interesting* (in S) if it

[3] Alternate definitions are possible. For example, we could define:

$$E_{\Pr(\widehat{Z})}[\Pr(Z)] = \frac{\Pr(\{z_1, \ldots, z_j\})}{\Pr(\{\widehat{z_1}, \ldots, \widehat{z_j}\})} \times \Pr(\widehat{Z}).$$

Rule #	Rule	Support
1	Clothes ⇒ Footwear	10%
2	Outerwear ⇒ Footwear	8%
3	Jackets ⇒ Footwear	4%

Item	Support
Clothes	5
Outerwear	2
Jackets	1

Figure 21: Example for Interest Measure definition

has no ancestors or is R-interesting with respect to at least one close ancestor among its interesting ancestors.

We motivate the reason for only considering close ancestors among all interesting ancestors with an example. Consider the rules shown in Figure 21. The support for the items in the antecedent are shown alongside. Assume we have the same taxonomy as in the previous example. Rule 1 has no ancestors and is hence interesting. The support for rule 2 is twice the expected support based on rule 1, and is thus interesting. The support for rule 3 is exactly the expected support based on rule 2, but twice the support based on rule 1. We do not want consider rule 3 to be interesting since its support can be predicted based on rule 2, even though its support is more than expected if we ignore rule 2 and look at rule 1.

3.2.2 Problem Statement

Given a set of transactions $\mathcal{D}$ and a user-specified minimum interest (called *min-interest*), the problem of mining association rules with taxonomies is to find all interesting association rules that have support and confidence greater than the user-specified minimum support (called *minsup*) and minimum confidence (called *minconf*) respectively.

For some applications, we may want to find partially interesting rules rather than just interesting rules. Note that if min-interest = 0, all rules are found, regardless of interest.

3.3 Algorithms

The problem of discovering generalized association rules can be decomposed into three parts:

1. Find all sets of items (*itemsets*) whose support is greater than the user-specified minimum support. Itemsets with minimum support are called *frequent* itemsets.

2. Use the frequent itemsets to generate the desired rules. The general idea is that if, say, $ABCD$ and AB are frequent itemsets, then we can determine if the rule $AB \Rightarrow CD$ holds by computing the ratio $conf = \text{support}(ABCD)/\text{support}(AB)$. If $conf \geq minconf$, then the rule holds. (The rule will have minimum support because $ABCD$ is frequent.)

3. Prune all uninteresting rules from this set.

In the rest of this section, we look at algorithms for finding all frequent itemsets where the items can be from any level of the taxonomy. Given the frequent itemsets, the algorithm in Section 2.3 can be used to generate rules. We first describe the obvious approach for finding frequent itemsets, and then present our two algorithms.

3.3.1 Basic

Consider the problem of deciding whether a transaction T supports an itemset X. If we take the raw transaction, this involves checking for each item $x \in X$ whether x or some descendant of x is present in the transaction. The task become much simpler if we first add to T all the ancestors of each item in T; let us call this extended transaction T'. Now T supports X if and only if T' is a superset of X. Hence a straight-forward way to find generalized association rules would be to run any of the algorithms for finding association

k-itemset	An itemset having k items.
L_k	Set of frequent k-itemsets (those with minimum support).
C_k	Set of candidate k-itemsets (potentially frequent itemsets).
$\mathcal{T}$	Taxonomy

Figure 22: Notation for Algorithms

rules from Chapter 2 on the extended transactions. We discuss below the generalization of the Apriori algorithm given in Section 2.2.1. Figure 23 gives an overview of the algorithm using the notation in Figure 22, with the additions to the Apriori algorithm highlighted.

The first pass of the algorithm simply counts item occurrences to determine the frequent 1-itemsets. Note that items in the itemsets can come from the leaves of the taxonomy or from interior nodes. A subsequent pass, say pass k, consists of two phases. First, the frequent itemsets L_{k-1} found in the $(k-1)$th pass are used to generate the candidate itemsets C_k, using the apriori candidate generation function described in Section 2.2.1. Next, the database is scanned and the support of candidates in C_k is counted. For fast counting, we need to efficiently determine the candidates in C_k that are contained in a given transaction T. We reuse the hash-tree data structure described in Section 2.2.1 for this purpose.

3.3.2 Cumulate

We now add several optimizations to the Basic algorithm to develop the algorithm Cumulate. The name indicates that all itemsets of a certain size are counted in one pass, unlike the Stratify algorithm in Section 3.3.3.

1. **Filtering ancestors.** We do not have to add all ancestors of the items in a transaction T to T. Instead, we just need to add ancestors that are in one (or more) of the candidate itemsets being counted in the current pass. In fact, if the

L_1 := {frequent 1-itemsets}; // *includes items at any level.*
$k := 2$; // k represents the pass number
while ($L_{k-1} \neq \emptyset$) **do**
begin
  C_k := New candidates of size k generated from L_{k-1}.
  forall transactions $T \in \mathcal{D}$ **do**
  begin
    Add all ancestors of each item in T to T, removing any duplicates.
    Increment the count of all candidates in C_k that are contained in T.
  end
  L_k := All candidates in C_k with minimum support.
  $k := k + 1$;
end
Answer := $\bigcup_k L_k$;

Figure 23: Basic Algorithm

original item is not in any of the itemsets, it can be dropped from the transaction.

For example, assume the parent of "Jacket" is "Outerwear", and the parent of "Outerwear" is "Clothes". Let {Clothes, Shoes} be the only itemset being counted. Then, in any transaction containing Jacket, we simply replace Jacket by Clothes. We do not need to keep Jacket in the transaction, nor do we need to add Outerwear to the transaction.

2. **Pre-computing ancestors.** Rather than finding ancestors for each item by traversing the taxonomy graph, we can pre-compute the ancestors for each item. We can drop ancestors that are not present in any of the candidates when pre-computing ancestors.

3. **Pruning itemsets containing an item and its ancestor.** We first present two lemmas to justify this optimization.

   Lemma 3 *The support for an itemset X that contains both an item x and its ancestor $\widehat{x}$ will be the same as the support for the itemset $X - \widehat{x}$.*

Proof: Clearly, any transaction that supports X will also support $X - \widehat{x}$, since $X - \widehat{x} \subset X$. By definition, any transaction that supports x supports $\widehat{x}$. Hence any transaction that supports $X - \widehat{x}$ will also support X. □

Lemma 4 *If L_k, the set of frequent k-itemsets, does not include any itemset that contains both an item and its ancestor, the set of candidates C_{k+1} generated by the candidate generation procedure in Section 3.3.1 will not include any itemset that contains both an item and its ancestor.*

Proof: Assume that the candidate generation procedure generates a candidate X that contains both an item x and its ancestor $\widehat{x}$. Let X' be any subset of X with k items that contains both x and $\widehat{x}$. Since X was not removed in the prune step of candidate generation, X' must have been in L_k. But this contradicts the statement that no itemset in L_k includes both an item and its ancestor. □

Lemma 3 shows that we need not count any itemset which contains both an item and its ancestor. We add this optimization by pruning the candidate itemsets of size two which consist of an item and its ancestor. Lemma 4 shows that pruning these candidates is sufficient to ensure that we never generate candidates in subsequent passes which contain both an item and its ancestor.

Figure 24 gives an overview of the Cumulate algorithm, with the additions to the Basic algorithm highlighted.

3.3.3 EstMerge

We motivate this next algorithm with an example. Let {Clothes, Shoes}, {Outerwear, Shoes} and {Jacket, Shoes} be candidate itemsets to be counted, with "Jacket" being

Compute $\mathcal{T}^*$, the set of ancestors of each item, from $\mathcal{T}$. // Opt. 2
L_1 := {frequent 1-itemsets};
$k := 2$; // k represents the pass number
while ($L_{k-1} \neq \emptyset$) **do**
begin
  C_k := New candidates of size k generated from L_{k-1}.
  if *($k = 2$)* **then**
    Delete any candidate in C_2 that consists of an item and its ancestor. // Opt. 3
  Delete all items in $\mathcal{T}^*$ that are not present in any of the candidates in C_k. // Opt. 1
  forall transactions $T \in \mathcal{D}$ **do**
  begin
    foreach item $x \in T$ **do**
      Add all ancestors of x in $\mathcal{T}^*$ to T.
    Remove any duplicates from T.
    Increment the count of all candidates in C_k that are contained in T.
  end
  L_k := All candidates in C_k with minimum support.
  $k := k + 1$;
end
Answer := $\bigcup_k L_k$;

Figure 24: Cumulate Algorithm

the child of "Outerwear", and "Outerwear" the child of "Clothes". If {Clothes, Shoes} does not have minimum support, we do not have to count either {Outerwear, Shoes} or {Jacket, Shoes}. Thus, rather than counting all candidates of a given size in the same pass as in Cumulate, it may be faster to first count the support of {Clothes, Shoes}, then count {Outerwear, Shoes} if {Clothes, Shoes} turns out to have minimum support, and finally count {Jacket, Shoes} if {Outerwear, Shoes} also has minimum support. Of course, the extra cost in making multiple passes over the database may cost more than the benefit of counting fewer itemsets. We will discuss this tradeoff in more detail shortly.

We develop this algorithm by first presenting a straight-forward version, Stratify, and then describing the use of sampling to increase its effectiveness, yielding the Estimate and EstMerge versions. The optimizations that we introduced for the Cumulate algorithm apply to this algorithm as well.

Stratify

Consider the partial ordering induced by the taxonomy DAG on a set of itemsets. Itemsets with no parents are considered to be at depth 0. For other itemsets, the depth of an itemset X is defined to be $(\max(\{\text{depth}(\widehat{X}) \mid \widehat{X} \text{ is a parent of } X\}) + 1)$.

We first count all itemsets C_0 at depth 0. After deleting candidates that are descendants of those itemsets in C_0 that did not have minimum support, we count the remaining itemsets at depth 1 (C_1). After deleting candidates that are descendants of the itemsets in C_1 without minimum support, we count the itemsets at depth 2, etc. If there are only a few candidates at depth n, we can count candidates at different depths (n, $n+1$, ...) together to reduce the overhead of making multiple passes.

There is a tradeoff between the number of itemsets counted (CPU time) and the number of passes over the database (IO+CPU time). One extreme would be to make a pass over the database for the candidates at each depth. This would result in a minimal number of itemsets being counted, but we may waste a lot of time in scanning the database multiple times. The other extreme would be to make just one pass for all the candidates, which is what Cumulate does. This would result in counting many itemsets that do not have minimum support and whose parents do not have minimum support. In our implementation of Stratify, we used the heuristic (empirically determined) that we should count at least 20% of the candidates in each pass.

Estimate

Rather than hoping that candidates which include items at higher levels of the taxonomy will not have minimum support, resulting in our not having to count candidates which include items at lower levels, we can use sampling as discussed below to estimate the support of candidates. We then count candidates that are expected to have minimum

support as well as candidates that are not expected to have minimum support but all of whose parents have minimum support. (We call this set C'_k, for candidates of size k.) We expect that the latter candidates will not have minimum support, and hence we will not have to count any of the descendants of those candidates. If some of those candidates turn out to have minimum support, we make an extra pass to count their descendants. (We call this set of candidates C''_k.) Note that if we only count candidates that are expected to have minimum support, we would have to make another pass to count their children, since we can only be sure that their children do not have minimum support if we actually count them.

In our implementation, we included candidates in C'_k whose support in the sample was 0.9 times the minimum support, and candidates all of whose parents had 0.9 times the minimum support, in order to reduce the effect of sampling error. We will discuss the effect of changing this sampling error margin shortly, when we also discuss how the sample size can be chosen. Given the sample size, we chose the sample during the first pass as follows. We generate a random number n between 1 and 2/sampling frequency, and add the nth transaction to the sample. If the next random number is m, we add the mth transaction after the previous transaction that was chosen to the sample and so on.

Example For example, consider the three candidates and two support scenarios shown in Figure 25. Let "Jacket" be a child of "Outerwear" and "Outerwear" a child of "Clothes". Let minimum support be 5%, and let the support for the candidates in a sample of the database be as shown in Figure 25. Hence, based on the sample, we expect only {Clothes, Shoes} to have minimum support over the database. We now find the support of *both* {Clothes, Shoes} and {Outerwear, Shoes} over the entire database. We count {Outerwear, Shoes} even though we do not expect it to have minimum support

Candidate Itemsets	Support in Sample	Support in Database	
		Scenario A	Scenario B
{Clothes, Shoes}	8%	7%	9%
{Outerwear, Shoes}	4%	4%	6%
{Jacket, Shoes}	2%		

Figure 25: Example for Estimate Algorithm

since we will not know for sure whether it has minimum support unless {Clothes, Shoes} does not have minimum support, and we expect {Clothes, Shoes} to have minimum support. Now, in scenario A, we do not have to find the support for {Jacket, Shoes} since {Outerwear, Shoes} does not have minimum support (over the entire database). However, in scenario B, we have to make an extra pass to count {Jacket, Shoes}.

EstMerge

Since the estimate (based on the sample) of which candidates have minimum support has some error, Estimate often makes a second pass to count the support for the candidates in C_k'' (the descendants of candidates in C_k that were wrongly expected to not have minimum support.) The number of candidates counted in this pass is usually small. Rather than making a separate pass to count these candidates, we can count them when we count candidates in C_{k+1}. However, since we do not know if the candidates in C_k'' will have minimum support or not, we assume all these candidates to be frequent when generating C_{k+1}. That is, we will consider L_k to be those candidates in C_k' with minimum support, as well as all candidates in C_k'', when generating C_{k+1}. This can generate more candidates in C_{k+1} than would be generated by Estimate, but does not affect correctness. The tradeoff is between the extra candidates counted by EstMerge against the extra pass made by Estimate. An overview of the EstMerge algorithm is given in Figure 26. (All the optimizations introduced for the Cumulate algorithm apply here, though we have omitted them in the figure.)

L_1 := {frequent 1-itemsets};
Generate $\mathcal{D}_S$, a sample of the database, in the first pass;
$k := 2$; // k represents the pass number
$C''_1 := \emptyset$; // C''_k represents candidates of size k to be counted with candidates of size $k+1$.
while ($L_{k-1} \neq \emptyset$ **or** $C''_{k-1} \neq \emptyset$) **do**
begin
  C_k := New candidates of size k generated from $L_{k-1} \cup C''_{k-1}$.
  Estimate the support of the candidates in C_k by making a pass over $\mathcal{D}_S$.
  C'_k := Candidates in C_k that are expected to have minimum support and candidates
    all of whose parents are expected to have minimum support.
  Find the support of the candidates in $C'_k \cup C''_{k-1}$ by making a pass over $\mathcal{D}$.
  Delete all candidates in C_k whose ancestors (in C'_k) do not have minimum support.
  C''_k := Remaining candidates in C_k that are not in C'_k.
  L_k := All candidates in C'_k with minimum support.
  Add all candidates in C''_{k-1} with minimum support to L_{k-1}.
  $k := k + 1$;
end
Answer := $\bigcup_k L_k$;

Figure 26: Algorithm EstMerge

Size of Sample

We now discuss how to select the sample size for estimating the support of candidates. Let p be the support (as a fraction) of a given itemset X. Consider a random sample with replacement of size n from the database. The number of transactions in the sample that contain X is a random variable s with binomial distribution of n trials, each having success probability p. We use the abbreviation $s \succeq k$ ("s is at least as extreme as k") defined by

$$s \succeq k \iff \begin{cases} x \geq k & \text{if } k \geq pn \\ x \leq k & \text{if } k < pn \end{cases}$$

Using Chernoff bounds [HR90] [AS92], the probability that the fractional support in the sample is at least as extreme as a is bounded by

$$Pr[s \succeq an] \leq \left[\left(\frac{p}{a}\right)^a \left(\frac{1-p}{1-a}\right)^{1-a} \right]^n \tag{1}$$

Table 5 presents probabilities that the support of an itemset in the sample is less than

	$p = 5\%$		$p = 1\%$		$p = 0.1\%$	
	a = .8p	a = .9p	a = .8p	a = .9p	a = .8p	a = .9p
$n = 1000$	0.32	0.76	0.80	0.95	0.98	0.99
$n = 10,000$	0.00	0.07	0.11	0.59	0.80	0.95
$n = 100,000$	0.00	0.00	0.00	0.01	0.12	0.60
$n = 1,000,000$	0.00	0.00	0.00	0.00	0.00	0.01

Table 5: Pr[support in sample $< a$], given values for the sample size n, the real support p and a

a when its real support is p, for various sample sizes n. For example, given a sample size of 10,000 transactions, the probability that the estimate of a candidate's support is less than 0.8% when its real support is 1% is less than 0.11.

Equation 1 suggests that the sample size should increase as the minimum support decreases. Also, the probability that the estimate is off by more than a certain fraction of the real support depends only on the sample size, not on the database size. Experiments showing the effect of sample size on the running time are given in Section 3.4.2.

3.4 Performance Evaluation

In this section, we evaluate the performance of the three algorithms on both synthetic and real-life datasets. First, we describe the synthetic data generation program in Section 3.4.1. We present some preliminary results comparing the three variants of the stratification algorithm and the effect of changing the sample size in Section 3.4.2. We then give the performance evaluation of the three algorithms on synthetic data in Section 3.4.3. We do a reality check of our results on synthetic data by running the algorithms against two real-life data sets in Section 3.4.4. Finally, we look at the effectiveness of the interest measure in pruning redundant rules in Section 3.4.5.

We performed our experiments on an IBM RS/6000 250 workstation with 128 MB of main memory running AIX 3.2.5. The data resided in the AIX file system and was

Parameter		Default Value
$\|\mathcal{D}\|$	Number of transactions	1,000,000
$\|T\|$	Average size of the transactions	10
$\|I\|$	Average size of the maximal potentially frequent itemsets	4
$\|\mathcal{I}\|$	Number of maximal potentially frequent itemsets	10,000
N	Number of items	100,000
R	Number of roots	250
L	Number of levels	4-5
F	Fanout	5
D	Depth-ratio $\left(\approx \frac{\text{probability that item in a rule comes from level } i}{\text{probability that item comes from level } i+1}\right)$	1

Table 6: Parameters for Synthetic Data Generation (with default values)

stored on a local 2GB SCSI 3.5" drive with measured sequential throughput of about 2 MB/second.

3.4.1 Synthetic Data Generation

Our synthetic data generation program is a generalization of the algorithm discussed in Section 2.4.3; the addition is the incorporation of taxonomies. The various parameters and their default vales are shown in Table 6. We now describe the extensions to the data generation algorithm in more detail.

The essential idea behind the synthetic data generation program in Section 2.4.3 was to first generate a table of potentially frequent itemsets $\mathcal{I}$, and then generate transactions by picking itemsets from $\mathcal{I}$ and inserting them in the transaction.

To extend this algorithm, we first build a taxonomy over the items.[4] For simplicity, we modeled the taxonomy as a forest rather than a DAG. For any internal node, the number of children is picked from a Poisson distribution with mean μ equal to fanout F. We first assign children to the roots, then to the nodes at depth 2, and so on, till we run out of items. With this algorithm, it is possible for the leaves of the taxonomy to be at

[4]Out of the four parameters R, L, F and N, only three need to be specified since any three of these determine the fourth parameter.

two different levels; this allows us to change parameters like the fanout or the number of roots in small increments.

Each item in the taxonomy tree (including non-leaf items) has a weight associated with it, which corresponds to the probability that the item will be picked for a frequent itemset. The weights are distributed such that the weight of an interior node x equals the sum of the weights of all its children divided by the depth-ratio. Thus with a high depth-ratio, items will be picked from the leaves or lower levels of the tree, while with a low depth-ratio, items will be picked from higher up the tree.

Each itemset in $\mathcal{I}$ has a weight associated with it, which corresponds to the probability that this itemset will be part of a transaction. This weight is picked from an exponential distribution with unit mean, and then multiplied by the geometric mean of the probabilities of all the items in the itemset. The weights are later normalized so that the sum of the weights for all the itemsets in $\mathcal{I}$ is 1. The next itemset to be put in a transaction is chosen from $\mathcal{I}$ by tossing an $|\mathcal{I}|$-sided weighted coin, where the weight for a side is the probability of picking the associated itemset.

When an itemset X in $\mathcal{I}$ is picked for adding to a transaction, it is first "specialized". For each item $\widehat{x}$ in X which is not a leaf in the taxonomy, we descend the subtree rooted at $\widehat{x}$ till we reach a leaf x, and replace $\widehat{x}$ with x. At each node, we decide what branch to follow by tossing a k-sided weighted coin, where k is the number of children, and the weights correspond to the weights of the children. We use this more complex model, rather than just generate itemsets in $\mathcal{I}$ with all the items as leaves, so that the depth-ratio can be varied.

We generate transactions as follows. We first determine the size of the next transaction. The size is picked from a Poisson distribution with mean μ equal to $|T|$. We then assign items to the transaction. Each transaction is assigned a series of potentially

frequent itemsets. The next itemset to be added to the transaction is chosen as described earlier. If the itemset on hand does not fit in the transaction, the itemset is put in the transaction anyway in half the cases, and the itemset is moved to the next transaction the rest of the cases. As before, only some of the items in the itemset are added to the transaction since items in a frequent itemset may not always be bought together. (Details are given in Section 2.4.3.)

3.4.2 Preliminary Experiments

Stratification Variants The results of comparing the three variants of the stratification algorithm on the default synthetic data are shown in Figure 27. At high minimum support, when there are only a few rules and most of the time is spent scanning the database, the performance of the three variants is nearly identical. At low minimum support, when there are more rules, EstMerge does slightly better than Estimate and significantly better than Stratify. The reason is that even though EstMerge counts a few more candidates than Estimate and Stratify, it makes fewer passes over the database, resulting in better performance.

Although we do not show the performance of Stratify and Estimate in the graphs in Section 3.4.3, the results were very similar to those in Figure 27. Both Estimate and Stratify always did somewhat worse than EstMerge, with Estimate beating Stratify.

Size of Sample We changed the size of the EstMerge sample from 0.25% to 8% on the synthetic data set generated with the default values. The results are shown in Figure 28. The running time was higher at both low sample sizes and high sample sizes. In the former case, the decrease in performance was due to the greater error in estimating which itemsets would have minimum support. In the latter case, it was due to the sampling overhead. Notice that the curve is quite flat around the minimum time at 2%;

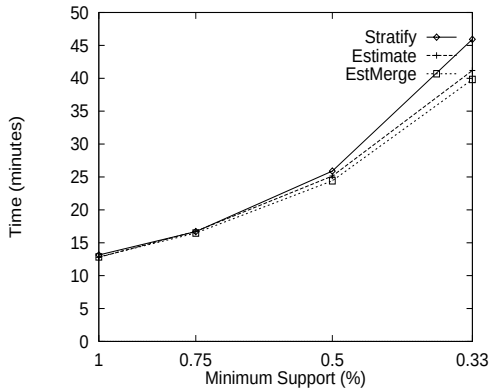


Figure 27: Variants of Stratify

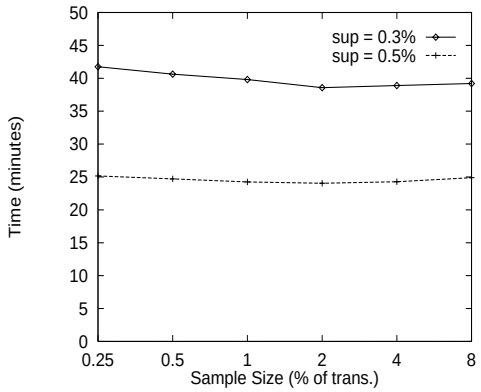


Figure 28: Changing Sample Size

there is no significant difference in performance if we sample a little less or a little more than 2%.

3.4.3 Comparison of Basic, Cumulate and EstMerge

We performed 6 experiments on synthetic datasets, changing a different parameter in each experiment. The results are shown in Figure 29. All the parameters except the one being varied were set to their default values. The minimum support was 0.5% (except for the first experiment, which varies minimum support). We obtained similar results at other levels of support, though the gap between the algorithms typically increased as we lowered the support.

Minimum Support: We changed minimum support from 2% to 0.3%. Cumulate and EstMerge were around 3 to 4 times faster than Basic, with the performance gap increasing as the minimum support decreased. At higher support levels, most of the candidates were at the upper levels of the taxonomy. Hence the optimizations of the Cumulate algorithm had less impact than at lower levels of support. At high support, Cumulate and EstMerge took about the same time since there were only a few rules and most of the time was spent scanning the database. At low support, EstMerge was

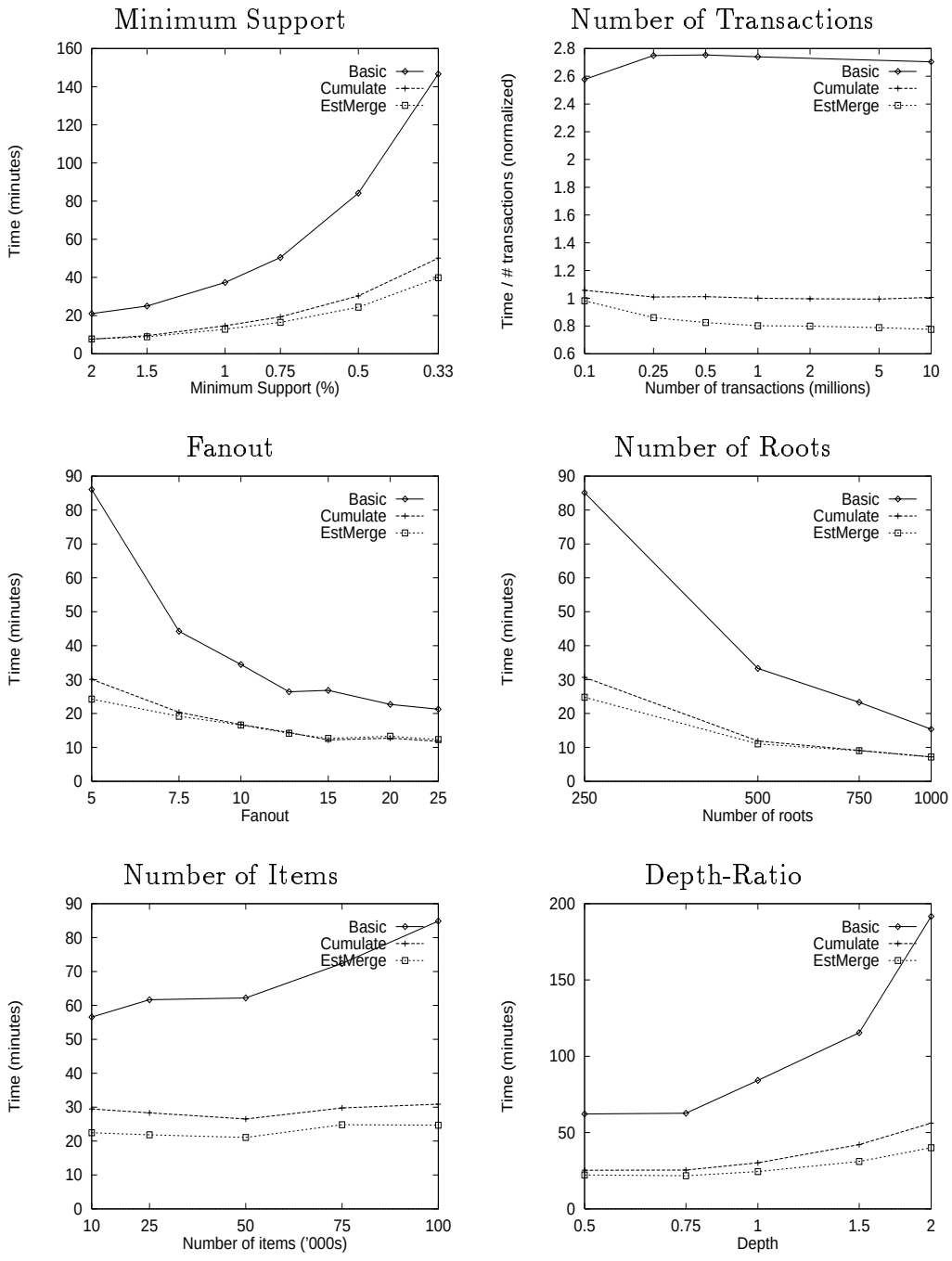


Figure 29: Experiments on Synthetic Data

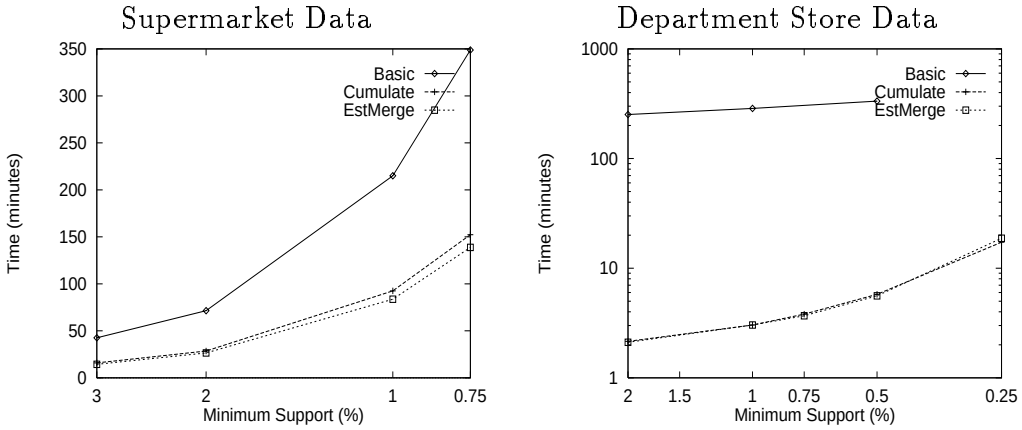


Figure 30: Comparison of algorithms on real data

about 20% faster than Cumulate since there were more candidates at lower levels of the taxonomy and such candidates are more likely to have ancestors without minimum support.

Number of Transactions: We varied the number of transactions from 100,000 to 10 million. Rather than showing the elapsed time, the graph shows the elapsed time divided by the number of transactions, such that the time taken by Cumulate for 1 million transactions is 1 unit. Again, EstMerge and Cumulate perform much better than Basic. The ratio of the time taken by EstMerge to the time taken by Cumulate decreases (i.e. improves) as the number of transactions increases, because when the sample size is a constant percentage, the accuracy of the estimates of the support of the candidates increases as the number of transactions increases.

Fanout: We changed the fanout from 5 to 25. This corresponded to decreasing the number of levels. While EstMerge did about 25% better than Cumulate at fanout 5, the performance advantage decreased as the fanout increased, and the two algorithms did about the same at high fanout. The reason is that at a fanout of 25, the leaves of the taxonomy were either at level 2 or level 3. Hence the percentage of candidates that

could be pruned by sampling became very small and EstMerge was not able to count significantly fewer candidates than Cumulate. The performance gap between Basic and the other algorithms decreases somewhat at high fanout since there were fewer rules and a greater fraction of the time was spent just scanning the database.

Number of Roots: We increased the number of roots from 250 to 1000. As shown by the figure, increasing the number of roots has an effect similar to decreasing the minimum support. The reason is that as the number of roots increases, the probability that a specific root is present in a transaction decreases.

Number of Items/Levels: We varied the number of items from 10,000 to 100,000. The main effect is to change the number of levels in the taxonomy tree, from most of the leaves being at level 3 (with a few at level 4) at 10,000 items to most of the leaves being at level 5 (with a few at level 4) at 100,000 items. Changing the number of items did not significantly affect the performance of Cumulate and EstMerge, but it did increase the time taken by Basic. Since few of the items in frequent itemsets come from the leaves of the taxonomy, the number of frequent itemsets did not change much for any of the algorithms. However, Basic had to do more work to find the candidates contained in the transaction since the transaction size (after adding ancestors) increased proportionately with the number of levels. Hence the time taken by Basic increased with the number of items. The time taken by the other two algorithms remained roughly constant, since the transaction size after dropping ancestors which were not present in any of the candidates did not change much.

Depth-Ratio: We changed the depth-ratio from 0.5 to 2. With high depth-ratios, items in frequent itemsets will tend to be picked from the leaves or lower levels of the

tree, while with low depth-ratios, items will be picked from higher up the tree. As shown in the figure, the performance gap between EstMerge and the other two algorithms increased as the depth-ratio increased. At a depth-ratio of 2, EstMerge did about 30% better than Cumulate, and about 5 times better than Basic. The reason is that EstMerge was able to prune a higher percentage of candidates at high depth-ratios.

Summary Cumulate and EstMerge were 2 to 5 times faster than Basic on all the synthetic datasets. EstMerge was 25% to 30% faster than Cumulate on many of the datasets. EstMerge's advantage decreased at high fanout, since most of the items in the rules came from the top levels of the taxonomy and EstMerge was not then able to prune many candidates. There was an increase in the performance gap between Cumulate and EstMerge as the number of transactions increased, since for a constant percentage sample size, the accuracy of the estimates of the support of the candidates increases as the number of transactions increases. All three algorithms exhibited linear scale-up with the number of transactions.

3.4.4 Reality Check

To see if our results on synthetic data hold in the "real world", we also ran the algorithms on two real-life datasets.

Supermarket Data This is data about grocery purchases of customers. There are a total of 548,000 items. The taxonomy (created by the user) has 4 levels, with 118 roots. There are around 1.5 million transactions, with an average of 9.6 items per transaction. Figure 30 shows the time taken by the three algorithms as the minimum support is decreased from 3% to 0.75%. These results are similar to those obtained on synthetic data, with EstMerge being a little faster than Cumulate, and both being about 3 times

as fast as Basic.

Department Store Data This is data from a department store. There are a total of 228,000 items. The taxonomy has 7 levels, with 89 roots. There are around 570,000 transactions, with an average of 4.4 items per transaction. Figure 30 shows the time taken by the three algorithms as the minimum support is decreased from 2% to 0.25%. The y-axis uses a log scale. Surprisingly, the Basic algorithm was more than 100 times slower than the other two algorithms. Since the taxonomy was very deep, the ratio of the number of frequent itemsets that contained both an item and its ancestor to the number of frequent itemsets that did not was very high. In fact, Basic counted around 300 times as many frequent itemsets as the other two algorithms, resulting in very poor performance.

3.4.5 Effectiveness of Interest Measure

We looked at the effectiveness of the interest measure in pruning rules for the two real-life datasets. Figure 31 shows the fraction of rules pruned for the supermarket and the department store datasets as the interest level is changed from 0 to 2 for different values of support and confidence. For the supermarket data, about 40% of the rules were pruned at an interest level of 1.1, while about 50% to 55% were pruned for the department store data at the same interest level.

In contrast, the interest measure based on statistical significance (i.e. correlation of the antecedent and consequent) did not prune any rules at 50% confidence, and it pruned less than 1% of the rules at 25% confidence (for both datasets). The reason is that the minimum support constraint already prunes most itemsets where the items are not correlated. Hence there is a statistically significant correlation between the antecedent and consequent for most frequent itemsets.

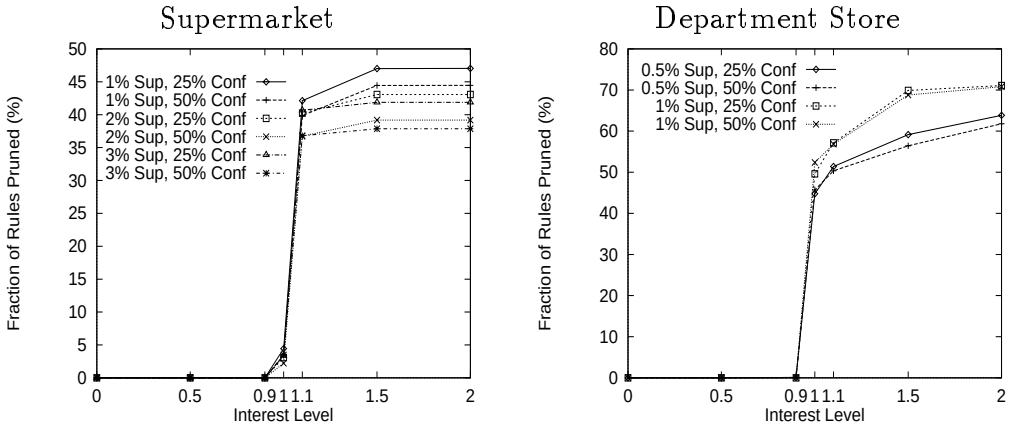


Figure 31: Effectiveness of Interest Measure

As an example of the way the interest measure prunes rules, the rule "[Carbonated beverages] and [Crackers] $\Rightarrow$ [Dairy-milk-refrigerated]" was pruned because because its support and confidence were less than 1.1 times the expected support and confidence (respectively) of ancestor "[Carbonated beverages] and [Crackers] $\Rightarrow$ [Milk]", where [Milk] was an ancestor of [Dairy-milk-refrigerated].

3.5 Summary

In this chapter, we introduced the problem of mining generalized association rules. Given a large database of customer transactions, where each transaction consists of a set of items, and a taxonomy (*isa* hierarchy) on the items, our approach finds associations between items at any level of the taxonomy. Earlier work on association rules did not consider the presence of taxonomies, thus restricting the items in the association rules to be leaf-level items in the taxonomy.

The "obvious" solution to this mining problem is to replace each transaction with an "extended transaction" that contains all the items in the original transaction as well as all the ancestors of each item in the original transaction. We could then run any of the earlier algorithms for mining association rules (e.g. Apriori) on these extended

transactions to obtain generalized association rules. However, this "Basic" approach is not very fast.

We presented two new algorithms, Cumulate and EstMerge. Empirical evaluation showed that these two algorithms run 2 to 5 times faster than Basic; for one real-life dataset, the performance gap was more than 100 times. Between the two algorithms, EstMerge performs somewhat better than Cumulate, with the performance gap increasing as the size of the database increases. Both EstMerge and Cumulate exhibit linear scale-up with the number of transactions.

A problem that many users experience in applying association rules to real problems is that many uninteresting or redundant rules are generated along with the interesting rules. We developed a new interest measure that uses the taxonomy information to prune redundant rules. The intuition behind this measure is that if the support and confidence of a rule are close to their expected values based on an ancestor of the rule, the more specific rule can be considered redundant. This approach was able to prune 40% to 60% of the rules on two real-life datasets. In contrast, an interest measure based on statistical significance rather than taxonomies was not able to prune even 1% of the rules.

Chapter 4

Mining Quantitative Association Rules

4.1 Introduction

In Chapters 2 and 3, we have looked at the problem of mining *association rules*. To recapitulate, given a set of transactions, where each transaction is a set of items, an association rule is an expression of the from $X \Rightarrow Y$, where X and Y are sets of items. The problem is to find all association rules that satisfy user-specified minimum support and minimum confidence constraints. Conceptually, this problem can be viewed as finding associations between the "1" values in a relational table where all the attributes are boolean. The table has an attribute corresponding to each item and a record corresponding to each transaction. The value of an attribute for a given record is "1" if the item corresponding to the attribute is present in the transaction corresponding to the record, else "0". In this chapter, we refer to this problem as the *Boolean Association Rules* problem.

Relational tables in most business and scientific domains have richer attribute types. Attributes can be quantitative (e.g. age, income) or categorical (e.g. zip code, make of car). Boolean attributes can be considered a special case of categorical attributes.

In this chapter, we define the problem of mining association rules over quantitative and categorical attributes in large relational tables and present techniques for discovering such rules. We refer to this mining problem as the *Quantitative Association Rules* problem. We give a formal statement of the problem in Section 4.2. For illustration, Figure 32

People

RecID	Age	Married	NumCars
100	23	No	1
200	25	Yes	1
300	29	No	0
400	34	Yes	2
500	38	Yes	2

(minimum support = 40%, minimum confidence = 50%)

Rules (Sample)	Support	Confidence
⟨Age: 30..39⟩ and ⟨Married: Yes⟩ ⇒ ⟨NumCars: 2⟩	40%	100%
⟨NumCars: 0..1⟩ ⇒ ⟨Married: No⟩	40%	66.6%

Figure 32: Example of Quantitative Association Rules

shows a People table with three non-key attributes. Age and NumCars are quantitative attributes, whereas Married is a categorical attribute. A quantitative association rule present in this table is: ⟨Age: 30..39⟩ and ⟨Married: Yes⟩ ⇒ ⟨NumCars: 2⟩.

4.1.1 Mapping to the Boolean Association Rules Problem

Let us examine whether the Quantitative Association Rules problem can be mapped to the Boolean Association Rules problem. If all attributes are categorical or the quantitative attributes have only a few values, this mapping is straightforward. Conceptually, instead of having just one field in the table for each attribute, we have as many fields as the number of attribute values. The value of a boolean field corresponding to ⟨*attribute1, value1*⟩ would be "1" if *attribute1* had *value1* in the original record, and "0" otherwise. If the domain of values for a quantitative approach is large, an obvious approach will be to first partition the values into intervals and then map each ⟨attribute, interval⟩ pair to a boolean attribute. We can now use any algorithm for finding Boolean Association Rules (e.g., the Apriori algorithm from Section2.2.1) to find quantitative association rules.

Figure 33 shows this mapping for the non-key attributes of the People table given in Figure 32. Age is partitioned into two intervals: 20..29 and 30..39. The categorical

RecID	Age: 20..29	Age: 30..39	Married: Yes	Married: No	NumCars: 0	NumCars: 1	NumCars: 2
100	1	0	0	1	0	1	0
200	1	0	1	0	0	1	0
300	1	0	0	1	1	0	0
400	0	1	1	0	0	0	1
500	0	1	1	0	0	0	1

Figure 33: Mapping to Boolean Association Rules Problem

attribute, Married, has two boolean attributes "Married: Yes" and "Married: No". Since the number of values for NumCars is small, NumCars is not partitioned into intervals; each value is mapped to a boolean field. Record 100, which had ⟨Age: 23⟩ now has "Age: 20..29" equal to "1", "Age: 30..39" equal to "0", etc.

Mapping Woes There are two problems with this simple approach when applied to quantitative attributes:

- *"MinSup"* If the number of intervals for a quantitative attribute (or values, if the attribute is not partitioned) is large, the support for any single interval can be low. Hence, without using larger intervals, some rules involving this attribute may not be found because they lack minimum support.

- *"MinConf"* There is some information lost whenever we partition values into intervals. Some rules may have minimum confidence only when an item in the antecedent consists of a single value (or a small interval). This information loss increases as the interval sizes become larger.

  For example, in Figure 33, the rule "⟨NumCars: 0⟩ ⇒ ⟨Married: No⟩" has 100% confidence. But if we had partitioned the attribute NumCars into intervals such that 0 and 1 cars end up in the same partition, then the closest rule is "⟨NumCars: 0..1⟩ ⇒ ⟨Married: No⟩", which only has 66.6% confidence.

There is a "catch-22" situation created by these two problems – if the intervals are too large, some rules may not have minimum confidence; if they are too small, some rules may not have minimum support.

Breaking the logjam To break the above catch-22 situation, we can consider all possible continuous ranges over the values of the quantitative attribute, or over the partitioned intervals. Thus if the partitions were [1..10], [11..20] and [21..30], we would also consider the intervals [1..20], [11..30] and [1..30]. The ""MinSup" problem now disappears since we combine adjacent intervals/values. The "MinConf" problem is still present; however, the information loss can be reduced by increasing the number of intervals, without encountering the "MinSup" problem.

Unfortunately, increasing the number of intervals while simultaneously combining adjacent intervals introduces two new problems:

- *"Execution Time"* If a quantitative attribute has n values (or intervals), there are on average $O(n^2)$ ranges that include a specific value or interval. Hence the number of items per record blows up, which will blow up the execution time.

- *"Many Rules"* If a value (or interval) of a quantitative attribute has minimum support, so will any range containing this value/interval. Thus, the number of rules blows up. Many of these rules will not be interesting. The notion of redundant rules that we will discuss later is analogous to taxonomies.

There is a tradeoff between faster execution time with fewer intervals (mitigating "Execution Time") and reducing information loss with more intervals (mitigating "MinConf"). We can reduce the information loss by increasing the number of intervals, at the cost of increasing the execution time and potentially generating many uninteresting

rules (the "Many Rules" problem).

It is not meaningful to combine categorical attribute values unless unless a taxonomy (*isa* hierarchy) is present on the attribute. In this case, the taxonomy can be used to implicitly combine values of a categorical attribute (see Chapter 3, or [HF95]). Using a taxonomy in this manner is somewhat similar to considering ranges over quantitative attributes.

4.1.2 Our Approach

We consider ranges over adjacent values/intervals of quantitative attributes to avoid the "MinSup" problem. To mitigate the "Execution Time" problem, we restrict the extent to which adjacent values/intervals may be combined by introducing a user-specified "maximum support" parameter; we stop combining intervals if their combined support exceeds this value. However, any single interval/value whose support exceeds maximum support is still considered.

But how do we decide whether to partition a quantitative attribute or not? And how many partitions should there be, and where should the cuts be, in case we do decide to partition? We introduce a *partial completeness measure* in Section 4.3 that gives us a handle on the information lost by partitioning and helps in making these decisions.

To address the "Many Rules" problem, we give an *interest measure* in Section 4.4. The interest measure is based on deviation from expectation and helps prune out uninteresting rules. This measure is an extension of the interest measure introduced in Section 3.2.1.

We give an algorithm for discovering quantitative association rules in Section 4.5. This algorithm shares the basic structure of the algorithm for finding boolean association rules given in Section2.2.1. However, to yield a fast implementation, the computational details of how candidates are generated and how their supports are counted are new.

We present our experience with this solution on a real-life dataset in Section 4.6.

4.1.3 Related Work

Apart from the work on mining association rules (discussed in Chapter 2) and the work on mining generalized association rules (discussed in Chapter 3), related work includes [PS91], where quantitative rules of the form $x = q_x \Rightarrow y = q_y$ are discovered. However, the antecedent and consequent are constrained to be a single ⟨attribute,value⟩ pair. There are suggestions about extending this work to rules where the antecedent is of the form $l \leq x \leq u$. This is done by partitioning the quantitative attributes into intervals; however, the intervals are not combined. The algorithm in [PS91] is fairly straightforward. To find the rules comprising $(A = a)$ as the antecedent, where a is a specific value of the attribute A, one pass over the data is made and each record is hashed by values of A. Each hash cell keeps a running summary of values of other attributes for the records with the same A value. The summary for $(A = a)$ is used to derive rules implied by $(A = a)$ at the end of the pass. To find rules for different attributes, the algorithm is run once on each attribute. Thus if we are interested in finding all rules, we must find these summaries for all combinations of attributes (which is exponentially large).

Fukuda et al. [FMMT96a] [FMMT96b] consider the problem of finding ranges for up to two quantitative attributes in the antecedent that maximize the support for a given minimum confidence, or maximize the confidence for a given minimum support. [FMMT96b] considers the case where there are two quantitative attributes in the antecedent, while [FMMT96b] considers the case where there is one quantitative attribute. Their work is complementary to ours: when the user finds an interesting rule as a result of running our algorithm, the rule can be optimized using their approach.

4.2 Problem Formulation

4.2.1 Problem Statement

We now give a formal statement of the problem of mining Quantitative Association Rules and introduce some terminology.

We use a simple device to treat categorical and quantitative attributes uniformly. For categorical attributes, the values of the attribute are mapped to a set of consecutive integers. (However, values of categorical attributes are not combined. That is, the implicit order imposed on the categorical variables is not used.) For quantitative attributes that are not partitioned into intervals, the values are mapped to consecutive integers such that the order of the values is preserved. If a quantitative attribute is partitioned into intervals, the intervals are mapped to consecutive integers, such that the order of the intervals is preserved. These mappings let us treat a database record as a set of $\langle$attribute, integer value$\rangle$ pairs, without loss of generality.

Now, let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called attributes. Let $\mathcal{P}$ denote the set of positive integers. Let $\mathcal{I}_\mathcal{V}$ denote the set $\mathcal{I} \times \mathcal{P}$. A pair $\langle x, v\rangle \in \mathcal{I}_\mathcal{V}$ denotes the attribute x, with the associated value v. Let $\mathcal{I}_\mathcal{R}$ denote the set $\{\langle x, l, u\rangle \in \mathcal{I} \times \mathcal{P} \times \mathcal{P} \mid l \leq u$, if x is quantitative; $l = u$, if x is categorical $\}$. Thus, a triple $\langle x, l, u\rangle \in \mathcal{I}_\mathcal{R}$ denotes either a quantitative attribute x with a value in the interval $[l, u]$, or a categorical attribute x with a value l. We will refer to this triple as an *item*. For any $X \subseteq \mathcal{I}_\mathcal{R}$, let *attributes*$(X)$ denote the set $\{x \mid \langle x, l, u\rangle \in X\}$.

Note that with the above definition, only values are associated with categorical attributes, while both values and ranges may be associated with quantitative attributes.

Let $\mathcal{D}$ be a set of records, where each record R is a set of attribute values such that $R \subseteq \mathcal{I}_\mathcal{V}$. We assume that each attribute occurs at most once in a record, i.e.

attributes are single-valued, not set-valued. We say that a record R *supports* $X \subseteq \mathcal{I}_{\mathcal{R}}$, if $\forall \langle x, l, u \rangle \in X$ $(\exists \langle x, q \rangle \in R$ such that $l \leq q \leq u)$.

A *quantitative association rule* is an implication of the form $X \Rightarrow Y$, where $X \subset \mathcal{I}_{\mathcal{R}}$, $Y \subset \mathcal{I}_{\mathcal{R}}$, and attributes$(X) \cap$ attributes$(Y) = \emptyset$. The rule $X \Rightarrow Y$ holds in the record set $\mathcal{D}$ with *confidence* c if $c\%$ of records in $\mathcal{D}$ that support X also support Y. The rule $X \Rightarrow Y$ has *support* s in the record set $\mathcal{D}$ if $s\%$ of records in $\mathcal{D}$ support $X \cup Y$.

Given a set of records $\mathcal{D}$, the problem of mining quantitative association rules is to find all (interesting) quantitative association rules that have support and confidence greater than the user-specified minimum support (called *minsup*) and minimum confidence (called *minconf*) respectively. Note that the fact that items in a rule can be categorical or quantitative has been hidden in the definition of an association rule.

Notation Recall that an *item* is a triple that represents either a categorical attribute with its value or a quantitative attribute with its range. We use the term *itemset* to represent a set of items. The support of an itemset $X \subset \mathcal{I}_{\mathcal{R}}$ is simply the percentage of records in $\mathcal{D}$ that support X. We use the term *frequent itemset* to represent an itemset with minimum support.

As before, let $\Pr(X)$ denote the probability that *all* the items in $X \subseteq \mathcal{I}_{\mathcal{R}}$ are supported by a given record. Then *support*$(X \Rightarrow Y) = \Pr(X \cup Y)$ and *confidence*$(X \Rightarrow Y) = \Pr(Y \mid X)$. (Note that $\Pr(X \cup Y)$ is the probability that all the items in $X \cup Y$ are present in the record.) We call $\widehat{X}$ a *generalization* of X (and X a *specialization* of $\widehat{X}$) if attributes$(X) =$ attributes$(\widehat{X})$ and $\forall x \in$ attributes(X) $[\langle x, l, u \rangle \in X \wedge \langle x, l', u' \rangle \in \widehat{X} \Rightarrow l' \leq l \leq u \leq u']$. For example, the itemset $\{$ $\langle$Age: 30..39$\rangle$, $\langle$Married: Yes$\rangle$ $\}$ is a generalization of $\{$ $\langle$Age: 30..35$\rangle$, $\langle$Married: Yes$\rangle$ $\}$. Thus generalizations are similar to ancestors in the taxonomy.

4.2.2 Problem Decomposition

We solve the problem of discovering quantitative association rules in five steps:

1. Determine the number of partitions and split-points for each quantitative attribute. (See Section 4.3.)

2. For categorical attributes, map the values of the attribute to a set of consecutive integers. For quantitative attributes that are not partitioned into intervals, the values are mapped to consecutive integers such that the order of the values is preserved. If a quantitative attribute is partitioned into intervals, the intervals are mapped to consecutive integers, such that the order of the intervals is preserved. From this point onwards, the algorithm only sees values (or ranges over values) for quantitative attributes. The fact that these values may represent intervals is transparent to the algorithm.

3. Find the support for each value of both quantitative and categorical attributes. Additionally, for quantitative attributes, adjacent values are combined as long as their support is less than the user-specified max support. We now know all ranges and values with minimum support for each quantitative attribute as well as all values with minimum support for each categorical attribute. These form the set of all *frequent items*.

   Next, find all sets of items whose support is greater than the user-specified minimum support. These are the *frequent itemsets*. (See Section 4.5.)

4. Use the algorithm in Section 2.3 to generate rules from the frequent itemsets.

5. Determine the interesting rules in the output. (See Section 4.4.)

Example Consider the "People" table shown in Figure 34a. There are two quantitative attributes, Age and NumCars. Assume that in Step 1, we decided to partition Age into 4 intervals, as shown in Figure 34b. Conceptually, the table now looks as shown in Figure 34c. After mapping the intervals to consecutive integers, using the mapping in Figure 34d, the table looks as shown in Figure 34e. Assuming minimum support of 40% and minimum confidence of 50%, Figure 34f shows some of the frequent itemsets, and Figure 34g shows some of the rules. (We have replaced the mapping numbers with the values in the original table in these last two figures.) Notice that the item ⟨Age: 20..29⟩ corresponds to a combination of the intervals 20..24 and 25..29, etc. We have not shown the step of determining the interesting rules in this example.

4.3 Partitioning Quantitative Attributes

In this section, we consider the questions of when we should partition the values of quantitative attributes into intervals, how many partitions there should be, and what the split-points should be. First, we present a measure of partial completeness which gives a handle on the amount of information lost by partitioning. We then show that equi-depth partitioning minimizes the number of intervals required to satisfy this partial completeness level. Thus equi-depth partitioning is, in some sense, optimal for this measure of partial completeness.

The intuition behind the partial completeness measure is as follows. Let R be the set of rules obtained by considering all ranges over the raw values of quantitative attributes. Let R' be the set of rules obtained by considering all ranges over the partitions of quantitative attributes. One way to measure the information loss when we go from R to R' is to see for each rule in R, how "far" the "closest" rule in R' is. The further away the closest rule, the greater the loss. By defining "close" rules to be generalizations, and

Minimum Support = 40% = 2 records
Minimum Confidence = 50%

People

RecID	Age	Married	NumCars
100	23	No	1
200	25	Yes	1
300	29	No	0
400	34	Yes	2
500	38	Yes	2

(a)

Partitions for Age

Interval
20..24
25..29
30..34
35..39

(b)

After partitioning Age

RecID	Age	Married	NumCars
100	20..24	No	0
200	25..29	Yes	1
300	25..29	No	1
400	30..34	Yes	2
500	35..39	Yes	2

(c)

Mapping Age

Interval	Integer
20..24	1
25..29	2
30..34	3
35..39	4

Mapping Married

Value	Integer
Yes	1
No	2

(d)

After mapping attributes

RecID	Age	Married	NumCars
100	1	2	0
200	2	1	1
300	2	2	1
400	3	1	2
500	4	1	2

(e)

Frequent Itemsets (Sample)

Itemset	Support
{ ⟨Age: 20..29⟩ }	3
{ ⟨Age: 30..39⟩ }	2
{ ⟨Married: Yes⟩ }	3
{ ⟨Married: No⟩ }	2
{ ⟨NumCars: 0..1⟩ }	3
{ ⟨Age: 30..39⟩, ⟨Married: Yes⟩ }	2

(f)

Rules (Sample)

Rule	Support	Confidence
⟨Age: 30..39⟩ and ⟨Married: Yes⟩ ⇒ ⟨NumCars: 2⟩	40%	100%
⟨Age: 20..29⟩ ⇒ ⟨NumCars: 0..1⟩	60%	66.6%

(g)

Figure 34: Example of Our Approach

using the ratio of the support of the rules as a measure of how far apart the rules are, we derive the measure of partial completeness given below.

4.3.1 Partial Completeness

We first define partial completeness over itemsets rather than rules, since we can guarantee that a close itemset will be found whereas we cannot guarantee that a close rule will be found. We then show that we can guarantee that a close rule will be found if the minimum confidence level for R' is less than that for R by a certain (computable) amount.

Let $\mathcal{C}$ denote the set of all frequent itemsets in $\mathcal{D}$. For any $K \geq 1$, we call $\mathcal{P}$ K-complete with respect to $\mathcal{C}$ if

- $\mathcal{P} \subseteq \mathcal{C}$,
- $X \in \mathcal{P}$ and $X' \subseteq X \Rightarrow X' \in \mathcal{P}$, and
- $\forall X \in \mathcal{C}$ [$\exists \widehat{X} \in \mathcal{P}$ such that
  1. $\widehat{X}$ is a generalization of X and $\text{support}(\widehat{X}) \leq K \times \text{support}(X)$, and
  2. $\forall Y \subseteq X$ $\exists \widehat{Y} \subseteq \widehat{X}$ such that $\widehat{Y}$ is a generalization of Y and $\text{support}(\widehat{Y}) \leq K \times \text{support}(Y)$].

The first two conditions ensure that $\mathcal{P}$ only contains frequent itemsets and that we can generate rules from $\mathcal{P}$. The first part of the third condition says that for any itemset in $\mathcal{C}$, there is a generalization of that itemset with at most K times the support in $\mathcal{P}$. The second part says that the property that the generalization has at most K times the support also holds for corresponding subsets of attributes in the itemset and its generalization. Notice that if $K = 1$, $\mathcal{P}$ becomes identical to $\mathcal{C}$.

For example, assume that in some table, the following are the frequent itemsets $\mathcal{C}$:

Number	Itemset	Support
1	{ ⟨Age: 20..30⟩ }	5%
2	{ ⟨Age: 20..40⟩ }	6%
3	{ ⟨Age: 20..50⟩ }	8%
4	{ ⟨Cars: 1..2⟩ }	5%
5	{ ⟨Cars: 1..3⟩ }	6%
6	{ ⟨Age: 20..30⟩, ⟨Cars: 1..2⟩ }	4%
7	{ ⟨Age: 20..40⟩, ⟨Cars: 1..3⟩ }	5%

The itemsets 2, 3, 5 and 7 would from a 1.5-complete set, since for any itemset X, either 2, 3, 5 or 7 is a generalization whose support is at most 1.5 times the support of X. For instance, itemset 2 is a generalization of itemset 1, and the support of itemset 2 is 1.2 times the support of itemset 1. Itemsets 3, 5 and 7 do not form a 1.5-complete set because for itemset 1, the only generalization among 3, 5 and 7 is itemset 3, and the support of 3 is more than 1.5 times the support of 1.

Lemma 5 *Let $\mathcal{P}$ be a K-complete set w.r.t. $\mathcal{C}$, the set of all frequent itemsets. Let $\mathcal{R}_\mathcal{C}$ be the set of rules generated from $\mathcal{C}$, for a minimum confidence level* minconf. *Let $\mathcal{R}_\mathcal{P}$ be the set of rules generated from $\mathcal{P}$ with the minimum confidence set to* minconf$/K$. *Then for any rule $A \Rightarrow B$ in $\mathcal{R}_\mathcal{C}$, there is a rule $\widehat{A} \Rightarrow \widehat{B}$ in $\mathcal{R}_\mathcal{P}$ such that*

- *$\widehat{A}$ is a generalization of A, $\widehat{B}$ is a generalization of B,*
- *the support of $\widehat{A} \Rightarrow \widehat{B}$ is at most K times the support of $A \Rightarrow B$, and*
- *the confidence of $\widehat{A} \Rightarrow \widehat{B}$ is at least $1/K$ times, and at most K times the confidence of $A \Rightarrow B$.*

Proof: Parts 1 and 2 follow directly from the definition of K-completeness. We now prove Part 3. Let $A \Rightarrow B$ be a rule in $\mathcal{R}_\mathcal{C}$. Then there is an itemset $A \cup B$ in $\mathcal{C}$. By definition of a K-complete set, there is an itemset $\widehat{A} \cup \widehat{B}$ in $\mathcal{P}$ such that (i) support$(\widehat{A} \cup \widehat{B}) \leq K \times$ support$(A \cup B)$, and (ii) support$(\widehat{A}) \leq K \times$ support(A). The confidence of

the rule $\widehat{A} \Rightarrow \widehat{B}$ (generated from $\widehat{A} \cup \widehat{B}$) is given by support$(\widehat{A} \cup \widehat{B})$/support$(\widehat{A})$. Hence

$$\frac{\text{confidence}(\widehat{A} \Rightarrow \widehat{B})}{\text{confidence}(A \Rightarrow B)} = \frac{\frac{\text{support}(\widehat{A} \cup \widehat{B})}{\text{support}(\widehat{A})}}{\frac{\text{support}(A \cup B)}{\text{support}(A)}} = \frac{\frac{\text{support}(\widehat{A} \cup \widehat{B})}{\text{support}(A \cup B)}}{\frac{\text{support}(\widehat{A})}{\text{support}(A)}}$$

Since both $\frac{\text{support}(\widehat{A} \cup \widehat{B})}{\text{support}(A \cup B)}$ and $\frac{\text{support}(\widehat{A})}{\text{support}(A)}$ lie between 1 and K (inclusive), the confidence of $\widehat{A} \Rightarrow \widehat{B}$ must be between $1/K$ and K times the confidence of $A \Rightarrow B$. □

Thus, given a set of frequent itemsets $\mathcal{P}$ which is K-complete w.r.t. the set of all frequent itemsets, the minimum confidence when generating rules from $\mathcal{P}$ must be set to $1/K$ times the desired level to guarantee that a close rule will be generated.

In the example given earlier, itemsets 2, 3 and 5 form a 1.5-complete set. The rule "$\langle$Age: 20..30$\rangle \Rightarrow \langle$Cars: 1..2$\rangle$" has 80% confidence, while the corresponding generalized rule "$\langle$Age: 20..40$\rangle \Rightarrow \langle$Cars: 1..3$\rangle$" has 83.3% confidence.

4.3.2 Determining the Number of Partitions

We first prove some properties of partitioned attributes (w.r.t. partial completeness), and then use these properties to decide the number of intervals given the partial completeness level.

Lemma 6 *Consider a quantitative attribute x, and some real $K > 1$. Assume we partition x into intervals (called* base *intervals) such that for any base interval B, either the support of B is less than* minsup $\times (K-1)/2$ *or B consists of a single value. Let $\mathcal{P}$ denote the set of all combinations of base intervals that have minimum support. Then $\mathcal{P}$ is K-complete w.r.t. the set of all ranges over x with minimum support.*

Proof: Let X be any interval with minimum support, and $\widehat{X}$ the smallest combination of base intervals which is a generalization of X (see Figure 35). There are at most two base intervals, one at each end, which are only partially spanned by X. Consider either

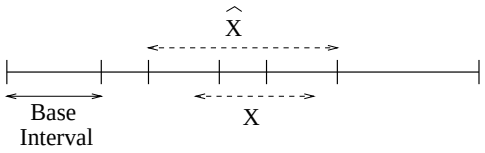


Figure 35: Illustration for Lemma 6

of these intervals. If X only partially spans this interval, the interval cannot be just a single value. Hence the support of this interval, as well as the support of the portion of the interval not spanned by X, must be less than $minsup \times (K-1)/2$. Thus

$$\begin{aligned} \text{support}(\widehat{X}) &\leq \text{support}(X) + 2 \times minsup \times (K-1)/2 \\ &\leq \text{support}(X) + \text{support}(X) \times (K-1) \\ &\quad (\text{since support}(X) > minsup) \\ &\leq \text{support}(X) \times K \end{aligned}$$

□

Lemma 7 *Consider a set of n quantitative attributes, and some real $K > 1$. Assume each quantitative attribute is partitioned such that for any base interval B, either the support of B is less than* minsup $\times (K-1)/(2 \times n)$ *or B consists of a single value. Let $\mathcal{P}$ denote the set of all frequent itemsets over the partitioned attributes. Then $\mathcal{P}$ is K-complete w.r.t the set of all frequent itemsets (obtained without partitioning).*

Proof: **Proof**: The proof is similar to that for Lemma 6. However, the difference in support between an itemset X and its generalization $\widehat{X}$ may be $2m$ times the support of a single base interval for a single attribute, where m is the number of quantitative attributes in X. Since X may have up to n attributes, the support of each base interval must be at most $minsup \times (K-1)/(2 \times n)$, rather than just $minsup \times (K-1)/2$ for $\mathcal{P}$ to be K-complete. A similar argument applies to subsets of X.

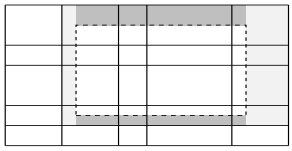


Figure 36: Example for Lemma 7

An illustration of this proof for 2 quantitative attributes is shown in Figure 36. The solid lines correspond to partitions of the attributes, and the dashed rectangle corresponds to an itemset X. The shaded areas show the extra area that must be covered to get its generalization $\widehat{X}$ using partitioned attributes. Each of the 4 shaded areas spans less than a single partition of a single attribute. (One partition of one attribute corresponds to a band from one end of the rectangle to another.) □

Note that Lemma 7 gives a bound for the partial completeness level of $\mathcal{P}$, rather than the actual level. If Lemma 7 says that $\mathcal{P}$ is K-complete w.r.t the set of all frequent itemsets, $\mathcal{P}$ may actually be $(K-\delta)$-complete, for some $\delta > 0$, depending on the data characteristics. However, it is probably not possible to find the largest δ such that $\mathcal{P}$ is $(K-\delta)$-complete without computing the set of all the frequent itemsets. The latter computation would be very expensive for most datasets.

For any given partitioning, we can use Lemma 7 to compute an upper bound on the partial completeness level for that partitioning. We first illustrate the procedure for a single attribute. In this case, we simply find the partition with highest support among those with more than one value. Let the support of this partition be s. Then, to find the partial completeness level K, we use the formula $s = minsup \times (K-1)/2$ from Lemma 6 to get $K = 1 + 2 \times s/minsup$. With n attributes, the formula becomes

$$K = 1 + \frac{2 \times n \times s}{minsup} \tag{2}$$

where s is the maximum support for a partition with more than one value, among all the quantitative attributes. Recall that the lower the level of partial completeness, the less

the information lost. The formula reflects this: as s decreases, implying more intervals, the partial completeness level decreases.

Lemma 8 *For any specified number of intervals, equi-depth partitioning minimizes the bound on the partial completeness level.*

Proof: From Lemma 7, if the support of each base interval is less than $minsup \times (K - 1)/(2 \times n)$, the partial completeness level is K. Since the maximum support of any base interval is minimized with equi-depth partitioning, equi-depth partitioning results in the lowest bound on the partial completeness level. □

Corollary 1 *For a given partial completeness level, equi-depth partitioning minimizes the number of intervals required to satisfy that partial completeness level.*

Given the level of partial completeness desired by the user, and the minimum support, we can calculate the number of partitions required (assuming equi-depth partitioning). From Lemma 7, we know that to get a partial completeness level K, the support of any partition with more than one value should be less than $minsup * (K - 1)/(2 \times n)$ where n is the number of quantitative attributes. Ignoring the special case of partitions that contain just one value[1], and assuming that equi-depth partitioning splits the support identically, there should be $1/s$ partitions in order to get the support of each partition to less than s. Thus we get

$$\text{Number of Intervals} = \frac{2 \times n}{m \times (K - 1)} \tag{3}$$

where

$$n = \text{Number of Quantitative Attributes}$$

[1]While this may overstate the number of partitions required, it will not increase the partial completeness level.

m = Minimum Support (as a fraction)

K = Partial Completeness Level

If there are no rules with more than n' quantitative attributes, we can replace n with n' in the above formula (see proof of Lemma 7).

4.4 Interest Measure

A potential problem with combining intervals for quantitative attributes is that the number of rules found may be very large. In this section, we present a "greater-than-expected-value" interest measure to identify the interesting rules in the output, based on intuitions similar to those for the interest measure presented in Section 3.2.1 for taxonomies. This interest measure looks at both generalizations and specializations of the rule to identify the interesting rules. The first half of this section focuses on generalizations, and uses techniques analogous to those in Section 3.2.1. We then incorporate specializations into the model.

To motivate our interest measure, consider the following rules; assume that roughly a quarter of people in the age group 20..30 are in the age group 20..25.

$\langle$Age: 20..30$\rangle \Rightarrow \langle$Cars: 1..2$\rangle$ (8% sup., 70% conf.)

$\langle$Age: 20..25$\rangle \Rightarrow \langle$Cars: 1..2$\rangle$ (2% sup., 70% conf.)

The second rule can be considered redundant since it does not convey any additional information and is less general than the first rule. Given the first rule, we expect that the second rule would have the same confidence as the first and support equal to a quarter of the support for the first. We try to capture this notion of "interest" by saying that we only want to find rules whose support and/or confidence is greater than expected. We now formalize this idea.

Expected Values Let $E_{\Pr(\widehat{Z})}[\Pr(Z)]$ denote the "expected" value of $\Pr(Z)$ (that is, the support of Z) based on $\Pr(\widehat{Z})$, where $\widehat{Z}$ is a generalization of Z. Let Z be the itemset $\{\langle z_1, l_1, u_1\rangle, \ldots, \langle z_n, l_n, u_n\rangle\}$ and $\widehat{Z}$ the set $\{\langle z_1, l'_1, u'_1\rangle, \ldots, \langle z_n, l'_n, u'_n\rangle\}$ (where $l'_i \leq l_i \leq u_i \leq u'_i$). Then we define

$$E_{\Pr(\widehat{Z})}[\Pr(Z)] = \frac{\Pr(\langle z_1, l_1, u_1\rangle)}{\Pr(\langle z_1, l'_1, u'_1\rangle)} \times \cdots \times \frac{\Pr(\langle z_n, l_n, u_n\rangle)}{\Pr(\langle z_n, l'_n, u'_n\rangle)} \times \Pr(\widehat{Z})$$

Similarly, we $E_{\Pr(\widehat{Y} \mid \widehat{X})}[\Pr(Y \mid X)]$ denote the "expected" confidence of the rule $X \Rightarrow Y$ based on the rule $\widehat{X} \Rightarrow \widehat{Y}$, where $\widehat{X}$ and $\widehat{Y}$ are generalizations of X and Y respectively. Let Y be the itemset $\{\langle y_1, l_1, u_1\rangle, \ldots, \langle y_n, l_n, u_n\rangle\}$ and $\widehat{Y}$ the set $\{\langle y_1, l'_1, u'_1\rangle, \ldots, \langle y_n, l'_n, u'_n\rangle\}$. Then we define

$$E_{\Pr(\widehat{Y} \mid \widehat{X})}[\Pr(Y \mid X)] = \frac{\Pr(\langle y_1, l_1, u_1\rangle)}{\Pr(\langle y_1, l'_1, u'_1\rangle)} \times \cdots \times \frac{\Pr(\langle y_n, l_n, u_n\rangle)}{\Pr(\langle y_n, l'_n, u'_n\rangle)} \times \Pr(\widehat{Y} \mid \widehat{X})$$

A Tentative Interest Measure We first introduce a measure similar to the one used in Section 3.2.1.

An itemset Z is *R-interesting* w.r.t an ancestor $\widehat{Z}$ if the support of Z is greater than or equal to R times the expected support based on $\widehat{Z}$. A rule $X \Rightarrow Y$ is *R-interesting* w.r.t an ancestor $\widehat{X} \Rightarrow \widehat{Y}$ if the support of the rule $X \Rightarrow Y$ is R times the expected support based on $\widehat{X} \Rightarrow \widehat{Y}$, or if the confidence is R times the expected confidence based on $\widehat{X} \Rightarrow \widehat{Y}$.

Given a set of rules, we call $\widehat{X} \Rightarrow \widehat{Y}$ a *close ancestor* of $X \Rightarrow Y$ if there is no rule $X' \Rightarrow Y'$ such that $\widehat{X} \Rightarrow \widehat{Y}$ is an ancestor of $X' \Rightarrow Y'$ and $X' \Rightarrow Y'$ is an ancestor of $X \Rightarrow Y$. A similar definition holds for itemsets.

Given a set of rules S and a minimum interest R, a rule $X \Rightarrow Y$ is *interesting* (in S) if it has no ancestors or it is R-interesting with respect to its close ancestors among its interesting ancestors.

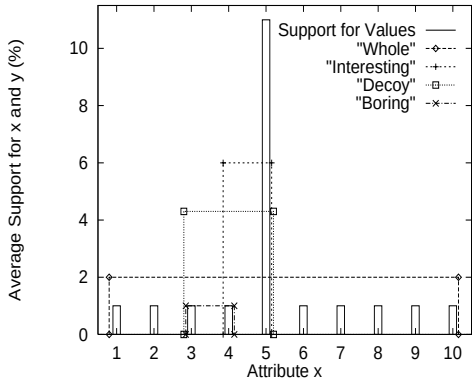


Figure 37: Example for Interest Measure definition

Why looking at generalizations is insufficient The above definition of interest has the following problem. Consider a single attribute x with the range [1,10], and another categorical attribute y. Assume the support for the values of x are uniformly distributed. Let the support for values of x together with y be as shown in Figure 37.[2] For instance, the support of $(\langle x, 5\rangle,\ y) = 11\%$, and the support for $(\langle x, 1\rangle,\ y) = 1\%$. This figure also shows the "average" support for the itemsets $(\langle x, 1, 10\rangle,\ y)$, $(\langle x, 3, 5\rangle,\ y)$, $(\langle x, 3, 4\rangle,\ y)$ and $(\langle x, 4, 5\rangle,\ y)$. Clearly, the only "interesting" set is $\{\langle x, 5, 5\rangle,\ y\}$. However, the interest measure given above may also find other itemsets "interesting". For instance, with an interest level of 2, interval "Decoy", $\{\langle x, 3, 5\rangle,\ y\}$ would also be considered interesting, as would $\{\langle x, 4, 6\rangle,\ y\}$ and $\{\langle x, 5, 7\rangle,\ y\}$.

If we have the support for each value of x along with y, it is easy to check that all specializations of an itemset are also interesting. However, in general, we will not have this information, since a single value of x together with y may not have minimum support. We will only have information about those specializations of x which (along with y) have minimum support. For instance, we may only have information about the support for the subinterval "Interesting" (for interval "Decoy").

[2] We cannot use a the attribute x without y in this example since expected values are computed from the support for values single attributes.

An obvious way to use this information is to check whether there are any specializations with minimum support that are not interesting. However, there are two problems with this approach. First, there may not be any specializations with minimum support that are not interesting. This case is true in the example given above unless the minimum support is less than or equal to 2%. Second, even if there are such specializations, there may not be any specializations with minimum support that *are* interesting. We do not want to discard the current itemset unless there is a specialization with minimum support that is interesting and some part of the current itemset is not interesting.

An alternative approach is to check whether there are any specializations that are more interesting than the itemset, and then subtract the specialization from the current itemset to see whether or not the difference is interesting. Notice that the difference need not have minimum support. Further, if there are no such specializations, we would want to keep this itemset. Thus, this approach is clearly to be preferred. We therefore change the definitions of interest given earlier to reflect these ideas.

Final Interest Measure An itemset X is *R-interesting* with respect to $\widehat{X}$ if the support of X is greater than or equal to R times the expected support based on $\widehat{X}$ and for any specialization X' such that X' has minimum support and $X - X' \subseteq \mathcal{I}_{\mathcal{R}}$, $X - X'$ is R-interesting with respect to $\widehat{X}$.

Similarly, a rule $X \Rightarrow Y$ is *R-interesting* w.r.t an ancestor $\widehat{X} \Rightarrow \widehat{Y}$ if the support of the rule $X \Rightarrow Y$ is R times the expected support based on $\widehat{X} \Rightarrow \widehat{Y}$, or the confidence is R times the expected confidence based on $\widehat{X} \Rightarrow \widehat{Y}$, and the itemset $X \cup Y$ is R-interesting w.r.t $\widehat{X} \cup \widehat{Y}$.

Note that with the specification of the interest level, the specification of the minimum confidence parameter can optionally be dropped. The semantics in that case will be that

we are interested in all those rules that have interest above the specified interest level.

4.5 Algorithm

In this section, we describe the algorithm for finding all frequent itemsets (Step 3 of the problem decomposition given in Section 4.2.2). At this stage, we have already partitioned quantitative attributes, and created combinations of intervals of the quantitative attributes that have minimum support. These combinations, along with those values of categorical attributes that have minimum support, form the frequent items.

Starting with the frequent items, we generate all frequent itemsets using an algorithm based on the Apriori algorithm for finding boolean association rules given in Section 2.2.1. The structure of this algorithm is the same as that of the Apriori algorithm shown in Figure 1. However, the proposed algorithm extends the candidate generation procedure to add pruning using the interest measure, and uses a different data structure for counting candidates. We now discuss these two issues in more detail.

4.5.1 Candidate Generation

Given L_{k-1}, the set of all frequent $k-1$-itemsets, the candidate generation procedure must return a superset of the set of all frequent k-itemsets. This procedure has now has three parts, the first two being similar to the corresponding phases for Apriori.

1. **Join Phase**. L_{k-1} is joined with itself, the join condition being that the lexicographically ordered first $k-2$ items are the same, and that the attributes of the last two items are different. For example, let L_2 consist of the following itemsets:

   $\{$ ⟨Married: Yes⟩ ⟨Age: 20..24⟩ $\}$

   $\{$ ⟨Married: Yes⟩ ⟨Age: 20..29⟩ $\}$

{ ⟨Married: Yes⟩ ⟨NumCars: 0..1⟩ }

{ ⟨Age: 20..29⟩ ⟨NumCars: 0..1⟩ }

After the join step, C_3 will consist of the following itemsets:

{ ⟨Married: Yes⟩ ⟨Age: 20..24⟩ ⟨NumCars: 0..1⟩ }

{ ⟨Married: Yes⟩ ⟨Age: 20..29⟩ ⟨NumCars: 0..1⟩ }

2. **Subset Prune Phase** All itemsets from the join result which have some $(k-1)$-subset that is not in L_{k-1} are deleted. Continuing the earlier example, the prune step will delete the itemset { ⟨Married: Yes⟩ ⟨Age: 20..24⟩ ⟨NumCars: 0..1⟩ } since its subset { ⟨Age: 20..24⟩ ⟨NumCars: 0..1⟩ } is not in L_2.

3. **Interest Prune Phase**. If the user specifies an interest level, and wants only itemsets whose support and confidence is greater than expected, the interest measure is used to prune the candidates further. Lemma 9, given below, says that we can delete any itemset that contains a quantitative item whose (fractional) support is greater than $1/R$, where R is the interest level. If we delete all items whose support is greater than $1/R$ at the end of the first pass, the candidate generation procedure will ensure that we never generate candidates that contain an item whose support is more than $1/R$.

Lemma 9 *Consider an itemset X, with a quantitative item x. Let $\widehat{X}$ be the generalization of X where x is replaced by the item corresponding to the full range of attribute(x). Let the user-specified interest level be R. If the support of x is greater than $1/R$, then the actual support of X cannot be more than R times the expected support based on $\widehat{X}$.*

Proof: The actual support of X cannot be greater than the actual support of $\widehat{X}$. The expected support of X w.r.t. $\widehat{X}$ is $\Pr(\widehat{X}) \times \Pr(x)$, since $\Pr(\widehat{x})$ equals 1. Thus the ratio of

the actual to the expected support of X is $\Pr(X)/(\Pr(\widehat{X}) \times \Pr(x)) = (\Pr(X)/\Pr(\widehat{X})) \times (1/\Pr(x))$. The first ratio is less than or equal to 1, and the second ratio is less than R. Hence the ratio of the actual to the expected support is less than R. □

4.5.2 Counting Support of Candidates

While making a pass, we read one record at a time and increment the support count of candidates supported by the record. Thus, given a set of candidate itemsets C and a record R, we need to find all itemsets in C that are supported by R.

We partition candidates into groups such that candidates in each group have the same attributes and the same values for their categorical attributes. We replace each such group with a single "super-candidate". Each "super-candidate" has two parts: (i) the common categorical attribute values, and (ii) a data structure representing the set of values of the quantitative attributes.

For example, consider the candidates:

{ ⟨Married: Yes⟩ ⟨Age: 20..24⟩, ⟨NumCars: 0..1⟩ }

{ ⟨Married: Yes⟩ ⟨Age: 20..29⟩, ⟨NumCars: 1..2⟩ }

{ ⟨Married: Yes⟩ ⟨Age: 24..29⟩, ⟨NumCars: 2..2⟩ }

These candidates have one categorical attribute, "Married", whose value, "Yes" is the same for all three candidates. Their quantitative attributes, "Age" and "NumCars" are also the same. Hence these candidates can be grouped together into a super-candidate. The categorical part of the super-candidate contains the item ⟨Married: Yes⟩. The quantitative part contains the following information.

Age	NumCars
20..24	0..1
20..29	1..2
24..29	2..2

We can now split the problem into two parts:

1. We first find which "super-candidates" are supported by the categorical attributes in the record. We re-use the *hash-tree* data structure described in Section 2.2.1 to reduce the number of super-candidates that need to be checked for a given record.

2. Once we know that the categorical attributes of a "super-candidate" are supported by a given record, we need to find which of the candidates in the super-candidate are supported. (Recall that while all candidates in a super-candidate have the same values for their categorical values, they have different values for their quantitative attributes.) We discuss this issue in the rest of this section.

Let a "super-candidate" have n quantitative attributes. The quantitative attributes are fixed for a given "super-candidate". Hence the set of values for the quantitative attributes correspond to a set of n-dimensional rectangles (each rectangle corresponding to a candidate in the super-candidate). The values of the corresponding quantitative attributes in a database record correspond to an n-dimensional point. Thus the problem reduces to finding which n-dimensional rectangles contain a given n-dimensional point, for a set of n-dimensional points. One classic solution to this problem is to put the rectangles in an R^*-tree [BKSS90].

If the number of dimensions is small, and the range of values in each dimension is also small, there is a faster solution. Namely, we use a n-dimensional array, where the number of array cells in the j-th dimension equals the number of partitions for the attribute corresponding to the j-th dimension. We use this array to get support counts for all possible combinations of values of the quantitative attributes in the super-candidate. The amount of work done per record is only O(number-of-dimensions), since we simply index into each dimension and increment the support count for a single cell.

At the end of the pass over the database, we iterate over all the cells covered by each of the rectangles and sum up the support counts.

Using a multi-dimensional array is cheaper than using an R^*-tree, in terms of CPU time. However, as the number of attributes (dimensions) in a super-candidate increases, the multi-dimensional array approach will need a huge amount of memory. Thus there is a tradeoff between less memory for the R^*-tree versus less CPU time for the multi-dimensional array. We use the following heuristic for choosing the data structure: if the ratio of the expected memory use of the R^*-tree to that of the multi-dimensional array is below a certain threshold, we use the array, else the R^* tree.

4.6 Experience with a Real-Life Dataset

We assessed the effectiveness of our approach by experimenting with a real-life dataset. The data had 7 attributes: 5 quantitative and 2 categorical. The quantitative attributes were monthly-income, credit-limit, current-balance, year-to-date balance, and year-to-date interest. The categorical attributes were employee-category and marital-status. There were 500,000 records in the data.

Our experiments were performed on an IBM RS/6000 250 workstation with 128 MB of main memory running AIX 3.2.5. The data resided in the AIX file system and was stored on a local 2GB SCSI 3.5" drive, with measured sequential throughput of about 2 MB/second.

Partial Completeness Level Figure 38 shows the number of interesting rules, and the percent of rules found to be interesting, for different interest levels as the partial completeness level increases from 1.5 to 5. The minimum support was set to 20%, minimum confidence to 25%, and maximum support to 40%. As expected, the number

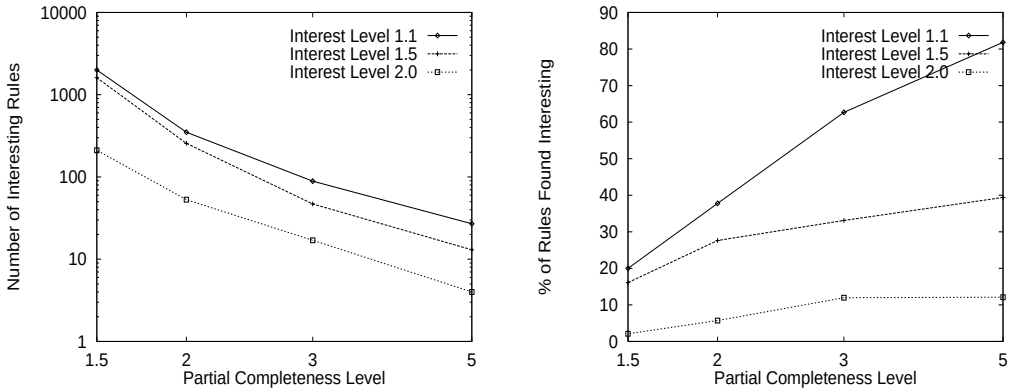


Figure 38: Changing the Partial Completeness Level

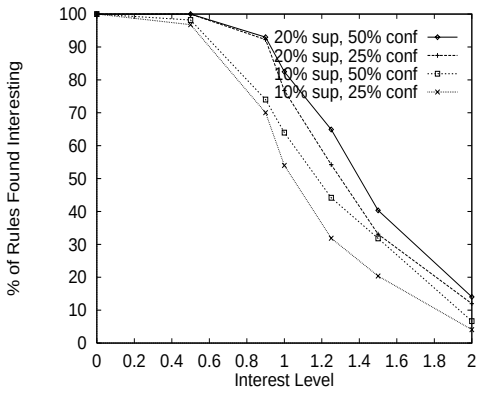


Figure 39: Interest Measure

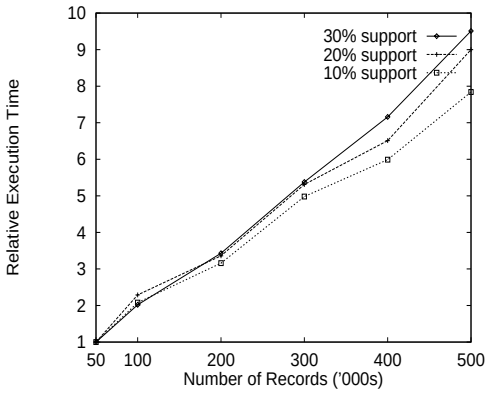


Figure 40: Scale-up : Number of records

of interesting rules decreases as the partial completeness level increases. The percentage of rules pruned also decreases, indicating that fewer similar rules are found as the partial completeness level increases and there are fewer intervals for the quantitative attributes.

Interest Measure Figure 39 shows the fraction of rules identified as "interesting" as the interest level was increased from 0 (equivalent to not having an interest measure) to 2. As expected, the percentage of rules identified as interesting decreases as the interest level increases.

Scale-up The running time for the algorithm can be split into two parts:

1. *Candidate generation* The time for this is independent of the number of records, assuming that the distribution of values in each record is similar.

2. *Counting support* The time for this is directly proportional to the number of records, again assuming that the distribution of values in each record is similar. When the number of records is large, this time will dominate the total time.

Thus we would expect the algorithm to have near-linear scaleup. This is confirmed by Figure 40, which shows the relative execution time as we increase the number of input records 10-fold from 50,000 to 500,000, for three different levels of minimum support. The times have been normalized with respect to the times for 50,000 records. The graph shows that the algorithm scales quite linearly for this dataset.

4.7 Summary

In this chapter, we introduced the problem of mining association rules in large relational tables containing both quantitative and categorical attributes. We dealt with quantitative attributes by fine-partitioning the values of the attribute and then combining adjacent partitions as necessary. We introduced a measure of partial completeness which quantifies the information lost due to partitioning. This measure is used to decide whether or not to partition a quantitative attribute, and if so, the number of partitions.

A direct application of this technique may generate too many similar rules. We tackled this problem by using a "greater-than-expected-value" interest measure to identify the interesting rules in the output. This interest measure looks at both generalizations and specializations of the rule to identify the interesting rules.

We gave an algorithm for mining such quantitative association rules. Our experiments on a real-life dataset indicate that the algorithm scales linearly with the number of records. They also showed that the interest measure was effective in identifying the

interesting association rules.

Chapter 5

Mining Sequential Patterns

5.1 Introduction

In this chapter, we introduce a new data mining problem, *discovering sequential patterns*, that is closely related to the problem of mining association rules. The input data is a set of sequences, called *data-sequences*. Each data-sequence is a list of *transactions*, where each transaction is a sets of literals, called *items*. Typically there is a transaction-time associated with each transaction. A *sequential pattern* also consists of a list of sets of items. The problem is to find all sequential patterns with a user-specified minimum *support*, where the support of a sequential pattern is the percentage of data-sequences that contain the pattern.

For example, in the database of a book club, each data-sequence may correspond to all book selections of a customer, and each transaction to the books selected by the customer in one order. A sequential pattern might be "5% of customers bought 'Foundation', then 'Foundation and Empire', and then 'Second Foundation'". The data-sequence corresponding to a customer who bought some other books in between these books still contains this sequential pattern; the data-sequence may also have other books in the same transaction as one of the books in the pattern. Elements of a sequential pattern can be sets of items, for example, "'Foundation' and 'Ringworld', followed by 'Foundation and Empire' and 'Ringworld Engineers', followed by 'Second Foundation'". However, all the items in an element of a sequential pattern must be present in a single transaction for the data-sequence to support the pattern.

This problem was motivated by applications in the retailing industry, including attached mailing, add-on sales, and customer satisfaction. In addition, the results apply to many scientific and business domains. For instance, in the medical domain, a data-sequence may correspond to the symptoms or diseases of a patient, with a transaction corresponding to the symptoms exhibited or diseases diagnosed during a visit to the doctor. The patterns discovered using this data could be used in disease research to help identify symptoms/diseases that precede certain diseases.

In [AS95], we had introduced the above problem formulation and presented an algorithm for this problem. However, the above problem definition has the following limitations:

1. **Absence of time constraints.** Users often want to specify maximum and/or minimum time gaps between adjacent elements of the sequential pattern. For example, a book club probably does not care if someone bought "Foundation", followed by "Foundation and Empire" three years later; they may want to specify that a customer should support a sequential pattern only if adjacent elements occur within a specified time interval, say three months. (So for a customer to support this pattern, the customer should have bought "Foundation and Empire" within three months of buying "Foundation").

2. **Rigid definition of a transaction.** For many applications, it does not matter if items in an element of a sequential pattern were present in two different transactions, as long as the transaction-times of those transactions are within some small time window. That is, each element of the pattern can be contained in the union of the items bought in a set of transactions as long as the difference between the maximum and minimum transaction-times is less than the size of a *sliding time*

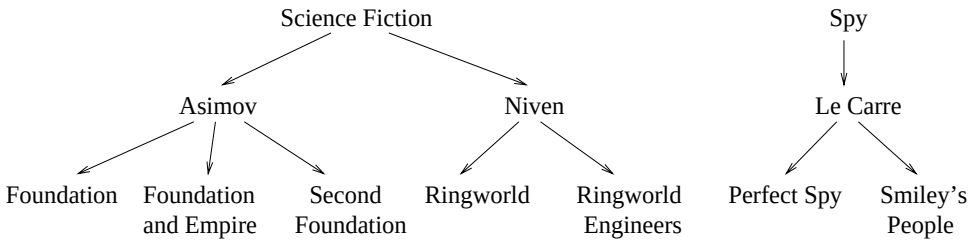


Figure 41: Example of a Taxonomy

window. For example, if the book club specifies a time window of a week, a customer who ordered the "Foundation" on Monday, "Ringworld" on Saturday, and then "Foundation and Empire" and "Ringworld Engineers" in a single order a few weeks later would still support the pattern "'Foundation' and 'Ringworld', followed by 'Foundation and Empire' and 'Ringworld Engineers'".

3. **Absence of taxonomies.** Many datasets have a user-defined taxonomy (*isa* hierarchy) over the items in the data, and users want to find patterns that include items across different levels of the taxonomy. An example of a taxonomy is given in Figure 41. With this taxonomy, a customer who bought "Foundation" followed by "Perfect Spy" would support the patterns "'Foundation' followed by 'Perfect Spy'", "'Asimov' followed by 'Perfect Spy'", "'Science Fiction' followed by 'Le Carre'", etc.

In this chapter, we generalize the problem definition given in [AS95] to incorporate time constraints, sliding time windows, and taxonomies in sequential patterns. We present GSP (Generalized Sequential Patterns), a new algorithm that discovers all such sequential patterns. Empirical evaluation shows that GSP scales linearly with the number of data-sequences, and has very good scale-up properties with respect to the number of transactions per data-sequence and number of items per transaction.

In addition to introducing the problem of sequential patterns, [AS95] presented three

algorithms for solving this problem, but these algorithms do not handle time constraints, sliding windows, or taxonomies. Two of these algorithms were designed to find only maximal sequential patterns; however, many applications require all patterns and their supports. The third algorithm, AprioriAll, finds all patterns; its performance was better than or comparable to the other two algorithms. We review AprioriAll in Section 5.4.1. Briefly, AprioriAll is a three-phase algorithm. It first finds all itemsets with minimum support (frequent itemsets), transforms the database so that each transaction is replaced by the set of all frequent itemsets contained in the transaction, and then finds sequential patterns. There are two problems with this approach. First, it is computationally expensive to do the data transformation on-the-fly during each pass while finding sequential patterns. The alternative, to transform the database once and store the transformed database, will be infeasible or unrealistic for many applications since it nearly doubles the disk space requirement (which could be prohibitive for large databases). Second, while it is possible to extend this algorithm to handle time constraints and taxonomies, it does not appear feasible to incorporate sliding windows. For the cases that the extended AprioriAll can handle, our empirical evaluation reported in Section 5.4 shows that new GSP algorithm is up to 20 times faster.

The rest of this Chapter is organized as follows. We give a formal description of the problem of mining generalized sequential patterns in Section 5.2. In Section 5.3, we describe GSP, our algorithm for finding such patterns. In Section 5.4, we compare the performance of GSP to the AprioriAll algorithm, show the scale-up properties of GSP, and study the performance impact of time constraints and sliding windows. We conclude with a summary in Section 5.5.

Related Work

The problem of mining association rules, described in Chapter 2, is relevant. Association rules are rules about what items are bought together within a transaction, and are thus intra-transaction patterns, unlike inter-transaction sequential patterns. The problem of finding association rules when there is a user-defined taxonomy on items has been addressed in Chapter 3.

The problem of discovering similarities in a database of genetic sequences, discussed in [WCM+94], is relevant. However, the patterns they wish to discover are subsequences made up of consecutive characters separated by a variable number of noise characters. A sequence in our problem consists of list of sets of characters (items), rather than being simply a list of characters. In addition, we are interested in finding *all* sequences with minimum support rather than some frequent patterns.

A problem of discovering frequent episodes in a sequence of events was presented in [MTV95]. Their patterns are arbitrary DAGs (directed acyclic graphs), where each vertex corresponds to a single event (or item) and an edge from event A to event B denotes that A occurred before B. They move a time window across the input sequence and find all patterns that occur in some user-specified percentage of all windows. Their algorithm is designed for counting the number of occurrences of a pattern when moving a window across a single sequence, while we are interested in finding patterns that occur in many different data-sequences.

Discovering patterns in sequences of events has been an area of active research in AI (see, for example, [DM85]). However, the focus in this body of work is on finding the rule underlying the generation of a given sequence in order to be able to predict a plausible sequence continuation (e.g. the rule to predict what number will come next, given a

sequence of numbers). We, on the other hand, are interested in finding all common patterns embedded in a database of sequences of sets of events (items).

Our problem is related to the problem of finding text subsequences that match a given regular expression (*c.f.* the UNIX grep utility). There also has been work on finding text subsequences that approximately match a given string (e.g. [CR93] [WM92]). These techniques are oriented toward finding matches for one pattern. In our problem, the difficulty is in figuring out what patterns to try and then efficiently finding out which of those patterns are contained in enough data sequences.

Techniques based on multiple alignment [Wat89] have been proposed to find entire text sequences that are similar. There also has been work to find locally similar subsequences [AGM+90] [Roy92] [VA89]. However, as pointed out in [WCM+94], these techniques apply when the discovered patterns consist of consecutive characters or multiple lists of consecutive characters separated by a fixed length of noise characters.

5.2 Problem Formulation

Definitions As before, let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called *items*. Let $\mathcal{T}$ be a directed acyclic graph on the literals. An edge in $\mathcal{T}$ represents an *isa* relationship, and $\mathcal{T}$ represents a set of taxonomies. If there is an edge in $\mathcal{T}$ from p to c, we call p a *parent* of c and c a *child* of p. (The item p represents a generalization of c). We call $\widehat{x}$ an *ancestor* of x (and x a *descendant* of $\widehat{x}$) if there is an edge from $\widehat{x}$ to x in transitive-closure($\mathcal{T}$).

An *itemset* is a non-empty set of items. A *sequence* is an ordered list of itemsets. We denote a sequence s by $\langle s_1 s_2 ... s_n \rangle$, where s_j is an itemset. We also call s_j an *element* of the sequence. We denote an element of a sequence by $(x_1, x_2, ..., x_m)$, where x_j is an item. An item can occur only once in an element of a sequence, but can occur multiple times

in different elements. An itemset is considered to be a sequence with a single element. We assume without loss of generality that the items in an element of a sequence (i.e. in an itemset) are in lexicographic order.

A sequence $\langle a_1 a_2 ... a_n \rangle$ is a *subsequence* of another sequence $\langle b_1 b_2 ... b_m \rangle$ if there exist integers $i_1 < i_2 < ... < i_n$ such that $a_1 \subseteq b_{i_1}$, $a_2 \subseteq b_{i_2}$, ..., $a_n \subseteq b_{i_n}$. For example, the sequence $\langle$ (3) (4 5) (8) $\rangle$ is a subsequence of $\langle$ (7) (3, 8) (9) (4, 5, 6) (8) $\rangle$, since (3) $\subseteq$ (3, 8), (4, 5) $\subseteq$ (4, 5, 6) and (8) $\subseteq$ (8). However, the sequence $\langle$ (3) (5) $\rangle$ is not a subsequence of $\langle$ (3, 5) $\rangle$ (and vice versa).

Input We are given a database $\mathcal{D}$ of sequences called *data-sequences*. Each data-sequence is a list of transactions, ordered by increasing transaction-time. A transaction has the following fields: sequence-id, transaction-id, transaction-time, and the items present in the transaction. While we expect the items in a transaction to be leaves in $\mathcal{T}$, we do not require this.

For simplicity, we assume that no data-sequence has more than one transaction with the same transaction-time, and we use the transaction-time as the transaction-identifier. We do not consider quantities of items in a transaction.

Support The *support count* (or simply *support*) for a sequence is defined as the fraction of total data-sequences that "contain" this sequence. (Although the word "contains" is not strictly accurate once we incorporate taxonomies, it captures the spirt of when a data-sequence contributes to the support of a sequential pattern.) We now define when a data-sequence *contains* a sequence, starting with the definition as in [AS95], and then adding taxonomies, sliding windows, and time constraints:

- **as in [AS95]:** In the absence of taxonomies, sliding windows, and time constraints, a data-sequence contains a sequence s if s is a subsequence of the data-sequence.

- **plus taxonomies:** We say that a transaction T *contains* an item $x \in \mathcal{I}$ if x is in T or x is an ancestor of some item in T. We say that a transaction T *contains* an itemset $y \subseteq \mathcal{I}$ if T contains every item in y. A data-sequence $d = \langle d_1 ... d_m \rangle$ contains a sequence $s = \langle s_1 ... s_n \rangle$ if there exist integers $i_1 < i_2 < ... < i_n$ such that s_1 is contained in d_{i_1}, s_2 is contained in d_{i_2}, ..., s_n is contained in d_{i_n}. If there is no taxonomy, this degenerates into a simple subsequence test.

- **plus sliding windows:** The sliding window generalization relaxes the definition of when a data-sequence contributes to the support of a sequence by allowing a set of transactions to contain an element of a sequence as long as the difference in transaction-times between the transactions in the set is less than the user-specified window-size. Formally, a data-sequence $d = \langle d_1 ... d_m \rangle$ contains a sequence $s = \langle s_1 ... s_n \rangle$ if there exist integers $l_1 \leq u_1 < l_2 \leq u_2 < ... < l_n \leq u_n$ such that

  1. s_i is contained in $\cup_{k=l_i}^{u_i} d_k$, $1 \leq i \leq n$, and
  2. transaction-time(d_{u_i}) − transaction-time$(d_{l_i}) \leq$ window-size, $1 \leq i \leq n$.

- **plus time constraints:** Time constraints restrict the time gap between sets of transactions that contain consecutive elements of the sequence. Given user-specified window-size, max-gap and min-gap, a data-sequence $d = \langle d_1 ... d_m \rangle$ contains a sequence $s = \langle s_1 ... s_n \rangle$ if there exist integers $l_1 \leq u_1 < l_2 \leq u_2 < ... < l_n \leq u_n$ such that

  1. s_i is contained in $\cup_{k=l_i}^{u_i} d_k$, $1 \leq i \leq n$,
  2. transaction-time(d_{u_i}) − transaction-time$(d_{l_i}) \leq$ window-size, $1 \leq i \leq n$,
  3. transaction-time(d_{l_i}) − transaction-time$(d_{u_{i-1}}) >$ min-gap, $2 \leq i \leq n$, and

4. transaction-time(d_{u_i}) − transaction-time($d_{l_{i-1}}$) ≤ max-gap, $2 \leq i \leq n$.

The first two conditions are the same as in the earlier definition of when a data-sequence contains a pattern. The third condition specifies the minimum time-gap constraint, and the last one specifies the maximum time-gap constraint.

We will refer to transaction-time(d_{l_i}) as *start-time*(s_i), and transaction-time(d_{u_i}) as *end-time*(s_i). In other-words, start-time(s_i) and end-time(s_i) correspond to the first and last transaction-times of the set of transactions that contain s_i.

Note that if there is no taxonomy, min-gap = 0, max-gap = ∞ and window-size = 0 we get the notion of sequential patterns as introduced in [AS95], where there are no time constraints and items in an element come from a single transaction.

5.2.1 Problem Statement

Given a database $\mathcal{D}$ of data-sequences, a taxonomy $\mathcal{T}$, user-specified min-gap and max-gap time constraints, and a user-specified sliding-window size, the generalized problem of mining sequential patterns is to find all sequences whose support is greater than the user-specified minimum support. Each such sequence represents a *sequential pattern*, also called a *frequent* sequence.

Given a frequent sequence $s = \langle s_1 ... s_n \rangle$, it is often useful to know the "support relationship" between the elements of the sequence, i.e., what fraction of the data-sequences that support $\langle s_1 ... s_i \rangle$ support the entire sequence s. Since $\langle s_1 ... s_i \rangle$ must also be a frequent sequence, this relationship can easily be computed.

5.2.2 Example

Consider the data-sequences shown in Figure 42. For simplicity, we have assumed that the transaction-times are integers; they could represent, for instance, the number of days

Database $\mathcal{D}$

Sequence-Id	Transaction Time	Items
C1	1	Ringworld
C1	2	Foundation
C1	15	Ringworld Engineers, Second Foundation
C2	1	Foundation, Ringworld
C2	20	Foundation and Empire
C2	50	Ringworld Engineers

Taxonomy $\mathcal{T}$

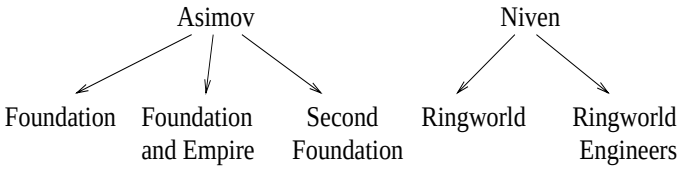


Figure 42: Dataset for Example

after January 1, 1995. We have used an abbreviated version of the taxonomy given in Figure 41. Assume that the minimum support has been set to 2 data-sequences.

With the [AS95] problem definition, the only 2-element sequential patterns are:

⟨(Ringworld) (Ringworld Engineers)⟩,

⟨(Foundation) (Ringworld Engineers)⟩

Setting a sliding-window of 7 days adds the pattern

⟨(Foundation, Ringworld) (Ringworld Engineers)⟩

since C1 now supports this pattern. ("Foundation" and "Ringworld" are present within a period of 7 days in data-sequence C1.)

Further setting a max-gap of 30 days results in all three patterns being dropped, since they are no longer supported by customer C2.

L_1 := {frequent 1-item sequences};
$k := 2$; // k represents the pass number
while ($L_{k-1} \neq \emptyset$) **do begin**
  C_k := New candidates of size k generated from L_{k-1};
  forall sequences $S \in \mathcal{D}$ **do begin**
    Increment the count of all candidates in C_k that are contained in S.
  end
  L_k := All candidates in C_k with minimum support.
  $k := k + 1$;
end
Answer := $\bigcup_k L_k$;

Figure 43: GSP Algorithm Overview

If we only add the taxonomy, but no sliding-window or time constraints, one of the patterns added is:

⟨ (Foundation) (Asimov) ⟩

Observe that this pattern is not simply a replacement of an item with its ancestor in an existing pattern. That is, without the taxonomy, there was no specialization of ⟨ (Foundation) (Asimov) ⟩ in the set of sequential patterns.

5.3 GSP Algorithm

The basic structure of the GSP algorithm is very similar to the Apriori algorithm (Figure 1), except that GSP deals with sequences rather than itemsets. The differences between Apriori and GSP are in the details of candidate generation and counting itemsets. Figure 43 shows an overview of the GSP algorithm. L_k and C_k refer to frequent k-sequences and candidate k-sequences respectively.

We need to specify two key details:

1. **Candidate generation:** how candidates sequences are generated before the pass

begins. We want to generate as few candidates as possible while maintaining completeness.

2. **Counting candidates:** how the support count for the candidate sequences is determined.

Candidate generation is discussed in Section 5.3.1, and candidate counting in Section 5.3.2. We incorporate time constraints and sliding windows in this discussion, but do not consider taxonomies. Extensions required to handle taxonomies are described in Section 5.3.3.

Our algorithm is not a main-memory algorithm; memory management is similar to that for the Apriori algorithm. The essential idea there was that we only generate as many candidates as fit in memory, make a pass to count their support, save the frequent sequences to disk, generate more candidates and so on. Thus we may make multiple passes to count C_k if there is insufficient memory.

5.3.1 Candidate Generation

We refer to a sequence with k items as a k-sequence. (If an item occurs multiple times in different elements of a sequence, each occurrence contributes to the value of k.) Let L_k denote the set of all frequent k-sequences, and C_k the set of candidate k-sequences.

Given L_{k-1}, the set of all frequent $(k-1)$-sequences, we want to generate a superset of the set of all frequent k-sequences. We first define the notion of a contiguous subsequence.

Definition Given a sequence $s = \langle s_1 s_2 ... s_n \rangle$ and a subsequence c, c is a *contiguous* subsequence of s if any of the following conditions hold:

1. c is derived from s by dropping an item from either s_1 or s_n.

2. c is derived from s by dropping an item from an element s_i which has at least 2 items.

3. c is a contiguous subsequence of c', and c' is a contiguous subsequence of s.

For example, consider the sequence $s = \langle (1, 2) (3, 4) (5) (6) \rangle$. The sequences $\langle (2) (3, 4) (5) \rangle$, $\langle (1, 2) (3) (5) (6) \rangle$ and $\langle (3) (5) \rangle$ are some of the contiguous subsequences of s. However, $\langle (1, 2) (3, 4) (6) \rangle$ and $\langle (1) (5) (6) \rangle$ are not.

As we will show in Lemma 10 below, any data-sequence that contains a sequence s will also contain any contiguous subsequence of s. If there is no max-gap constraint, the data-sequence will contain all subsequences of s (including non-contiguous subsequences). This property provides the basis for the candidate generation procedure.

Candidates are generated in two steps:

1. **Join Phase.** We generate candidate sequences by joining L_{k-1} with L_{k-1}. A sequence s_1 joins with s_2 if the subsequence obtained by dropping the first item of s_1 is the same as the subsequence obtained by dropping the last item of s_2. The candidate sequence generated by joining s_1 with s_2 is the sequence s_1 extended with the last item in s_2. The added item becomes a separate element if it was a separate element in s_2, and part of the last element of s_1 otherwise. When joining L_1 with L_1, we need to add the item in s_2 both as part of an itemset and as a separate element, since both $\langle (\mathrm{x}) (\mathrm{y}) \rangle$ and $\langle (\mathrm{x\ y}) \rangle$ give the same sequence $\langle (\mathrm{y}) \rangle$ upon deleting the first item. (Observe that s_1 and s_2 are contiguous subsequences of the new candidate sequence.)

2. **Prune Phase.** We delete candidate sequences that have a contiguous $(k-1)$-subsequence whose support count is less than the minimum support. If there is no max-gap constraint, we also delete candidate sequences that have any subsequence

Frequent 3-Sequences	Candidate 4-Sequences	
	after join	after pruning
⟨ (1, 2) (3) ⟩	⟨ (1, 2) (3, 4) ⟩	⟨ (1, 2) (3, 4) ⟩
⟨ (1, 2) (4) ⟩	⟨ (1, 2) (3) (5) ⟩	
⟨ (1) (3, 4) ⟩		
⟨ (1, 3) (5) ⟩		
⟨ (2) (3, 4) ⟩		
⟨ (2) (3) (5) ⟩		

Figure 44: Candidate Generation: Example

without minimum support.

The above procedure is reminiscent of the candidate generation procedure for finding association rules given in Section 2.2.1; however its details are quite different.

Example Figure 44 shows L_3 and C_4 after the join and prune phases. In the join phase, the sequence ⟨ (1, 2) (3) ⟩ joins with ⟨ (2) (3, 4) ⟩ to generate ⟨ (1, 2) (3, 4) ⟩ and with ⟨ (2) (3) (5) ⟩ to generate ⟨ (1, 2) (3) (5) ⟩. The remaining sequences do not join with any sequence in L_3. For instance, ⟨ (1, 2) (4) ⟩ does not join with any sequence since there is no sequence of the form ⟨ (2) (4 x) ⟩ or ⟨ (2) (4) (x) ⟩. In the prune phase, ⟨ (1, 2) (3) (5) ⟩ is dropped since its contiguous subsequence ⟨ (1) (3) (5) ⟩ is not in L_3.

Correctness We need to show that $C_k \supseteq L_k$. We first prove the following lemma.

Lemma 10 *If a data-sequence d contains a sequence s, d will also contain any contiguous subsequence of s. If there is no max-gap constraint, d will contain any subsequences of s.*

Proof: Let c denote any contiguous subsequence of s obtained by dropping just one item from s. If we show that any data-sequence that contains s also contains c, we can use induction to show that the data-sequence will contain any contiguous subsequence of s.

Let s have n elements, that is, $s = \langle s_1..s_n \rangle$. Now, c either has n elements or $n-1$ elements. Let us first consider the case where c has n elements; so $c = \langle c_1..c_n \rangle$. Let $l_1, u_1, ..., l_n, u_n$ define the transactions in d that supported s; that is, s_i is contained in $\cup_{k=l_i}^{u_i} d_k$, $1 \leq i \leq n$. In other words, l_i and u_i together define the set of transactions in d that contain s_i. Now, since $c_i \subseteq s_i$, c_i is also contained in $\cup_{k=l_i}^{u_i} d_k$, $1 \leq i \leq n$. Since $l_1, u_1, ..., l_n, u_n$ satisfied the min-gap, max-gap and window-size constraints for s, they also satisfy the constraints for c. Thus d contains c.

If c has $n-1$ elements, either the first or the last element of s consisted of a single item and was dropped completely. In this case, we use a similar argument to show that d contains c, except that we just look at the transactions corresponding to $l_1, u_1, ..., l_{n-1}, u_{n-1}$ or those corresponding to $l_2, u_2, ..., l_n, u_n$. □

Theorem 2 *Given L_{k-1}, the set of all frequent $(k-1)$-sequences, the candidate generation procedure produces a superset of L_k, the set of all frequent k-sequences.*

Proof: From Lemma 10, if we extended each sequence in L_{k-1} with every frequent item, and then deleted all those whose contiguous $(k-1)$-subsequences were not in L_{k-1}, we would be left with a superset of the sequences in L_k. This join is equivalent to extending L_{k-1} with each frequent item and then deleting those sequences for which the $(k-1)$-subsequence obtained by deleting the first item is not in L_{k-1}. Note that the subsequence obtained by deleting the first item is a contiguous subsequence. Thus, after the join step, $C_k \supseteq L_k$. By similar reasoning, the prune step, where we delete from C_k all sequences whose contiguous $(k-1)$-subsequences are not in L_{k-1}, also does not delete any sequence that could be in L_k. □

5.3.2 Counting Support of Candidates

While making a pass, we read one data-sequence at a time and increment the support count of candidates contained in the data-sequence. Thus, given a set of candidate sequences C and a data-sequence d, we need to find all sequences in C that are contained in d. We use two techniques to solve this problem:

1. We use a *hash-tree* data structure to reduce the number of candidates in C that are checked for a data-sequence.
2. We transform the representation of the data-sequence d so that we can efficiently find whether a specific candidate is a subsequence of d.

Reducing the number of candidates that need to be checked

We adapt the hash-tree data structure of Section 2.2.1 for this purpose. A node of the hash-tree either contains a list of sequences (a *leaf* node) or a hash table (an *interior* node). In an interior node, each non-empty bucket of the hash table points to another node. The root of the hash-tree is defined to be at depth 1. An interior node at depth p points to nodes at depth $p+1$.

Adding candidate sequences to the hash-tree When we add a sequence s, we start from the root and go down the tree until we reach a leaf. At an interior node at depth p, we decide which branch to follow by applying a hash function to the pth item of the sequence. (Note that we apply the hash function to the pth item, not the pth element.) All nodes are initially created as leaf nodes. When the number of sequences in a leaf node exceeds a threshold, the leaf node is converted to an interior node.

Finding the candidates contained in a data-sequence Starting from the root node, we find all the candidates contained in a data-sequence d. We apply the following

procedure, based on the type of node that we are currently at:

- *Interior node, if it is the root*: We apply the hash function to each item in d, and recursively apply this procedure to the node in the corresponding bucket. For any sequence s contained in the data-sequence d, the first item of s must be in d. By hashing on every item in d, we ensure that we only ignore sequences that start with an item not in d.

- *Interior node, if it is not the root*: Assume we reached this node by hashing on an item x whose transaction-time is t. We apply the hash function to each item in d whose transaction-time is in $[t - \text{window-size}, t + \max(\text{window-size}, \text{max-gap})]$ and recursively apply this procedure to the node in the corresponding bucket.

  To see why this returns the desired set of candidates, consider a candidate sequence s with two consecutive items x and y. Let x be contained in a transaction in d whose transaction-time is t. For d to contain s, the transaction-time corresponding to y must be in $[t - \text{window-size}, t + \text{window-size}]$ if y is part of the same element as x, or in the interval $(t, t + \text{max-gap}]$ if y is part of the next element. Hence, if we reached this node by hashing on an item x with transaction-time t, y must be contained in a transaction whose transaction-time is in the interval $[t - \text{window-size}, t + \max(\text{window-size}, \text{max-gap})]$ for the data-sequence to support the sequence. Thus, we only need to apply the hash function to the items in d whose transaction-times are in the above interval and check the corresponding nodes.

- *Leaf node*: For each sequence s in the leaf, we check whether d contains s and add s to the answer set if necessary. (We will discuss below exactly how to find out whether or not d contains a specific candidate sequence.) Since we check each sequence contained in this node, we don't miss any sequences.

Checking whether a data-sequence contains a specific sequence

Let d be a data-sequence, and let $s = \langle s_1 ... s_n \rangle$ be a candidate sequence. We first describe the algorithm for checking if d contains s, assuming the existence of a procedure that finds the first occurrence of an element of s in d after a given time; we then describe this procedure as well.

Contains test The algorithm for checking if the data-sequence d contains a candidate sequence s alternates between two phases. The algorithm starts in the forward phase from the first element.

- **Forward phase:** The algorithm finds successive elements of s in d as long as the difference between the end-time of the element just found and the start-time of the previous element is less than max-gap. (Recall that for an element s_i, start-time(s_i) and end-time(s_i) correspond to the first and last transaction-times of the set of transactions that contain s_i.) If the difference is more than max-gap, the algorithm switches to the backward phase. If an element is not found, the data-sequence does not contain s.

- **Backward phase:** The algorithm backtracks and "pulls up" previous elements. If s_i is the current element and end-time(s_i) $= t$, the algorithm finds the first set of transactions containing s_{i-1} whose transaction-times are after $t -$ max-gap. The start-time for s_{i-1} (after s_{i-1} is pulled up) could be after the end-time for s_i. Pulling up s_{i-1} may necessitate pulling up s_{i-2} because the max-gap constraint between s_{i-1} and s_{i-2} may no longer be satisfied. The algorithm moves backwards until either the max-gap constraint between the element just pulled up and the previous element is satisfied, or the first element has been pulled up. The algorithm then switches to the forward phase, finding elements of s in d starting from the element

after the last element pulled up. If any element cannot be pulled up (that is, there is no subsequent set of transactions which contain the element), the data-sequence does not contain s.

This procedure is repeated, switching between the backward and forward phases, until all the elements are found. Though the algorithm moves back and forth among the elements of s, it still terminates. For any element s_i, the algorithm always checks whether a later set of transactions contains s_i. So the transaction-times for an element always increase.

Example Consider the data-sequence shown in Figure 45. Consider the case when max-gap is 30, min-gap is 5, and window-size is 0. For the candidate-sequence $\langle (1, 2)\ (3)\ (4) \rangle$, we would first find (1, 2) at transaction-time 10, and then find (3) at time 45. Since the gap between these two elements (35 days) is more than max-gap, we "pull up" (1, 2). We search for the first occurrence of (1, 2) after time 15, because end-time((3)) = 45 and max-gap is 30, and so occurences of (1, 2) at some time before 15 will not satisfy the max-gap constraint. We find (1, 2) at time 50. Since this is the first element, we do not have to check to see if the max-gap constraint between (1, 2) and the element before that is satisfied. We now move forward. Since (3) no longer occurs more than 5 days after (1, 2), we search for the next occurrence of (3) after time 55. We find (3) at time 65. Since the max-gap constraint between (3) and (1, 2) is satisfied, we continue to move forward and find (4) at time 90. The max-gap constraint between (4) and (3) is satisfied, so we are done.

Finding a single element To describe the procedure for finding the first occurrence of an element in a data sequence, we first discuss how to efficiently find a single item. A straightforward approach would be to scan consecutive transactions of the data-sequence

Transaction-Time	Items
10	1, 2
25	4, 6
45	3
50	1, 2
65	3
90	2, 4
95	6

Figure 45: Example Data-Sequence

Item	Times
1	$\rightarrow 10 \rightarrow 50 \rightarrow$ NULL
2	$\rightarrow 10 \rightarrow 50 \rightarrow 90 \rightarrow$ NULL
3	$\rightarrow 45 \rightarrow 65 \rightarrow$ NULL
4	$\rightarrow 25 \rightarrow 90 \rightarrow$ NULL
5	$\rightarrow$ NULL
6	$\rightarrow 25 \rightarrow 95 \rightarrow$ NULL

Figure 46: Alternate Representation

until we find the item. A faster alternative is to transform the representation of d as follows.

Create an array that has as many elements as the number of items in the database. For each item in the data-sequence d, store in this array a list of transaction-times of the transactions of d that contain the item. To find the first occurrence of an item after time t, the procedure simply traverses the list corresponding to the item till it finds a transaction-time greater than t. Figure 46 shows the tranformed representation of the data-sequence in Figure 45. This transformation has a one-time overhead of O(total-number-of-items-in-dataset) over the whole execution (to allocate and initialize the array), plus an overhead of O(no-of-items-in-d) for each data-sequence.

Now, to find the first occurrence of an element after time t, the algorithm makes one pass through the items in the element and finds the first transaction-time greater than t for each item. If the difference between the start-time and end-time is less than or equal to the window-size, we are done. Otherwise, t is set to the end-time minus the window-size, and the procedure is repeated.[1]

[1] An alternate approach would be to "pull up" previous items as soon as we find that the transaction-time for an item is too high. Such a procedure would be similar to the algorithm that does the contains test for a sequence.

Example Consider the data-sequence shown in Figure 45. Assume window-size is set to 7 days, and we have to find the first occurrence of the element (2, 6) after time $t = 20$. We find 2 at time 50, and 6 at time 25. Since end-time((2,6)) − start-time((2,6)) > 7, we set t to 43 (= end-time((2,6)) − window-size) and try again. Item 2 remains at time 50, while item 6 is found at time 95. The time gap is still greater than the window-size, so we set t to 88, and repeat the procedure. We now find item 2 at time 90, while item 6 remains at time 95. Since the time gap between 90 and 95 is less than the window size, we are done.

5.3.3 Taxonomies

The ideas presented in Chapter 3 for discovering association rules with taxonomies carry over to the current problem. The basic approach was to replace each data-sequence d with an "extended-sequence" d', where each transaction d'_i of d' contains the items in the corresponding transaction d_i of d, as well as all the ancestors of each item in d_i. For example, with the taxonomy shown in Figure 41, a data-sequence $\langle$ (Foundation, Ringworld) (Second Foundation) $\rangle$ would be replaced with the extended-sequence $\langle$ (Foundation, Ringworld, Asimov, Niven, Science Fiction) (Second Foundation, Asimov, Science Fiction) $\rangle$. We now run GSP on these "extended-sequences". The optimizations for the Cumulate and EstMerge algorithm (Sections 3.3.2 and 3.3.3) also apply. For example, the third optimization of the Cumulate algorithm translates to not counting sequential patterns with an element that contains both an item x and its ancestor y.

Similarly, the interest measure introduced in Section 3.2.1 can be adapted to prune redundant sequential patterns. The essential idea was that, given a user-specified interest-level I, we display patterns that have no ancestors or whose actual support is at least I times their expected support (based on the support of their ancestors).

5.4 Performance Evaluation

We now compare the performance of GSP to the AprioriAll algorithm given in [AS95] using both synthetic and real-life datasets. We also show the scale-up properties of GSP, and study the effect of time constraints and sliding-window transactions on the performance of GSP. Our experiments were performed on an IBM RS/6000 250 workstation with 128 MB of main memory running AIX 3.2.5. The data resided in the AIX file system and was stored on a local 2GB SCSI 3.5" drive with measured sequential throughput of about 2 MB/second.

5.4.1 Overview of the AprioriAll Algorithm

In order to explain performance trends, we must first give the essential details of the AprioriAll algorithm [AS95]. This algorithm splits the problem of finding sequential patterns into three phases:

1. **Itemset Phase.** All itemsets with minimum support are found. These also correspond to the sequential patterns with exactly 1 element. The Apriori algorithm for finding frequent itemsets given in Section 2.2.1 is used in this phase.

2. **Transformation Phase.** The frequent itemsets are mapped to integers. The database is then transformed, with each transaction being replaced by the set of all frequent itemsets contained in the transaction. This transformation can either be done on-the-fly, each time the algorithm makes a pass over the data in the sequence phase, or done once and cached. The latter option would be infeasible in many real applications, as the transformed data may be larger than the original database, especially at low levels of support.

Each data-sequence is now a list of sets of integers, where each integer represents a frequent itemset. Sequential patterns can now be considered lists of integers rather than lists of sets of items. (Any element of a sequential pattern with minimum support must be a frequent itemset.)

3. **Sequence Phase.** All frequent sequential patterns are found. The basic computational structure of this phase is similar to the one described for GSP. Starting with a seed of sequences found in the previous pass (found in the itemset phase for the first pass), the algorithm generates candidates, makes a pass over the data to find the support count of candidates, and uses those candidates with minimum support as the seed set for generating the next set of candidates. However, the candidates generated and counted during the kth pass correspond to all candidates with k elements rather than candidates with k items. Candidate generation is somewhat similar to the one for GSP since it is based on the intuition that all subsets/subsequences of an itemset/sequence with minimum support also have minimum support. However, it is much simpler as its candidates are lists of integers rather than a list of sets of integers.

5.4.2 Synthetic Data Generation

To evaluate the performance of the algorithms over a large range of data characteristics, we generated synthetic customer transactions. Our model is a generalization of the synthetic data generation model given in Section 2.4.3, and uses the parameters shown in Table 7. We now highlight the extensions to the earlier model.

We determine the number of transactions for the next customer, and the average size of each transaction for this customer as follows. The number of transactions is picked from a Poisson distribution with mean μ equal to $|C|$, and the average size of the

$\|D\|$	Number of customers (= size of Database)
$\|C\|$	Average number of transactions per Customer
$\|T\|$	Average number of items per Transaction
$\|S\|$	Average length of maximal potentially large Sequences
$\|I\|$	Average size of Itemsets in maximal potentially large sequences
N_S	Number of maximal potentially large Sequences
N_I	Number of maximal potentially large Itemsets
N	Number of items

Table 7: Parameters for Synthetic Data Generation

transaction is picked form a Poisson distribution with μ equal to $|T|$.

We then assign items to the transactions of the customer. Each customer is assigned a series of potentially large sequences. If the large sequence on hand does not fit in the customer-sequence, the itemset is put in the customer-sequence anyway in half the cases, and the itemset is moved to the next customer-sequence the rest of the cases.

Large sequences are chosen from a table $\mathcal{I}$ of such sequences. The number of sequences in $\mathcal{I}$ is set to N_S. There is an inverse relationship between N_S and the average support for potentially large sequences. A sequence in $\mathcal{I}$ is generated by first picking the number of itemsets in the sequence from a Poisson distribution with mean μ equal to $|S|$. Itemsets in the first sequence are chosen randomly from a table of large itemsets. The table of large itemsets is similar to the table of large sequences. Since the sequences often have common items, some fraction of itemsets in subsequent sequences are chosen from the previous sequence generated. We use an exponentially distributed random variable with mean equal to the *correlation level* to decide this fraction for each itemset. The remaining itemsets are picked at random. In the datasets used in the experiments, the correlation level was set to 0.25.

Each sequence in $\mathcal{I}$ has a weight associated with it that corresponds to the probability that this sequence will be picked. This weight is picked from an exponential distribution with unit mean, and is then normalized so that the sum of the weights for all the itemsets

Name	$\|C\|$	$\|T\|$	$\|S\|$	$\|I\|$
C10-T2.5-S4-I1.25	10	2.5	4	1.25
C10-T5-S4-I1.25	10	5	4	1.25
C10-T5-S4-I2.5	10	5	4	2.5
C20-T2.5-S4-I1.25	20	2.5	4	1.25
C20-T2.5-S4-I2.5	20	2.5	4	2.5
C20-T2.5-S8-I1.25	20	2.5	8	1.25

Table 8: Parameter values for synthetic datasets

in $\mathcal{I}$ is 1. As before, we assign each sequence in $\mathcal{I}$ a *corruption level* c to model the fact that all the items in a frequent sequence are not always bought together,

We generated datasets by setting $N_S = 5000$, $N_I = 25000$ and $N = 10000$. The number of data-sequences, $|D|$ was set to 100,000. Table 8 summarizes the dataset parameter settings.

5.4.3 Real-life Datasets

We used the following real-life datasets:

Mail Order: Clothes A transaction consists of items ordered by a customer in a single mail order. This dataset has 16,000 items. The average size of a transaction is 2.62 items. There are 214,000 customers and 2.9 million transactions; the average is 13 transactions per customer. The time period is about 10 years.

Mail Order: Packages A transaction consists of "packaged offers" ordered by a customer in a single mail order. This dataset has 69,000 items. The average size of a transaction is 1.65 items. There are 570,000 customers and 1.7 million transactions; the average is 1.6 transactions per customer. The time period is about 3 years.

Mail Order: General A transaction consists of items ordered by a customer in a single mail order. There are 71,000 items. The average size of a transaction is 1.74 items. There are 3.6 million customers and 6.3 million transactions; the average is 1.6 transactions per customer. The time period is about 3 months.

5.4.4 Comparison of GSP and AprioriAll

On the synthetic datasets, we varied minimum support from 1% to 0.25%. The results are shown in Figure 47. "AprioriAll" refers to the version that does the data transformation on-the-fly. Also, although it may be practically infeasible to transform the database once and cache it to disk, we have included this version in the comparison for completeness, referring to it as "AprioriAll-Cached". We did not include in these experiments those features not supported by AprioriAll (e.g. time constraints and sliding windows).

As the support decreases, more sequential patterns are found and the time increases. GSP is between 30% to 5 times faster than AprioriAll on the synthetic datasets, with the performance gap often increasing at low levels of minimum support. GSP is also slightly faster to 3 times faster than AprioriAll-Cached.

The execution times on the real datasets are shown in Figure 48. Except for the Mail Order: General dataset at 0.01% support, GSP runs 2 to 3 times faster than AprioriAll, with AprioriAll-Cached coming in between. These match the results on synthetic data. For the Mail Order: General dataset at 0.01% support, GSP is around 20 times faster than AprioriAll and around 9 times faster than AprioriAll-Cached. We explain this behavior below.

There are two main reasons why GSP does better than AprioriAll.

1. GSP counts fewer candidates than AprioriAll. AprioriAll prunes candidate sequences by checking if the subsequences obtained by dropping an *element* have

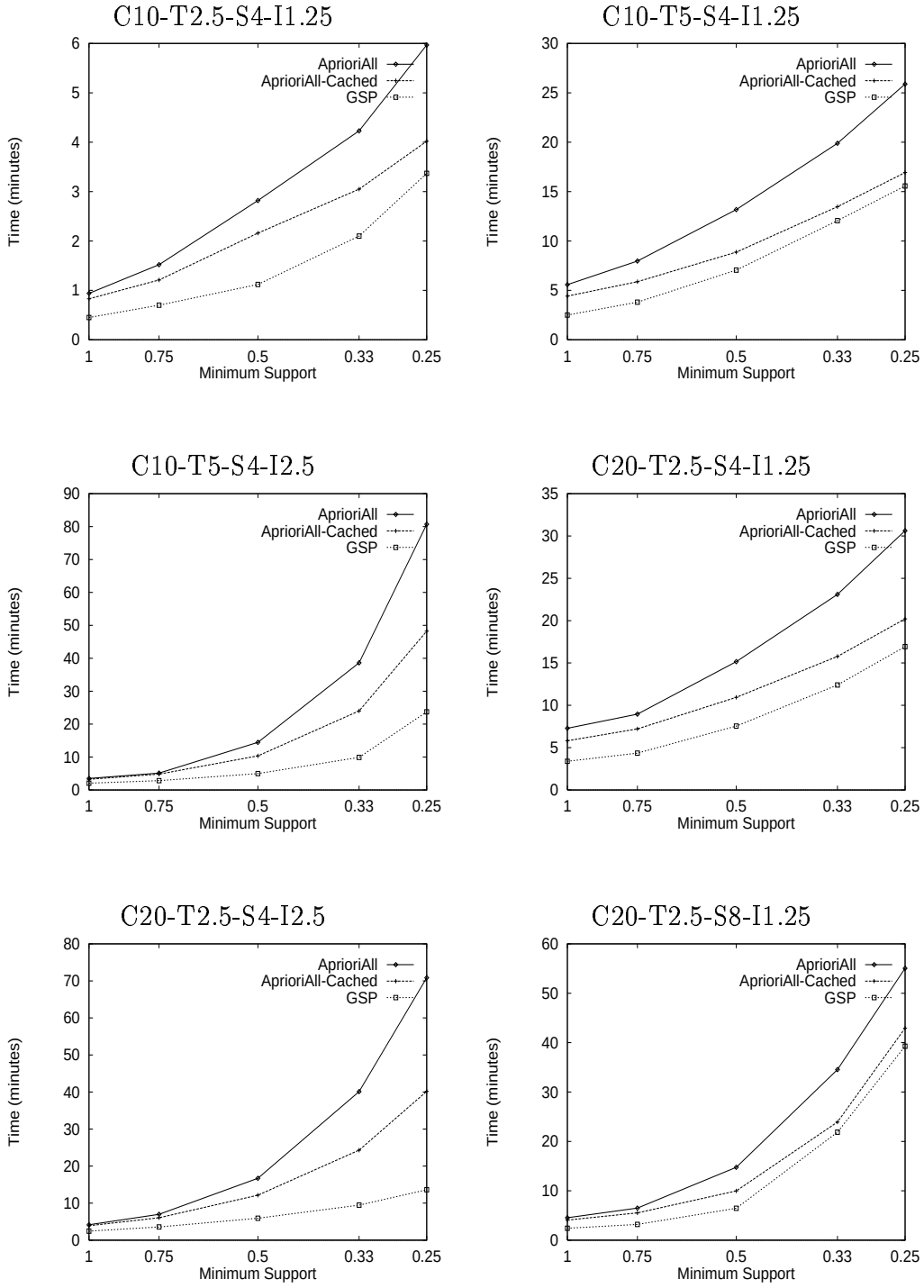


Figure 47: Performance Comparison: Synthetic Data

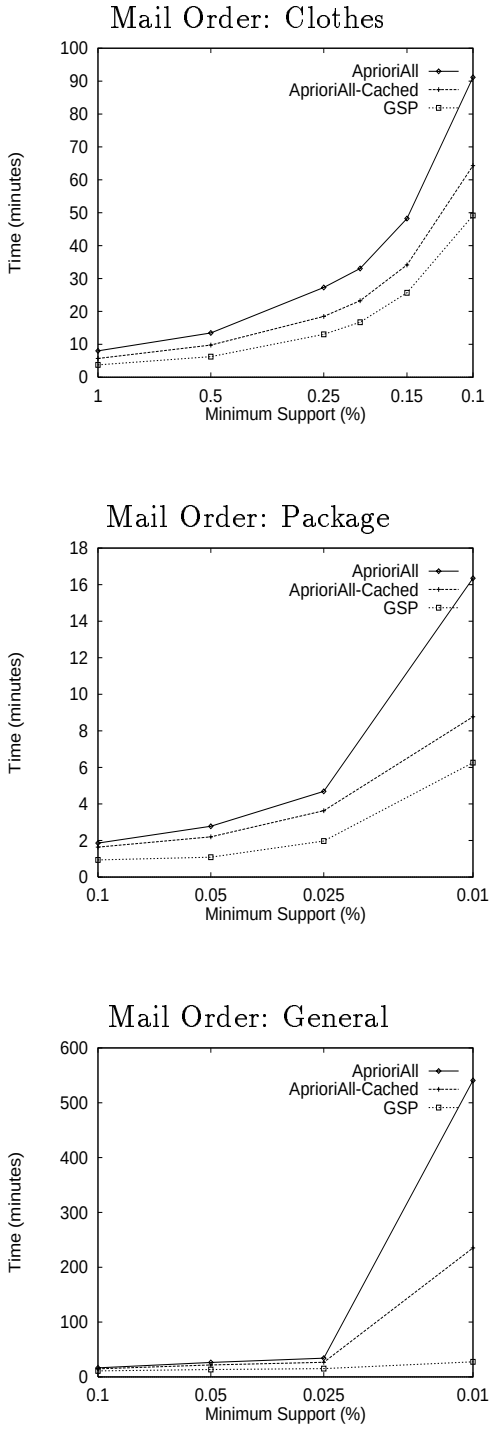


Figure 48: Performance Comparison: Real Data

minimum support, while GSP checks if the subsequences obtained by dropping an *item* have minimum support. Thus GSP always counts fewer candidates than AprioriAll. The difference in the number of candidates can be quite large for candidate sequences with 2 elements. AprioriAll has to do a cross-product of the frequent itemsets found in the itemset phase. GSP first counts sequences obtained by doing a cross-product on the frequent items, and generates 2-element candidates with more than 2 items later. If the number of large items is much smaller than the number of large itemsets, the performance gap can be dramatic. This is what happens in the Mail Order: General dataset at 0.01% support.

2. AprioriAll (the non-cached version) has to first find which frequent itemsets are present in each element of a data-sequence during the data transformation, and then find which candidate sequences are present in it. This is typically somewhat slower than directly finding the candidate sequences. For the cached version, the procedure used by AprioriAll to find which candidates are present in a data-sequence is either about as fast or slightly faster than the procedure use by GSP. However, AprioriAll still has to do the conversion once.

5.4.5 Scale-up

Fig. 49 shows how GSP scales up as the number of data-sequences is increased ten times from 100,000 to 1 million. We show the results for the dataset C10-T2.5-S4-I1.25 with three levels of minimum support. The execution times are normalized with respect to the times for the 100,000 data-sequences dataset. As shown, the execution times scale quite linearly. We observed similar behavior for the other datasets as well.

Next, we investigated the scale-up as we increased the total number of items in a data-sequence. This increase was realized in two different ways: i) by increasing

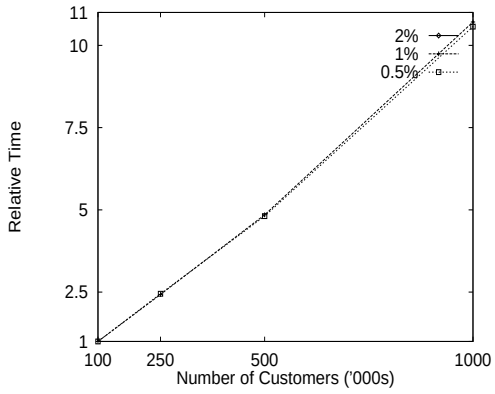


Figure 49: Scale-up: Number of data-sequences

the average number of transactions per data-sequence, keeping the average number of items per transaction the same; and ii) by increasing the average number of items per transaction, keeping the average number transactions per data-sequence the same. The aim of this experiment was to see how our data structures scale with the data-sequence size, independent of other factors like the database size and the number of frequent sequences. We kept the size of the database roughly constant by keeping the product of the average data-sequence size and the number of data-sequences constant. We fixed the minimum support in terms of the number of transactions in this experiment. Fixing the minimum support as a percentage would have led to large increases in the number of frequent sequences and we wanted to keep the size of the answer set roughly the same.

The results are shown in Fig. 50. All the experiments had the frequent sequence length set to 4 and the frequent itemset size set to 1.25. The average transaction size was set to 2.5 in the first graph, while the number of transactions per data-sequence was set to 10 in the second. The numbers in the key (e.g. 800) refer to the minimum support.

As shown, the execution times usually increased with the data-sequence size, but only gradually. There are two reasons for the increase. First, finding the candidates present in a data-sequence took a little more time. Second, despite setting the minimum support

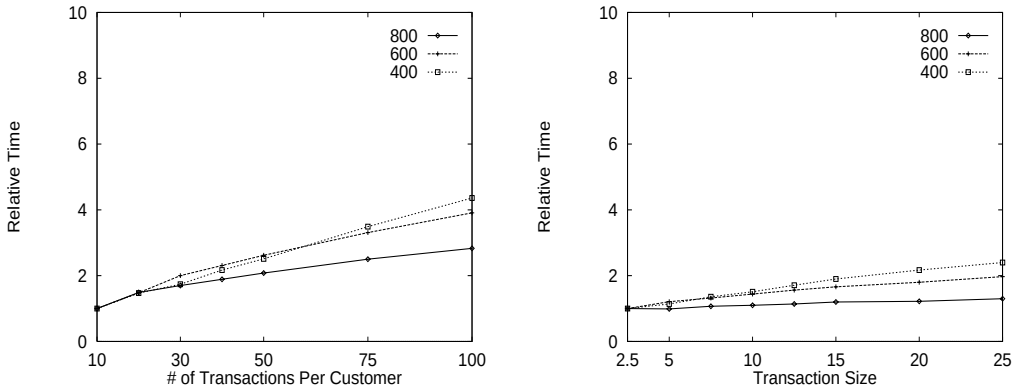


Figure 50: Scale-up: Number of Items per Data-Sequence

in terms of the number of data-sequences, the number of frequent sequences increased with increasing data-sequence size.

5.4.6 Effects of Time Constraints and Sliding Windows

To see the effect of the sliding window and time constraints on performance, we ran GSP on the three real datasets, using the min-gap, max-gap, sliding-window, and max-gap+sliding-window constraints.[2] The sliding-window was set to 1 day, so that the effect on the number of sequential patterns would be small. Similarly, the max-gap was set to more than the total time-span of the transactions in the dataset, and the min-gap was set to 1 day. Figure 51 shows the results.

The min-gap constraint comes for "free"; there was no performance degradation. The reason is that the min-gap constraint does not affect candidate generation, effectiveness of the hash-tree in reducing the number of candidates that need to be checked, or the speed of the contains test. However, there was a performance penalty of 5% to 30% for running the max-gap constraint or sliding windows. There are several reasons for this:

1. The time for the contains test increases when either the max-gap or sliding window

[2] We could not study the performance impact of running with and without the taxonomy. For a fixed minimum support, the number of sequential patterns found will be much higher when there is a taxonomy. If we try to fix the number of sequential patterns found, other factors such as the number of passes differ for the two runs.

option is used.

2. The number of candidates increases when the max-gap constraint is specified, since we can no longer prune non-contiguous subsequences.

3. When a sliding-window option is used, the effect of the hash-tree in pruning the number of candidates that we have to check against the data-sequence decreases somewhat. If we reach a node by hashing on item x, rather than just applying the hash function to the items after x and checking those nodes, we also have to apply the hash function to the items before x whose transaction-times are within window-size of the transaction-time for x.

For realistic values of max-gap, GSP will usually run significantly faster with the constraint than without, since there will be fewer candidate sequences. However, specifying a sliding window will increase the execution time, since both the overhead and the number of sequential patterns will increase.

5.5 Summary

We are given a database of sequences, where each sequence is a list of transactions ordered by transaction-time, and each transaction is a set of items. The problem of mining sequential patterns introduced in [AS95] is to discover all sequential patterns with a user-specified minimum support, where the support of a pattern is the number of data-sequences that contain the pattern.

In this chapter, we addressed some critical limitations of the earlier work in order to make sequential patterns useful for real applications. In particular, we generalized the definition of sequential patterns to admit max-gap and min-gap time constraints between adjacent elements of a sequential pattern. We also relaxed the restriction that all the

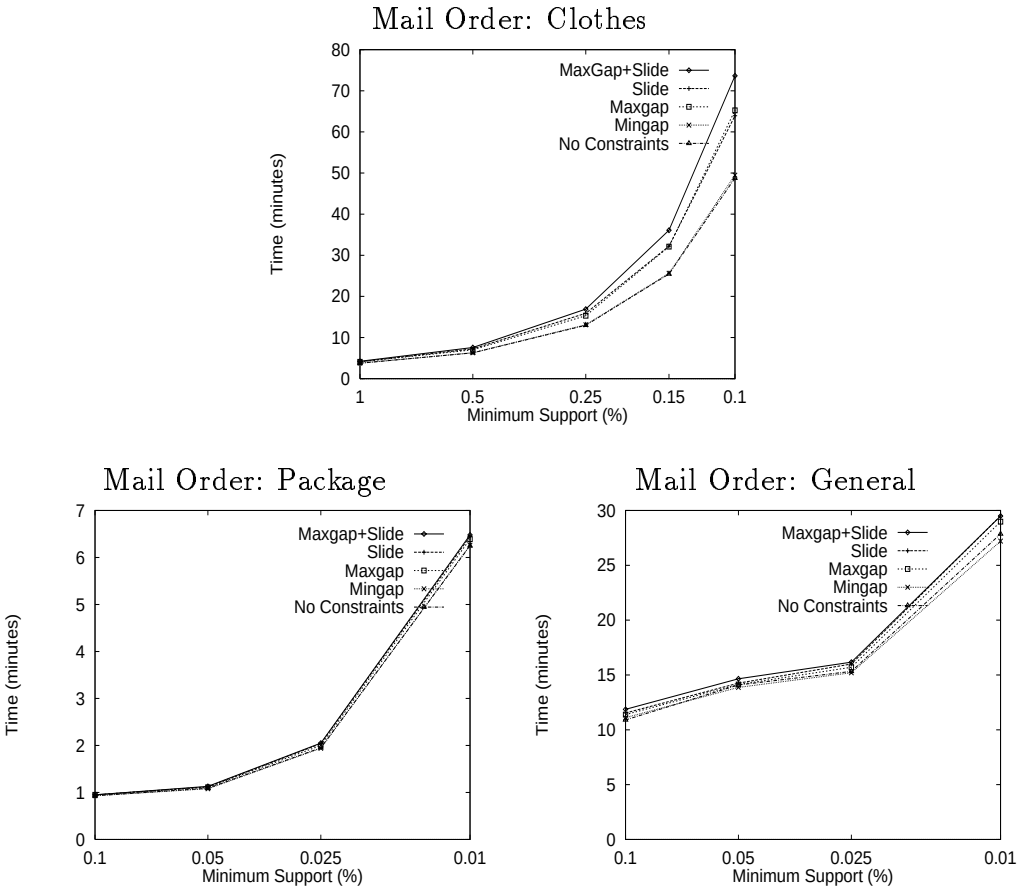


Figure 51: Effects of Extensions on Performance

items in an element of a sequential pattern must come from the same transaction, and allowed for a user-specified window-size within which the items can be present. Finally, if a user-defined taxonomy over the items in the database is available, the sequential patterns may include items across different levels of the taxonomy.

We presented GSP, a new algorithm that discovers these generalized sequential patterns. It is a complete algorithm in that it guarantees finding all patterns that have a user-specified minimum support. Empirical evaluation using synthetic and real-life data indicates that GSP is much faster than the AprioriAll algorithm presented in [AS95]. GSP scales linearly with the number of data-sequences, and has very good scale-up properties with respect to the average data-sequence size.

Chapter 6

Conclusions

In this chapter, we summarize the dissertation, discuss future work, and then conclude with some closing remarks.

6.1 Summary

We first considered the problem of discovering association rules between items in a large database of sales transactions. We presented two new algorithms for solving this problem that are fundamentally different from previously known algorithms. Experiments with synthetic as well as real-life data showed that these algorithms outperform the prior algorithms by factors ranging from three for small problems to more than an order of magnitude for large problems. We also showed how the best features of the two proposed algorithms can be combined into a hybrid algorithm, called AprioriHybrid. Scale-up experiments show that AprioriHybrid scales linearly with the number of transactions. AprioriHybrid also has excellent scale-up properties with respect to the transaction size and the number of items in the database.

Next, we generalized the problem by incorporating taxonomies. Given a large database of transactions, where each transaction consists of a set of items, and a taxonomy on the items, we find associations between items at any level of the taxonomy. An obvious solution to the problem is to add all ancestors of each item in a transaction to the transaction, and then run any of the algorithms for mining association rules on these "extended transactions". However, this "Basic" algorithm is not very fast; we presented two other

algorithms, Cumulate and EstMerge, which run 2 to 5 times faster than Basic (and more than 100 times faster on one real-life dataset). Between the two algorithms, EstMerge performs somewhat better than Cumulate, with the performance gap increasing as the size of the database increases.

We also presented a new interest-measure for rules which uses the information present in the taxonomy. The intuition behind this measure was that if the support and confidence of a rule are close to their expected values based on an ancestor of the rule, the more specific rule can be considered redundant. Given a user-specified "minimum-interest-level", this measure can prune a large number of redundant rules. We observed that it could prune 40% to 60% of all the generalized association rules on two real-life datasets.

Next, we introduced the problem of mining association rules in large relational tables that contain both quantitative and categorical attributes. An example of such an association rule might be "10% of married people between age 50 and 60 have at least 2 cars". We dealt with quantitative attributes by fine-partitioning the values of the attribute and then combining adjacent partitions as necessary. We introduced measures of partial completeness to quantify the information lost due to partitioning. A direct application of this technique can generate too many similar rules. We tackled this problem by extending the "greater-than-expected-value" interest measure to identify the interesting rules in the output. We gave an algorithm for mining such quantitative association rules, and described the results of using this approach on a real-life dataset.

Finally, we introduced the problem of mining sequential patterns, which is closely related to the problem of mining association rules. Given a database of sequences, where each sequence is a list of transactions ordered by transaction-time, and each transaction is a set of items, the problem is to discover all sequential patterns with a user-specified

minimum support. The support of a pattern is the number of data-sequences that contain the pattern. An example of a sequential pattern is "5% of customers bought 'Foundation' and 'Ringworld' in one transaction, followed by 'Second Foundation' in a later transaction". We generalized the definition of sequential patterns to admit max-gap and min-gap time constraints between adjacent elements of a sequential pattern. We also relaxed the restriction that all items in an element of a sequential pattern must come from the same transaction by allowing a user-specified window-size within which the items can be present. Finally, if a user-defined taxonomy over the items in the database is available, the sequential patterns may include items across different levels of the taxonomy. We presented GSP, a new algorithm that discovers these generalized sequential patterns. GSP scales linearly with the number of data-sequences, and has very good scale-up properties with respect to the average data-sequence size.

6.2 Future Work

We now discuss topics for future work.

Quantitative Association: Other Measures of Partial Completeness We presented a measure of partial completeness based on the support of the rules. Alternate measures may be useful for some applications. For instance, we may generate a partial completeness measure based on the range of the attributes in the rules. (For any rule, we will have a generalization such that the range of each attribute is at most K times the range of the corresponding attribute in the original rule.)

Quantitative Association: Other Partitioning Methods Equi-depth partitioning may not be the best approach for highly skewed data. This is because it tends to split adjacent values with high support into separate intervals even though their behavior

would typically be similar. It may be worth exploring the use of clustering algorithms [JD88] for partitioning and their relationship to partial completeness.

Quantitative Sequential Patterns The concepts of partial completeness and the interest measure that we developed for quantitative associations can be directly applied to sequential patterns. However, the algorithm for finding quantitative associations does not map directly. Developing an efficient algorithm for mining quantitative sequential patterns is an open problem.

Quantitative Associations and Clustering Each quantitative association rule, after pruning by the interest measure, corresponds to a cluster in a lower-dimensional space. It would be interesting to explore the relationship between quantitative associations and clustering algorithms.

Rule Interest Finding the interesting patterns in the output of association rules or sequential patterns is an area where more work needs to be done. While the greater-than-expected value interest measure helps, incorporating additional domain knowledge will reduce the number of uninteresting rules even further. One of the challenges is identifying types of domain knowledge (apart from taxonomies) that are applicable across several domains and do not require a lot of effort by the user to create them.

Visualization Visualizing the output of running associations or sequential patterns is an interesting research problem, since standard techniques do not scale beyond a few hundred items and very simple association rules. A related problem is coming up with high-quality, easy-to-generate domain-specific or application-specific visualizations.

6.3 Closing Remarks

This dissertation presented fast algorithms and data structures for discovering association rules. The problem of mining association rules was then generalized by incorporating taxonomies and quantitative attributes, and the algorithms extended to discover these generalized rules. Finally, the dissertation presented a fast algorithm for mining associations over time, called sequential patterns. The algorithms for mining associations and sequential patterns have been successfully applied in many domains, including retail, direct marketing, fraud detection and medical research.

Bibliography

[ABN92] Tarek M. Anwar, Howard W. Beck, and Shamkant B. Navathe. Knowledge mining by imprecise querying: A classification-based approach. In *IEEE 8th Int'l Conference on Data Engineering*, Phoenix, Arizona, February 1992.

[AFS93] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. Efficient similarity search in sequence databases. In *Proc. of the Fourth Int'l Conference on Foundations of Data Organization and Algorithms*, Chicago, October 1993. Also in Lecture Notes in Computer Science 730, Springer Verlag, 1993, 69–84.

[AGM+90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. A basic local alignment search tool. *Journal of Molecular Biology*, 1990.

[AIS93a] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993.

[AIS93b] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207–216, Washington, D.C., May 1993.

[ALSS95] Rakesh Agrawal, King-Ip Lin, Harpreet S. Sawhney, and Kyuseok Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, September 1995.

[ANB92] Tarek M. Anwar, Shamkant B. Navathe, and Howard W. Beck. Knowledge mining in databases: A unified approach through conceptual clustering. Technical report, Georgia Institute of Technology, May 1992.

[AP95] Rakesh Agrawal and Giuseppe Psaila. Active data mining. In *Proc. of the 1st Int'l Conference on Knowledge Discovery in Databases and Data Mining*, Montreal, Canada, August 1995.

[APWZ95] Rakesh Agrawal, Giuseppe Psaila, Edward L. Wimmers, and Mohamed Zaït. Querying shapes of histories. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, September 1995.

[AS92] Noga Alon and Joel H. Spencer. *The Probabilistic Method.* John Wiley Inc., New York, 1992.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, Santiago, Chile, September 1994.

[AS95] Rakesh Agrawal and Ramakrishnan Srikant. Mining Sequential Patterns. In *Proc. of the 11th Int'l Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[Ass90] David Shepard Associates. The new direct marketing. Business One Irwin, Illinois, 1990.

[Ass92] Direct Marketing Association. Managing database marketing technology for success, 1992.

[B+93] R. J. Brachman et al. Integrated support for data archeology. In *AAAI-93 Workshop on Knowledge Discovery in Databases*, July 1993.

[BFOS84] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth, Belmont, 1984.

[Bit92] D. Bitton. Bridging the gap between database theory and practice, 1992.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: an efficient and robust access method for points and rectangles. In *Proc. of ACM SIGMOD*, pages 322–331, Atlantic City, NJ, May 1990.

[Cat91] J. Catlett. Megainduction: A test flight. In *8th Int'l Conference on Machine Learning*, June 1991.

[CHNW96] D. Cheung, J. Han, V. Ng, and C.Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating techniques. In *Proc. of 1996 Int'l Conference on Data Engineering*, New Orleans, USA, February 1996.

[CKS+88] P. Cheeseman, James Kelly, Matthew Self, et al. AutoClass: A Bayesian classification system. In *5th Int'l Conference on Machine Learning*. Morgan Kaufman, June 1988.

[CR93] Andrea Califano and Isidore Rigoutsos. FLASH: A fast look-up algorithm for string homology. In *Proc. of the 1st Int'l Conference on Intelligent Systems for Molecular Biology*, pages 353–359, Bethesda, MD, July 1993.

[DM85] Thomas G. Dietterich and Ryszard S. Michalski. Discovering patterns in sequences of events. *Artificial Intelligence*, 25:187–232, 1985.

[Fis87] Douglas H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2(2), 1987.

[FMMT96a] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Montreal, Canada, June 1996.

[FMMT96b] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Mining optimized association rules for numeric attributes. In *Proc. of the 15th ACM Symposium on Principles of Database Systems*, Montreal, Canada, June 1996.

[FPSSU95] Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1995.

[FRM94] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, May 1994.

[FWD93] Usama Fayyad, Nicholas Weir, and S. G. Djorgovski. Skicat: A machine learning system for automated cataloging of large scale sky surveys. In *10th Int'l Conference on Machine Learning*, June 1993.

[HCC92] Jiawei Han, Yandong Cai, and Nick Cercone. Knowledge discovery in databases: An attribute oriented approach. In *Proc. of the VLDB Conference*, pages 547–559, Vancouver, British Columbia, Canada, 1992.

[HF95] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, September 1995.

[HR90] Torben Hagerup and Christine Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33:305–308, 1989/90.

[HS94] M. Holsheimer and A. Siebes. Data mining: The search for knowledge in databases. Technical Report CS-R9406, CWI, Netherlands, 1994.

[HS95] Maurice Houtsma and Arun Swami. Set-oriented mining of association rules. In *Int'l Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[JD88] A. K. Jain and R. C. Dubes. *Algorithms for clustering data.* Prentice Hall, 1988.

[KI91] Ravi Krishnamurthy and Tomasz Imielinski. Practitioner problems in need of database research: Research directions in knowledge discovery. *SIGMOD RECORD*, 20(3):76–78, September 1991.

[MKKR92] R. S. Michalski, L. Kerschberg, K. A. Kaufman, and J. S. Ribeiro. Mining for knowledge in databases: The INLEN architecture, initial implementation, and first results. *Journal of Intelligent Information Systems*, 1:85–113, 1992.

[MR87] Heikki Mannila and Kari-Jouku Raiha. Dependency inference. In *Proc. of the VLDB Conference*, pages 155–158, Brighton, England, 1987.

[MTV94] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In *KDD-94: AAAI Workshop on Knowledge Discovery in Databases*, pages 181–192, Seattle, Washington, July 1994.

[MTV95] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovering frequent episodes in sequences. In *Proc. of the 1st Int'l Conference on Knowledge Discovery in Databases and Data Mining*, Montreal, Canada, August 1995.

[NRV96] J.P. Nearhos, M.J. Rothman, and M.S. Viveros. Applying data mining techniques to a health insurance information system. In *Proc. of the 22nd Int'l*

Conference on Very Large Databases, Mumbai (Bombay), India, September 1996.

[PCY95] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash based algorithm for mining association rules. In *Proc. of the ACM-SIGMOD Conference on Management of Data*, San Jose, California, May 1995.

[PS91] G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 229–248. AAAI/MIT Press, Menlo Park, CA, 1991.

[PSF91] G. Piatetsky-Shapiro and W. J. Frawley, editors. *Knowledge Discovery in Databases*. AAAI/MIT Press, Menlo Park, CA, 1991.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1993.

[Roy92] M. A. Roytberg. A search for common patterns in many sequences. *Computer Applications in the Biosciences*, 8(1):57–64, 1992.

[SA95] Ramakrishnan Srikant and Rakesh Agrawal. Mining Generalized Association Rules. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, September 1995.

[SA96a] Ramakrishnan Srikant and Rakesh Agrawal. Mining Quantitative Association Rules in Large Relational Tables. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Montreal, Canada, June 1996.

[SA96b] Ramakrishnan Srikant and Rakesh Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996.

[SAD+93] Michael Stonebraker, Rakesh Agrawal, Umeshwar Dayal, Erich J. Neuhold, and Andreas Reuter. The DBMS research at crossroads. In *Proc. of the VLDB Conference*, pages 688–692, Dublin, August 1993.

[SON95] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the VLDB Conference*, Zurich, Switzerland, September 1995.

[ST95] Avi Silberschatz and Alexander Tuzhilin. On Subjective Measures of Interestingness in Knowledge Discovery. In *Proc. of the First Int'l Conference on Knowledge Discovery and Data Mining*, Montreal, Canada, August 1995.

[Tsu90] S. Tsur. Data dredging. *IEEE Data Engineering Bulletin*, 13(4):58–63, December 1990.

[VA89] M. Vingron and P. Argos. A fast and sensitive multiple sequence alignment algorithm. *Computer Applications in the Biosciences*, 5:115–122, 1989.

[Wat89] M. S. Waterman, editor. *Mathematical Methods for DNA Sequence Analysis*. CRC Press, 1989.

[WCM+94] Jason Tsong-Li Wang, Gung-Wei Chirn, Thomas G. Marr, Bruce Shapiro, Dennis Shasha, and Kaizhong Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Minneapolis, May 1994.

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, October 1992.