transnationale
UNIVERSITEIT LIMBURG

School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

Efficient Frequent Pattern Mining

Proefschrift voorgelegd tot het behalen van de graad van
Doctor in de Wetenschappen, richting Informatica,
te verdedigen door

Bart GOETHALS

Promotor : Prof. dr. J. Van den Bussche

2002

OCTORAATSPROEFSCHRIFT

Universiteit Maastricht

LIMBURGS UNIVERSITAIR CENTRUM
PARTNER IN DE UNIVERSITEIT LIMBURG

681.37

transnationale
UNIVERSITEIT LIMBURG

School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

Efficient Frequent Pattern Mining

Proefschrift voorgelegd tot het behalen van de graad van Doctor in de Wetenschappen, richting Informatica, te verdedigen door

Bart GOETHALS

Promotor : Prof. dr. J. Van den Bussche

2002

LIMBURGS UNIVERSITAIR CENTRUM
PARTNER IN DE UNIVERSITEIT LIMBURG

According to the guidelines of the Limburgs Universitair Centrum,
a copy of this publication
has been filed in the Royal Library Albert I, Brussels,as publication
D/2002/2451/46

Acknowledgements

Many people have contributed to the realization of this thesis.

First and foremost, I am grateful to my advisor Jan Van den Bussche for his guidance throughout my doctoral studies and all the time and effort he put in the development of me and my work. The amount of decibels we produced during our vivid discussions, are directly related to the amount of knowledge he passed on to me.

I am much in debt to my office-mate Floris Geerts. His help, encouragement and interest in my research resulted in the work presented in Chapter 4. I also thank the other members of our research group, the department and the administrative staff for creating a stimulating environment.

This thesis further benefitted from pleasant discussions with, among others, Tom Brijs, Toon Calders, Christian Hidber, Paolo Palmerini, Hannu Toivonen and Jean-François Boulicaut.

Also many thanks to my parents, sister, other family members and friends for the support and encouragement they have given me during my long career as a student.

Finally, I am very much in debt for the unconditional support, endless patience and constant encouragement I have received from my companion in life, Eva. Thank you.

You are all part of the "we" used throughout this thesis.

Diepenbeek, December 2002

Contents

1 Introduction

Progress in digital data acquisition, distribution, retrieval and storage technology has resulted in the growth of massive databases. One of the greatest challenges facing organizations and individuals is how to turn their rapidly expanding data collections into accessible, and actionable knowledge.

The attempts to counter these challenges gathered researchers from areas such as statistics, machine learning, databases and probably many more, resulting in a new area of research, called *Data Mining*.

Data mining is usually mentioned in the broader setting of *Knowledge discovery in databases*, or KDD, and is viewed as a single step in a larger process called the *KDD process* [27]. This process includes:

- Developing an understanding of the application domain, the relevant prior knowledge, and the goals of the end-user.
- Selecting the target data set on which discovery is to be performed, and cleaning and transforming this data if necessary.
- Choosing the data mining task, the algorithm, and deciding which models and parameters may be appropriate.
- Performing the actual data mining to extract patterns and models.
- Visualizing, interpreting and consolidating the discovered knowledge.

This process is iterative in the sense that each step can inspire rectifications to preceding steps. It is interactive in the sense that a user must be able to limit the amount of work done by the system to that what he is really interested in.

The data mining step is concerned with the task of automated information extraction from data that might be valuable to the owner of the data store. A working definition of this discipline is the following [43]:

> Data mining is the analysis of (often large) observational data sets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner.

In order to do this analysis, several different types of *tasks* have been identified, corresponding to the objectives of what needs to be analyzed and more importantly, what the intended outcome should describe. These tasks can be categorized as follows [43].

Exploratory Data Analysis The goal is here to explore the data without any clear ideas of what is wanted to be found. Typical techniques include graphical display methods, projection techniques and summarization methods.

Retrieval by Content The user has a specific pattern in mind in advance and is looking for similar patterns in the data set. This task is most commonly used for the retrieval of information from large collections of text or image data. The main challenge here is to define similarity and how to find all similar patterns according to this definition. A well known example is the Google search engine (`http://www.google.com`) of Brin and Page [17], which finds web pages that contain information similar to the set of key-words given by the user.

Descriptive Modelling As the name suggests, descriptive models try to describe all of the collected data. Typical descriptions include several statistical models, clusters and dependency models.

Predictive Modelling Predictive techniques such as classification and regression try to answer questions by examining prior knowledge and answers in order to generalize them for future occasions. An impressive example of classification is the SKICAT system of Fayyad et al. [26], that can perform as well as human experts in classifying stars and galaxies. Their system is in routine use at NASA for automatically cataloging millions of stars and galaxies from digital images of the sky.

Pattern Discovery The aim here is to find local patterns that occur frequently within a database. A lot of algorithms have been studied for several types of patterns, such as sets [3], tree structures [82], graph structures [54, 49], or arbitrary relational structures [23, 32], and association rules over these structures. The most well studied type of patterns are sets of items that occur frequently together in transaction databases such as market basket logs of retail stores. An interesting application of association rules is in cross-selling applications in a retail context [15].

During the past few years, several very good books and surveys have been published on these topics, to which we refer the interested reader for more information [43, 39, 47].

In this thesis we focus on the Frequent Pattern Discovery task and how it can be efficiently solved in the specific context of itemsets and association rules.

The original motivation for searching association rules came from the need to analyze so called supermarket transaction data, that is, to examine customer behavior in terms of the purchased products. Association rules describe how often items are purchased together. For example, an association rule "beer $\Rightarrow$ chips (80%)" states that four out of five customers that bought beer also bought chips. Such rules can be useful for decisions concerning product pricing, promotions, store layout and many others.

Since their introduction in 1993 by Argawal et al. [3], the frequent itemset and association rule mining problems have received a great deal of attention. Within the past decade, hundreds of research papers have been published presenting new algorithms or improvements on existing algorithms to solve these mining problems more efficiently.

In Chapter 2, we explain the frequent itemset and association rule mining problems. We present an in depth analysis of the most influential algorithms which made significant contributions to several efficiency issues of these mining problems.

Since the data mining process is an essentially interactive process, it motivated the idea of a "data mining query language" [37, 38, 47, 48, 60]. A data mining query language allows the user to ask for specific subsets of association rules by specifying several constraints within each query.

In Chapter 3, we present new techniques in order to efficiently find all frequent patterns that satisfy the constraints given by the user. For that purpose, we study a class of constraints on associations to be generated, which should be expressible in any reasonable data mining query language: Boolean combinations of atomic conditions, where an atomic condition can either specify that a certain item occurs in the antecedent of the rule or the consequent of the rule. Efficiently supporting data mining query language environments is a challenging task. Towards this goal, we present and compare three approaches. In the first extreme, the *integrated querying* approach, every individual data mining query will be answered by running an adaptation of the mining algorithm in which the constraints on the rules and sets to be generated are directly incorporated. The second extreme, the *post-processing* approach, first mines as much associations as possible, by performing one major, global mining operation. After this relatively expensive operation, the actual data mining queries issued by the user then amount to standard lookups in the set of materialized associations. A third approach, the *incremental querying* approach, combines the advantages of both previous approaches.

In Chapter 4, we describe a combinatorial problem which is implicit to a wide range of frequent pattern mining algorithms. Our contribution is to solve this problem by providing hard and tight combinatorial upper bounds on the amount of work that a typical range of frequent itemset algorithms will need to perform. By computing our upper bounds, we have at all times an airtight guarantee of what is still to come, on which then various optimization decisions can be based, depending on the specific algorithm that is used.

2

Survey on Frequent Pattern Mining

Frequent itemsets play an essential role in many data mining tasks that try to find interesting patterns from databases, such as association rules, correlations, sequences, episodes, classifiers, clusters and many more of which the mining of association rules is one of the most popular problems. The original motivation for searching association rules came from the need to analyze so called supermarket transaction data, that is, to examine customer behavior in terms of the purchased products. Association rules describe how often items are purchased together. For example, an association rule "beer $\Rightarrow$ chips (80%)" states that four out of five customers that bought beer also bought chips. Such rules can be useful for decisions concerning product pricing, promotions, store layout and many others.

Since their introduction in 1993 by Argawal et al. [3], the frequent itemset and association rule mining problems have received a great deal of attention. Within the past decade, hundreds of research papers have been published presenting new algorithms or improvements on existing algorithms to solve these mining problems more efficiently.

In this chapter, we explain the basic frequent itemset and association rule mining problems. We describe the main techniques used to solve these problems and give a comprehensive survey of the most influential algorithms that were proposed during the last decade.

2.1 Problem Description

Let $\mathcal{I}$ be a set of items. A set $X = \{i_1, \ldots, i_k\} \subseteq \mathcal{I}$ is called an *itemset*, or a *k-itemset* if it contains k items.

A *transaction* over $\mathcal{I}$ is a couple $T = (tid, I)$ where *tid* is the transaction identifier and I is an itemset. A transaction $T = (tid, I)$ is said to *support* an itemset $X \subseteq \mathcal{I}$, if $X \subseteq I$.

A *transaction database* $\mathcal{D}$ over $\mathcal{I}$ is a set of transactions over $\mathcal{I}$. We omit $\mathcal{I}$ whenever it is clear from the context.

The *cover* of an itemset X in $\mathcal{D}$ consists of the set of transaction identifiers of transactions in $\mathcal{D}$ that support X:

$$cover(X, \mathcal{D}) := \{tid \mid (tid, I) \in \mathcal{D}, X \subseteq I\}.$$

The *support* of an itemset X in $\mathcal{D}$ is the number of transactions in the cover of X in $\mathcal{D}$:

$$support(X, \mathcal{D}) := |cover(X, \mathcal{D})|.$$

The *frequency* of an itemset X in $\mathcal{D}$ is the probability of X occurring in a transaction $T \in \mathcal{D}$:

$$frequency(X, \mathcal{D}) := P(X) = \frac{support(X, \mathcal{D})}{|\mathcal{D}|}.$$

Note that $|\mathcal{D}| = support(\{\}, \mathcal{D})$. We omit $\mathcal{D}$ whenever it is clear from the context.

An itemset is called *frequent* if its support is no less than a given absolute *minimal support threshold* σ_{abs}, with $0 \leq \sigma_{abs} \leq |\mathcal{D}|$. When working with frequencies of itemsets instead of their supports, we use a relative *minimal frequency threshold* σ_{rel}, with $0 \leq \sigma_{rel} \leq 1$. Obviously, $\sigma_{abs} = \lceil \sigma_{rel} \cdot |\mathcal{D}| \rceil$. In this thesis, we will only work with the absolute minimal support threshold for itemsets and omit the subscript *abs* unless explicitly stated otherwise.

Definition 2.1. Let $\mathcal{D}$ be a transaction database over a set of items $\mathcal{I}$, and σ a minimal support threshold. The collection of frequent itemsets in $\mathcal{D}$ with respect to σ is denoted by

$$\mathcal{F}(\mathcal{D}, \sigma) := \{X \subseteq \mathcal{I} \mid support(X, \mathcal{D}) \geq \sigma\},$$

or simply $\mathcal{F}$ if $\mathcal{D}$ and σ are clear from the context.

Problem 2.1. (Itemset Mining) *Given a set of items $\mathcal{I}$, a transaction database $\mathcal{D}$ over $\mathcal{I}$, and minimal support threshold σ, find $\mathcal{F}(\mathcal{D}, \sigma)$.*

In practice we are not only interested in the set of itemsets $\mathcal{F}$, but also in the actual supports of these itemsets.

An *association rule* is an expression of the form $X \Rightarrow Y$, where X and Y are itemsets, and $X \cap Y = \{\}$. Such a rule expresses the association that if a transaction contains all items in X, then that transaction also contains all items in Y. X is called the *body* or *antecedent*, and Y is called the *head* or *consequent* of the rule.

The support of an association rule $X \Rightarrow Y$ in $\mathcal{D}$, is the support of $X \cup Y$ in $\mathcal{D}$, and similarly, the frequency of the rule is the frequency of $X \cup Y$. An association rule is called *frequent* if its support (frequency) exceeds a given minimal support (frequency) threshold σ_{abs} (σ_{rel}). Again, we will only work with the absolute minimal support threshold for association rules and omit the subscript *abs* unless explicitly stated otherwise.

The *confidence* or *accuracy* of an association rule $X \Rightarrow Y$ in $\mathcal{D}$ is the conditional probability of having Y contained in a transaction, given that X is contained in that transaction:

$$\mathit{confidence}(X \Rightarrow Y, \mathcal{D}) := P(Y|X) = \frac{\mathit{support}(X \cup Y, \mathcal{D})}{\mathit{support}(X, \mathcal{D})}.$$

The rule is called *confident* if $P(Y|X)$ exceeds a given *minimal confidence threshold* γ, with $0 \leq \gamma \leq 1$.

Definition 2.2. Let $\mathcal{D}$ be a transaction database over a set of items $\mathcal{I}$, σ a minimal support threshold, and γ a minimal confidence threshold. The collection of frequent and confident association rules with respect to σ and γ is denoted by

$$\begin{aligned}\mathcal{R}(\mathcal{D}, \sigma, \gamma) := \{X \Rightarrow Y \mid X, Y \subseteq \mathcal{I}, X \cap Y = \{\},\\ X \cup Y \in \mathcal{F}(\mathcal{D}, \sigma), \mathit{confidence}(X \Rightarrow Y, \mathcal{D}) \geq \gamma\},\end{aligned}$$

or simply $\mathcal{R}$ if $\mathcal{D}, \sigma$ and γ are clear from the context.

Problem 2.2. (Association Rule Mining) *Given a set of items $\mathcal{I}$, a transaction database $\mathcal{D}$ over $\mathcal{I}$, and minimal support and confidence thresholds σ and γ, find $\mathcal{R}(\mathcal{D}, \sigma, \gamma)$.*

Besides the set of all association rules, we are also interested in the support and confidence of each of these rules.

Note that the Itemset Mining problem is actually a special case of the Association Rule Mining problem. Indeed, if we are given the support and confidence thresholds σ and γ, then every frequent itemset X also represents the trivial rule $X \Rightarrow \{\}$ which holds with 100% confidence. Obviously, the support of the rule equals the support of X. Also note that for every itemset

I, all rules $X \Rightarrow Y$, with $X \cup Y = I$, hold with at least σ_{rel} confidence. Hence, the minimal confidence threshold must be higher than the minimal frequency threshold to be of any effect.

Example 2.1. Consider the database shown in Table 2.1 over the set of items

$$\mathcal{I} = \{\text{beer}, \text{chips}, \text{pizza}, \text{wine}\}.$$

tid	X
100	{beer, chips, wine}
200	{beer, chips}
300	{pizza, wine}
400	{chips, pizza}

Table 2.1: An example transaction database $\mathcal{D}$.

Table 2.2 shows all frequent itemsets in $\mathcal{D}$ with respect to a minimal support threshold of 1. Table 2.3 shows all frequent and confident association rules with a support threshold of 1 and a confidence threshold of 50%.

The first algorithm proposed to solve the association rule mining problem was divided into two phases [3]. In the first phase, all frequent itemsets are generated (or all frequent rules of the form $X \Rightarrow \{\}$). The second phase consists of the generation of all frequent and confident association rules. Almost all association rule mining algorithms comply with this two phased strategy. In the following two sections, we discuss these two phases in further detail. Nevertheless, there exist a successful algorithm, called MagnumOpus, that uses another strategy to immediately generate a large subset of all association rules [79]. We will not discuss this algorithm here, as the main focus of this survey is on frequent itemset mining of which association rules are a natural extension.

Next to the support and confidence measures, a lot of other interestingness measures have been proposed in order to get better or more interesting association rules. Recently, Tan et al. presented an overview of various measures proposed in statistics, machine learning and data mining literature [76]. In this survey, we only consider algorithms within the support-confidence framework as presented before.

Itemset	Cover	Support	Frequency
$\{\}$	$\{100, 200, 300, 400\}$	4	100%
{beer}	{100,200}	2	50%
{chips}	{100,200,400}	3	75%
{pizza}	{300,400}	2	50%
{wine}	{100,300}	2	50%
{beer, chips}	{100,200}	2	50%
{beer, wine}	{100}	1	25%
{chips, pizza}	{400}	1	25%
{chips, wine}	{100}	1	25%
{pizza, wine}	{300}	1	25%
{beer, chips, wine}	{100}	1	25%

Table 2.2: Itemsets and their support in $\mathcal{D}$.

Rule	Support	Frequency	Confidence
{beer} $\Rightarrow$ {chips}	2	50%	100%
{beer} $\Rightarrow$ {wine}	1	25%	50%
{chips} $\Rightarrow$ {beer}	2	50%	66%
{pizza} $\Rightarrow$ {chips}	1	25%	50%
{pizza} $\Rightarrow$ {wine}	1	25%	50%
{wine} $\Rightarrow$ {beer}	1	25%	50%
{wine} $\Rightarrow$ {chips}	1	25%	50%
{wine} $\Rightarrow$ {pizza}	1	25%	50%
{beer, chips} $\Rightarrow$ {wine}	1	25%	50%
{beer, wine} $\Rightarrow$ {chips}	1	25%	100%
{chips, wine} $\Rightarrow$ {beer}	1	25%	100%
{beer} $\Rightarrow$ {chips, wine}	1	25%	50%
{wine} $\Rightarrow$ {beer, chips}	1	25%	50%

Table 2.3: Association rules and their support and confidence in $\mathcal{D}$.

2.2 Itemset Mining

The task of discovering all frequent itemsets is quite challenging. The search space is exponential in the number of items occurring in the database. The support threshold limits the output to a hopefully reasonable subspace. Also, such databases could be massive, containing millions of transactions, making support counting a tough problem. In this section, we will analyze these two aspects into further detail.

2.2.1 Search Space

The search space of all itemsets contains exactly $2^{|\mathcal{I}|}$ different itemsets. If $\mathcal{I}$ is large enough, then the naive approach to generate and count the supports of all itemsets over the database can't be achieved within a reasonable period of time. For example, in many applications, $\mathcal{I}$ contains thousands of items, and then, the number of itemsets is more than the number of atoms in the universe ($\approx 10^{79}$).

Instead, we could generate only those itemsets that occur at least once in the transaction database. More specifically, we generate all subsets of all transactions in the database. Of course, for large transactions, this number could still be too large. Therefore, as an optimization, we could generate only those subsets of at most a given maximum size. This technique has been studied by Amir et al. [8] and has proven to pay off for very sparse transaction databases. Nevertheless, for large or dense databases, this algorithm suffers from massive memory requirements. Therefore, several solutions have been proposed to perform a more directed search through the search space.

During such a search, several collections of *candidate itemsets* are generated and and their supports computed until all frequent itemsets have been generated. Formally,

Definition 2.3. (Candidate itemset) Given a transaction database $\mathcal{D}$, a minimal support threshold σ, and an algorithm that computes $\mathcal{F}(\mathcal{D}, \sigma)$, an itemset I is called a *candidate* if that algorithm evaluates whether I is frequent or not.

Obviously, the size of a collection of candidate itemsets may not exceed the size of available main memory. Moreover, it is important to generate as few candidate itemsets as possible, since computing the supports of a collection of itemsets is a time consuming procedure. In the best case, only the frequent itemsets are generated and counted. Unfortunately, this ideal is impossible in general, which will be shown later in this section. The main underlying property exploited by most algorithms is that support is monotone decreasing with respect to extension of an itemset.

Proposition 2.1. (Support monotonicity) *Given a transaction database $\mathcal{D}$ over $\mathcal{I}$, let $X, Y \subseteq \mathcal{I}$ be two itemsets. Then,*

$$X \subseteq Y \Rightarrow \mathit{support}(Y) \leq \mathit{support}(X).$$

Proof. This follows immediately from

$$\mathit{cover}(Y) \subseteq \mathit{cover}(X).$$

□

Hence, if an itemset is infrequent, all of its supersets must be infrequent. In the literature, this monotonicity property is also called the downward closure property, since the set of frequent itemsets is closed with respect to set inclusion.

The search space of all itemsets can be represented by a *subset-lattice*, with the empty itemset at the bottom and the set containing all items at the top. The collection of frequent itemsets $\mathcal{F}(\mathcal{D}, \sigma)$ can be represented by the collection of *maximal* frequent itemsets, or the collection of *minimal* infrequent itemsets, with respect to set inclusion. For this purpose, Mannila and Toivonen introduced the notion of the *Border* of a downward closed collection of itemsets [58].

Definition 2.4. (Border) Let $\mathcal{F}$ be a downward closed collection of subsets of $\mathcal{I}$. The *Border* $\mathcal{B}d(\mathcal{F})$ consists of those itemsets $X \subseteq \mathcal{I}$ such that all subsets of X are in $\mathcal{F}$, and no superset of X is in $\mathcal{F}$:

$$\mathcal{B}d(\mathcal{F}) := \{X \subseteq \mathcal{I} \mid \forall Y \subset X : Y \in \mathcal{F}\}.$$

Those itemsets in $\mathcal{B}d(\mathcal{F})$ that are in $\mathcal{F}$ are called the *positive border* $\mathcal{B}d^+(\mathcal{F})$:

$$\mathcal{B}d^+(\mathcal{F}) := \{X \subseteq \mathcal{I} \mid \forall Y \subseteq X : Y \in \mathcal{F}\},$$

and those itemsets in $\mathcal{B}d(\mathcal{F})$ that are not in $\mathcal{F}$ are called the *negative border* $\mathcal{B}d^-(\mathcal{F})$:

$$\mathcal{B}d^-(\mathcal{F}) := \{X \subseteq \mathcal{I} \mid \forall Y \supseteq X : Y \notin \mathcal{F}\}.$$

The lattice for the frequent itemsets from Example 2.1, together with its borders, is shown in Figure 2.1.

Several efficient algorithms have been proposed to find only the positive border of all frequent itemsets, but if we want to know the supports of all itemsets in the collection, we still need to count them. Therefore, these algorithms are not discussed in this survey. From a theoretical point of view, the border gives some interesting insights into the frequent itemset mining problem, and still poses several interesting open problems [35, 58, 57].

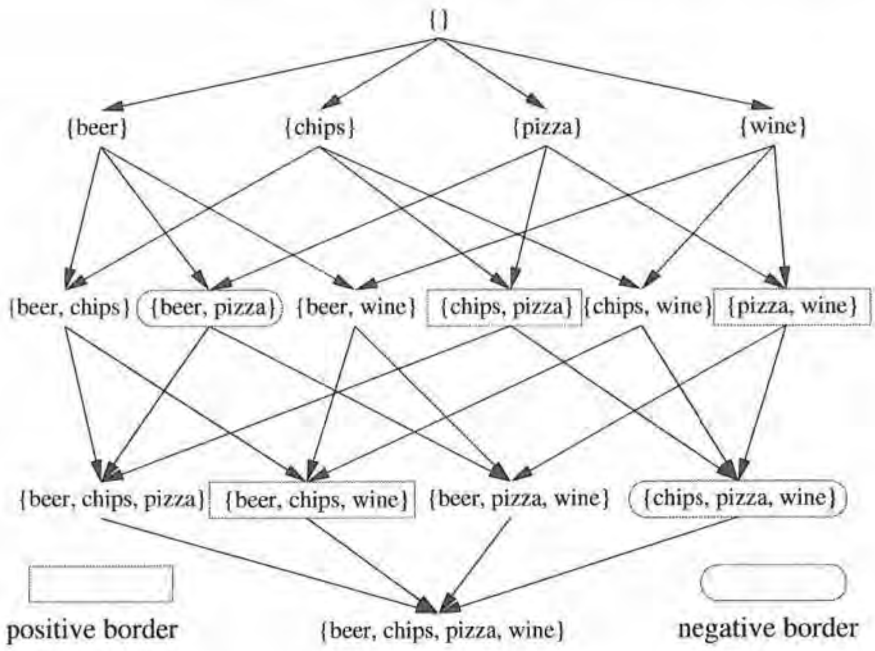


Figure 2.1: The lattice for the itemsets of Example 2.1 and its border.

Theorem 2.2. *[58] Let $\mathcal{D}$ be a transaction database over $\mathcal{I}$, and σ a minimal support threshold. Finding the collection $\mathcal{F}(\mathcal{D},\sigma)$ requires that at least all itemsets in the negative border $\mathcal{B}d^-(\mathcal{F})$ are evaluated.*

Note that the number of itemsets in the positive or negative border of any given downward closed collection of itemsets over $\mathcal{I}$ can still be large, but it is bounded by $\binom{|\mathcal{I}|}{\lfloor|\mathcal{I}|/2\rfloor}$. In combinatorics, this upper bound is well known as Sperner's theorem.

If the number of frequent itemsets for a given database is large, it could become infeasible to generate them all. Moreover, if the transaction database is dense, or the minimal support threshold is set too low, then there could exist a lot of very large frequent itemsets, which would make sending them all to the output infeasible to begin with. Indeed, a frequent itemset of size k includes the existence of at least $2^k - 1$ other frequent itemsets, i.e. all of its subsets. To overcome this problem, several proposals have been made to generate only a concise representation of all frequent itemsets for a given transaction database such that, if necessary, the support of a frequent itemset not in that representation can be efficiently computed or estimated without accessing the database [56, 66, 14, 18, 19]. These techniques are based on the observation that the support of some frequent itemsets can be deduced from the supports of other itemsets. We will not discuss these algorithms in this survey because all frequent itemsets need to be considered to generate association rules anyway. Nevertheless, several of these techniques can still be used to improve the performance of the algorithms that do generate all frequent itemsets, as will be explained later in this chapter.

2.2.2 Database

To compute the supports of a collection of itemsets, we need to access the database. Since such databases tend to be very large, it is not always possible to store them into main memory.

An important consideration in most algorithms is the representation of the transaction database. Conceptually, such a database can be represented by a binary two-dimensional matrix in which every row represents an individual transaction and the columns represent the items in $\mathcal{I}$. Such a matrix can be implemented in several ways. The most commonly used layout is the *horizontal data layout*. That is, each transaction has a transaction identifier and a list of items occurring in that transaction. Another commonly used layout is the *vertical data layout*, in which the database consists of a set of items, each followed by its cover [70, 80]. Table 2.4 shows both layouts for the database from Example 2.1. Note that for both layouts, it is also possible to use the exact bit-strings from the binary matrix [71, 64]. Also a combination of both layouts can be used, as will be explained later in this chapter.

	beer	wine	chips	pizza
100	1	1	1	0
200	1	0	1	0
300	0	1	0	1
400	0	0	1	1

	beer	wine	chips	pizza
100	1	1	1	0
200	1	0	1	0
300	0	1	0	1
400	0	0	1	1

Table 2.4: Horizontal and Vertical database layout of $\mathcal{D}$.

To count the support of an itemset X using the horizontal database layout, we need to scan the database completely, and test for every transaction T whether $X \subseteq T$. Of course, this can be done for a large collection of itemsets at once. An important misconception about frequent pattern mining is that scanning the database is a very I/O intensive operation. However, in most cases, this is not the major cost of such counting steps. Instead, updating the supports of all candidate itemsets contained in a transaction consumes considerably more time than reading that transaction from a file or from a database cursor. Indeed, for each transaction, we need to check for every candidate itemset whether it is included in that transaction, or similarly, we need to check for every subset of that transaction whether it is in the set of candidate itemsets. On the other hand, the number of transactions in a database is often correlated to the maximal size of a transaction in the database. As such, the number of transactions does have an influence on the time needed for support counting, but it is by no means the dictating factor.

The vertical database layout has the major advantage that the support of an itemset X can be easily computed by simply intersecting the covers of any two subsets $Y, Z \subseteq X$, such that $Y \cup Z = X$. However, given a set of candidate itemsets, this technique requires that the covers of a lot of sets are available in main memory, which is of course not always possible. Indeed, the covers of all singleton itemsets already represent the complete database.

2.3 Association Rule Mining

The search space of all association rules contains exactly $3^{|\mathcal{I}|}$ different rules. However, given all frequent itemsets, this search space immediately shrinks tremendously. Indeed, for every frequent itemset I, there exists at most $2^{|I|}$ rules of the form $X \Rightarrow Y$, such that $X \cup Y = I$. Again, in order to efficiently traverse this search space, sets of candidate association rules are iteratively generated and evaluated, until all frequent and confident association rules are found. The underlying technique to do this, is based on a similar monotonicity property as was used for mining all frequent itemsets.

Proposition 2.3. (Confidence monotonicity) *Let $X, Y, Z \subseteq \mathcal{I}$ be three itemsets, such that $X \cap Y = \{\}$. Then,*

$$\mathit{confidence}(X \setminus Z \Rightarrow Y \cup Z) \leq \mathit{confidence}(X \Rightarrow Y).$$

Proof. Since $X \cup Y \subseteq X \cup Y \cup Z$, and $X \setminus Z \subseteq X$, we have

$$\frac{\mathit{support}(X \cup Y \cup Z)}{\mathit{support}(X \setminus Z)} \leq \frac{\mathit{support}(X \cup Y)}{\mathit{support}(X)}.$$

□

In other words, confidence is monotone decreasing with respect to extension of the head of a rule. If an item in the extension is included in the body, then it is removed from the body of that rule. Hence, if a certain head of an association rule over an itemset I causes the rule to be unconfident, all of the head's supersets must result in unconfident rules.

As already mentioned in the problem description, the association rule mining problem is actually more general than the frequent itemset mining problem in the sense that every itemset I can be represented by the rule $I \Rightarrow \{\}$, which holds with 100% confidence, given its support is not zero. On the other hand,

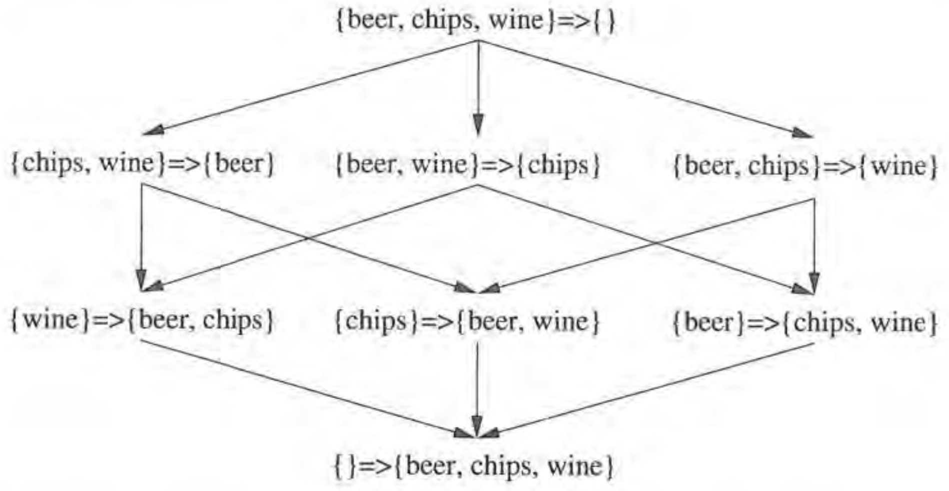


Figure 2.2: An example of a lattice representing a collection of association rules for {beer, chips, wine}.

for every itemset I, the frequency of the rule $\{\} \Rightarrow I$ equals its confidence. Hence, if the frequency of I is above the minimal confidence threshold, then so are all other association rules that can be constructed from I.

For a given frequent itemset I, the search space of all possible association rules $X \Rightarrow Y$, such that $X \cup Y = I$, can be represented by a subset-lattice with respect to the head of a rule, with the rule with an empty head at the bottom and the rule with all items in the head at the top. Figure 2.2 shows such a lattice for the itemset {beer, chips, wine}, which was found to be frequent on the artificial data set used in Example 2.4.

Given all frequent itemsets and their supports, the computation of all frequent and confident association rules becomes relatively straightforward. Indeed, to compute the confidence of an association rule $X \Rightarrow Y$, with $X \cup Y = I$, we only need to find the supports of I and X, which can be easily retrieved from the collection of frequent itemsets.

2.4 Example Data Sets

For all experiments we performed in this thesis, we used four data sets with different characteristics. We have experimented using three real data sets, of which two are publicly available, and one synthetic data set generated by the program provided by the Quest research group at IBM Almaden [5]. The mushroom data set contains characteristics of various species of mushrooms, and was originally obtained from the UCI repository of machine learn-

Data set	#Items	#Transactions	$min\|T\|$	$max\|T\|$	$avg\|T\|$
T40I10D100K	942	100 000	4	77	39
mushroom	119	8 124	23	23	23
BMS-Webview-1	497	59 602	1	267	2
basket	13 103	41 373	1	52	9

Table 2.5: Data Set Characteristics.

ing databases [11]. The BMS-WebView-1 data set contains several months worth of clickstream data from an e-commerce web site, and is made publicly available by Blue Martini Software [52]. The basket data set contains transactions from a Belgian retail store, but can unfortunately not be made publicly available. Table 2.5 shows the number of items and the number of transactions in each data set, and the minimum, maximum and average length of the transactions.

Additionally, Table 2.6 shows for each data set the lowest minimal support threshold that was used in our experiments, the number of frequent items and itemsets, and the size of the longest frequent itemset that was found.

Data set	σ	$\|\mathcal{F}_1\|$	$\|\mathcal{F}\|$	$\max\{k \mid \|\mathcal{F}_k\| > 0\}$
T40I10D100K	700	804	550 126	18
mushroom	600	60	945 309	16
BMS-Webview-1	36	368	461 521	15
basket	5	8 051	285 758	11

Table 2.6: Data Set Characteristics.

2.5 The Apriori Algorithm

The first algorithm to generate all frequent itemsets and confident association rules was the AIS algorithm by Agrawal et al. [3], which was given together with the introduction of this mining problem. Shortly after that, the algorithm was improved and renamed Apriori by Agrawal et al., by exploiting the monotonicity property of the support of itemsets and the confidence of association rules [6, 73]. The same technique was independently proposed by Mannila et al. [59]. Both works were cumulated afterwards [4].

2.5.1 Itemset Mining

For the remainder of this thesis, we assume for simplicity that items in transactions and itemsets are kept sorted in their lexicographic order unless stated otherwise.

The itemset mining phase of the Apriori algorithm is given in Algorithm 1. We use the notation $X[i]$, to represent the ith item in X. The *k-prefix* of an itemset X is the k-itemset $\{X[1], \ldots, X[k]\}$.

Algorithm 1 Apriori - Itemset mining

```
Input: $\mathcal{D}, \sigma$
Output: $\mathcal{F}(\mathcal{D}, \sigma)$
 1: $C_1 := \{\{i\} \mid i \in \mathcal{I}\}$
 2: $k := 1$
 3: while $C_k \neq \{\}$ do
 4:   // Compute the supports of all candidate itemsets
 5:   for all transactions $(tid, I) \in \mathcal{D}$ do
 6:     for all candidate itemsets $X \in C_k$ do
 7:       if $X \subseteq I$ then
 8:         $X.support$++
 9:       end if
10:     end for
11:   end for
12:   // Extract all frequent itemsets
13:   $\mathcal{F}_k := \{X \mid X.support \geq \sigma\}$
14:   // Generate new candidate itemsets
15:   for all $X, Y \in \mathcal{F}_k, X[i] = Y[i]$ for $1 \leq i \leq k-1$, and $X[k] < Y[k]$ do
16:     $I = X \cup \{Y[k]\}$
17:     if $\forall J \subset I, |J| = k : J \in \mathcal{F}_k$ then
18:       $C_{k+1} := C_{k+1} \cup I$
19:     end if
20:   end for
21:   $k$++
22: end while
```

The algorithm performs a breadth-first search through the search space of all itemsets by iteratively generating candidate itemsets C_{k+1} of size $k+1$, starting with $k = 0$ (line 1). An itemset is a candidate if all of its subsets are known to be frequent. More specifically, C_1 consists of all items in $\mathcal{I}$, and at a certain level k, all itemsets of size $k+1$ in $\mathcal{B}d^-(\mathcal{F}_k)$ are generated. This is done in two steps. First, in the *join* step, $\mathcal{F}_k$ is joined with itself. The union $X \cup Y$ of itemsets $X, Y \in \mathcal{F}_k$ is generated if they have the same $k-1$-prefix (lines 20–21). In the *prune* step, $X \cup Y$ is only inserted into C_{k+1} if all of its k-subsets occur in $\mathcal{F}_k$ (lines 22–24).

To count the supports of all candidate k-itemsets, the database, which retains on secondary storage in the horizontal database layout, is scanned one transaction at a time, and the supports of all candidate itemsets that are included in that transaction are incremented (lines 6–12). All itemsets that turn out to be frequent are inserted into $\mathcal{F}_k$ (lines 14–18).

Note that in this algorithm, the set of all itemsets that were ever generated as candidate itemsets, but turned out to be infrequent, is exactly $\mathcal{B}d^-(\mathcal{F})$.

If the number of candidate $k+1$-itemsets is too large to retain into main memory, the candidate generation procedure stops and the supports of all generated candidates is computed as if nothing happened. But then, in the next iteration, instead of generating candidate itemsets of size $k+2$, the remainder of all candidate $k+1$-itemsets is generated and counted repeatedly until all frequent itemsets of size $k+1$ are generated.

2.5.2 Association Rule Mining

Given all frequent itemsets, we can now generate all frequent and confident association rules. The algorithm is very similar to the frequent itemset mining algorithm and is given in Algorithm 2.

Algorithm 2 Apriori - Association Rule mining

```
Input: $\mathcal{D}, \sigma, \gamma$
Output: $\mathcal{R}(\mathcal{D}, \sigma, \gamma)$
 1: Compute $\mathcal{F}(\mathcal{D}, \sigma)$
 2: $\mathcal{R} := \{\}$
 3: for all $I \in \mathcal{F}$ do
 4:   $\mathcal{R} := \mathcal{R} \cup I \Rightarrow \{\}$
 5:   $C_1 := \{\{i\} \mid i \in I\}$;
 6:   $k := 1$;
 7:   while $C_k \neq \{\}$ do
 8:     // Extract all heads of confident association rules
 9:     $H_k := \{X \in C_k \mid \mathit{confidence}(I \setminus X \Rightarrow X, \mathcal{D}) \geq \gamma\}$
10:     // Generate new candidate heads
11:     for all $X, Y \in H_k, X[i] = Y[i]$ for $1 \leq i \leq k-1$, and $X[k] < Y[k]$ do
12:       $I = X \cup \{Y[k]\}$
13:       if $\forall J \subset I, |J| = k : J \in H_k$ then
14:         $C_{k+1} := C_{k+1} \cup I$
15:       end if
16:     end for
17:     $k$++
18:   end while
19:   // Cumulate all association rules
20:   $\mathcal{R} := \mathcal{R} \cup \{I \setminus X \Rightarrow X \mid X \in H_1 \cup \cdots \cup H_k\}$
21: end for
```

First, all frequent itemsets are generated using Algorithm 1. Then, every frequent itemset I is divided into a candidate head Y and a body $X = I \setminus Y$. This process starts with $Y = \{\}$, resulting in the rule $I \Rightarrow \{\}$, which always holds with 100% confidence (line 4). After that, the algorithm iteratively generates candidate heads C_{k+1} of size $k+1$, starting with $k = 0$ (line 5). A head is a candidate if all of its subsets are known to represent confident rules. This candidate head generation process is exactly like the candidate itemset generation in Algorithm 1 (lines 11–16). To compute the confidence of a candidate head Y, the support of I and X is retrieved from $\mathcal{F}$. All heads that result in confident rules are inserted into H_k (line 9). In the end, all confident rules are inserted into $\mathcal{R}$ (line 20).

It can be seen that this algorithm does not fully exploit the monotonicity of confidence. Given an itemset I and a candidate head Y, representing the rule $I \setminus Y \Rightarrow Y$, the algorithm checks for all $Y' \subset Y$ whether the rule $I \setminus Y' \Rightarrow Y'$ is confident, but not whether the rule $I \setminus Y \Rightarrow Y'$ is confident. Nevertheless, this is perfectly possible if all rules are generated from an itemset I, only if all rules are already generated for all itemsets $I' \subset I$.

However, exploiting monotonicity as much as possible is not always the best solution. Since computing the confidence of a rule only requires the lookup of the support of at most 2 itemsets, it might even be better not to exploit the confidence monotonicity at all and simply remove the prune step from the candidate generation process, i.e., remove lines 13 and 15. Of course, this depends on the efficiency of finding the support of an itemset or a head in the used data structures.

Luckily, if the number of frequent and confident association rules is not too large, then the time needed to find all such rules consists mainly of the time that was needed to find all frequent sets.

Since the proposal of this algorithm for the association rule generation phase, no significant optimizations have been proposed anymore and almost all research has been focused on the frequent itemset generation phase.

2.5.3 Data Structures

The candidate generation and the support counting processes require an efficient data structure in which all candidate itemsets are stored since it is important to efficiently find the itemsets that are contained in a transaction or in another itemset.

Hash-tree

In order to efficiently find all k-subsets of a potential candidate itemset, all frequent itemsets in $\mathcal{F}_k$ are stored in a hash table.

Candidate itemsets are stored in a hash-tree [4]. 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 leaves.

When we add a k-itemset X during the candidate generation process, 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 $X[d]$ item of the itemset, and following the pointer in the corresponding bucket. All nodes are initially created as leaf nodes. When the number of itemsets in a leaf node at depth d exceeds a specified threshold, the leaf node is converted into an interior node, only if $k > d$.

In order to find the candidate-itemsets that are contained in a transaction T, we start from the root node. If we are at a leaf, we find which of the itemsets in the leaf are contained in T and increment their support. If we are at an 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.

Trie

Another data structure that is commonly used is a trie (or prefix-tree) [8, 13, 16, 9]. In a trie, every k-itemset has a node associated with it, as does its $k-1$-prefix. The empty itemset is the root node. All the 1-itemsets are attached to the root node, and their branches are labelled by the item they represent. Every other k-itemset is attached to its $k-1$-prefix. Every node stores the last item in the itemset it represents, its support, and its branches. The branches of a node can be implemented using several data structures such as a hash table, a binary search tree or a vector.

At a certain iteration k, all candidate k-itemsets are stored at depth k in the trie. In order to find the candidate-itemsets that are contained in a transaction T, we start at the root node. To process a transaction for a node of the trie, (1) follow the branch corresponding to the first item in the transaction and process the remainder of the transaction recursively for that branch, and (2) discard the first item of the transaction and process it recursively for the node itself. This procedure can still be optimized, as is described in [13].

Also the join step of the candidate generation procedure becomes very simple using a trie, since all itemsets of size k with the same $k-1$-prefix are represented by the branches of the same node (that node represents the $k-1$-prefix). Indeed, to generate all candidate itemsets with $k-1$-prefix X, we simply copy all siblings of the node that represents X as branches of that node.

Moreover, we can try to minimize the number of such siblings by reordering the items in the database in support ascending order [13, 16, 9]. Using this heuristic, we reduce the number of itemsets that is generated during the join step, and hence, we implicitly reduce the number of times the prune step needs to be performed. Also, to find the node representing a specific k-itemset in the trie, we have to perform k searches within a set of branches. Obviously, the performance of such a search can be improved when these sets are kept as small as possible.

An in depth study on the implementation details of a trie for Apriori can be found in [13].

All implementations of all frequent itemsets mining algorithms presented in this thesis are implemented using this trie data structure.

2.5.4 Optimizations

A lot of other algorithms proposed after the introduction of Apriori retain the same general structure, adding several techniques to optimize certain steps within the algorithm. Since the performance of the Apriori algorithm is almost completely dictated by its support counting procedure, most research has focused on that aspect of the Apriori algorithm. As already mentioned before, the performance of this procedure is mainly dependent on the number of candidate itemsets that occur in each transaction.

AprioriTid, AprioriHybrid

Together with the proposal of the Apriori algorithm, Agrawal et al. [6, 4] proposed two other algorithms, AprioriTid and AprioriHybrid. The AprioriTid algorithm reduces the time needed for the support counting procedure by replacing every transaction in the database by the set of candidate itemsets that occur in that transaction. This is done repeatedly at every iteration k. The adapted transaction database is denoted by $\overline{C}_k$. The algorithm is given in Algorithm 3.

More implementation details of this algorithm can be found in [7]. Although the AprioriTid algorithm is much faster in later iterations, it performs much slower than Apriori in early iterations. This is mainly due to the additional overhead that is created when $\overline{C}_k$ does not fit into main memory and has to be written to disk. If a transaction does not contain any candidate k-itemsets, then $\overline{C}_k$ will not have an entry for this transaction. Hence, the number of entries in $\overline{C}_k$ may be smaller than the number of transactions in the database, especially at later iterations of the algorithm. Additionally, at later iterations, each entry may be smaller than the corresponding transaction because very few candidates may be contained in the transaction. However, in

Algorithm 3 AprioriTid

```
Input: $\mathcal{D}, \sigma$
Output: $\mathcal{F}(\mathcal{D}, \sigma)$
 1: Compute $\mathcal{F}_1$ of all frequent items
 2: $\overline{C}_1 := \mathcal{D}$ (with all items not in $\mathcal{F}_1$ removed)
 3: k := 2
 4: while $\mathcal{F}_{k-1} \neq \{\}$ do
 5:   Compute $C_k$ of all candidate $k$-itemsets
 6:   $\overline{C}_k := \{\}$
 7:   // Compute the supports of all candidate itemsets
 8:   for all transactions $(tid, T) \in \overline{C}_k$ do
 9:     $C_T := \{\}$
10:     for all $X \in C_k$ do
11:       if $\{X[1], \ldots, X[k-1]\} \in T \wedge \{X[1], \ldots, X[k-2], X[k]\} \in T$ then
12:         $C_T := C_T \cup \{X\}$
13:         $X.support{+}{+}$
14:       end if
15:     end for
16:     if $C_T \neq \{\}$ then
17:       $\overline{C}_k := \overline{C}_k \cup \{(tid, C_T)\}$
18:     end if
19:   end for
20:   Extract $\mathcal{F}_k$ of all frequent $k$-itemsets
21:   $k{+}{+}$
22: end while
```

early iterations, each entry may be larger than its corresponding transaction. Therefore, another algorithm, AprioriHybrid, has been proposed [6, 4] that combines the Apriori and AprioriTid algorithms into a single hybrid. This hybrid algorithm uses Apriori for the initial iterations and switches to AprioriTid when it is expected that the set $\overline{C}_k$ fits into main memory. Since the size of $\overline{C}_k$ is proportional with the number of candidate itemsets, a heuristic is used that estimates the size that $\overline{C}_k$ would have in the current iteration. If this size is small enough and there are fewer candidate patterns in the current iteration than in the previous iteration, the algorithm decides to switch to AprioriTid. Unfortunately, this heuristic is not airtight as will be shown in Chapter 4. Nevertheless, AprioriHybrid performs almost always better than Apriori.

Counting candidate 2-itemsets

Shortly after the proposal of the Apriori algorithms described before, Park et al. proposed another optimization, called DHP (Direct Hashing and Pruning) to reduce the number of candidate itemsets [65]. During the kth iteration, when the supports of all candidate k-itemsets are counted by scanning the database, DHP already gathers information about candidate itemsets of size $k+1$ in such a way that all $(k+1)$-subsets of each transaction after some pruning are hashed to a hash table. Each bucket in the hash table consists of a counter to represent how many itemsets have been hashed to that bucket so far. Then, if a candidate itemset of size $k+1$ is generated, the hash function is applied on that itemset. If the counter of the corresponding bucket in the hash table is below the minimal support threshold, the generated itemset is not added to the set of candidate itemsets. Also, during the support counting phase of iteration k, every transaction trimmed in the following way. If a transaction contains a frequent itemset of size $k+1$, any item contained in that $k+1$ itemset will appear in at least k of the candidate k-itemsets in C_k. As a result, an item in transaction T can be trimmed if it does not appear in at least k of the candidate k-itemsets in C_k. These techniques result in a significant decrease in the number of candidate itemsets that need to be counted, especially in the second iteration. Nevertheless, creating the hash tables and writing the adapted database to disk at every iteration causes a significant overhead.

Although DHP was reported to have better performance than Apriori and AprioriHybrid, this claim was countered by Ramakrishnan if the following optimization is added to Apriori [72]. Instead of using the hash-tree to store and count all candidate 2-itemsets, a triangular array C is created, in which the support counter of a candidate 2-itemset $\{i, j\}$ is stored at location $C[i][j]$. Using this array, the support counting procedure reduces to a simple two-level for-loop over each transaction. A similar technique was later used by Orlando et al. in their DCP and DCI algorithms [63, 64].

Since the number of candidate 2-itemsets is exactly $\binom{|\mathcal{F}_1|}{2}$, it is still possible that this number is too large, such that only part of the structure can be generated and multiple scans over the database need to be performed. Nevertheless, from experience, we discovered that a lot of candidate 2-itemsets do not even occur at all in the database, and hence, their support remains 0. Therefore, we propose the following optimization. When all single items are counted, resulting in the set of all frequent items $\mathcal{F}_1$, we do not generate any candidate 2-itemset. Instead, we start scanning the database, and remove from each transaction all items that are not frequent, on the fly. Then, for each trimmed transaction, we increase the support of all candidate 2-itemsets contained in that transaction. However, if the candidate 2-itemset does not yet exists, we generate the candidate itemset and initialize its support to 1. In this way, only those candidate 2-itemsets that occur at least once in the database are generated. For example, this technique was especially useful for the basket data set used in our experiments, since in that data set there exist 8 051 frequent items, and hence Apriori would generate $\binom{8\,051}{2} = 32\,405\,275$ candidate 2-itemsets. Using this technique, this number was drastically reduced to 1 708 203.

Support lower bounding

As we already mentioned earlier in this chapter, apart from the monotonicity property, it is sometimes possible to derive information on the support of an itemset, given the support of all of its subsets. The first algorithm that uses such a technique was proposed by Bayardo in his MaxMiner and Apriori-LB algorithms [9]. The presented technique is based on the following property which gives a lower bound on the support of an itemset.

Proposition 2.4. *Let $X, Y, Z \subseteq \mathcal{I}$ be itemsets.*

$$\mathit{support}(X \cup Y \cup Z) \geq \mathit{support}(X \cup Y) + \mathit{support}(X \cup Z) - \mathit{support}(X)$$

Proof.

$$\begin{aligned}
\mathit{support}(X \cup Y \cup Z) &= |\mathit{cover}(X \cup Y) \cap \mathit{cover}(X \cup Z)| \\
&= |\mathit{cover}(X \cup Y) \setminus (\mathit{cover}(X \cup Y) \setminus \mathit{cover}(X \cup Z))| \\
&\geq |\mathit{cover}(X \cup Y) \setminus (\mathit{cover}(X) \setminus \mathit{cover}(X \cup Z))| \\
&\geq |\mathit{cover}(X \cup Y)| - |(\mathit{cover}(X) \setminus \mathit{cover}(X \cup Z))| \\
&= |\mathit{cover}(X \cup Y)| - (|\mathit{cover}(X)| - |\mathit{cover}(X \cup Z)|) \\
&= \mathit{support}(X \cup Y) + \mathit{support}(X \cup Z) - \mathit{support}(X)
\end{aligned}$$

□

In practice, this lower bound can be used in the following way. Every time a candidate $k+1$-itemset is generated by joining two of its subsets of size k, we can easily compute this lower bound for that candidate. Indeed, suppose the candidate itemset $X \cup \{i_1, i_2\}$ is generated by joining $X \cup \{i_1\}$ and $X \cup \{i_2\}$, we simply add up the supports of these two itemsets and subtract the support of X. If this lower bound is higher than the minimal support threshold, then we already know that it is frequent and hence, we can already generate candidate itemsets of larger sizes for which this lower bound can again be computed. Nevertheless, we still need to count the exact supports of all these itemsets, but this can be done all at once during the support counting procedure. Using the efficient support counting mechanism as we described before, this optimization could result in significant performance improvements.

Additionally, we can exploit a special case of Proposition 2.4 even more.

Corollary 2.5. *Let $X, Y, Z \subseteq \mathcal{I}$ be itemsets.*

$$\mathit{support}(X \cup Y) = \mathit{support}(X) \Rightarrow \mathit{support}(X \cup Y \cup Z) = \mathit{support}(X \cup Z)$$

This specific property was later exploited by Pasquier et al. in order to find a concise representation of all frequent itemsets [66, 14]. Nevertheless, it can already be used to improve the Apriori algorithm.

Suppose we have generated and counted the support of the frequent itemset $X \cup \{i\}$ and that its support is equal to the support of X. Then we already know that the supports of every superset $X \cup \{i\} \cup Y$ is equal to the support of $X \cup Y$ and hence, we do not have to generate all such supersets anymore, but only have to keep the information that every superset of $X \cup \{i\}$ is also represented by a superset of X.

Recently, Calders and Goethals presented a generalization of all these techniques resulting in a system of deduction rules that derive tight bounds on the support of candidate itemsets [19]. These deduction rules allow for constructing a minimal representation of all frequent itemsets, but can also be used to efficiently generate the set of all frequent itemsets. Unfortunately, for a given candidate itemset, an exponential number of rules in the length of the itemset need to be evaluated. The rules presented in this section, which are part of the complete set of derivation rules, are shown to result in significant performance improvements, while the other rules only show a marginal improvement.

Combining passes

Another improvement of the Apriori algorithm, which is part of the folklore, tries to combine as many iterations as possible in the end, when only few candidate patterns can still be generated. The potential of such a combination technique was realized early on [6], but the modalities under which it can be applied were never further examined. In Chapter 4, we study this problem and provide several upper bounds on the number of candidate itemsets that can yet be generated after a certain iteration in the Apriori algorithm.

Dynamic Itemset Counting

The DIC algorithm, proposed by Brin et al. tries to reduce the number of passes over the database by dividing the database into intervals of a specific size [16]. First, all candidate patterns of size 1 are generated. The supports of the candidate sets are then counted over the first interval of the database. Based on these supports, a new candidate pattern of size 2 is already generated if all of its subsets are already known to be frequent, and its support is counted over the database together with the patterns of size 1. In general, after every interval, candidate patterns are generated and counted. The algorithm stops if no more candidates can be generated and all candidates have been counted over the complete database. Although this method drastically reduces the number of scans through the database, its performance is also heavily dependent on the distribution of the data.

Although the authors claim that the performance improvement of reordering all items in support ascending order is negligible, this is not true for Apriori in general. Indeed, the reordering used in DIC was based on the supports of the 1-itemsets that were computed only in the first interval. Obviously, the success of this heuristic also becomes highly dependent on the distribution of the data.

The CARMA algorithm (Continuous Association Rule Mining Algorithm), proposed by Hidber [45] uses a similar technique, reducing the interval size to 1. More specifically, candidate itemsets are generated on the fly from every transaction. After reading a transaction, it increments the supports of all candidate itemsets contained in that transaction and it generates a new candidate itemset contained in that transaction, if all of its subsets are suspected to be relatively frequent with respect to the number of transactions that has already been processed. As a consequence, CARMA generates a lot more candidate itemsets than DIC or Apriori. (Note that the number of candidate itemsets generated by DIC is exactly the same as in Apriori.) Additionally, CARMA allows the user to change the minimal support threshold during the execution of the algorithm. After the database has been processed once, CARMA is guaranteed to have generated a superset of all frequent itemsets relative to some threshold which depends on how the user changed the minimal support threshold during its execution. However, when the minimal support threshold was kept fixed during the complete execution of the algorithm, at least all frequent itemsets have been generated. To determine the exact supports of all generated itemsets, a second scan of the database is required.

Sampling

The sampling algorithm, proposed by Toivonen [77], performs at most two scans through the database by picking a random sample from the database, then finding all relatively frequent patterns in that sample, and then verifying the results with the rest of the database. In the cases where the sampling method does not produce all frequent patterns, the missing patterns can be found by generating all remaining potentially frequent patterns and verifying their supports during a second pass through the database. The probability of such a failure can be kept small by decreasing the minimal support threshold. However, for a reasonably small probability of failure, the threshold must be drastically decreased, which can cause a combinatorial explosion of the number of candidate patterns.

Partitioning

The Partition algorithm, proposed by Savasere et al. uses an approach which is completely different from all previous approaches [70]. That is, the database is stored in main memory using the vertical database layout and the support of an itemset is computed by intersecting the covers of two of its subsets. More specifically, for every frequent item, the algorithm stores its cover. To compute the support of a candidate k-itemset I, which is generated by joining two of its subsets X, Y as in the Apriori algorithm, it intersects the covers of X and Y, resulting in the cover of I.

Of course, storing the covers of all items actually means that the complete database is read into main memory. For large databases, this could be impossible. Therefore, the Partition algorithm uses the following trick. The database is partitioned into several disjoint parts and the algorithm generates for every part all itemsets that are relatively frequent within that part, using the algorithm described in the previous paragraph and shown in Algorithm 4. The parts of the database are chosen in such a way that each part fits into main memory on itself.

The algorithm merges all relatively frequent itemsets of every part together. This results in a superset of all frequent itemsets over de complete database, since an itemset that is frequent in the complete database must be relatively frequent in one of the parts. Then, the actual supports of all itemsets are computed during a second scan through the database. Again, every part is read into main memory using the vertical database layout and the support of every itemset is computed by intersecting the covers of all items occurring in that itemset. The exact Partition algorithm is given in Algorithm 5.

The exact computation of the supports of all itemsets can still be optimized, but we refer to the original article for further implementation de-

Algorithm 4 Partition - Local Itemset Mining

```
Input: $\mathcal{D}, \sigma$
Output: $\mathcal{F}(\mathcal{D}, \sigma)$
 1: Compute $\mathcal{F}_1$ and store with every frequent item its cover
 2: $k := 2$
 3: while $\mathcal{F}_{k-1} \neq \{\}$ do
 4:     $\mathcal{F}_k := \{\}$
 5:     for all $X, Y \in \mathcal{F}_{k-1}, X[i] = Y[i]$ for $1 \leq i \leq k-2$, and $X[k-1] < Y[k-1]$
        do
 6:         $I = \{X[1], \ldots, X[k-1], Y[k-1]\}$
 7:         if $\forall J \subset I : J \in \mathcal{F}_{k-1}$ then
 8:             $I.cover := X.cover \cap Y.cover$
 9:             if $|I.cover| \geq \sigma$ then
10:                 $\mathcal{F}_k := \mathcal{F}_k \cup I$
11:             end if
12:         end if
13:     end for
14:     $k{+}{+}$
15: end while
```

Algorithm 5 Partition

```
Input: $\mathcal{D}, \sigma$
Output: $\mathcal{F}(\mathcal{D}, \sigma)$
 1: Partition $\mathcal{D}$ in $D_1, \ldots, D_n$
 2: // Find all local frequent itemsets
 3: for $1 \leq p \leq n$ do
 4:     Compute $C^p := \mathcal{F}(D_p, \lceil \sigma_{rel} \cdot |D_p| \rceil)$
 5: end for
 6: // Merge all local frequent itemsets
 7: $C_{global} := \bigcup_{1 \leq p \leq n} C^p$
 8: // Compute actual support of all itemsets
 9: for $1 \leq p \leq n$ do
10:     Generate cover of each item in $D_p$
11:     for all $I \in C_{global}$ do
12:         $I.support := I.support + |I[1].cover \cap \cdots \cap I[|I|].cover|$
13:     end for
14: end for
15: // Extract all global frequent itemsets
16: $\mathcal{F} := \{I \in C_{global} \mid I.support \geq \sigma\}$
```

tails [70].

Although the covers of all items can be stored in main memory, during the generation of all local frequent itemsets for every part, it is still possible that the covers of all local candidate k-itemsets can not be stored in main memory. Also, the algorithm is highly dependent on the heterogeneity of the database and can generate too many local frequent itemsets, resulting in a significant decrease in performance. However, if the complete database fits into main memory and the total of all covers at any iteration also does not exceed main memory limits, then the database must not be partitioned at all and outperforms Apriori by several orders of magnitude. Of course, this is mainly due to the fast intersection based counting mechanism.

2.6 Depth-First Algorithms

As explained in the previous section, the intersection based counting mechanism made possible by using the vertical database layout shows significant performance improvements. However, this is not always possible since the total size of all covers at a certain iteration of the local itemset generation procedure could exceed main memory limits. Nevertheless, it is possible to significantly reduce this total size by generating collections of candidate itemsets in a depth-first strategy. The first algorithm proposed to generate all frequent itemsets in a depth-first manner is the Eclat algorithm by Zaki [80, 84]. Later, several other depth-first algorithms have been proposed [1, 2, 41] of which the FP-growth algorithm by Han et al. [41, 40] is the most well known. In this section, we explain both the Eclat and FP-growth algorithms.

Given a transaction database $\mathcal{D}$ and a minimal support threshold σ, denote the set of all frequent k-itemsets with the same $k-1$-prefix $I \subseteq \mathcal{I}$ by $\mathcal{F}[I](\mathcal{D}, \sigma)$. (Note that $\mathcal{F}[\{\}](\mathcal{D}, \sigma) = \mathcal{F}(\mathcal{D}, \sigma)$.) Both Eclat and FP-growth recursively generate for every item $i \in \mathcal{I}$ the set $\mathcal{F}[\{i\}](\mathcal{D}, \sigma)$.

For the sake of simplicity and presentation, we assume that all items that occur in the transaction database are frequent. In practice, all frequent items can be computed during an initial scan over the database, after which all infrequent items will be ignored.

2.6.1 Eclat

Eclat uses the vertical database layout and uses the intersection based approach to compute the support of an itemset. The Eclat algorithm is given in Algorithm 6.

Note that a candidate itemset is now represented by each set $I \cup \{i, j\}$ of which the support is computed at line 6 of the algorithm. Since the algorithm doesn't fully exploit the monotonicity property, but generates a candidate

Algorithm 6 Eclat

```
Input: $\mathcal{D}, \sigma, I \subseteq \mathcal{I}$
Output: $\mathcal{F}[I](\mathcal{D}, \sigma)$
 1: $\mathcal{F}[I] := \{\}$
 2: for all $i \in \mathcal{I}$ occurring in $\mathcal{D}$ do
 3:   $\mathcal{F}[I] := \mathcal{F}[I] \cup \{I \cup \{i\}\}$
 4:   // Create $\mathcal{D}^i$
 5:   $\mathcal{D}^i := \{\}$
 6:   for all $j \in \mathcal{I}$ occurring in $\mathcal{D}$ such that $j > i$ do
 7:     $C := cover(\{i\}) \cap cover(\{j\})$
 8:     if $|C| \geq \sigma$ then
 9:       $\mathcal{D}^i := \mathcal{D}^i \cup \{(j, C)\}$
10:     end if
11:   end for
12:   // Depth-first recursion
13:   Compute $\mathcal{F}[I \cup \{i\}](\mathcal{D}^i, \sigma)$
14:   $\mathcal{F}[I] := \mathcal{F}[I] \cup \mathcal{F}[I \cup \{i\}]$
15: end for
```

itemset based on only two of its subsets, the number of candidate itemsets that are generated is much larger as compared to the breadth-first approaches presented in the previous section. As a comparison, Eclat essentially generates candidate itemsets using only the join step from Apriori, since the itemsets necessary for the prune step are not available. Again, we can reorder all items in the database in support ascending order to reduce the number of candidate itemsets that is generated, and hence, reduce the number of intersections that need to be computed and the total size of the covers of all generated itemsets. In fact, such reordering can be performed at every recursion step of the algorithm between line 10 and line 11 in the algorithm. In comparison with Apriori, counting the supports of all itemsets is performed much more efficiently. In comparison with Partition, the total size of all covers that is kept in main memory is on average much less. Indeed, in the breadth-first approach, at a certain iteration k, all frequent k-itemsets are stored in main memory together with their covers. On the other hand, in the depth-first approach, at a certain depth d, the covers of at most all k-itemsets with the same $k-1$-prefix are stored in main memory, with $k \leq d$. Because of the item reordering, this number is kept small.

Recently, Zaki and Gouda [81, 83] proposed a new approach to efficiently compute the support of an itemset using the vertical database layout. Instead of storing the cover of a k-itemset I, the difference between the cover of I and the cover of the $k-1$-prefix of I is stored, denoted by the *diffset* of I. To compute the support of I, we simply need to subtract the size of the diffset from the support of its $k-1$-prefix. Note that this support does not need to be stored within each itemset but can be maintained as a parameter within the recursive function calls of the algorithm. The diffset of an itemset $I \cup \{i, j\}$, given the two diffsets of its subsets $I \cup \{i\}$ and $I \cup \{j\}$, with $i < j$, is computed as follows:

$$\mathit{diffset}(I \cup \{i,j\}) := \mathit{diffset}(I \cup \{j\}) \setminus \mathit{diffset}(I \cup \{i\}).$$

This technique has experimentally shown to result in significant performance improvements of the algorithm, now designated as *dEclat* [81]. The original database is still stored in the original vertical database layout. Observe an arbitrary recursion path of the algorithm starting from the itemset $\{i_1\}$, up to the k-itemset $I = \{i_1, \ldots, i_k\}$. The itemset $\{i_1\}$ has stored its cover and for each recursion step that generates a subset of I, we compute its diffset. Obviously, the total size of all diffsets generated on the recursion path can be at most $|\mathit{cover}(\{i_1\})|$. On the other hand, if we generate the cover of each generated itemset, the total size of all generated covers on that path is at least $(k-1) \cdot \sigma$ and can be at most $(k-1) \cdot |\mathit{cover}(\{i_1\})|$. Of course, not all generated diffsets or covers are stored during all recursions, but only for the last two of them. This observation indicates that the total size of all diffsets that are stored in main memory at a certain point in the algorithm is less than the total size of all covers. These predictions were supported by several experiments [81].

Using this depth-first approach, it remains possible to exploit a technique presented as an optimization of the Apriori algorithm in the previous section. More specifically, suppose we have generated and counted the support of the frequent itemset $X \cup \{i\}$ and that its support is equal to the support of X (hence, its diffset is empty). Then we already know that the support of every superset $X \cup \{i\} \cup Y$ is equal to the support of $X \cup Y$ and hence, we do not have to generate all such supersets anymore, but only have to retain the information that every superset of $X \cup \{i\}$ is also represented by a superset of X.

If the database does not fit into main memory, the Partition algorithm can be used in which the local frequent itemsets are found using Eclat.

Another optimization proposed by Hipp et al. combines Apriori and Eclat into a single Hybrid [46]. More specifically, the algorithm starts generating frequent itemsets in a breadth-first manner using Apriori, and switches after a certain iteration to a depth-first strategy using Eclat. The exact switching point must be given by the user. The main performance improvement of this strategy occurs at the generation of all candidate 2-itemsets if these are generated online as described in Section 2.5.4. Indeed, when a lot of items in the database are frequent, Eclat generates every possible 2-itemset whether or not it occurs in the database. On the other hand, if the transaction database contains a lot of large transactions of frequent items, such that Apriori needs to generate all its subsets of size 2, Eclat still outperforms Apriori. Of course, as long as the number of transactions that still contain candidate itemsets is too high to store into main memory, switching to Eclat might be impossible, while Apriori nicely marches on.

2.6.2 FP-growth

In order to count the supports of all generated itemsets, FP-growth uses a combination of the vertical and horizontal database layout to store the database in main memory. Instead of storing the cover for every item the database, it stores the actual transactions from the database in a trie structure and every item has a linked list going through all transactions that contain that item. This new data structure is denoted by *FP-tree* (Frequent-Pattern tree) and is created as follows [41]. Again, we order the items in the database in support ascending order for the same reasons as before. First, create the root node of the tree, labelled with "null". For each transaction in the database, the items are processed in reverse order (hence, support descending) and a branch is created for each transaction. Every node in the FP-tree additionally stores a counter which keeps track of the number of transactions that share that node. Specifically, when considering the branch to be added for a transaction, the count of each node along the common prefix is incremented by 1, and nodes for the items in the transaction following the prefix are created and linked accordingly. Additionally, an item header table is built so that each item points to its occurrences in the tree via a chain of node-links. Each item in this header table also stores its support. The reason to store transactions in the FP-tree in support descending order is that in this way, it is hoped that the FP-tree representation of the database is kept as small as possible since the more frequently occurring items are arranged closer to the root of the FP-tree and thus are more likely to be shared.

Example 2.2. Assume we are given a transaction database and a minimal support threshold of 2. First, the supports of all items is computed, all infrequent items are removed from the database and all transactions are reordered according to the support descending order resulting in the example transaction database in Table 2.7. The FP-tree for this database is shown in Figure 2.3.

Given such an FP-tree, the supports of all frequent items can be found in the header table. Obviously, the FP-tree is just like the vertical and horizontal database layouts a lossless representation of the complete transaction database for the generation of frequent itemsets. Indeed, every linked list starting from

tid	X
100	$\{a, b, c, d, e, f\}$
200	$\{a, b, c, d, e\}$
300	$\{a, d\}$
400	$\{b, d, f\}$
500	$\{a, b, c, e, f\}$

Table 2.7: An example preprocessed transaction database.

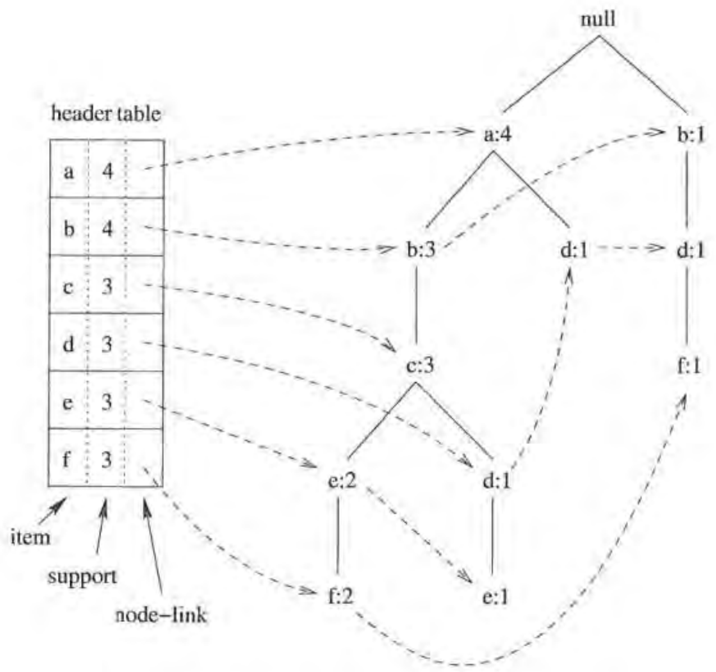


Figure 2.3: An example of an FP-tree.

an item in the header table actually represents a compressed form of the cover of that item. On the other hand, every branch starting from the root node represents a compressed form of a set of transactions.

Apart from this FP-tree, the FP-growth algorithm is very similar to Eclat, but it uses some additional steps to maintain the FP-tree structure during the recursion steps, while Eclat only needs to maintain the covers of all generated itemsets. More specifically, in order to generate for every $i \in \mathcal{I}$ all frequent itemsets in $\mathcal{F}[\{i\}](\mathcal{D}, \sigma)$, FP-growth creates the so called *i-projected* database of $\mathcal{D}$. Essentially, the $\mathcal{D}^i$ used in Eclat is the vertical database layout of the i-projected database considered here. The FP-growth algorithm is given in Algorithm 7.

Algorithm 7 FP-growth

```
Input: $\mathcal{D}, \sigma, I \subseteq \mathcal{I}$
Output: $\mathcal{F}[I](\mathcal{D}, \sigma)$
 1: $\mathcal{F}[I] := \{\}$
 2: for all $i \in \mathcal{I}$ occurring in $\mathcal{D}$ do
 3:   $\mathcal{F}[I] := \mathcal{F}[I] \cup \{I \cup \{i\}\}$
 4:   // Create $\mathcal{D}^i$
 5:   $\mathcal{D}^i := \{\}$
 6:   $H := \{\}$
 7:   for all $j \in \mathcal{I}$ occurring in $\mathcal{D}$ such that $j > i$ do
 8:     if $support(I \cup \{i, j\}) \geq \sigma$ then
 9:       $H := H \cup \{j\}$
10:     end if
11:   end for
12:   for all $(tid, X) \in \mathcal{D}$ with $i \in X$ do
13:     $\mathcal{D}^i := \mathcal{D}^i \cup \{(tid, X \cap H)\}$
14:   end for
15:   // Depth-first recursion
16:   Compute $\mathcal{F}[I \cup \{i\}](\mathcal{D}^i, \sigma)$
17:   $\mathcal{F}[I] := \mathcal{F}[I] \cup \mathcal{F}[I \cup \{i\}]$
18: end for
```

The only difference between Eclat an FP-growth is the way they count the supports of every candidate itemset and how they represent and maintain the i-projected database. I.e., only lines 5–10 of the Eclat algorithm are renewed. First, FP-growth computes all frequent items for $\mathcal{D}^i$ at lines 6–10, which is of course different in every recursion step. This can be efficiently done by simply following the linked list starting from the entry of i in the header table. Then at every node in the FP-tree it follows its path up to the root node and increments the support of each item it passes by its count. Then, at lines

11–13, the FP-tree for the i-projected database is built for those transactions in which i occurs, intersected with the set of all frequent items in $\mathcal{D}$ greater than i. These transactions can be efficiently found by following the node-links starting from the entry of item i in the header table and following the path from every such node up to the root of the FP-tree and ignoring all items that are not in H. If this node has count n, then the transaction is added n times. Of course, this is implemented by simply incrementing the counters, on the path of this transaction in the new i-projected FP-tree, by n. However, this technique does require that every node in the FP-tree also stores a link to its parent. Additionally, we can also use the technique that generates only those candidate itemsets that occur at least once in the database. Indeed, we can dynamically add a counter initialized to 1 for every item that occurs on each path in the FP-tree that is traversed.

These steps can be further optimized as follows. Suppose that the FP-tree consists of a single path. Then, we can stop the recursion and simply enumerate every combination of the items occurring on that path with the support set to the minimum of the supports of the items in that combination. Essentially, this technique is similar to the technique used by all other algorithms when the support of an itemset is equal to the support of any of its subsets. However, FP-growth is able to detect this one recursion step ahead of Eclat.

As can be seen, at every recursion step, an item j occurring in $\mathcal{D}^i$ actually represents the itemset $I \cup \{i, j\}$. In other words, for every frequent item i occurring in $\mathcal{D}$, the algorithm recursively finds all frequent 1-itemsets in the i-projected database $\mathcal{D}^i$.

Although the authors of the FP-growth algorithm claim that their algorithm [40, 41] does not generate any candidate itemsets, we have shown that the algorithm actually generates a lot more candidate itemsets since it essentially uses the same candidate generation technique as is used in Apriori but without its prune step.

The only main advantage FP-growth has over Eclat is that each linked list, starting from an item in the header table representing the cover of that item, is stored in a compressed form. Unfortunately, to accomplish this gain, it needs to maintain a complex data structure and perform a lot of dereferencing, while Eclat only has to perform simple and fast intersections. Also, the intended gain of this compression might be much less than is hoped for. In Eclat, the cover of an item can be implemented using an array of transaction identifiers. On the other hand, in FP-growth, the cover of an item is compressed using the linked list starting from its node-link in the header table, but, every node in this linked list needs to store its label, a counter, a pointer to the next node, a pointer to its branches and a pointer to its parent. Therefore, the size of an FP-tree should be at most 20% of the size of all covers in Eclat in order to profit from this compression. Table 2.8 shows for all four used data sets

Data set	$\|\|\mathcal{D}\|\|$	$\|$FP-tree$\|$	$\frac{\|cover\|}{\|\text{FP-tree}\|}$
T40I10D100K	3 912 459 : 15 283K	3 514 917 : 68 650K	89% : 174%
mushroom	174 332 : 680K	16 354 : 319K	9% : 46%
BMS-Webview-1	148 209 : 578K	55 410 : 1 082K	37% : 186%
basket	399 838 : 1 561K	294 311 : 5 748K	73% : 368%

Table 2.8: Memory usage of Eclat versus FP-growth.

the size of the total length of all arrays in Eclat ($||\mathcal{D}||$), the total number of nodes in FP-growth (|FP-tree|) and the corresponding compression rate of the FP-tree. Additionally, for each entry, we show the size of the data structures in bytes and the corresponding compression of the FP-tree.

As can be seen, the only data set for which FP-growth becomes an actual compression of the database is the mushroom data set. For all other data sets, there is no compression at all, on the contrary, the FP-tree representation is often much larger than the plain array based representation.

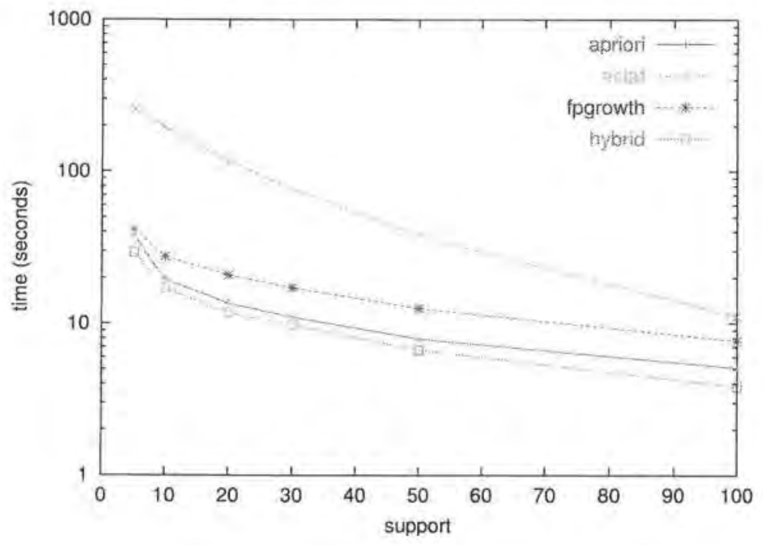
2.7 Experimental Evaluation

We implemented the Apriori implementation using the online candidate 2-itemset generation optimization. Additionally, we implemented the Eclat, Hybrid and FP-growth algorithms as presented in the previous section. All these algorithms were implemented in C++ using several of the data structures provided by the C++ Standard Template Library [75]. All experiments reported in this thesis were performed on a 400 MHz Sun Ultra Sparc 10 with 512 MB main memory, running Sun Solaris 8.

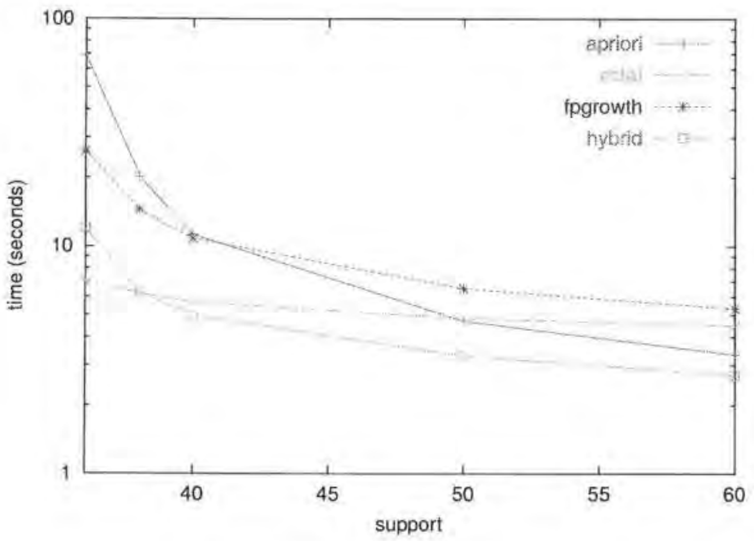
Figure 2.7 shows the performance of the algorithms on each of the data sets described in Section 2.4 for varying minimal support thresholds.

The first interesting behavior can be observed in the experiments for the basket data. Indeed, Eclat performs much worse than all other algorithms. Nevertheless, this behavior has been predicted since the number of frequent items in the basket data set is very large and hence, a huge amount of candidate 2-itemsets is generated. The other algorithms all use dynamic candidate generation of 2-itemsets resulting in much better performance results. The Hybrid algorithm performed best when Apriori was switched to Eclat after the second iteration, i.e., when all frequent 2-itemsets were generated.

Another remarkable result is that Apriori performs better than FP-growth for the basket data set. This result is due to the overhead created by the maintenance of the FP-tree structure, while updating the supports of all candidate itemsets contained in each transaction is performed very fast due to the sparseness of this data set.

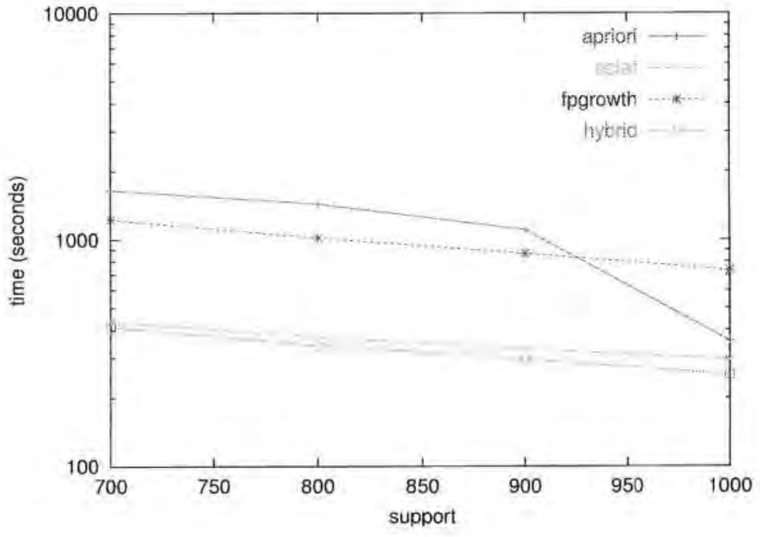


(a) basket

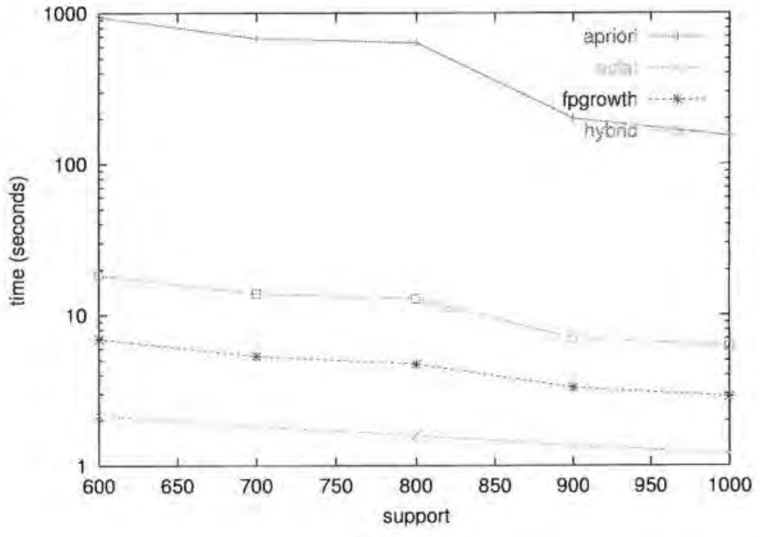


(b) BMS-Webview-1

Figure 2.4: Frequent itemset mining performance.



(c) T40I10D100K



(d) mushroom

Figure 2.4: Frequent itemset mining performance.

For the BMS-Webview-1 data set, the Hybrid algorithm again performed best when switched after the second iteration. For all minimal support thresholds higher than 40, the differences in performance are negligible and are mainly due to the initialization and destruction routines of the used data structures. For very low support thresholds, Eclat clearly outperforms all other algorithms. The reason for the lousy performance of Apriori is because of some very large transactions for which the subset generation procedure for counting the supports of all candidate itemsets consumes most of the time. To support this claim we did some additional experiments which indeed showed that only 34 transactions containing more than 100 frequent items consumed most of the time of during the support counting of all candidate itemsets of sized 5 to 10. For example, counting the supports of all 7-itemsets takes 10 seconds of which 9 seconds were used for these 34 transactions.

For the synthetic data set, all experiments showed the normal behavior as was predicted by the analysis in this survey. However, this time, the switching point for which the Hybrid algorithm performed best was after the third iteration.

Also the mushroom data set shows some interesting results. The performance differences of Eclat and FP-growth are negligible and are again mainly due to the differences in initialization and destruction. Obviously, because of the small size of the database, both run extremely fast. Apriori on the other hand runs extremely slow because each transaction contains exactly 23 items and of which a many have very high supports. Here, the Hybrid algorithm doesn't perform well at all and only performed well when Apriori is not used at all. We show the time of Hybrid when the switch is performed after the second iteration.

2.8 Conclusions

Throughout the last decade, a lot of people have implemented and compared several algorithms that try to solve the frequent itemset mining problem as efficiently as possible. Unfortunately, only a very small selection of researchers put the source codes of their algorithms publicly available such that fair empirical evaluations and comparisons of their algorithms become very difficult. Moreover, we experienced that different implementations of the same algorithms could still result in significantly different performance results. As a consequence, several claims that were presented in some articles were later contradicted in other articles. For example, a very often used implementation of the Apriori algorithm is that by Christian Borgelt [13]. Nevertheless, when we compared his implementation with ours, the performance of both algorithms showed immense differences. While Borgelt his implementation performed much better for high support thresholds, it performed much worse for small thresholds. This is mainly due to differences in the implementation of some of the used data structures and procedures. Indeed, different compilers and different machine architectures sometimes showed different behavior for the same algorithms. Also different kinds of data sets on which the algorithms were tested showed remarkable differences in the performance of such algorithms. An interesting example of this is presented by Zheng et al. in their article on the real world performance of association rule algorithms [85] in which five well-known association rule mining algorithms are compared on three new real-world data sets. They discovered different performance behaviors of the algorithms as was previously claimed by their respective authors.

In this survey, we presented an in depth analysis of a lot of algorithms which made a significant contribution to improve the efficiency of frequent itemset mining.

We have shown that as long as the database fits in main memory, the Hybrid algorithm, as a combination of an optimized version of Apriori and Eclat is by far the most efficient algorithm. However, for very dense databases, the Eclat algorithm is still better.

If the database does not fit into memory, the best algorithm depends on the density of the database. For sparse databases the Hybrid algorithm seems the best choice if the switch from Apriori to Eclat is made as soon as the database fits into main memory. For dense databases, we envisage that the partition algorithm, using Eclat to compute all local frequent itemsets, performs best.

For our experiments, we did not implement Apriori with all possible optimizations as presented in this survey. Nevertheless, the main cost of the algorithm can be dictated by only a few very large transactions, for which the presented optimizations will not always be sufficient.

Several experiments on four different data sets confirmed the reasoning presented in the analysis of the various algorithms.

3

Interactive Constrained Association Rule Mining

The interactive nature of the mining process has been acknowledged from the start [27]. It motivated the idea of a "data mining query language" [37, 38, 47, 48, 60] and was stressed again by Ng et al. [62]. A data mining query language allows the user to ask for specific subsets of association rules by specifying several constraints within each query.

In this chapter, we consider a class of conditions on associations to be generated, which should be expressible in any reasonable data mining query language: Boolean combinations of atomic conditions, where an atomic condition can either specify that a certain item occurs in the body of the rule or the head of the rule, or set a threshold on the support or on the confidence. A *mining session* then consists of a sequence of such Boolean combinations (henceforth referred to as *queries*). Efficiently supporting data mining query language environments is a challenging task. Towards this goal, we present and compare three approaches. In the first extreme, the *integrated querying* approach, every individual data mining query will be answered by running an adaptation of the mining algorithm in which the constraints on the rules and sets to be generated are directly incorporated. The second extreme, the *post-processing* approach, first mines as much associations as possible, by performing one major, global mining operation. After this expensive operation, the actual data mining queries issued by the user then amount to standard lookups in the set of materialized associations. A third approach, the *incremental querying* approach, combines the advantages of both previous approaches.

Our contributions We present the first algorithm to support interactive mining sessions efficiently. We measure efficiency in terms of the total number of itemsets that are generated, but do not satisfy the query, and the number of scans over the database that have to be performed. Specifically, our results are the following:

1. Although our results show significant improvements of performance, we will also show that exploiting constraints is not always the best solution. More specifically, if mining without constraints is feasible to begin with, then the presented post-processing approach will eventually outperform integrated querying.

2. The querying achieved by exploiting the constraints is *optimal*, in the sense that it never generates an itemset that could give rise to a rule that does not satisfy the query, apart from the minimal support and confidence thresholds. Therefore, the number of generated itemsets during the execution of a query, becomes proportional to the strength of the constraints in the query: the more specific the query, the faster its execution.

3. Not only is the number of passes through the database reduced, but also the size of the database itself, again proportionally to the strength of the constraints in the query.

4. Within a session, a generated itemset will never be regenerated as a candidate itemset: results of earlier queries are reused when answering a new query.

This chapter is further organized as follows. Section 3.1 gives an overview of related work on constrained mining. In Section 3.2, we present a way of incorporating query-constraints inside a frequent set mining algorithm. In Section 3.3, we discuss ways of supporting interactive mining sessions. We conclude the chapter in Section 3.4.

Bibliographical note Parts of this chapter have been published before in the Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery [31].

3.1 Related Work

The idea that queries can be integrated in the mining algorithm was initially launched by Srikant, Vu, and Agrawal [74], who considered queries that are Boolean expressions over the presence or absence of certain items in the rules.

Queries specified on bodies or heads were not discussed. The authors considered three different approaches to the problem. The proposed algorithms are not optimal: they generate and test several itemsets that do not satisfy the query, and their optimizations also do not always become more efficient for more specific queries.

Also Lakshmanan, Ng, Han and Pang worked on the integration of constraints on itemsets in mining, considering conjunctions of conditions on itemsets such as those considered here, as well as others (arbitrary Boolean combinations were not discussed) [55, 62]. Of the various strategies for the so-called "CAP" algorithm they present, the one that can handle the queries considered here, limited to conjunctions, is their "strategy II". Again, this strategy generates and tests itemsets that do not satisfy the query. Also, their algorithms implement a rule-query by separately mining for possible heads and for possible bodies, while we tightly couple the querying of rules with the querying of sets. This work has also been further studied by Pei, Han and Lakshmanan [67, 68], and employed within the FP-growth algorithm.

Still other work focused on other kinds of constraints over association rules and frequent sets, such as *correlation* [33], and *improvement* [10]. These and other statistical measures of interestingness will not be discussed.

All previously mentioned works do not discuss the reuse of results acquired from earlier queries within a session. Nag, Deshpande, and DeWitt proposed the use of a knowledge cache for this purpose [61]. Several caching strategies were studied for different cache sizes. However, their work only considers mining sessions of queries where only constraints on the support of the itemsets are allowed. No solutions were provided for other constraints like those studied here. Also Jeudy and Boulicaut have studied the use of a knowledge cache for finding a condensed representation of all itemsets, based on the concept of *free sets* [50].

3.2 Exploiting Constraints

As already mentioned in the introduction, the constraints we consider are Boolean combinations of atomic conditions. An atomic condition can either specify that a certain item i occurs in the body of the rule or the head of the rule, denoted respectively by Body(i) or Head(i), or set a threshold on the support or on the confidence.

In this section, we explain how we can incorporate these constraints in the mining algorithm. We first consider the special case of constraints where only conjunctions of atomic conditions or their negations are allowed.

3.2.1 Conjunctive Constraints

Let b_1, ..., b_ℓ be the items that must be in the body by the constraint; b'_1, ..., $b'_{\ell'}$ those that must not; h_1, ..., h_m those that must be in the head; and h'_1, ..., $h'_{m'}$ those that must not.

Recall that an association rule $X \Rightarrow Y$ is only generated if $X \cup Y$ is a frequent set. Hence, we only have to generate those frequent sets that contain every b_i and h_i, plus some of the subsets of these frequent sets that can serve as bodies or heads. Therefore we will create a set-query corresponding to the rule-query, which is also a conjunctive expression, but now over the presence or absence of an item i in a frequent set, denoted by $\text{Set}(i)$ and $\neg\text{Set}(i)$. We do this as follows:

1. For each positive literal $\text{Body}(i)$ or $\text{Head}(i)$ in the rule-query, add the literal $\text{Set}(i)$ in the set-query.

2. If for an item i both $\neg\text{Body}(i)$ and $\neg\text{Head}(i)$ are in the rule-query, add the negated literal $\neg\text{Set}(i)$ to the set-query.

3. Add the minimal support threshold to the set-query.

4. All other literals in the rule-query are ignored because they do not restrict the frequent sets that must be generated.

   Formally, the following is readily verified:

Lemma 3.1. *An itemset Z satisfies the set-query if and only if there exists itemsets X and Y such that $X \cup Y = Z$ and the rule $X \Rightarrow Y$ satisfies the rule-query, apart from the confidence threshold.* □

So, once we have generated all sets Z satisfying the set-query, we can generate all rules satisfying the rule-query by splitting all these Z in all possible ways in a body X and a head Y such that the rule-query is satisfied. Lemma 3.1 guarantees that this method is "sound and complete".

So, we need to explain two things:

1. Finding all frequent Z satisfying the set-query.

2. Finding, for each such Z, the frequencies of all bodies and heads X and Y such that $X \cup Y = Z$ and $X \Rightarrow Y$ satisfies the rule-query.

Finding the frequent sets satisfying the set-query Let $Pos := \{i \mid \text{Set}(i) \text{ in set-query}\}$ and $Neg := \{i \mid \neg\text{Set}(i) \text{ in set-query}\}$. Note that $Pos = \{b_1, \ldots, b_\ell, h_1, \ldots, h_m\}$. Denote the data set of transactions by $\mathcal{D}$. We define the following derived data set $\mathcal{D}_0$:

$$\mathcal{D}_0 := \{t - (Pos \cup Neg) \mid t \in \mathcal{D} \text{ and } Pos \subseteq t\}$$

In other words, we ignore all transactions that are not supersets of *Pos* and from all transactions that are not ignored, we remove all items in *Pos* plus all items that are in *Neg*.

We observe:

Lemma 3.2. *Let p be the support threshold defined in the query. Let $\mathcal{S}_0$ be the set of itemsets over the new data set $\mathcal{D}_0$, without any further conditions, except that their support is at least p. Let $\mathcal{S}$ be the set of itemsets over the original data set $\mathcal{D}$ that satisfy the set-query, and whose support is also at least p. Then*

$$\mathcal{S} = \{s \cup Pos \mid s \in \mathcal{S}_0\}.$$

Proof. To show the inclusion from left to right, consider $Z \in \mathcal{S}$. We show that $s := Z - Pos$ is in $\mathcal{S}_0$. Thereto, it suffices to establish an injection $t \mapsto t_0$ from the transactions t in the support set of Z in $\mathcal{D}$ (i.e., the set of all transactions in $\mathcal{D}$ containing Z) into the transactions t_0 in the support set of s in $\mathcal{D}_0$.

Let t be in $\mathcal{D}$ and containing Z. Since Z satisfies the set-query, Z contains Pos, and hence t contains Pos as well. Thus, $t_0 := t - (Pos \cup Neg)$ is in $\mathcal{D}_0$. Since $Z \cap Neg = \emptyset$ (again because Z satisfies the set-query), t_0 contains $Z - Pos = s$. Hence, t_0 is in the support set of s in $\mathcal{D}_0$, as desired.

To show the inclusion from right to left, consider $s \in \mathcal{S}_0$. We show that $Z := s \cup Pos$ is in $\mathcal{S}$. Thereto, it suffices to establish an injection $t_0 \mapsto t$ from the transactions t_0 in the support set of s in $\mathcal{D}_0$ into the transactions t in the support set of Z in $\mathcal{D}$.

Let t_0 be in $\mathcal{D}_0$ and containing s. Obviously, a transaction $t \in \mathcal{D}$ exists, such that $t_0 \cup Pos \subseteq t - Neg \subseteq t$. Since t contains $s \cup Pos$, it is in the support set of Z in $\mathcal{D}$, as desired. □

We can thus perform any frequent set generation algorithm, using only $\mathcal{D}_0$ instead of $\mathcal{D}$. Note that the number of transactions in $\mathcal{D}_0$ is exactly the support of Pos in $\mathcal{D}$. Also, the search space of all itemsets is halved for every item in $Pos \cup Neg$. In practice, the search space of all frequent itemsets is at least halved for every item in *Pos* and at most halved for every item in *Neg*. Still put differently: we are mining in a world where itemsets that do not satisfy the query simply do not exist. The correctness and optimality of our method is thus automatically guaranteed.

Note however that now an itemset I, actually represents the itemset $I \cup Pos$! We thus head-start with a lead of k, where k is the cardinality of *Pos*, in comparison with standard, non-constrained mining.

Finding the frequencies of bodies and heads We now have all frequent sets containing every b_i and h_i, from which rules that satisfy the rule-query can be generated. Recall that in the standard association rule mining algorithm rules are generated by taking every item in a frequent set as a head and the others as body. All heads that result in a confident rule, with respect to the minimal confidence threshold, can then be combined to generate more general rules. But, because we now only want rules that satisfy the query, a head must always be a superset of $\{h_1, \ldots, h_m\}$ and may not include any of the h'_i and b_i (the latter because bodies and heads of rules are disjoint). In this way, we head-start with a lead of m. Similarly, a body must always be a superset of $\{b_1, \ldots, b_\ell\}$ and may not include any of the b'_i and h_i.

The following lemma (which follows immediately from Lemma 3.2) tells us that these potential heads and bodies are already present, albeit implicitly, in $\mathcal{S}_0$:

Lemma 3.3. *Let $\mathcal{S}_0$ be as in Lemma 3.2. Let $\mathcal{B}$ ($\mathcal{H}$) be the set of bodies (heads) of those association rules over $\mathcal{D}$ that satisfy the rule-query. Then*

$$\mathcal{B} = \{s \cup \{b_1, \ldots, b_\ell\} \mid s \in \mathcal{S}_0 \text{ and } s \cap \{b'_1, \ldots, b'_{\ell'}, h_1, \ldots, h_m\} = \emptyset\}$$

and

$$\mathcal{H} = \{s \cup \{h_1, \ldots, h_m\} \mid s \in \mathcal{S}_0 \text{ and } s \cap \{h'_1, \ldots, h'_{m'} b_1, \ldots, b_\ell\} = \emptyset\}.$$

So, for the potential bodies (heads), we use, in $\mathcal{S}_0$, all sets that do not include any of the b'_i and h_i (h'_i and b_i), and add all b_i (h_i). Hence, all we have to do is to determine the frequencies of these subsets by performing one additional scan through the data set. (We do not necessarily yet have these frequencies because these sets do not contain either items b_i or h_i, while we ignored transactions that did not contain all items b_i and h_i.)

Each generated itemset can thus have up to three different "personalities:"

1. A frequent set that satisfies the set-query;
2. A frequent set that can act as body of a rule that satisfies the rule-query;
3. A frequent set that can act as head of a rule that satisfies the rule-query.

Hence, we finally have at most three families of sets, i.e., those sets from which rules must be generated, the *rule-sets* ($\mathcal{S}_0$ with all b_i and h_i added); a family of possible bodies, the *body-sets* ($\mathcal{S}_0$ with all b_i added, minus all those sets that include any of the b'_i and h_i); and yet another family of possible heads, the *head-sets* ($\mathcal{S}_0$ with all h_i added, minus all those sets that include any of the h'_i and b_i). Note that the frequencies of the body-sets and head-sets need not necessarily to be recounted since their frequencies are equal to the frequencies of their corresponding sets in $\mathcal{S}_0$ if the query consists of negated atoms only. We finally generate the desired association rules from the rule-sets, by looking for possible bodies and heads only within the body-sets and head-sets respectively, on condition that they have enough confidence.

$\mathcal{S}_0$	$\mathcal{S}$	$\mathcal{B}$	$\mathcal{H}$
$\{\}$	$\{1,3\}$	$\{1\}$	$\{3\}$
$\{2\}$	$\{1,2,3\}$	-	$\{2,3\}$
$\{4\}$	$\{1,3,4\}$	$\{1,4\}$	-
$\{6\}$	$\{1,3,6\}$	$\{1,6\}$	$\{3,6\}$
$\{8\}$	$\{1,3,8\}$	$\{1,8\}$	$\{3,8\}$
$\{2,6\}$	$\{1,2,3,6\}$	-	$\{2,3,6\}$
$\{4,8\}$	$\{1,3,4,8\}$	$\{1,4,8\}$	-

Table 3.1: An example of generated sets, which can represent a frequent set, as well as a body, as well as a head.

Optimality Note that every rule-set, body-set, and head-set is needed to construct the rules potentially satisfying the rule-query so that these can be tested for confidence, and moreover, no other sets are ever needed. In this precise sense, our method is *optimal*.

Example 3.1. We illustrate our method with an example. Assume we are given the rule-query

$$\begin{aligned}&\text{Body}(1) \wedge \neg\text{Body}(2) \wedge \text{Head}(3) \wedge \neg\text{Head}(4)\\&\qquad \wedge \neg\text{Body}(5) \wedge \neg\text{Head}(5) \wedge \text{support} \geq 1 \wedge \text{confidence} \geq 50\%.\end{aligned}$$

We begin by converting it to the set-query

$$\text{Set}(1) \wedge \text{Set}(3) \wedge \neg\text{Set}(5) \wedge \text{support} \geq 1.$$

Hence $Pos = \{1,3\}$ and $Neg = \{5\}$. Consider a database consisting of the three transactions $\{2,3,5,6,9\}$, $\{1,2,3,5,6\}$ and $\{1,3,4,8\}$. We ignore the first transaction because it is not a superset of Pos. We remove items 1 and 3 from the second transaction because they are in Pos, and we also remove 5 because it is in Neg. We only remove items 1 and 3 from the third transaction. Table 3.1 shows the itemsets that result from the mining algorithm after reading, according to Lemma 3.1 and 3.2, the two resulting transactions. For example, the itemset $\{4,8\}$ actually represents the set $\{1,3,4,8\}$. It also represents a potential body, namely $\{1,4,8\}$, but it does not represent a head, because it includes item 4, which must not be in the head according to the given rule-query. As another example, the empty set now represents the set $\{1,3\}$ from which a rule can be generated. It also represents a potential body and a potential head.

3.2.2 Boolean Constraints

Assume now given a rule-query that is an arbitrary Boolean combination of atomic conditions. We can put it in disjunctive normal form (DNF) and then generate all frequent itemsets for every disjunct (which is a conjunction) in parallel by feeding every transaction of the database to every disjunct, and processing them there as described in the previous subsection.

However, this approach is a bit simplistic, as it might generate some sets and rules multiple times. For example, consider the following query: $\text{Body}(1) \vee \text{Body}(2)$. If we convert it to its corresponding set-query (disjunct by disjunct), we get $\text{Set}(1) \vee \text{Set}(2)$. Then, we would generate for both disjuncts all supersets of $\{1, 2\}$. We can avoid this problem by putting the set-query to *disjoint DNF*.[*] Then, no itemset can satisfy more than one set-disjunct. On the other hand this does not solve the problem of generating some rules multiple times. Consider the equivalent disjoint DNF of the above set-query: $\text{Set}(1) \vee (\text{Set}(2) \wedge \neg\text{Set}(1))$. The first disjunct thus contains the set $\{1, 2\}$ and all of its supersets. If we generate for every itemset all potential bodies and heads according to every rule-disjunct, both rule-disjuncts will still generate all rules with the itemset $\{1, 2\}$ in the body. The easiest way to avoid this problem is to already put the rule-query in disjoint DNF. Obviously, this does not mean its corresponding set-query is also in disjoint DNF, and hence, we still have to put it in disjoint DNF.

After all sets have been generated according to the set-query, we still have to generate all rules according to the rule-query. This can be done for every rule-disjunct (which is a conjunction) in parallel after some modifications to the algorithm described in the previous subsection.

Indeed, a single set-disjunct can now contain sets from which rules can be generated satisfying several rule-disjuncts. Hence, a set generated in one set-disjunct has now possibly even more personalities. More specifically, for every rule disjunct, it can possibly represent a set from which rules can be generated, a body of such a rule and a head of such a rule. We illustrate this with the rule-query given in the previous paragraph.

Example 3.2. Assume we are given the rule-query

$$\text{Body}(1) \wedge (\text{Body}(2) \vee \text{Head}(2)).$$

In disjoint DNF, this gives

$$(\text{Body}(1) \wedge \text{Body}(2)) \vee (\text{Body}(1) \wedge \text{Head}(2)).$$

[*]In disjoint DNF, the conjunction of any two disjuncts is unsatisfiable. Any boolean expression has an equivalent disjoint DNF.

Converted to its corresponding set-query in disjoint DNF, we get

$$\mathrm{Set}(1) \wedge \mathrm{Set}(2).$$

Obviously, this single set-disjunct contains sets from which rules satisfying the first rule-disjunct can be generated. Following the methodology described in the previous subsection, this means we still have to count the frequencies of all these sets without item 1 and item 2 included, since they will occur as heads in the rules satisfying the first rule disjunct. But now, the set-disjunct also contains sets from which rules satisfying the second rule-disjunct can be generated. Hence, we still have to count the frequencies of the generated sets with item 1 included, which can serve as bodies for the rules satisfying the second rule-disjunct, and the sets with item 2 included, which can serve as heads.

Until now, we have disregarded the possible presence of negated thresholds in the queries, which can come from the conversion to disjoint DNF, or from the user himself. In the latter case, it would not be possible to exploit this constraint in an Apriori-like algorithm, because it is an essentially bottom-up algorithm. Algorithms that generate sets also in a top-down strategy could exploit this constraint. Another source for negated thresholds is the conversion from the user's query to a disjoint DNF formula. Before we discuss this, we first have to explain how we are going to convert a given formula to disjoint DNF.

We first put the Boolean expression ϕ in DNF, obtaining an expression of the form $\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n$, in which ϕ_i is a conjunction of atomic conditions or their negations. Of course, any two of these disjuncts may not be disjoint. A good way to obtain a disjoint DNF is to add to every disjunct ϕ_i the negated disjuncts ϕ_j with $j < i$. We thus become the equivalent formula $\phi_1 \vee (\phi_2 \wedge \neg\phi_1) \vee \cdots \vee (\phi_n \wedge \neg\phi_{n-1} \wedge \cdots \wedge \neg\phi_1)$ in which all disjuncts are pairwise disjoint. Our problem is not yet solved, because our formula is not even in DNF anymore. We thus still have to convert every disjunct on itself to disjoint DNF. For example, take $(\phi_2 \wedge \neg\phi_1)$ with $\phi_1 \equiv p_1 \wedge p_2 \wedge \cdots \wedge p_\ell$ in which p_i is an atomic condition or its negation. The disjunct thus becomes $(\phi_2 \wedge \neg p_1) \vee (\phi_2 \wedge p_1 \wedge \neg p_2) \vee \cdots \vee (\phi_2 \wedge p_1 \wedge p_2 \wedge \cdots \wedge p_{\ell-1} \wedge \neg p_\ell)$, which is in disjoint DNF.

An example showing that negated thresholds can be introduced in this process, is the following.

Example 3.3. Assume we are given the rule-query

$$(\mathrm{Body}(1) \wedge \mathrm{support} \geq 10) \vee (\mathrm{Body}(2) \wedge \mathrm{support} \geq 5).$$

As equivalent disjoint DNF, we obtain

$$\begin{aligned}(\text{Body}(1) \wedge \text{support} \geq 10) \vee (\text{Body}(2) \wedge \text{support} \geq 5 \wedge \neg\text{Body}(1))\\ \vee (\text{Body}(2) \wedge \text{support} \geq 5 \wedge \text{Body}(1) \wedge \text{support} < 10).\end{aligned}$$

Notice the maximal support threshold in the last disjunct, which is needed to avoid generating itemsets satisfying Body(2) $\wedge$ support $\geq 10 \wedge$ Body(1) that are already generated by the first disjunct.

Negated support thresholds can be avoided however. After putting the user's formula in DNF, but before putting the DNF in disjoint DNF, we sort all disjuncts on their support threshold, in ascending order. This guarantees that the conversion to disjoint DNF does not introduce any negated support thresholds.

Note that we cannot avoid negated *confidence* thresholds at the same time: we have already sorted on support, and thus cannot sort anymore on confidence at the same time. Since we are here already in phase 2, it is less of an efficiency issue to just ignore maximal confidence thresholds.

Furthermore, if a set-disjunct (rule-disjunct) consists of nothing but a negated support (confidence) threshold, we can of course easily switch the generation algorithm and generate the candidate sets (heads) in a top-down manner.

3.2.3 Experimental Evaluation

For our experiments, we have implemented an extensively optimized version of the Apriori algorithm, equipped with the querying optimizations as described in the previous sections.

For each data set, we generated 100 random Boolean queries consisting of at most three atomic conditions. Figure 3.1 shows the improvement on the performance of the algorithm exploiting the constraints. The y-axis shows the time needed for the algorithm exploiting our queries, relative to the time needed without exploiting the queries. The x-axis shows the number of patterns satisfying the given query, relative to the total number of patterns. As can be seen, the time improvement is proportional to the selectivity of the constraints. Notice that the proportionality factor is around 1.

3.3 Interactive Mining

3.3.1 Integrated Querying or Post-Processing?

In the previous section, we have seen a way to integrate constraints tightly into the mining of association rules. We call this *integrated querying*. At the

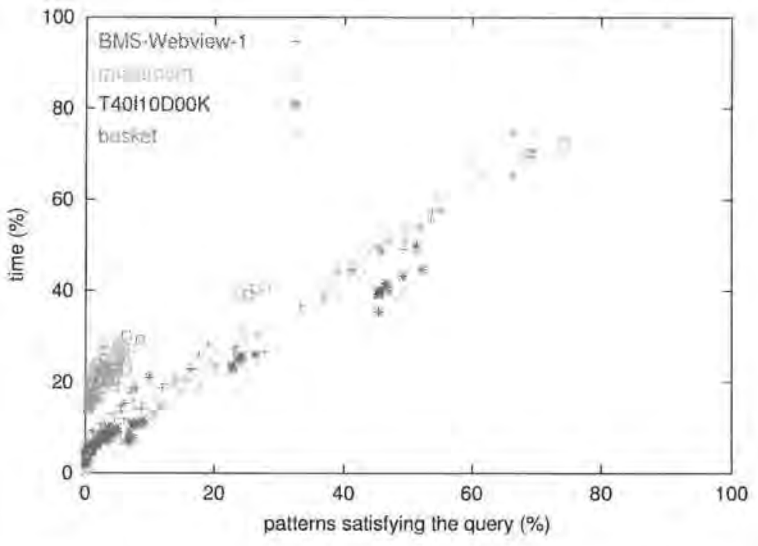


Figure 3.1: Improvement after exploiting constraints.

other end of the spectrum we have *post-processing*, where we perform standard, non-constrained mining, save the resulting itemsets and rules, and then query those results for the constraints.

Integrated querying has the following two obvious advantages over post-processing:

1. Answering one single data mining query using integrated querying is much more efficient than answering it using post-processing.

2. It is well known that, by setting parameters such as minimal support too low, or by the nature of the data, association rule mining can be infeasible, simply because of a combinatorial explosion involved in the generation of rules or frequent itemsets. Under such circumstances, of course, post-processing is infeasible as well; yet, integrated querying can still be executed, if the query conditions can be effectively exploited to reduce the number of itemsets and rules from the outset.

However, as already mentioned in the introduction, data mining query language environments must support an interactive, iterative mining process, where a user repeatedly issues new queries based on what he found in the answers of his previous queries. Now consider a situation where minimal support requirements and data set particulars are favorable enough so that post-processing is not infeasible to begin with. Then the global, non-constrained mining operation, on the result of which the querying will be performed by post-processing, *can be executed once and its result materialized for the remainder of the data mining session.*

In that case, if the session consists of, say, 20 data mining queries, these 20 queries amount to standard retrieval queries on the materialized mining results. In contrast, answering every single one of the 20 queries by integrated querying will involve at least one, and often many more, passes over the data, as each query involves a separate mining operation. Also, several queries could have a non-empty intersection, such that a lot of work is repeated several times. Hence, the total time needed to answer the integrated queries is guaranteed to grow beyond the post-processing total time.

The naively conceived advantages of integrated querying over post-processing become much less clear now. Indeed, if the number of data mining queries issued by the user is large enough, then the post-processing approach clearly outperforms the integrated querying approach. We have performed several experiments which all confirmed this predicted effect. For the post-processing approach, we only materialized all frequent itemsets, since the time needed to generate all association rules that satisfy the query turned out to be as fast as finding all such rules from the materialized results. Moreover, storing all frequent and confident association rules requires huge storage capabilities, and hence, it is preferable to generate the necessary association rules on the fly. Figure 3.2 shows the total time needed for answering up to 20 different queries on the BMS-Webview-1 data set. Since the time needed to generate all association rules is the same for both approaches, we only recorded the time to generate all itemsets that were needed to generate all association rules. The queries were randomly generated, only those queries with an empty output were replaced, but all used the same support threshold as was used for the initial mining operation of the post-processing approach. As can be seen, the cut-off point from where the post-processing approach outperforms the integrated querying approach occurs already after the eighth query.

3.3.2 Incremental Querying: Basic Approach

From the above discussion it is clear that we should try to combine the advantages of integrated querying and post-processing. We now introduce such an approach, which we call *incremental querying*.

In the incremental approach, all itemsets that result from every posed query, as well as all intermediate generated itemsets, are stored into a cache. Initially, when the user issues his first query, nothing has been mined yet, and thus we answer it using integrated querying.

Every subsequent query is first converted to its corresponding rule- and set-query in disjoint DNF. For every disjunct in the set-query, the system adds all currently cached itemsets that satisfy the disjunct to the data structure holding

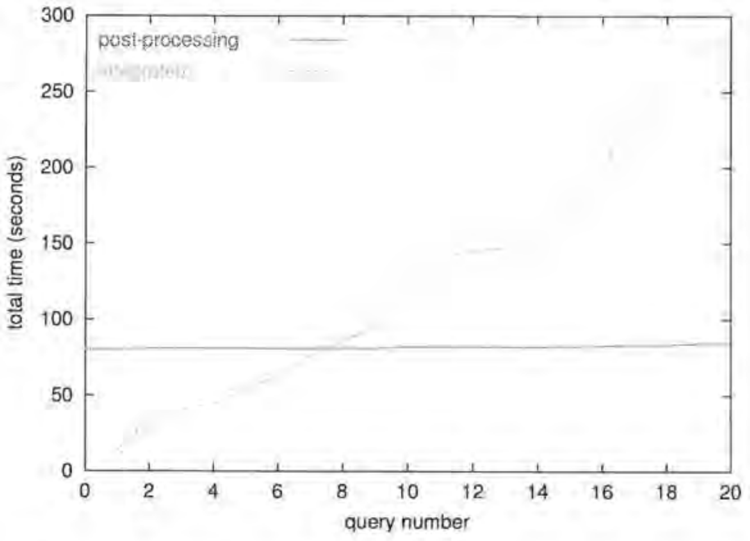


Figure 3.2: Integrated querying versus post-processing.

itemsets, that is used for mining that disjunct, as well as all of its subsets that satisfy the disjunct (note that these subsets may not all be cached; if they are not, we have to count their supports during the first scan through the data set). We also immediately add all candidate itemsets.

If no new candidate itemsets can be generated, which means that all necessary itemsets were already cached, we are done. However, if this is not the case, we can now begin our queried mining algorithm with the important generalization that in each iteration, candidate itemsets of different cardinalities are now generated. In order for this to work, candidate itemsets that turn out to be infrequent must be kept such that they are not regenerated in later iterations. This generalization was first used by Toivonen in his sampling algorithm [77].

Caching all generated itemsets gives us another advantage that can be exploited by the integrated querying algorithm. Consider a set-query stating that items 1 and 2 must be in the itemsets. In the first iteration of the algorithm, all single itemsets are generated as candidate sets over the new data set $\mathcal{D}_0$ (cf. Section 3.2.1). We explained that these single itemsets actually represent supersets of $\{1,2\}$. Normally, before we generate a candidate itemset, we check if all of its subsets are frequent. Of course, this is impossible if these subsets do not even exist in $\mathcal{D}_0$. Now, however, we can check in the cache for a subset with too low support; if we find this, we avoid generating the candidate.

We thus obtain an algorithm which reuses previously generated itemsets as if they had been generated in previous iterations of the algorithm. We are optimal in the sense that we never generate and test itemsets that were generated before. For rule generation, we again did not cache the results, but instead generated all association rules when needed for the same reasons as explained in the previous subsection.

In the worst case, the cached results do not contain anything that can be reused for answering a query, and hence the time needed to generate the itemsets and rules that satisfy the query is equal to the time needed when answering that query using the integrated querying approach. In the best case, all requested itemsets are already cached, and hence the time needed to find all itemsets and rules that satisfy the query is equal to the time needed for answering that query using post-processing. In the average case, part of the needed itemsets are cached and will then be used to speed up the integrated querying approach. If the time gained by this speedup is more than the time needed to find the reusable sets, then the incremental approach will always be faster than the integrated querying approach. In the limit, all itemsets will be materialized, and hence all subsequent queries will be answered using post-processing.

3.3.3 Incremental Querying: Overhead

Could it be that the time gained by the speedup in the integrated querying approach is less than the time needed to find and reuse the reusable itemsets? This could happen when a lot of itemsets are already cached, but almost none of them satisfy the constraints. It is also possible that the reusable itemsets give only a marginal improvement. We can however counter this phenomenon by estimating what is currently cached, as follows.

We keep track of a set-query ϕ_{sets} which describes the stored sets. This query is initially *false*. Given a new query (rule-query) ψ, the system now goes through the following steps: (step 1 was described in Section 3.2.1)

1. Convert the rule-query ψ to the set-query ϕ
2. $\phi_{mine} := \phi \wedge \neg\phi_{sets}$
3. $\phi_{sets} := \phi_{sets} \vee \phi$

After this, we perform:

1. Generate all frequent sets according to ϕ_{mine}, using the basic incremental approach.
2. Retrieve all cached sets satisfying $\phi \wedge \neg\phi_{mine}$.
3. Add all needed subsets that can serve as bodies or heads.

4. Generate all rules satisfying ψ.

Note that the query ϕ_{mine} is much more specific than the original query ϕ. We thus obtain a speedup, because we have shown in Section 3.2 that the speed of integrated querying is proportional to the selectivity of the query.

3.3.4 Avoiding Exploding Queries

The improvement just described incurs a new problem. The formula ϕ_{sets} becomes longer with the session. When, given the next query ϕ, we mine for $\phi \wedge \neg\phi_{sets}$, and convert this to disjoint DNF which could explode.

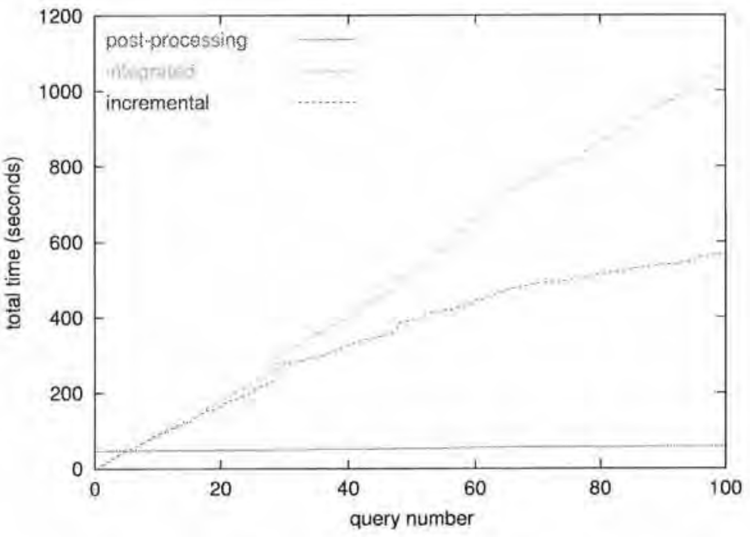
To avoid this, consider ϕ_{sets} in DNF: $\phi_1 \vee \cdots \vee \phi_n$. Instead of the full query $\phi \wedge \neg\phi_{sets}$, we are going to use a query $\phi \wedge \neg\phi'_{sets}$, where ϕ'_{sets} is obtained from ϕ_{sets} by keeping only the least selective disjuncts ϕ_i (their negation will thus be most selective). In this way $\phi \wedge \neg\phi'_{sets}$ is kept short.

But how do we measure selectiveness of a ϕ_i? Several heuristics come to mind. A simple one is to keep for each ϕ_i the number of cached sets that satisfy it. These numbers can be maintained incrementally.

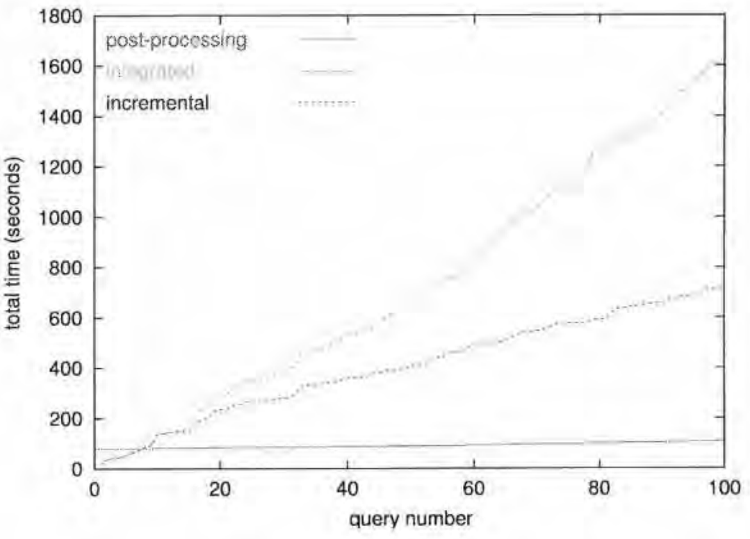
3.3.5 Experimental Evaluation

For each data set, we experimented with a session of 100 queries using the integrated querying approach, the post-processing approach and the incremental approach. For the same reasons as explained in the previous section, we only show the time needed to generate the necessary itemsets. Again, the queries used for the sessions were randomly generated. Figure 3.3 shows the evolution of the sessions in time.

For all four sessions, the cut-off point where the integrated querying approach loses against the post-processing approach is the same for the incremental querying approach since not enough itemsets could be reused before that. Except for the mushroom data set, the incremental approach starts paying off after the twentieth query. However, as can be seen, the incremental approach shows a significant improvement on the integrated querying approach. Only for the mushroom data set, the cut-off point occurs at the fifth query, and almost all itemsets have been generated after the eighteenth query. As can be seen, the performance of the post-processing approach is very good compared to the other approaches. Nevertheless, if we still lowered the support thresholds, the post-processing approach became infeasible to begin with, due to an overload of frequent itemsets. In that case, the integrated and incremental approach are still feasible and perform very similarly as they do in the presented experiments.

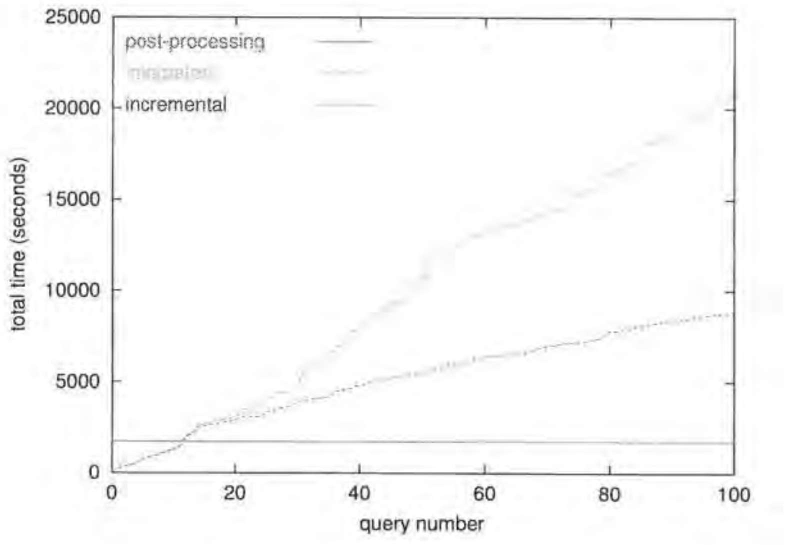


(a) basket

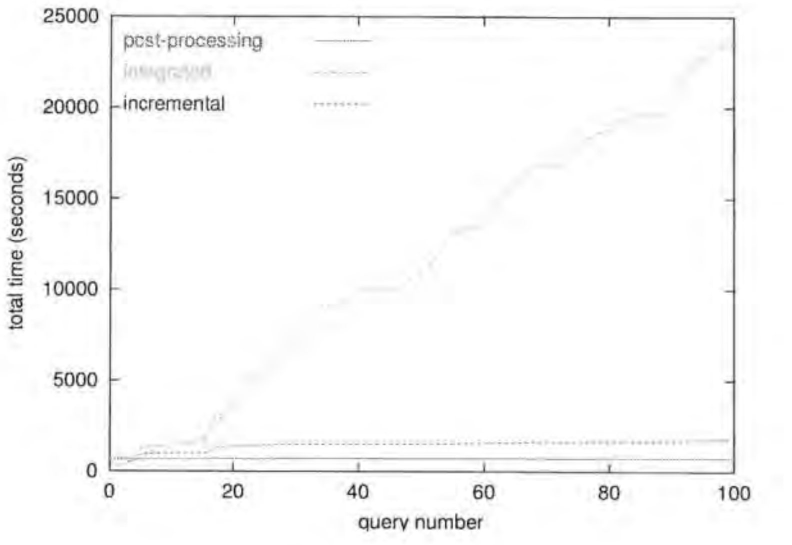


(b) BMS-Webview-1

Figure 3.3: Performance comparison of the three approaches.



(c) T40I10D100K



(d) mushroom

Figure 3.3: Performance comparison of the three approaches.

3.4 Conclusions

This study revealed several insights into the association rule mining problem. First, due to recent advances on association rule mining algorithms, the performance has been significantly improved, such that the advantages of integrating constraints into the mining algorithm suddenly become less clear. Indeed, we showed that as long as mining without any constraints is feasible, that is, if the number of frequent itemsets does not reach massive amounts, the total time spent to query the frequent itemsets and confident association rules becomes less after a certain amount of queries, compared to integrated querying, in which every query is pushed into the mining algorithm. The incremental approach still improves the integrated approach by reusing as much previously generated results as possible. If the cut-off point would lie beyond the number of queries in which the user is interested, the incremental approach is obviously the best choice to use.

Of course, if the user is interested in some frequent itemsets and association rules which have very low frequencies, and hence mining without any constraints becomes infeasible, the incremental approach can still be performed.

Also note that if a user is still interested in all frequent sets and association rules, but mining without constraints is infeasible, our queries can be used to divide the task over several runs, without spending much more time. For example, one can ask different queries of which the disjunction still gives all sets and rules. Essentially, this technique forms the basis of the Eclat [80] and FP-growth [41] algorithms.

4 Tight Upper Bounds on the Number of Candidate Patterns

Several improvements on the Apriori algorithm try to reduce the number of scans through the database by estimating the number of candidate patterns that can still be generated.

At the heart of all these techniques lies the following purely combinatorial problem, that must be solved first before we can seriously start applying them: *given the current set of frequent patterns at a certain iteration of the algorithm, what is the maximal number of candidate patterns that can be generated in the iterations yet to come?*

Our contribution is to solve this problem by providing a hard and tight combinatorial upper bound. By computing our upper bound after every iteration of the algorithm, we have at all times an airtight guarantee on the size of what is still to come, on which we can then base various optimization decisions, depending on the specific algorithm that is used.

In the next section, we will discuss existing techniques to reduce the number of database scans, and point out the dangers of using existing heuristics for this purpose. Using our upper bound, these techniques can be made watertight. In Section 4.2, we derive our upper bound, using a combinatorial result from the sixties by Kruskal and Katona. In Section 4.3, we show how to get even more out of this upper bound by applying it recursively. We will then generalize the given upper bounds such that they can be applied by a wider range of algorithms in Section 4.4. In Section 4.5, we discuss several issues concerning the implementation of the given upper bounds on top of Apriori-like algorithms. In Section 4.6, we give experimental results, showing the effectiveness of our result in estimating, far ahead, how much will still be generated in the future. Finally, we conclude the chapter in Section 4.7.

Bibliographical note Parts of this chapter have been published before in the Proceedings of the 2001 IEEE International Conference on Data Mining [30].

4.1 Related Work

Nearly all frequent pattern mining algorithms developed after the proposal of the Apriori algorithm, rely on its levelwise candidate generation and pruning strategy. Most of them differ in how they generate and count candidate patterns as we described in Chapter 2.

Several strategies try to reduce the number of scans through the database. However, such a reduction often causes an increase in the number of candidate patterns that need to be explored during a single scan. This tradeoff between the reduction of scans and the number of candidate patterns is important since, as we recall, the time needed to process a transaction is dependent on the number of candidates that are covered in that transaction, which might blow up exponentially. Our upper bound can be used to predict whether or not this blowup will occur.

The Partition algorithm, proposed by Savasere et al. [70], reduces the number of database scans to two. Nevertheless, its performance is heavily dependent on the distribution of the data, and could generate much too many candidates.

The Sampling algorithm proposed by Toivonen [77] performs at most two scans through the database by first mining a random sample from the database. In the cases where this sample does not produce all frequent patterns, the missing patterns can be found by generating all remaining potentially frequent patterns and verifying their frequencies during a second scan through the database. The probability of such a failure can be kept small by decreasing the minimal support threshold. However, for a reasonably small probability of failure, the threshold must be drastically decreased, which can again cause a combinatorial explosion of the number of candidate patterns.

Other successful algorithms attempt to generate frequent patterns using a depth-first search. Generating patterns in a depth-first manner implies that the monotonicity property cannot be exploited anymore. Hence, a lot more patterns will be generated and need to be counted, compared to the breadth-first algorithms.

The first heuristic specifically proposed to estimate the number of candidate patterns that can still be generated was used in the AprioriHybrid algorithm [6, 4], as we explained in Chapter 2. This algorithm uses Apriori in the initial iterations and switches to AprioriTid if it expects it to run faster. This AprioriTid algorithm does not use the database at all for counting the support of candidate patterns. Rather, an encoding of the candidate patterns used in the previous iteration is employed for this purpose. The AprioriHybrid algorithm switches to AprioriTid when it expects this encoding of the candidate patterns to be small enough to fit in main memory. The size of the encoding grows with the number of candidate patterns. Therefore, it calculates the size the encoding would have in the current iteration. If this size is small enough and there are fewer candidate patterns in the current iteration than the in previous iteration, the heuristic decides to switch to AprioriTid.

This heuristic (like all heuristics) is not waterproof, however. Take, for example, two disjoint data sets. The first data set consists of all subsets of a frequent pattern of size 20. The second data set consists of all subsets of 1 000 disjoint frequent patterns of size 5. If we merge these two data sets, we get $\binom{20}{3} + 1\,000\binom{5}{3} = 11\,140$ patterns of size 3 and $\binom{20}{4} + 1\,000\binom{5}{4} = 9\,845$ patterns of size 4. If we have enough memory to store the encoding for all these patterns, then the heuristic decides to switch to AprioriTid. This decision is premature, however, because the number of new patterns in each pass will start growing exponentially afterwards.

Another improvement of the Apriori algorithm, which is part of the folklore, tries to combine as many iterations as possible in the end, when only few candidate patterns can still be generated. The potential of such a combination technique was realized early on [6], but the modalities under which it can be applied were never further examined. Our work does exactly that.

4.2 The Basic Upper Bounds

In all that follows, L is some family of patterns of size k.

Definition 4.1. A *candidate pattern* for L is a pattern (of size larger than k) of which all k-subsets are in L. For a given $p > 0$, we denote the set of all size-$k+p$ candidate patterns for L by $C_{k+p}(L)$.

For any $p \geq 1$, we will provide an upper bound on $|C_{k+p}(L)|$ in terms of $|L|$. The following lemma is central to our approach. (A simple proof was given by Katona [51].)

Lemma 4.1. *Given n and k, there exists a unique representation*

$$n = \binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_r}{r},$$

with $r \geq 1$, $m_k > m_{k-1} > \ldots > m_r$, and $m_i \geq i$ for $i = r, r+1, \ldots, k$.

This representation is called the *k-canonical representation of n* and can be computed as follows: The integer m_k satisfies $\binom{m_k}{k} \leq n < \binom{m_k+1}{k}$, the integer m_{k-1} satisfies $\binom{m_{k-1}}{k-1} \leq n - \binom{m_k}{k} < \binom{m_{k-1}+1}{k-1}$, and so on, until $n - \binom{m_k}{k} - \binom{m_{k-1}}{k-1} - \cdots - \binom{m_r}{r}$ is zero.

We now establish:

Theorem 4.2. *If*

$$|L| = \binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_r}{r}$$

in k-canonical representation, then

$$|C_{k+p}(L)| \leq \binom{m_k}{k+p} + \binom{m_{k-1}}{k-1+p} + \cdots + \binom{m_{s+1}}{s+p+1},$$

where s is the smallest integer such that $m_s < s+p$. If no such integer exists, we set $s = r-1$.

Proof. Suppose, for the sake of contradiction, that

$$|C_{k+p}(L)| \geq \binom{m_k}{k+p} + \binom{m_{k-1}}{k-1+p} + \cdots + \binom{m_{s+1}}{s+p+1} + \binom{s+p}{s+p}.$$

Note that this is in $k+p$-canonical representation. A theorem by Kruskal and Katona [29, 51, 53] says that

$$|L| \geq \binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_{s+1}}{s+1} + \binom{s+p}{s}.$$

But this is impossible, because

$$\begin{aligned}
|L| &= \binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_{s+1}}{s+1} + \binom{m_s}{s} + \cdots + \binom{m_r}{r} \\
&\leq \binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_{s+1}}{s+1} + \sum_{1 \leq i \leq s} \binom{i+p-1}{i} \\
&< \binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_{s+1}}{s+1} + \sum_{0 \leq i \leq s} \binom{i+p-1}{i} \\
&= \binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_{s+1}}{s+1} + \binom{s+p}{s}.
\end{aligned}$$

The first inequality follows from the observation that $m_s \leq s+p-1$ implies $m_i \leq i+p-1$ for all $i = s, s-1, \ldots, r$. The last equality follows from a well-known binomial identity. □

Notation We will refer to the upper bound provided by the above theorem as $KK_k^{k+p}(|L|)$ (for Kruskal-Katona). The subscript k, the level at which we are predicting, is important, as the only parameter is the cardinality $|L|$ of L, not L itself. The superscript $k+p$ denotes the level we are predicting.

Proposition 4.3 (Tightness). *The upper bound provided by Theorem 4.2 is* tight: *for any given n and k there always exists an L with $|L| = n$ such that for any given p, $|C_{k+p}(L)| = KK_k^{k+p}(|L|)$.*

Proof. Let us write a finite set of natural numbers as a string of natural numbers by writing its members in decreasing order. We can then compare two such sets by comparing their strings in lexicographic order. The resulting order on the sets is known as the *colexicographic* (or *colex*) order. An intuitive proof of the Kruskal-Katona theorem, based on this colex order, was given by Bollobás [12]. Let

$$\binom{m_k}{k} + \binom{m_{k-1}}{k-1} + \cdots + \binom{m_r}{r}$$

be the k-canonical representation of n. Then, Bollobás has shown that all $k-p$-subsets of the first n k-sets of natural numbers in colex order, are exactly the first

$$\binom{m_k}{k-p} + \binom{m_{k-1}}{k-1-p} + \cdots + \binom{m_s}{r-s}$$

$k-p$-sets of natural numbers in colex order, with s the smallest integer such that $s > p$. Using the same reasoning as above, we can conclude that all $k+p$-supersets of the first n k-sets of natural numbers in colex order are exactly the first $KK_k^{k+p}(n)$ $k+p$-sets of natural numbers in colex order. □

Analogous tightness properties hold for all upper bounds we will present in this chapter, but we will no longer explicitly state this.

Example 4.1. Let L be the set of 13 patterns of size 3:

$$\{\{3,2,1\},\{4,2,1\},\{4,3,1\},\{4,3,2\},$$
$$\{5,2,1\},\{5,3,1\},\{5,3,2\},\{5,4,1\},\{5,4,2\},\{5,4,3\},$$
$$\{6,2,1\},\{6,3,1\},\{6,3,2\}\}.$$

The 3-canonical representation of 13 is $\binom{5}{3} + \binom{3}{2}$ and hence the maximum number of candidate patterns of size 4 is $KK_3^4(13) = \binom{5}{4} + \binom{3}{3} = 6$ and the maximum number of candidate patterns of size 5 is $KK_3^5(13) = \binom{5}{5} = 1$. This is tight indeed, because

$$C_4(L) = \{\{4,3,2,1\},\{5,3,2,1\},\{5,4,2,1\},$$
$$\{5,4,3,1\},\{5,4,3,2\},\{6,3,2,1\}\}$$

and

$$C_5(L) = \{\{5, 4, 3, 2, 1\}\}.$$

Estimating the number of levels The k-canonical representation of $|L|$ also yields an upper bound on the maximal size of a candidate pattern, denoted by $\text{maxsize}(L)$. Recall that this size equals the number of iterations the standard Apriori algorithm will perform. Indeed, since $|L| < \binom{m_k+1}{k}$, there cannot be a candidate pattern of size $m_k + 1$ or higher, so:

Proposition 4.4. *If $\binom{m_k}{k}$ is the first term in the k-canonical representation of $|L|$, then* $\text{maxsize}(L) \leq m_k$.

We denote this number m_k by $\mu_k(|L|)$. From the form of KK_k^{k+p} as given by Theorem 4.2, it is immediate that μ also tells us the last level before which KK becomes zero. Formally:

Proposition 4.5.

$$\mu_k(|L|) = k + \min\{p \mid KK_k^{k+p}(|L|) = 0\} - 1.$$

Estimating all levels As a result of the above, we can also bound, at any given level k, the *total* number of candidate patterns that can be generated, as follows:

Proposition 4.6. *The total number of candidate patterns that can be generated from a set L of k-patterns is at most*

$$KK_k^{\text{total}}(|L|) := \sum_{p \geq 1} KK_k^{k+p}(|L|).$$

4.3 Improved Upper Bounds

The upper bound KK on itself is neat and simple as it takes as parameters only two numbers: the current size k, and the number $|L|$ of current frequent patterns. However, in reality, when we have arrived at a certain level k, we do not merely have the cardinality: we have the actual set L of current k-patterns! For example, if the frequent patterns in the current pass are all disjoint, our current upper bound will still estimate their number to a certain non-zero figure. However, by the pairwise disjointness, it is clear that no further patterns will be possible at all. In sum, because we have richer information than a mere cardinality, we should be able to get a better upper bound.

To get inspiration, let us recall that the candidate generation process of the Apriori algorithm works in two steps. In the *join* step, we join L with itself to obtain a superset of C_{k+1}. The union $p \cup q$ of two patterns $p, q \in L$ is inserted in C_{k+1} if they share their $k-1$ smallest items:

```
insert into $C_{k+1}$
select $p[1], p[2], \ldots, p[k], q[k]$
from $L_k$ $p$, $L_k$ $q$
where $p[1] = q[1], \ldots, p[k-1] = q[k-1], p[k] < q[k]$
```

Next, in the *prune* step, we delete every pattern $c \in C_{k+1}$ such that some k-subset of c is not in L.

Let us now take a closer look at the join step from another point of view. Consider a family of all frequent patterns of size k that share their $k-1$ smallest items, and let its cardinality be n. If we now remove from each of these patterns all these shared $k-1$ smallest items, we get exactly n distinct single-item patterns. The number of pairs that can be formed from these single items, being $\binom{n}{2}$, is exactly the number of candidates the join step will generate for the family under consideration. We thus get an obvious upper bound on the total number of candidates by taking the sum of all $\binom{n_f}{2}$, for every possible family f.

This obvious upper bound on $|C_{k+1}|$, which we denote by $obvious_{k+1}(L)$, can be recursively computed in the following manner. Let I denote the set of items occurring in L. For an arbitrary item x, define the set L^x as

$$L^x = \{s - \{x\} \mid s \in L \text{ and } x = \min s\}.$$

Then

$$obvious_{k+1}(L) := \begin{cases} \binom{|L|}{2} & \text{if } k = 1; \\ \sum_{x \in I} obvious_k(L^x) & \text{if } k > 1. \end{cases}$$

This upper bound is much too crude, however, because it does not take the prune step into account, only the join step. The join step only checks two k-subsets of a potential candidate instead of all $k+1$ k-subsets.

However, we can generalize this method such that more subsets will be considered. Indeed, instead of taking a family of all frequent patterns sharing their $k-1$ smallest items, we can take all frequent patterns sharing only their k' smallest items, for some $k' \leq k-1$. If we then remove these k' shared items from each pattern in the family, we get a new set L' of n patterns of size $k - k'$. If we now consider the set C' of candidates (of size $k - k' + 1$) for L', and add back to each of them the previously removed k' items, we obtain a pruned set of candidates of size $k+1$, where instead of just two (as in the join step), $k - k' + 1$ of the k-subsets were checked in the pruning. Note that we can get the estimate $KK_{k-k'}^{k-k'+1}(|L'|)$ on the cardinality of C' from our upper bound Theorem 4.2.

Doing this for all possible values of k' yields an improved upper bound on $|C_{k+1}|$, which we denote by $\mathit{improved}_{k+1}(L)$, and which is computed by refining the recursive procedure for the obvious upper bound as follows:

$$\mathit{improved}_{k+1}(L) := \begin{cases} \binom{|L|}{2} & \text{if } k = 1; \\ \min\{KK_k^{k+1}(|L|), \sum_{x \in I} \mathit{improved}_k(L^x)\} & \text{if } k > 1. \end{cases}$$

Actually, as in the previous section, we can do this not only to estimate $|C_{k+1}|$, but also more generally to estimate $|C_{k+p}|$ for any $p \geq 1$. Henceforth we will denote our general improved upper bound by $KK^*_{k+p}(L)$. The general definition is as follows:

$$KK^*_{k+p}(L) := \begin{cases} KK_k^{k+p}(|L|) & \text{if } k = 1; \\ \min\{KK_k^{k+p}(|L|), \sum_{x \in I} KK^*_{k+p-1}(L^x)\} & \text{if } k > 1. \end{cases}$$

(For the base case, note that $KK_k^{k+p}(|L|)$, when $k = 1$, is nothing but $\binom{|L|}{p+1}$.)

By definition, KK^*_{k+p} is always smaller than KK_k^{k+p}. We now prove formally that it is still an upper bound on the number of candidate patterns of size $k + p$:

Theorem 4.7.

$$|C_{k+p}(L)| \leq KK^*_{k+p}(L).$$

Proof. By induction on k. The base case $k = 1$ is clear. For $k > 1$, it suffices to show that for all $p > 0$

$$C_{k+p}(L) \subseteq \bigcup_{x \in I} C_{k+p-1}(L^x) + x. \tag{4.1}$$

(For any set of patterns H, we denote $\{h \cup \{x\} \mid h \in H\}$ by $H + x$.)

From the above containment we can conclude

$$\begin{aligned} |C_{k+p}(L)| &\leq |\bigcup_{x \in I} C_{k+p-1}(L^x) + x| \\ &\leq \sum_{x \in I} |C_{k+p-1}(L^x) + x| \\ &= \sum_{x \in I} |C_{k+p-1}(L^x)| \\ &\leq \sum_{x \in I} KK^*_{k+p-1}(L^x) \end{aligned}$$

where the last inequality is by induction.

To show (4.1), we need to show that for every $p > 0$ and every $s \in C_{k+p}(L)$, $s - \{x\} \in C_{k+p-1}(L^x)$, where $x = \min s$. This means that every subset of $s - \{x\}$ of size $k - 1$ must be an element of L^x. Let $s - \{x\} - \{y_1, \dots, y_p\}$ be such a subset. This subset is an element of L^x iff $s - \{y_1, \dots, y_p\} \in L$ and $x = \min(s - \{y_1, \dots, y_p\})$. The first condition follows from $s \in C_{k+p}(L)$, and the second condition is trivial. Hence the theorem. □

A natural question is why we must take the minimum in the definition of KK^*. The answer is that the two terms of which we take the minimum are incomparable. The example of an L where all patterns are pairwise disjoint, already mentioned in the beginning of this section, shows that, for example, $KK_k^{k+1}(|L|)$ can be larger than the summation $\sum_{x \in I} KK_k^*(L^x)$. But the converse is also possible: consider $L = \{\{1,2\},\{1,3\}\}$. Then $KK_2^3(L) = 0$, but the summation yields 1.

Example 4.2. Let L consist of all 19 3-subsets of $\{1,2,3,4,5\}$ and $\{3,4,5,6,7\}$ plus the sets $\{5,7,8\}$ and $\{5,8,9\}$. Because $21 = \binom{6}{3} + \binom{2}{2}$, we have $KK_3^4(21) = 15$, $KK_3^5(21) = 6$ and $KK_3^6(21) = 1$. On the other hand,

$$\begin{aligned}
KK_4^*(L) &= KK_3^*(L^1) + KK_3^*(L^2) + KK_3^*(L^3) + KK_3^*(L^4) \\
&\quad + KK_2^*((L^5)^6) + KK_2^*((L^5)^7) + KK_2^*((L^5)^8) + KK_2^*((L^5)^9) \\
&\quad + KK_3^*(L^6) + KK_3^*(L^7) + KK_3^*(L^8) + KK_3^*(L^9) \\
&= 4 + 1 + 4 + 1 + 0 + \cdots + 0 \\
&= 10
\end{aligned}$$

and

$$\begin{aligned}
KK_5^*(L) &= KK_4^*(L^1) + KK_4^*(L^2) + KK_4^*(L^3) + KK_4^*(L^4) \\
&\quad + KK_3^*((L^5)^6) + KK_3^*((L^5)^7) + KK_3^*((L^5)^8) + KK_3^*((L^5)^9) \\
&\quad + KK_4^*(L^6) + KK_4^*(L^7) + KK_4^*(L^8) + KK_4^*(L^9) \\
&= 1 + 0 + 1 + 0 + 0 + \cdots + 0 \\
&= 2.
\end{aligned}$$

Indeed, we have 10 4-subsets of $\{1,2,3,4,5\}$ and $\{3,4,5,6,7\}$, and the two 5-sets themselves.

We can also improve the upper bound $\mu_k(|L|)$ on maxsize(L). In analogy with Proposition 4.5, we define:

$$\mu_k^*(L) := k + \min\{p \mid KK_{k+p}^*(L) = 0\} - 1.$$

We then have:

Proposition 4.8.

$$\operatorname{maxsize}(L) \leq \mu_k^*(L) \leq \mu_k(L).$$

We finally use Theorem 4.7 for improving the upper bound KK_k^{total} on the total number of candidate patterns. We define:

$$KK^*_{\text{total}}(L) := \sum_{p \geq 1} KK^*_{k+p}(L).$$

Then we have:

Proposition 4.9. *The total number of candidate patterns that can be generated from a set L of k-patterns is bounded by $KK^*_{\text{total}}(L)$. Moreover,*

$$KK^*_{\text{total}}(L) \leq KK_k^{\text{total}}(L).$$

4.4 Generalized Upper Bounds

The upper bounds presented in the previous sections work well for algorithms that generate and test candidate patterns of one specific size at a time. However, a lot of algorithms generate and test patterns of different sizes within the same pass of the algorithm [16, 9, 77]. Hence, these algorithms know in advance that several patterns of size larger than k are frequent or not. Since our upper bound is solely based on the patterns of a certain length k, it does not use information about patterns of length larger than k.

Nevertheless, these larger sets could give crucial information. More specifically, suppose we have generated all frequent patterns of size k, and we also already know in advance that a certain set of size larger than k is not frequent. Our upper bound on the total number of candidate patterns that can still be generated, would disregard this information. We will therefore generalize our upper bound such that it will also incorporate this additional information.

4.4.1 Generalized KK-Bounds

From now on, L is some family of sets of patterns $L_k, L_{k+1}, \ldots, L_{k+q}$ which are known to be frequent, such that L_{k+p} contains patterns of size $k+p$, and all $k+p-1$-subsets of all patterns in L_{k+p} are in L_{k+p-1}. We denote by $|L|$ the sequence of numbers $|L_k|, |L_{k+1}|, \ldots, |L_{k+q}|$.

Similarly, let I be a family of sets of patterns $I_k, I_{k+1}, \ldots, I_{k+q}$ which are known to be infrequent, such that I_{k+p} contains patterns of size $k+p$ and all $k+p-1$-subsets of all patterns in I_{k+p} are in L_{k+p-1}. We denote by $|I|$ the sequence of numbers $|I_k|, |I_{k+1}|, \ldots, |I_{k+q}|$. Note that for each $p \geq 0$, L_{k+p} and I_{k+p} are disjoint.

Before we present the general upper bounds, we also generalize our notion of a candidate pattern.

Definition 4.2. A *candidate pattern for* (L, I) of size $k+p$ is a pattern which is not in L_{k+p} or I_{k+p}, all of its k-subsets are in L_k, and none of its subsets of size larger than k is included in $I_k \cup I_{k+1} \cup \cdots \cup I_{k+q}$. For a given p, we denote the set of all $k+p$-size candidate patterns for (L, I) by $C_{k+p}(L, I)$.

We note:

Lemma 4.10.

$$C_{k+p}(L,I) = \begin{cases} C_{k+1}(L_k) \setminus (L_{k+1} \cup I_{k+1}) & \textit{if } p = 1; \\ C_{k+p}\big(C_{k+p-1}(L,I) \cup L_{k+p-1}\big) \setminus (L_{k+p} \cup I_{k+p}) & \textit{if } p > 1. \end{cases}$$

Proof. The case $p = 1$ is clear. For $p > 1$, we show the inclusion in both directions.

- $\supseteq$ For every set in $C_{k+p}\big(C_{k+p-1}(L,I) \cup L_{k+p-1}\big)$, we know that all of its k-subsets are always contained in a $k+p-1$ subset, and these are in $C_{k+p-1}(L,I) \cup L_{k+p-1}$. By definition, we know that for every set in $C_{k+p-1}(L,I)$, all of its k-subsets are in L_k. Also, for every set in L_{k+p-1}, all of its k-subsets are in L_k. By definition, for every set in $C_{k+p-1}(L,I)$, all of its $k+p-i$-subsets are not in I_{k+p-i}. Also, for every set in L_{k+p-1}, all of its $k+p-i$-subsets are in L_{k+p-i} and hence they are not in I_{k+p-i} since they are disjoint. By definition, none of the patterns in $L_{k+p} \cup I_{k+p}$ are in $C_{k+p}(L,I)$.

- $\subseteq$ It suffices to show that for every set in $C_{k+p}(L,I)$, every $k+p-1$-subset s is in $C_{k+p-1}(L,I) \cup L_{k+p-1}$. Obviously, this is true, since if it is not already in L_{k+p-1}, still all k-subsets of s must be in L_k, s can not be in I_{k+p-1} and none of its subsets can be in any $I_{k+p-\ell}$ with $\ell > 1$.

□

Hence, we define

$$gKK_k^{k+p}(|L|,|I|) := \begin{cases} KK_k^{k+1}(|L_k|) - |L_{k+1}| - |I_{k+1}| & \text{if } p = 1; \\ KK_{k+p-1}^{k+p}(gKK_k^{k+p-1}(|L|,|I|) + |L_{k+p-1}|) - |L_{k+p}| - |I_{k+p}| & \text{if } p > 1, \end{cases}$$

and obtain:

Theorem 4.11.

$$|C_{k+p}(L,I)| \le gKK_k^{k+p}(|L|,|I|) \le KK_k^{k+p}(|L_k|) - |L_{k+p}| - |I_{k+p}|.$$

Proof. The first inequality is clear by Lemma 4.10. The second inequality is by induction on p. The base case $p = 1$ is by definition. For $p > 1$, we have:

$$\begin{aligned} gKK_k^{k+p}(|L|, |I|) &= KK_{k+p-1}^{k+p}(gKK_k^{k+p-1}(|L|, |I|) + |L_{k+p-1}|) \\ &\quad - |L_{k+p}| - |I_{k+p}| \\ &\leq KK_{k+p-1}^{k+p}(KK_k^{k+p-1}(|L_k|) - |I_{k+p-1}|) - |L_{k+p}| - |I_{k+p}| \\ &\leq KK_{k+p-1}^{k+p}(KK_k^{k+p-1}(|L_k|)) - |L_{k+p}| - |I_{k+p}| \\ &= KK_k^{k+p}(|L_k|) - |L_{k+p}| - |I_{k+p}| \end{aligned}$$

where the first inequality is by induction and because of the monotonicity of KK, the second inequality also because of the monotonicity of KK and the last equality follows from

$$KK_k^{k+p}(|L_k|)) = KK_{k+p-1}^{k+p}(KK_k^{k+p-1}(|L_k|)).$$

□

Again, we can also generalize the upper bound on the maximal size of a candidate pattern, denoted by maxsize(L, I), and the upper bound on the total number of candidate patterns, both also incorporating (L, I):

$$g\mu(|L|, |I|) := k + \min\{p \mid gKK_k^{k+p}(|L|, |I|) = 0\} - 1$$
$$gKK_k^{\text{total}}(|L|, |I|) := \sum_{p\geq 1} gKK_k^{k+p}(|L|, |I|).$$

We obtain:

Proposition 4.12.

$$\text{maxsize}(L, I) \leq g\mu(|L|, |I|) \leq \mu(|L|).$$

Proposition 4.13. *The total number of candidate patterns that can be generated from (L, I) is bounded by $gKK_k^{\text{total}}(|L|, |I|)$. Moreover,*

$$gKK_k^{\text{total}}(|L|, |I|) \leq KK_k^{\text{total}}(|L_k|).$$

Example 4.3. Suppose L_3 consists of all subsets of size 3 of the set $\{1, 2, 3, 4, 5, 6\}$. Now assume we already know that I_4 contains patterns $\{1, 2, 3, 4\}$ and $\{3, 4, 5, 6\}$. The KK upper bound presented in the previous section would estimate the number of candidate patterns of sizes $4, 5$, and 6 to be at most $\binom{6}{4} = 15$, $\binom{6}{5} = 6$, and $\binom{6}{6} = 1$ respectively. Nevertheless, using the additional information, gKK can already reduce these numbers to $13, 3$, and 0. Also, μ would predict the maximal size of a candidate pattern to be 6, while $g\mu$ can already predict this number to be at most 5. Similarly, KK_{total} would predict the total number of candidate patterns that can still be generated to be at most 22, while gKK_{total} can already deduce this number to be at most 16.

4.4.2 Generalized KK^*-Bounds

Using the generalized basic upper bound, we can now also generalize our improved upper bound KK^*. For an arbitrary item x, define the family of sets L^x as $L^x_k, L^x_{k+1}, \ldots, L^x_{k+q}$, and I^x as $I^x_k, I^x_{k+1}, \ldots, I^x_{k+q}$. We define:

$$gKK^*_{k+p}(L,I) := \begin{cases} gKK^{k+p}_k(|L|,|I|) & \text{if } k=1; \\ \min\{gKK^{k+p}_k(|L|,|I|), \sum_{x\in I} gKK^*_{k+p-1}(L^x,I^x)\} & \text{if } k>1. \end{cases}$$

We then have:

Theorem 4.14.

$$|C_{k+p}(L,I)| \leq gKK^*_{k+p}(L,I) \leq KK^*_{k+p}(L_k) - |L_{k+p}| - |I_{k+p}|.$$

Proof. The proof of the first inequality is similar to the proof of Theorem 4.7, instead that we now need to show that for all $p>0$,

$$C_{k+p}(L,I) \subseteq \bigcup_{x\in I} C_{k+p-1}(L^x,I^x) + x.$$

Therefore, we need to show that for every $s \in C_{k+p}(L,I)$ the subset $s-\{x\}$ is in $C_{k+p-1}(L^x,I^x)$, where $x = \min s$. First, this means that every subset of $s-\{x\}$ of size $k-1$ must be in L^x_k. Let $s-\{x\}-\{y_1,\ldots,y_p\}$ be such a subset. This subset is an element of L^x_k if and only if $s-\{y_1,\ldots,y_p\} \in L_k$ and $x = \min(s-\{y_1,\ldots,y_p\})$. The first condition follows from $s \in C_{k+p}(L,I)$, and the second condition is trivial. Second, we need to show that $s-\{x\}$ is not in L^x_{k+p}. Since $s \in C_{k+p}(L,I)$, s is not in L_{k+p} and hence $s-\{x\}$ cannot be in L^x_{k+p}. Finally, we need to show that none of the subsets of $s-\{x\}$ of size greater than $k-1$ are in $I^x_{k+1},\ldots,I^x_{k+p-1}$. Let $s-\{x\}-\{y_1,\ldots,y_m\}$ be such a subset. Since $s \in C_{k+p}(L,I)$, $s-\{y_1,\ldots,y_m\}$ is not in I_{k+p-m}, and hence $s-\{x\}-\{y_1,\ldots,y_m\}$ cannot be in I^x_{k+p-m}.

We prove the second inequality by induction on k. The base case $k=1$ is clear. For all $k>0$, we have

$$\begin{aligned} gKK^*_{k+p}(L,I) &= \min\{gKK^{k+p}_k(|L|,|I|), \sum_{x\in I} gKK^*_{k+p-1}(L^x,I^x)\} \\ &\leq \min\{KK^{k+p}_k(|L_k|) - |L_{k+p}| - |I_{k+p}|, \sum_{x\in I} KK^*_{k+p-1}(L^x_k) - |L^x_{k+p}| - |I^x_{k+p}|\} \\ &= \min\{KK^{k+p}_k(|L|), \sum_{x\in I} KK^*_{k+p-1}(L^x)\} - |L_{k+p}| - |I_{k+p}| \\ &= KK^*_{k+p}(L_k) - |L_{k+p}| - |I_{k+p}| \end{aligned}$$

where the left hand side of the minimum in the inequality is by Theorem 4.11 and the right hand side is by induction. □

Again, we get an upper bound on maxsize(L, I):

$$g\mu^*(L, I) := k + \min\{p \mid gKK^*_{k+p}(L, I) = 0\} - 1,$$

and on the total number of candidate patterns that can still be generated:

$$gKK^*_{\text{total}}(L, I) := \sum_{p \geq 1} gKK^*_{k+p}(L, I).$$

We then have the following analogous propositions to 4.8 and 4.9:

Proposition 4.15.

$$\text{maxsize}(L, I) \leq g\mu^*(L, I) \leq \mu^*(L).$$

Proposition 4.16. *The total number of candidate patterns that can be generated from (L, I) is bounded by $gKK^*_{\text{total}}(L, I)$. Moreover,*

$$gKK^*_{\text{total}}(L, I) \leq KK^*_{\text{total}}(L_k).$$

Example 4.4. Consider the same set of patterns as in the previous example. I.e., L_3 consists of all subsets of size 3 of the set $\{1, 2, 3, 4, 5, 6\}$ and $\{1, 2, 3, 4\}$ and $\{3, 4, 5, 6\}$ are included in I_4. The KK^* upper bound presented in the previous section would also estimate the number of candidate patterns of sizes 4, 5, and 6 to be at most $\binom{6}{4} = 15$, $\binom{6}{5} = 6$, and $\binom{6}{6} = 1$ respectively. Nevertheless, using the additional information, gKK^* can perfectly predict these numbers to be 13, 2, and 0. Again, μ^* would predict the maximal size of a candidate pattern to be 6, while $g\mu^*$ can already predict this number to be at most 5. Similarly, KK^*_{total} would predict the total number of candidate patterns that can still be generated to be at most 22, while gKK^*_{total} can already deduce this number to be at most 15.

4.5 Efficient Implementation

For simplicity reasons, we will restrict ourselves to the explanation of how the improved upper bounds can be implemented. The proposed implementation can be easily extended to support the computation of the general upper bounds.

To evaluate our upper bounds we implemented an optimized version of the Apriori algorithm using the trie data structure to store all generated patterns. This trie structure makes it cheap and straightforward to implement the computation of all upper bounds. Indeed, a top-level subtrie (rooted at some singleton pattern $\{x\}$) represents exactly the set L^x we defined in Section 4.3. Every top-level subtrie of this subtrie (rooted at some two-element pattern $\{x, y\}$) then represents $(L^x)^y$, and so on. Hence, we can compute the recursive bounds while traversing the trie, after the frequencies of all candidate patterns are counted, and we have to traverse the trie once more to remove all candidate patterns that turned out to be infrequent. This can be done as follows.

Remember, at that point, we have the current set of frequent patterns of size k stored in the trie. For every node at depth d smaller than k, we compute the $k-d$-canonical representation of the number of descendants this node has at depth k, which can be used to compute μ_{k-d} (cf. Proposition 4.4), KK^{ℓ}_{k-d} for any $\ell \leq \mu_{k-d}$ (cf. Theorem 4.2) and hence also KK^{total}_{k-d} (cf. Proposition 4.6). For every node at depth $k-1$, its KK^* and μ^* values are equal to its KK and μ values respectively. Then compute for every $p > 0$, the sum of the $KK^*_{k-d+p-1}$ values of all its children, and let KK^*_{k-d+p} be the smallest of this sum and KK^{k-d+p}_{k-d} until this minimum becomes zero, which also gives us the value of μ^*. Finally, we can compute KK^*_{total} for this node. If this is done for every node, traversed in a depth-first manner, then finally the root node will contain the upper bounds on the number of candidate patterns that can still be generated, and on the maximum size of any such pattern. The soundness and completeness of this method follows directly from the theorems and propositions of the previous sections.

We should also point out that, since the numbers involved can become exponentially large (in the number of items), an implementation should take care to use arbitrary-length integers such as provided by standard mathematical packages. Since the length of an integer is only logarithmic in its value, the lengths of the numbers involved will remain polynomially bounded.

4.6 Experimental Evaluation

The algorithm was implemented in C++ and uses the GNU MP library for arbitrary-length integers [34].

The results from the experiment with the real data sets were not immediately as good as the results from the synthetic data set. The reason for this, however, turned out to be the bad ordering of the items, as explained next.

Reordering

From the form of L^x, it can be seen that the order of the items can affect the recursive upper bounds. By computing the upper bound only for a subset of all frequent patterns (namely L^x), we win by incorporating the structure of the current collection of frequent patterns, but we also lose some information. Indeed, whenever we recursively restrict ourselves to a subtrie L^x, then for

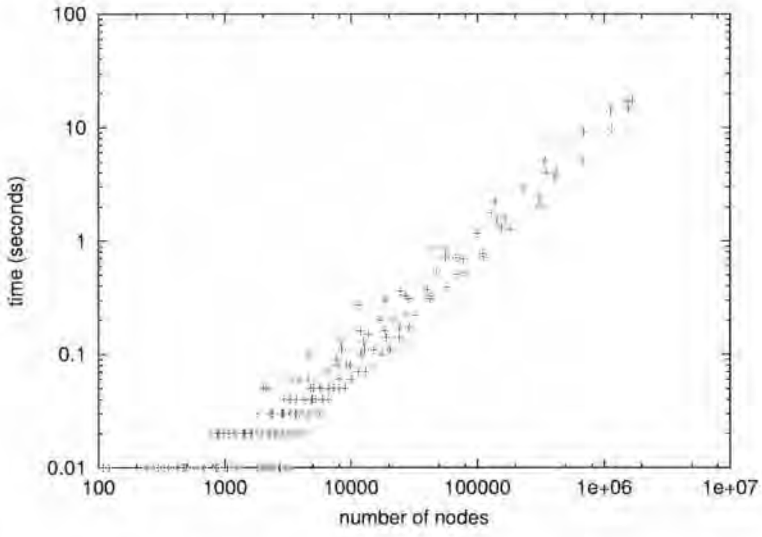


Figure 4.1: Time needed to compute upper bounds is linear in the number of nodes.

every candidate pattern s with $x = \min s$, we lose the information about exactly one subpattern in L, namely $s - x$.

We therefore would like to make it likely that many of these excluded patterns are frequent. A good heuristic, which has already been used for several other optimizations in frequent pattern mining [9, 16, 2], is to force the most frequent items to appear in the most candidate patterns, by reordering the single item patterns in ascending order of support.

After reordering the items in the real life data set, using this heuristic, the results became very analogous with the results using the synthetic data sets.

Efficiency

The cost for the computation of the upper bounds is negligible compared to the cost of the complete algorithm. Indeed, the time T needed to calculate the upper bounds is largely dictated by the number n of currently known frequent sets. We have shown experimentally that T scales linearly with n. Moreover, the constant factor in our implementation is very small (around 0.00001). We ran several experiments using the different data sets and varying minimal support thresholds. After every pass of the algorithm, we registered the number of known frequent sets and the time spent to compute all upper bounds, resulting in 145 different data points. Figure 4.1 shows these results.

Upper Bounds

- Figure 4.2 shows, after each level k, the computed upper bound KK and improved upper bound KK^* for the number of candidate patterns of size $k+1$, as well as the actual number $|C_{k+1}|$ it turned out to be. We omitted the upper bound for $k+1=2$, since this upper bound is simply $\binom{|L|}{2}$, with $|L|$ the number of frequent items.

- Figure 4.3 shows the upper bounds on the total number of candidate patterns that could still be generated, compared to the actual number of candidate patterns, $|C_{\text{total}}|$, that were effectively generated. Again, we omitted the upper bound for $k=1$, since this number is simply $2^{|L|}-|L|-1$, with $|L|$ the number of frequent items.

- Figure 4.4 shows the computed upper bounds μ and μ^* on the maximal size of a candidate pattern. Also here we omitted the result for $k=1$, since this number is exactly the number of frequent items.

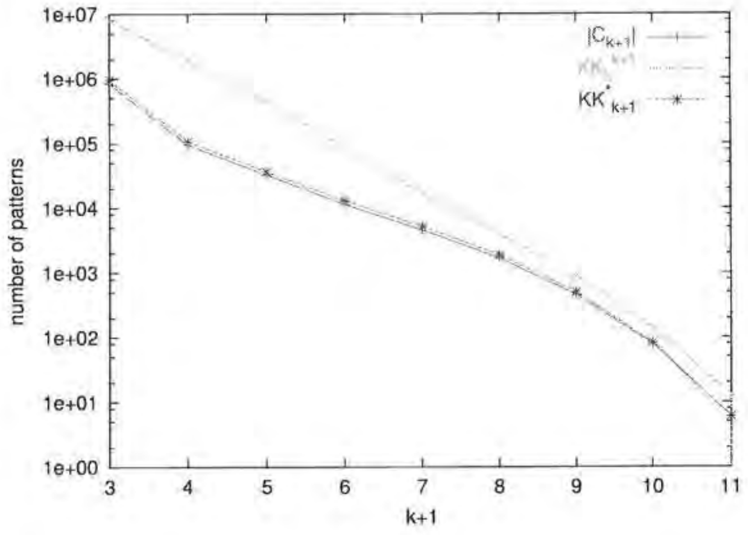
The results are pleasantly surprising:

- Note that the improvement of KK^* over KK, and of μ^* over μ, anticipated by our theoretical discussion, is indeed dramatic.

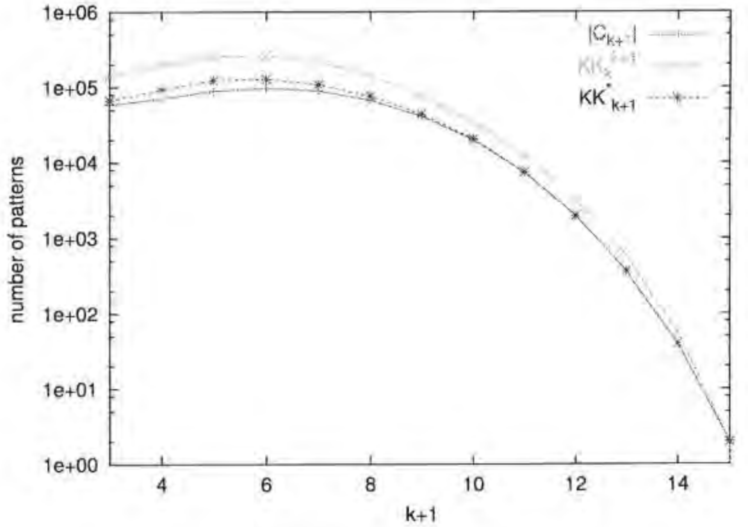
- Comparing the computed upper bounds with the actual numbers, we observe the high accuracy of the estimations given by KK^*. Indeed, the estimations of KK^*_{k+1} match almost exactly the actual number of candidate patterns that has been generated at level $k+1$. Also note that the number of candidate patterns in T40I10D100K is decreasing in the first four iterations and then increases again. This perfectly illustrates that the heuristic used for AprioriHybrid, as explained in the related work section, would not work on this data set. Indeed, since the current number of candidate patterns is small enough and there are fewer candidate patterns in the current iteration than in the previous iteration, these observations would be falsely interpreted. The presented upper bounds perfectly predict this increase.

- The upper bounds on the total number of candidate patterns are still very large when estimated in the first few passes, which is not surprising because at these initial stages, there is not much information yet. For the mushroom and the artificial data sets, the upper bound is almost exact when the frequent patterns of size 3 are known. For the basket data set, this result is obtained when the frequent patterns of size 4 are known and size 6 for the BMS-Webview-1 data set.

- The results obtained from experimenting with varying minimal support thresholds were entirely similar to those presented above.

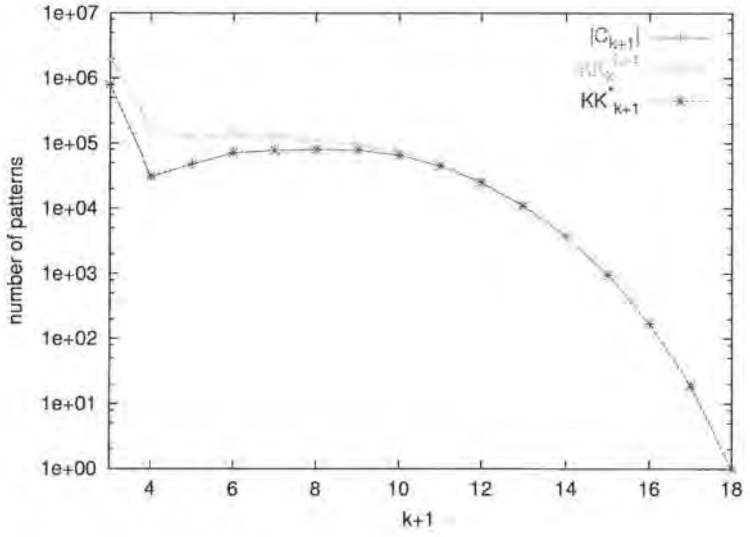


(a) basket

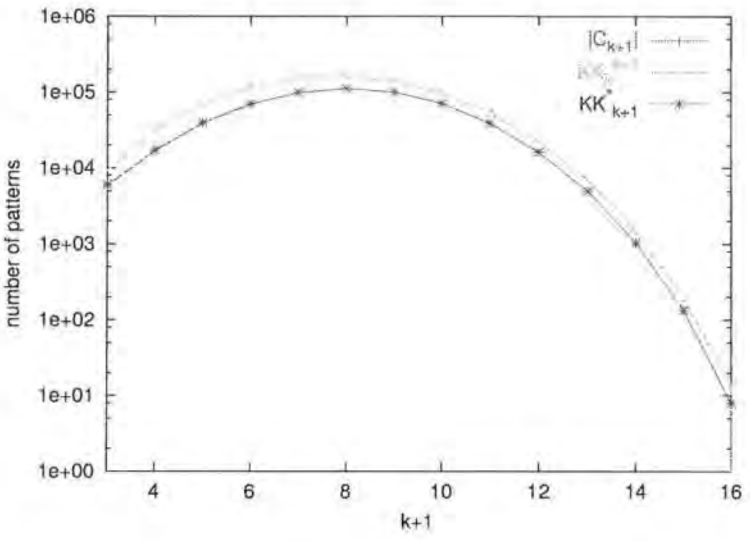


(b) BMS-Webview-1

Figure 4.2: Actual and estimated number of candidate patterns.

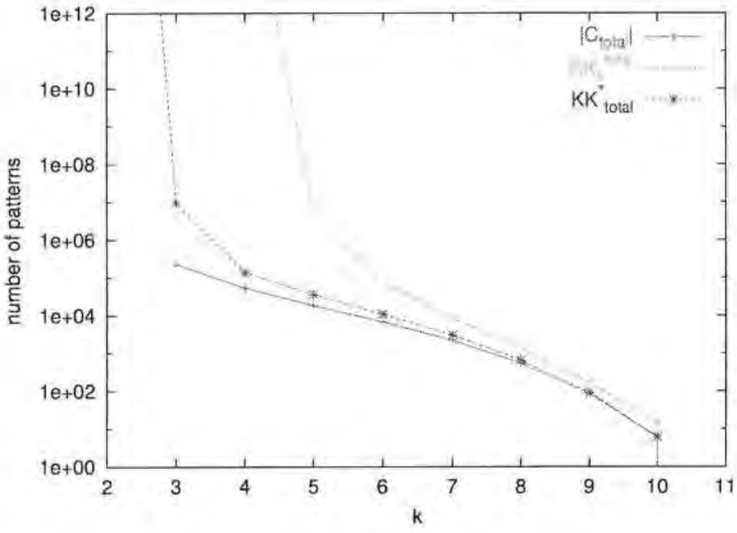


(c) T40I10D100K

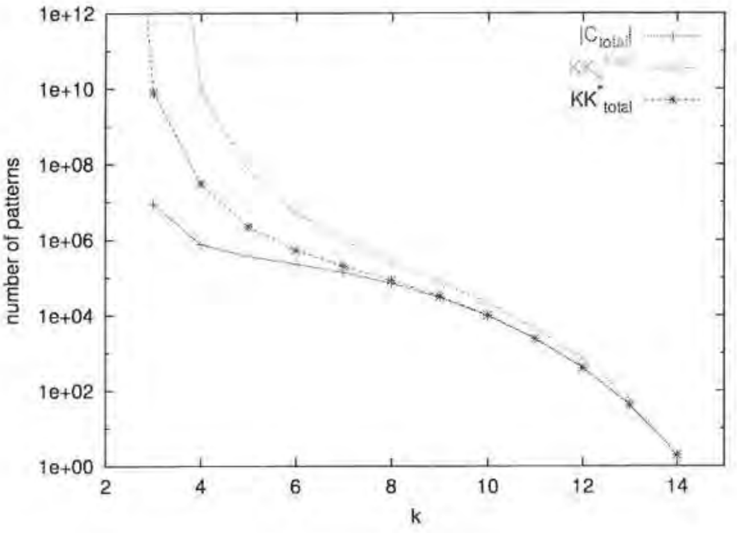


(d) mushroom

Figure 4.2: Actual and estimated number of candidate patterns.

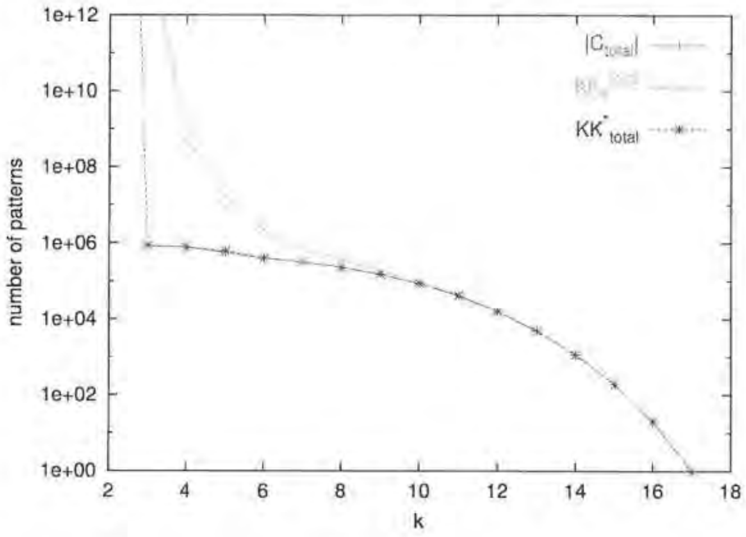


(a) basket

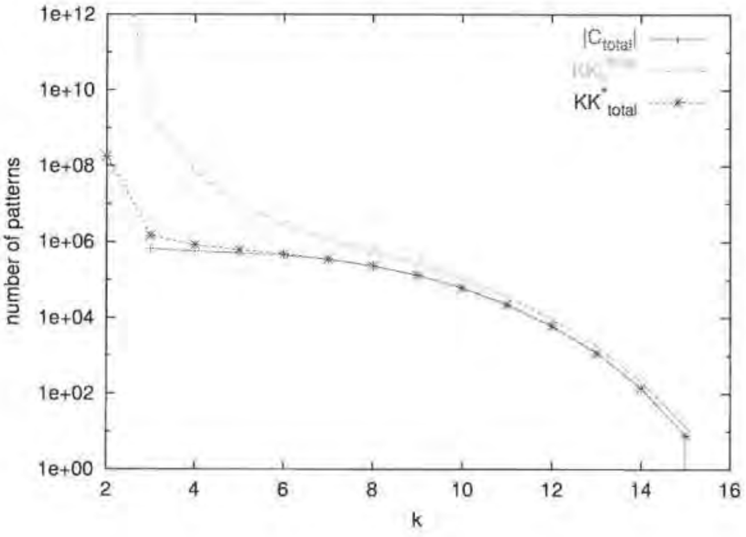


(b) BMS-Webview-1

Figure 4.3: Actual and estimated total number of future candidate patterns.

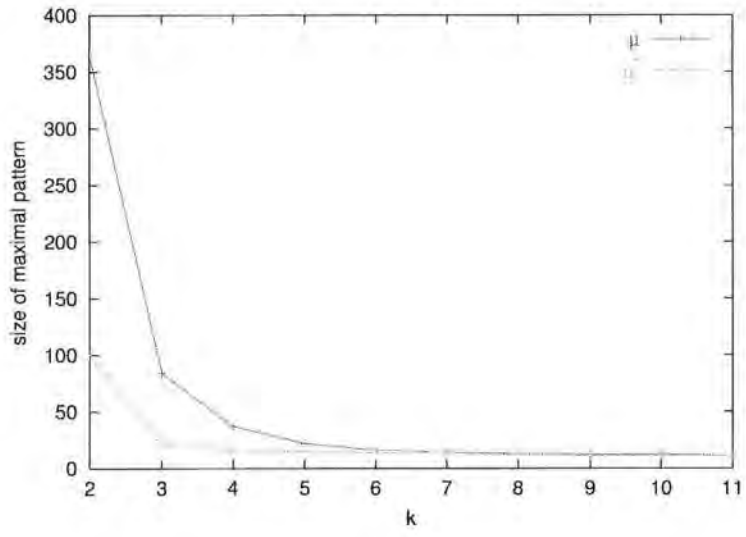


(c) T40I10D100K

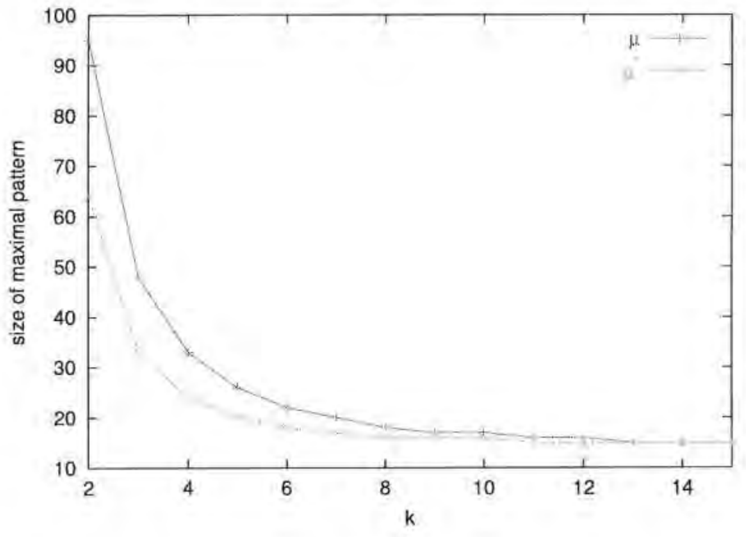


(d) mushroom

Figure 4.3: Actual and estimated total number of future candidate patterns.

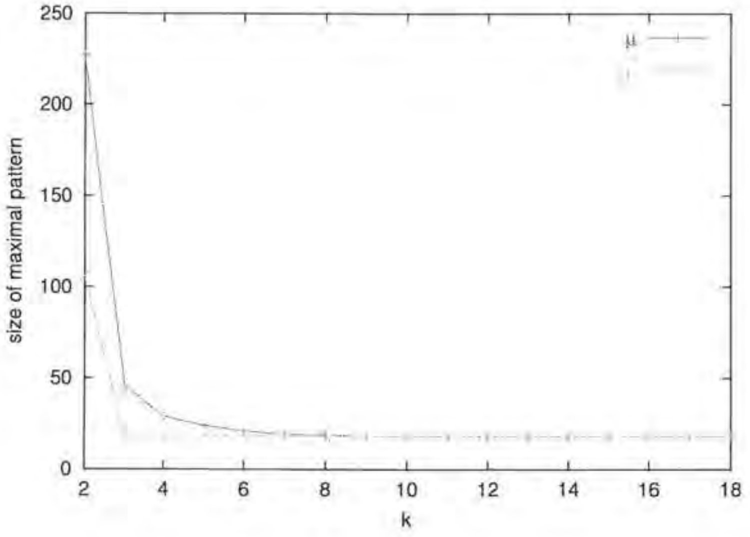


(a) basket

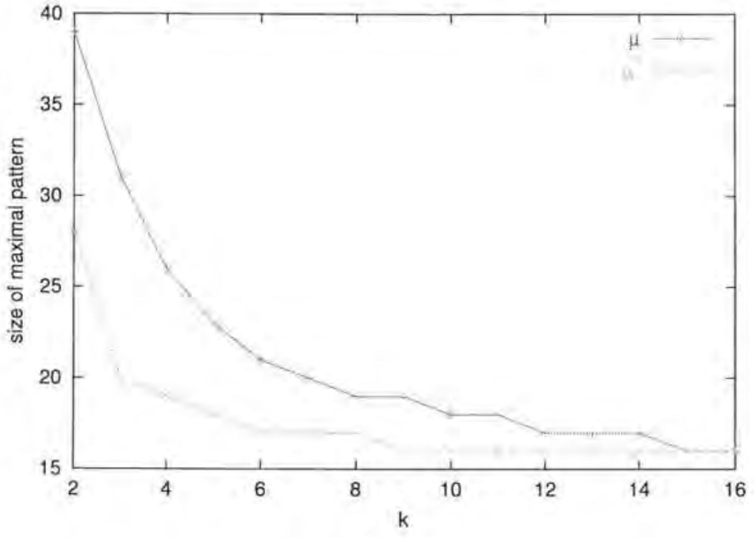


(b) BMS-Webview-1

Figure 4.4: Estimated size of the largest possible candidate pattern.



(c) T40I10D100K



(d) mushroom

Figure 4.4: Estimated size of the largest possible candidate pattern.

Combining Iterations

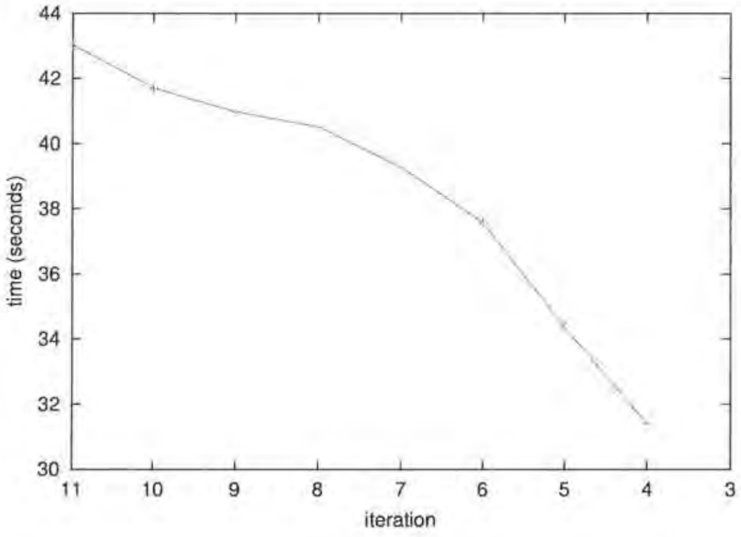
As discussed in the introduction, the proposed upper bound can be used to protect several improvements of the Apriori algorithm from generating too many candidate patterns. One such improvement tries to combine as many iterations as possible in the end, when only few candidate patterns can still be generated. We have implemented this technique within our implementation of the Apriori algorithm.

We performed several experiments on each data set and limited the number of candidate patterns that is allowed to be generated. If the upper bound on the total number of candidate patterns is below this limit, the algorithm generates and counts all possible candidate patterns within the next iteration. Figure 4.5 shows the results. The x-axis shows the total number of iterations in which the algorithm completed, and the y-axis shows the total time the algorithm needed to complete.

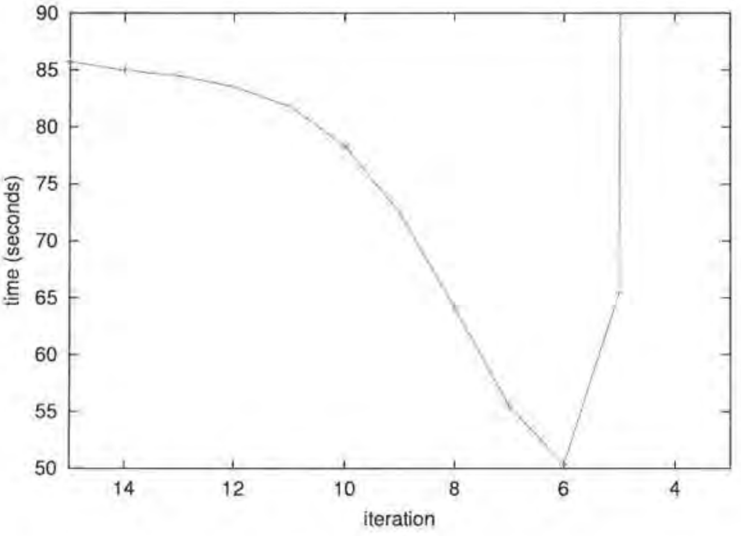
As can be seen, for all data sets, the algorithm can already combine all remaining iterations into one very early in the algorithm. For example, the BMS-Webview-1 data set, which normally performs 15 iteration, could be reduced to six iterations to give an optimal performance. If the algorithm already generated all remaining candidate patterns in the fifth iteration, the number of candidate patterns that turned out to be infrequent was too large, such that the gain of reducing iterations has been consumed by the time needed to count all these candidate patterns. Nevertheless, it is still more effective than not combining any passes at all. If we allowed the generation of all candidate patterns to occur in even earlier iterations, although the upper bound predicted a too large number of candidate patterns, this number became indeed too large to keep into main memory.

4.7 Conclusions

Motivated by several heuristics to reduce the number of database scans in the context of frequent pattern mining, we provide a hard and tight combinatorial upper bound on the number of candidate patterns and on the size of the largest possible candidate pattern, given a set of frequent patterns. Our findings are not restricted to a single algorithm, but can be applied to any frequent pattern mining algorithm which is based on the levelwise generation of candidate patterns. Using the standard Apriori algorithm, on which most frequent pattern mining algorithms are based, our experiments showed that these upper bounds can be used to considerably reduce the number of database scans without taking the risk of getting a combinatorial explosion of the number of candidate patterns.

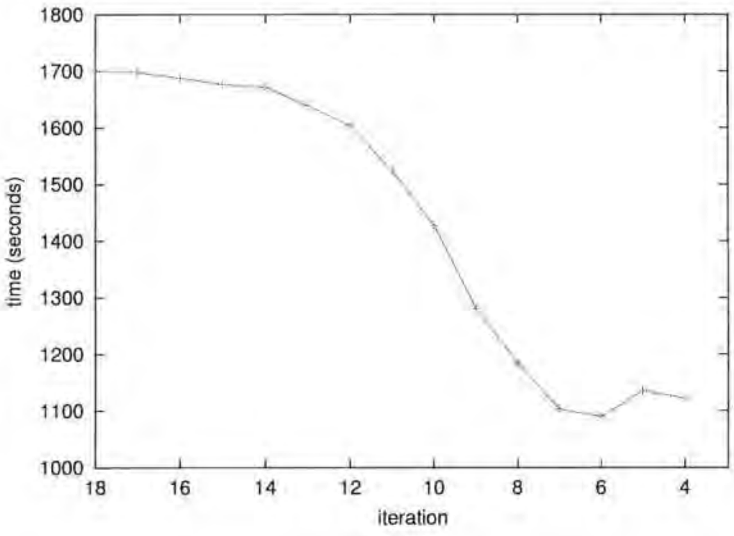


(a) basket

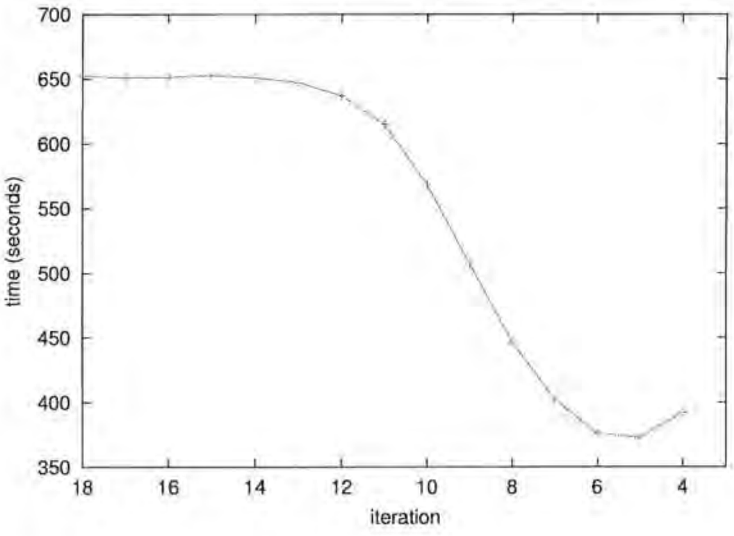


(b) BMS-Webview-1

Figure 4.5: Combining iterations.



(c) T40I10D100K



(d) mushroom

Figure 4.5: Combining iterations.

Bibliography

[1] R.C. Agarwal, C.C. Aggarwal, and V.V.V. Prasad. Depth first generation of long patterns. In Ramakrishnan et al. [69], pages 108–118.

[2] R.C. Agarwal, C.C. Aggarwal, and V.V.V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, March 2001.

[3] R. Agrawal, T. Imielinski, and A.N. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22(2) of *SIGMOD Record*, pages 207–216. ACM Press, 1993.

[4] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In Fayyad et al. [28], pages 307–328.

[5] R. Agrawal and R. Srikant. *Quest Synthetic Data Generator.* IBM Almaden Research Center, San Jose, California, `http://www.almaden.ibm.com/cs/quest/syndata.html`.

[6] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J.B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings 20th International Conference on Very Large Data Bases*, pages 487–499. Morgan Kaufmann, 1994.

[7] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. IBM Research Report RJ9839, IBM Almaden Research Center, San Jose, California, June 1994.

[8] A. Amir, R. Feldman, and R. Kashi. A new and versatile method for association generation. *Information Systems*, 2:333–347, 1997.

[9] R.J. Bayardo, Jr. Efficiently mining long patterns from databases. In Haas and Tiwary [36], pages 85–93.

[10] R.J. Bayardo, Jr., R. Agrawal, and D. Gunopulos. Constraint-based rule mining on large, dense data sets. In *Proceedings of the 15th International Conference on Data Engineering*, pages 188–197. IEEE Computer Society, 1999.

[11] C.L. Blake and C.J. Merz. *UCI Repository of machine learning databases.* University of California, Irvine, Dept. of Information and Computer Sciences, `http://www.ics.uci.edu/~mlearn/MLRepository.html`, 1998.

[12] B. Bollobás. *Combinatorics.* Cambridge University Press, 1986.

[13] C. Borgelt and R. Kruse. Induction of association rules: Apriori implementation. In W. Härdle and B. Rönz, editors, *Proceedings of the 15th Conference on Computational Statistics*, pages 395–400, `http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html`, 2002. Physica-Verlag.

[14] J.-F. Boulicaut, A. Bykowski, and C. Rigotti. Free-sets: A condensed representation of boolean data for the approximation of frequency queries. *Data Mining and Knowledge Discovery*, 2003. To appear.

[15] T. Brijs, B. Goethals, G. Swinnen, K. Vanhoof, and G. Wets. A data mining framework for optimal product selection in retail supermarket data: The generalized profset model. In Ramakrishnan et al. [69], pages 300–304.

[16] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, volume 26(2) of *SIGMOD Record*, pages 255–264. ACM Press, 1997.

[17] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International World-Wide Web Conference*, volume 30(1–7) of *Computer Networks*, pages 107–117. Elsevier Science, 1998.

[18] A. Bykowski and C. Rigotti. A condensed representation to find frequent patterns. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 267–273. ACM Press, 2001.

[19] T. Calders and B. Goethals. Mining all non-derivable frequent itemsets. In Elomaa et al. [25], pages 74–85.

[20] N. Cercone, T.Y. Lin, and X. Wu, editors. *Proceedings of the 2001 IEEE International Conference on Data Mining.* IEEE Computer Society, 2001.

[21] W. Chen, J.F. Naughton, and P.A. Bernstein, editors. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, volume 29(2) of *SIGMOD Record*. ACM Press, 2000.

[22] U. Dayal, P.M.D. Gray, and S. Nishio, editors. *Proceedings 21th International Conference on Very Large Data Bases*. Morgan Kaufmann, 1995.

[23] L. Dehaspe and H. Toivonen. Discovery of relational association rules. In S. Dzeroski and N. Lavrac, editors, *Relational data mining*, pages 189–212. Springer, 2001.

[24] A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors. *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, volume 28(2) of *SIGMOD Record*. ACM Press, 1999.

[25] T. Elomaa, H. Mannila, and H. Toivonen, editors. *Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, volume 2431 of *Lecture Notes in Computer Science*. Springer, 2002.

[26] U.M. Fayyad, S.G. Djorgovski, and N. Weir. Automating the analysis and cataloging of sky surveys. In Fayyad et al. [28], pages 471–494.

[27] U.M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. In Fayyad et al. [28], pages 1–34.

[28] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. MIT Press, 1996.

[29] P. Frankl. A new short proof for the Kruskal–Katona theorem. *Discrete Mathematics*, 48:327–329, 1984.

[30] F. Geerts, B. Goethals, and J. Van den Bussche. A tight upper bound on the number of candidate patterns. In Cercone et al. [20], pages 155–162.

[31] B. Goethals and J. Van den Bussche. On supporting interactive association rule mining. In Y. Kambayashi, M.K. Mohania, and A.M. Tjoa, editors, *Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery*, volume 1874 of *Lecture Notes in Computer Science*, pages 307–316. Springer, 2000.

[32] B. Goethals and J. Van den Bussche. Relational association rules: getting warmer. In D. Hand, R. Bolton, and N. Adams, editors, *Proceedings of the ESF Exploratory Workshop on Pattern Detection and Discovery in Data Mining*, volume 2447 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2002.

[33] G. Grahne, L.V.S. Lakshmanan, and X. Wang. Efficient mining of constrained correlated sets. In *Proceedings of the 16th International Conference on Data Engineering*, pages 512–521. IEEE Computer Society, 2000.

[34] T. Granlund and K. Ryde. *GNUmp, Library for arithmetic on arbitrary precision numbers.* `http://www.gnu.org/directory/gnump.html`.

[35] D. Gunopulos, R. Khardon, H. Mannila, and H. Toivonen. Data mining, hypergraph transversals, and machine learning. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 209–216. ACM Press, 1997.

[36] L.M. Haas and A. Tiwary, editors. *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, volume 27(2) of *SIGMOD Record.* ACM Press, 1998.

[37] J. Han, Y. Fu, K. Koperski, W. Wang, and O. Zaiane. DMQL: A data mining query language for relational databases. Presented at SIGMOD'96 Workshop on Research Issues on Data Mining and Knowledge Discovery, 1996.

[38] J. Han, Y. Fu, W. Wang, J. Chiang, W. Gong, K. Koperski, D. Li, Y. Lu, A. Rajan, N. Stefanovic, B. Xia, and O.R. Zaïane. DBMiner: A system for mining knowledge in large relational databases. In E. Simoudis, J. Han, and U. Fayyad, editors, *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 250–255. AAAI Press, 1996.

[39] J. Han and M. Kamber. *Data Mining: Concepts and Techniques.* Morgan Kaufmann, 2001.

[40] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In Chen et al. [21], pages 1–12.

[41] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 2003. To appear.

[42] D. Hand, D. Keim, and R.T. Ng, editors. *Proceedings of the Eight ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM Press, 2002.

[43] D. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining.* MIT Press, 2001.

[44] D. Heckerman, H. Mannila, and D. Pregibon, editors. *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining.* AAAI Press, 1997.

[45] C. Hidber. Online association rule mining. In Delis et al. [24], pages 145–156.

[46] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Mining association rules: Deriving a superior algorithm by analyzing today's approaches. In Zighed et al. [86], pages 159–168.

[47] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of the ACM*, 39(11):58–64, 1996.

[48] T. Imielinski and A. Virmani. MSQL: A query language for database mining. *Data Mining and Knowledge Discovery*, 3(4):373–408, December 1999.

[49] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In Zighed et al. [86], pages 13–23.

[50] B. Jeudy and J.-F. Boulicaut. Using condensed representations for interactive association rule mining. In Elomaa et al. [25], pages 225–236.

[51] G.O.H. Katona. A theorem of finite sets. In *Theory Of Graphs*, pages 187–207. Akadémia Kiadó, 1968.

[52] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000. `http://www.ecn.purdue.edu/KDDCUP`.

[53] J.B. Kruskal. The number of simplices in a complex. In *Mathematical Optimization Techniques*, pages 251–278. Univ. of California Press, 1963.

[54] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In Cercone et al. [20], pages 313–320.

[55] L.V.S. Lakshmanan, R.T. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In Delis et al. [24], pages 157–168.

[56] H. Mannila. Inductive databases and condensed representations for data mining. In J. Maluszynski, editor, *Proceedings of the 1997 International Symposium on Logic Programming*, pages 21–30. MIT Press, 1997.

[57] H. Mannila. Global and local methods in data mining: basic techniques and open problems. In P. Widmayer, F.T. Ruiz, R. Morales, M. Hennessy, S. Eidenbenz, and R. Conejo, editors, *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 57–68. Springer, 2002.

[58] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, November 1997.

[59] H. Mannila, H. Toivonen, and A.I. Verkamo. Efficient algorithms for discovering association rules. In U.M. Fayyad and R. Uthurusamy, editors, *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, pages 181–192. AAAI Press, 1994.

[60] R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In Vijayaraman et al. [78], pages 122–133.

[61] B. Nag, P. Deshpande, and D.J. DeWitt. Using a knowledge cache for interactive discovery of association rules. In U. Fayyad, S. Chaudhuri, and D. Madigan, editors, *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 244–253. ACM Press, 1999.

[62] R.T. Ng, L.V.S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained association rules. In Haas and Tiwary [36], pages 13–24.

[63] S. Orlando, P. Palmerini, and R. Perego. Enhancing the apriori algorithm for frequent set counting. In Y. Kambayashi, W. Winiwarter, and M. Arikawa, editors, *Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery*, volume 2114 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 2001.

[64] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In V. Kumar, S. Tsumoto, P.S. Yu, and N.Zhong, editors, *Proceedings of the 2002 IEEE International Conference on Data Mining*. IEEE Computer Society, 2002. To appear.

[65] J.S. Park, M.-S. Chen, and P.S. Yu. An effective hash based algorithm for mining association rules. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, volume 24(2) of *SIGMOD Record*, pages 175–186. ACM Press, 1995.

[66] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In C. Beeri and P. Buneman, editors, *Proceedings of the 7th International Conference on Database Theory*, volume 1540 of *Lecture Notes in Computer Science*, pages 398–416. Springer, 1999.

[67] J. Pei and J. Han. Can we push more constraints into frequent pattern mining? In Ramakrishnan et al. [69], pages 350–354.

[68] J. Pei, J. Han, and L.V.S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proceedings of the 17th International Conference on Data Engineering*, pages 433–442. IEEE Computer Society, 2001.

[69] R. Ramakrishnan, S. Stolfo, R.J. Bayardo, Jr., and I. Parsa, editors. *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2000.

[70] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In Dayal et al. [22], pages 432–444.

[71] P. Shenoy, J.R. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In Chen et al. [21], pages 22–33.

[72] R. Srikant. *Fast algorithms for mining association rules and sequential patterns*. PhD thesis, University of Wisconsin, Madison, 1996.

[73] R. Srikant and R. Agrawal. Mining generalized association rules. In Dayal et al. [22], pages 407–419.

[74] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In Heckerman et al. [44], pages 66–73.

[75] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[76] P. Tan, V. Kumar, and J. Srivastava. Selecting the right interestingness measure for association patterns. In Hand et al. [42], pages 32–41.

[77] H. Toivonen. Sampling large databases for association rules. In Vijayaraman et al. [78], pages 134–145.

[78] T.M. Vijayaraman, A.P. Buchmann, C. Mohan, and N.L. Sarda, editors. *Proceedings 22nd International Conference on Very Large Data Bases*. Morgan Kaufmann, 1996.

[79] G.I. Webb. Efficient search for association rules. In Ramakrishnan et al. [69], pages 99–107.

[80] M.J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, May/June 2000.

[81] M.J. Zaki. Fast vertical mining using diffsets. Technical Report 01-1, Rensselaer Polytechnic Institute, Troy, New York, 2001.

[82] M.J. Zaki. Efficiently mining frequent trees in a forest. In Hand et al. [42], pages 71–80.

[83] M.J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In R. Grossman, J. Han, V. Kumar, H. Mannila, and R. Motwani, editors, *Proceedings of the Second SIAM International Conference on Data Mining*, 2002.

[84] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In Heckerman et al. [44], pages 283–286.

[85] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In F. Provost and R. Srikant, editors, *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 401–406. ACM Press, 2001.

[86] D.A. Zighed, H.J. Komorowski, and J.M. Zytkow, editors. *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, volume 1910 of *Lecture Notes in Computer Science*. Springer, 2000.

Samenvatting

Een enorme technologische vooruitgang in hardware en software heeft de voorbije decennia geresulteerd in de opbouw en groei van gigantische hoeveelheden gegevens die opgeslagen zitten in databanken. Eén van de voornaamste uitdagingen waar tal van ondernemingen en individuen nu voor staan, is hoe ze deze gegevens kunnen analyseren en omzetten in handelbare en bruikbare kennis.

Pogingen om deze uitdagingen aan te gaan brachten onderzoekers samen uit verschillende disciplines zoals statistiek, kunstmatige intelligentie, databanken en waarschijnlijk nog veel meer, resulterende in het nieuwe onderzoeksgebied *Data Mining*.

Data mining wordt meestal vernoemd in de bredere context van *Knowledge Discovery in Databases* (KDD) en wordt beschouwd als een enkele stap in het zogenaamde *KDD proces* [27].

In dit proefschrift concentreren we ons op het zoeken naar frequent voorkomende patronen in databanken en de efficiëntie van de methoden die daarvoor gebruikt worden. De patronen die we hier beschouwen zijn verzamelingen van items en associatieregels in zogenaamde transactie databanken.

De motivatie om naar zulke patronen te zoeken kwam oorspronkelijk voort uit de behoefte om de transactiegegevens van supermarkten te analyseren. Meer bepaald, het zoeken naar patronen in het aankoopgedrag van klanten. Een associatieregel beschrijft dan hoe frequent verschillende items (of producten) samen aangekocht worden. Bijvoorbeeld, de associatieregel "bier $\Rightarrow$ chips (80%)" drukt uit dat vier van de vijf klanten die bier aankopen ook chips aankopen. Zulke regels kunnen dan gebruikt worden voor beslissingen in verband met prijstoekenningen, promoties, productplaatsing en dergelijke meer.

Sinds hun introductie in 1993 door Agrawal et al. [3], hebben het frequente item-verzameling-probleem en associatieregel-probleem enorm veel aandacht gekregen. In de laatste tien jaar zijn honderden onderzoeksartikels gepubliceerd, die elk nieuwe algoritmes of verbeteringen op algoritmes voorstellen om deze problemen efficiënter op te lossen.

Probleembeschrijving en overzicht: Het aantal keren dat een bepaalde verzameling van items in een databank voorkomt, noemen we de *support* van die verzameling. Om aan te duiden wanneer een verzameling van items frequent is, wordt er een minimale support vastgelegd. Een verzameling van items is dan frequent als zijn support groter is dan deze minimale drempelwaarde. Een associatieregel bestaat uit een antecedent X en een consequent Y die allebei verzamelingen van items zijn, en wordt genoteerd $X \Rightarrow Y$. De support van een regel is gedefinieerd als de support van de unie van het antecedent en het consequent. De *betrouwbaarheid* van een regel wordt berekend door de support van de regel te delen door de support van het antecedent. We noemen een regel *interessant* als zijn betrouwbaarheid hoger ligt dan een vastgelegde *minimale betrouwbaarheid* en als de verzameling van items bestaande uit de unie van het antecedent en het consequent frequent is.

Het eerste algoritme dat alle interessante associatieregels efficiënt kon genereren was het Apriori algoritme van Agrawal et al. en werd onafhankelijk verkregen door Mannila et al. [6, 59, 4]. Het algoritme werd opgedeeld in twee fasen. In de eerste fase werden alle frequente verzamelingen gegenereerd en in een tweede fase alle interessante associatieregels.

Het onderliggende principe dat in het Apriori algoritme en in tal van zijn opvolgers wordt gebruikt is het *monotoniciteitsprincipe*, dat zegt dat de support van een verzameling van items niet groter kan zijn dan de support van één van zijn deelverzamelingen. Met andere woorden, als een verzameling niet frequent is, dan kan geen enkele uitbreiding van die verzameling frequent zijn. Het Apriori algoritme werkt als volgt.

Gegeven is een databank bestaande uit een collectie van verzamelingen van items, ook transacties genoemd. Het Apriori algoritme is een iteratief algoritme dat in elke iteratie k alle *kandidaat verzamelingen* bestaande uit k items genereert. Een verzameling is een kandidaat verzameling als al zijn deelverzamelingen frequent zijn. In de eerste iteratie wordt de support van elk item apart geteld door heel de database transactie per transactie te scannen. Telkens een item voorkomt in een transactie wordt zijn support met 1 verhoogd. Wanneer alle transacties op deze manier verwerkt zijn, bezitten we de support van elk item in de databank. Vermits we enkel geïnteresseerd zijn in frequente verzamelingen verwijderen we alle items waarvan de support kleiner is dan de gegeven minimale support. In elke volgende iteratie k, worden verzamelingen bestaande uit $k-1$ items gecombineerd tot kandidaat verzamelingen bestaande uit k items zodat voor elke gegenereerde verzameling geldt dat al zijn deelverzamelingen frequent zijn. Daarna wordt telkens de databank volledig doorlopen om de supports te berekenen van alle gegenereerde kandidaat verzamelingen. Wanneer er geen kandidaat verzamelingen meer gegenereerd kunnen worden, eindigt het algoritme en zijn alle frequent voorkomende verzamelingen gevonden.

Op basis van deze frequente verzamelingen kunnen we nu in de tweede fase alle interessante associatieregels genereren. Daarvoor dienen we enkel elke frequente verzameling op te delen in een antecedent en consequent en te berekenen wat de betrouwbaarheid is van de bekomen regel. Vermits we daarvoor enkel de support van die verzameling en de support van het antecedent nodig hebben kan dit zeer efficiënt berekend worden.

Later werden nog tal van artikels gepubliceerd waarin vele mogelijke optimalisaties werden voorgesteld, al dan niet voor speciale situaties. Het onderliggende laagsgewijze algoritme werd echter zelden gewijzigd. De meeste pogingen die werden ondernomen, trachten het aantal keren dat de database doorlopen moet worden te verminderen [77, 70, 16, 45]. Andere pogingen trachten aan de hand van allerhande heuristieken en speciale technieken het aantal kandidaat verzamelingen te verminderen die dienen geteld te worden. Deze twee aspecten zijn immers de belangrijkste kostfactoren van het Apriori algoritme.

Een zeer eenvoudig algoritme, Eclat, dat in 1997 werd voorgesteld door Zaki blijkt in verschillende ordes van grootte efficiënter te zijn dan Apriori [80]. Dit algoritme werkt echter wel alleen zoals gewenst wanneer de volledige databank in het werkgeheugen van de computer geladen kan worden. Het algoritme is gebaseerd op de eigenschap dat het aantal transacties waarin een bepaalde verzameling I voorkomt, gelijk is aan het aantal transacties waarin twee van zijn deelverzamelingen X, Y voorkomen, zodat $I = X \cup Y$. Essentieel werkt dit algoritme zeer gelijkaardig als het Apriori algoritme, met als grootste verschil dat we nu voor elke verzameling een lijst bijhouden van alle transacties waarin die verzameling voorkomt. Om dan de support te kennen van een verzameling I dienen we enkel de doorsnede te nemen van de transactielijst van twee van zijn deelverzamelingen X, Y, zodat $I = X \cup Y$. Met andere woorden, als we beginnen door aan elk item zijn lijst van transacties toe te kennen door een enkele keer door de databank te scannen, kunnen we vanaf dan recursief telkens twee frequente verzamelingen X en Y combineren tot een grotere verzameling I, waarvan we de support direct kunnen berekenen door simpelweg de doorsnede te nemen van de transactielijsten van X en Y.

Recentelijk werd een ander algoritme voorgesteld door Han et al., genaamd FP-growth [41], waarvan werd beweerd dat het alle frequente verzamelingen kan vinden zonder daarvoor kandidaat verzamelingen te moeten genereren. Ook werd een nieuwe datastructuur voorgesteld waarvan werd beweerd dat deze zou resulteren in een veel kleiner geheugengebruik en het mede daardoor de meest efficiënte methode zou zijn om alle frequente verzamelingen te vinden. Wij tonen echter aan dat deze beweringen onjuist zijn en besluiten dat een combinatie van het Apriori algoritme en Eclat momenteel de beste uitkomst biedt.

Interactieve methodes voor data mining: Vermits data mining een essentieel interactief proces is, waarin de gebruiker herhaaldelijk bepaalde beperkingen moet kunnen leggen op het soort van patronen dat hij zoekt aan de hand van *queries*, hebben wij drie verschillende methoden bestudeerd om in zulke interactieve data mining sessies deze beperkingen zo goed mogelijk te gebruiken en het data mining proces alsdusdanig efficiënter te kunnen laten verlopen. Meer specifiek bestuderen we een klasse van beperkingen bestaande uit Booleaanse combinaties van atomaire condities, waarin zo een atomaire conditie kan specificeren of een bepaald item in het antecedent of in het consequent van een associatieregel moet voorkomen. Doordat het genereren van associatieregels voornamelijk bestaat uit het genereren van verzamelingen van items, vertalen wij zulke condities onmiddellijk naar beperkingen die eisen dat een item al dan niet mag voorkomen in een verzameling van items. Deze vertaling gebeurt optimaal, in die zin dat we enkel en alleen die verzamelingen genereren die nodig zijn om alle gevraagde associatieregels te kunnen construeren.

In de eerste methode die we voorstellen, wordt elke aparte data mining query geïntegreerd in de mining algoritmes aan de hand van de volgende techniek. Wanneer een query stelt dat alleen die verzamelingen van items mogen gegenereerd worden waarin de items $i_1, \ldots, i_n$ voorkomen, dienen we enkel de databank aan te passen door alle transacties die die items niet bevatten te verwijderen en de items zelf ook nog te verwijderen uit alle overgebleven transacties. Wanneer een query bovendien eist dat alleen die verzamelingen van items dienen gegenereerd te worden waarin de items $j_1, \ldots, j_m$ niet voorkomen dienen we evenzeer enkel de databank aan te passen door uit alle overgebleven transacties ook die items te verwijderen. Als we dan op deze aangepaste databank om het even welk algoritme dat frequente verzamelingen van items genereert uitvoeren, verkrijgen we de correcte uitkomst door achteraf aan elke verzameling de items $i_1, \ldots, i_n$ terug toe te voegen. Door elke Booleaanse query om te zetten in disjuncte disjunctieve normaalvorm, kunnen we de disjuncts op deze manier beantwoorden.

In een tweede methode bekijken we de mogelijkheid om eerst alle mogelijke frequente verzamelingen te genereren voor een zo laag mogelijke minimale support, om daarna elke query te beantwoorden door gewoon de nodige verzamelingen te zoeken aan de hand van zeer efficiënte technieken voor databanken. We tonen aan dat deze methode na verloop van de data mining sessie uiteindelijk veel efficiënter wordt dan de eerst voorgestelde methode.

In een derde methode gebruiken we een combinatie van de twee vorige, door in het begin van een data mining sessie elke query te beantwoorden door de gegeven condities te integreren in het algoritme om de frequente verzamelingen te genereren zoals hierboven werd beschreven, en telkens te resulterende verzamelingen op te slaan in een aparte databank. Daaropvolgende queries worden dan opgedeeld in twee delen, namelijk het deel dat kan beantwoord worden aan de hand van de tweede methode en het deel dat dient beantwoord te worden aan de hand van de eerste methode.

Aan de hand van tal van experimenten met deze drie methoden besluiten we dat de tweede voorgestelde methode in veel gevallen de meest efficiënte uitkomst biedt. Wanneer deze methode echter niet mogelijk is om de reden dat het aantal gegenereerde frequente verzamelingen te groot zou worden, blijft de laatste gecombineerde methode de meest efficiënte oplossing.

Bovengrenzen op het aantal kandidaat verzamelingen Zoals reeds vermeld, trachten verschillende optimalisatietechnieken het aantal scans door de databank te reduceren om het Apriori algoritme efficiënter te laten verlopen. Veel van deze technieken houden echter een groot risico in doordat zij mogelijk een te groot aantal kandidaat verzamelingen zouden genereren met een tegengesteld effect tot gevolg. Aan de basis van deze technieken ligt volgend puur combinatorisch probleem dat eerst opgelost dient te worden vooraleer deze technieken effectief toegepast kunnen worden: *gegeven het huidige aantal frequente verzamelingen in een bepaalde iteratie, hoeveel kandidaat patronen kunnen er nog maximaal gegenereerd worden tijdens komende iteraties?*

Wij beantwoorden deze vraag door een aantal combinatorische bovengrenzen voor te stellen die na elke iteratie van het apriori algoritme efficiënt berekend kunnen worden en als dusdanig een waterdichte garantie geven over het aantal kandidaat verzamelingen dat nog maximaal gegenereerd kan worden.

Experimenten tonen aan dat onze theoretisch verkregen bovengrenzen opmerkelijk goede resultaten vertonen in de praktijk en al zeer snel kunnen voorspellen hoeveel kandidaat verzamelingen er nog maximaal gegenereerd kunnen worden.