Survey on Frequent Pattern Mining

Bart Goethals
HIIT Basic Research Unit
Department of Computer Science
University of Helsinki
P.O. box 26, FIN-00014 Helsinki
Finland

1 Introduction

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 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 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 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:

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

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

Definition 2. 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

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

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

Problem 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 frequent 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 1. Consider the database shown in Table 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 1: An example transaction database $\mathcal{D}$.

Table 2 shows all frequent itemsets in $\mathcal{D}$ with respect to a minimal support threshold of 1. Table 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 [40]. 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 [38]. 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: 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 3: Association rules and their support and confidence in $\mathcal{D}$.

3 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.

3.1 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 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 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 top and the set containing all items at the bottom. 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 [26].

Definition 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 $\mathcal{F}$:

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

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

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

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

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

The lattice for the frequent itemsets from Example 1, together with its borders, is shown in Figure 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

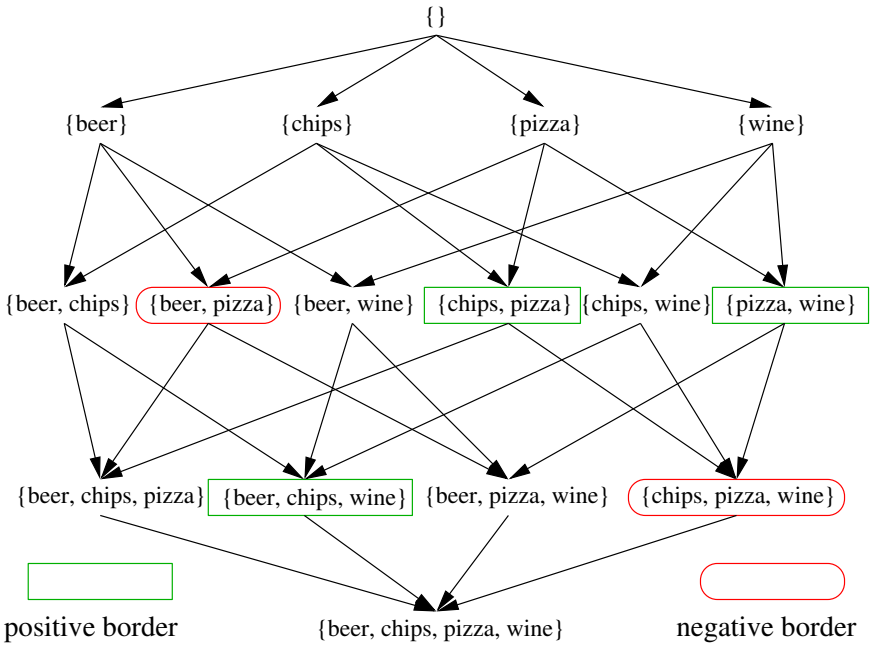


Figure 1: The lattice for the itemsets of Example 1 and its border.

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 [18, 26, 25].

Theorem 2. *[26] 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 [24, 31, 12, 14, 15]. 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.

3.2 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 [33, 41]. Table 4 shows both layouts for the database from Example 1. Note that for both layouts, it is also possible to use the exact bit-strings from the binary matrix [34, 29]. 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 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.

4 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 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,

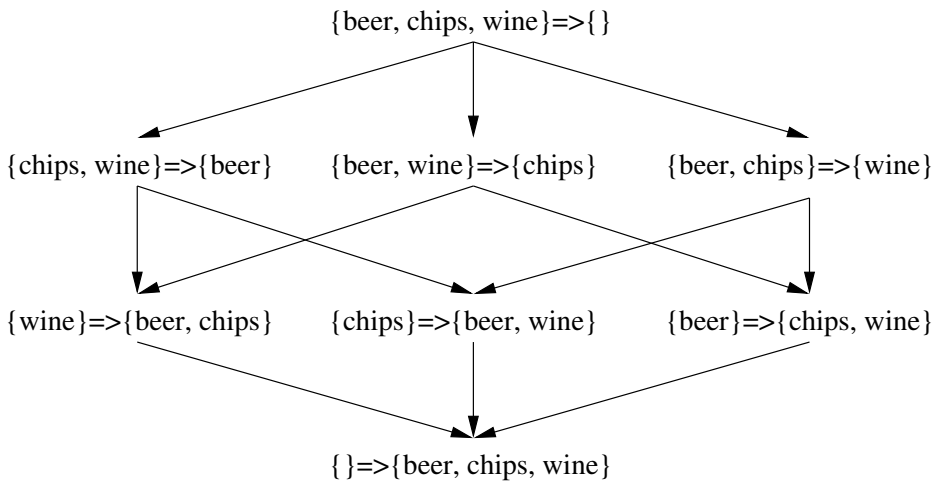


Figure 2: An example of a lattice representing a collection of association rules for $\{\text{beer}, \text{chips}, \text{wine}\}$.

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, 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 top and the rule with all items in the head at the bottom. Figure 2 shows such a lattice for the itemset $\{\text{beer}, \text{chips}, \text{wine}\}$, which was found to be frequent on the artificial data set used in Example 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.

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 5: Data Set Characteristics.

5 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 learning databases [10]. 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 [23]. The basket data set contains transactions from a Belgian retail store, but can unfortunately not be made publicly available. Table 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 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 6: Data Set Characteristics.

6 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, 36]. The same technique was independently proposed by Mannila et al. [27]. Both works were cumulated afterwards [4].

6.1 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} | i ∈ I}
2:  k := 1
3:  while C_k ≠ {} do
4:    // Compute the supports of all candidate itemsets
5:    for all transactions (tid, I) ∈ D do
6:      for all candidate itemsets X ∈ C_k do
7:        if X ⊆ I then
8:          X.support++
9:        end if
10:     end for
11:   end for
12:   // Extract all frequent itemsets
13:   F_k := {X | X.support ≥ σ}
14:   // Generate new candidate itemsets
15:   for all X, Y ∈ F_k, X[i] = Y[i] for 1 ≤ i ≤ k − 1, and X[k] < Y[k] do
16:     I = X ∪ {Y[k]}
17:     if ∀J ⊂ I, |J| = k : J ∈ F_k then
18:       C_{k+1} := C_{k+1} ∪ 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.

6.2 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.

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 confi-

Algorithm 2 Apriori - Association Rule mining

Input: $\mathcal{D}, \sigma, \gamma$
Output: $\mathcal{R}(\mathcal{D}, \sigma, \gamma)$

Compute $\mathcal{F}(\mathcal{D}, \sigma)$
$\mathcal{R} := \{\}$
for all $I \in \mathcal{F}$ **do**
  $\mathcal{R} := \mathcal{R} \cup I \Rightarrow \{\}$
  $C_1 := \{\{i\} \mid i \in I\}$;
  $k := 1$;
  while $C_k \neq \{\}$ **do**
    // Extract all heads of confident association rules
    $H_k := \{X \in C_k \mid \mathit{confidence}(I \setminus X \Rightarrow X, \mathcal{D}) \geq \gamma\}$
    // Generate new candidate heads
    for all $X, Y \in H_k, X[i] = Y[i]$ for $1 \leq i \leq k-1$, and $X[k] < Y[k]$ **do**
      $I = X \cup \{Y[k]\}$
      if $\forall J \subset I, |J| = k : J \in H_k$ **then**
        $C_{k+1} := C_{k+1} \cup I$
      end if
    end for
    k++
  end while
  // Cumulate all association rules
  $\mathcal{R} := \mathcal{R} \cup \{I \setminus X \Rightarrow X \mid X \in H_1 \cup \cdots \cup H_k\}$
end for

dent. 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.

6.3 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.

6.3.1 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.

6.3.2 Trie

Another data structure that is commonly used is a trie (or prefix-tree) [8, 11, 13, 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 [11].

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 [11, 13, 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 [11].

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

6.4 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.

6.4.1 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 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.

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**

6.4.2 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 [30]. 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 [35]. 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 [28, 29].

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.

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

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

Proof.

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

□

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 4 even more.

Corollary 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 [31, 12]. 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 [15]. 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.

6.4.4 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.

6.4.5 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 [13]. 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 [21] 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.

6.4.6 Sampling

The sampling algorithm, proposed by Toivonen [39], 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.

6.4.7 Partitioning

The Partition algorithm, proposed by Savasere et al. uses an approach which is completely different from all previous approaches [33]. 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.

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: $\quad \mathcal{F}_k := \{\}$
5: $\quad$ **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: $\quad\quad I = \{X[1], \ldots, X[k-1], Y[k-1]\}$
7: $\quad\quad$ **if** $\forall J \subset I : J \in \mathcal{F}_{k-1}$ **then**
8: $\quad\quad\quad I.cover := X.cover \cap Y.cover$
9: $\quad\quad\quad$ **if** $|I.cover| \geq \sigma$ **then**
10: $\quad\quad\quad\quad \mathcal{F}_k := \mathcal{F}_k \cup I$
11: $\quad\quad\quad$ **end if**
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: $\quad k{+}{+}$
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: $\quad$ 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: $\quad$ Generate cover of each item in D_p
11: $\quad$ **for all** $I \in C_{global}$ **do**
12: $\quad\quad I.support := I.support + |I[1].cover \cap \cdots \cap I[|I|].cover|$
13: $\quad$ **end for**
14: **end for**
15: // Extract all global frequent itemsets
16: $\mathcal{F} := \{I \in C_{global} \mid I.support \geq \sigma\}$

The exact computation of the supports of all itemsets can still be optimized, but we refer to the original article for further implementation details [33].

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.

7 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 [41, 44]. Later, several other depth-first algorithms have been proposed [1, 2, 20] of which the FP-growth algorithm by Han et al. [20, 19] 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.

7.1 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.

Algorithm 6 Eclat

Input: $\mathcal{D}, \sigma, I \subseteq \mathcal{I}$
Output: $\mathcal{F}[I](\mathcal{D}, \sigma)$

```
1:  F[I] := {}
2:  for all i ∈ I occurring in D do
3:     F[I] := F[I] ∪ {I ∪ {i}}
4:     // Create D^i
5:     D^i := {}
6:     for all j ∈ I occurring in D such that j > i do
7:        C := cover({i}) ∩ cover({j})
8:        if |C| ≥ σ then
9:           D^i := D^i ∪ {(j, C)}
10:       end if
11:    end for
12:    // Depth-first recursion
13:    Compute F[I ∪ {i}](D^i, σ)
14:    F[I] := F[I] ∪ F[I ∪ {i}]
15: end for
```

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 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 [42, 43] 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* [42]. 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 [42].

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 [22]. 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 6.4. 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.

7.2 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 [20]. 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. 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

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 7: An example preprocessed transaction database.

reordered according to the support descending order resulting in the example transaction database in Table 7. The FP-tree for this database is shown in Figure 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 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.

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

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

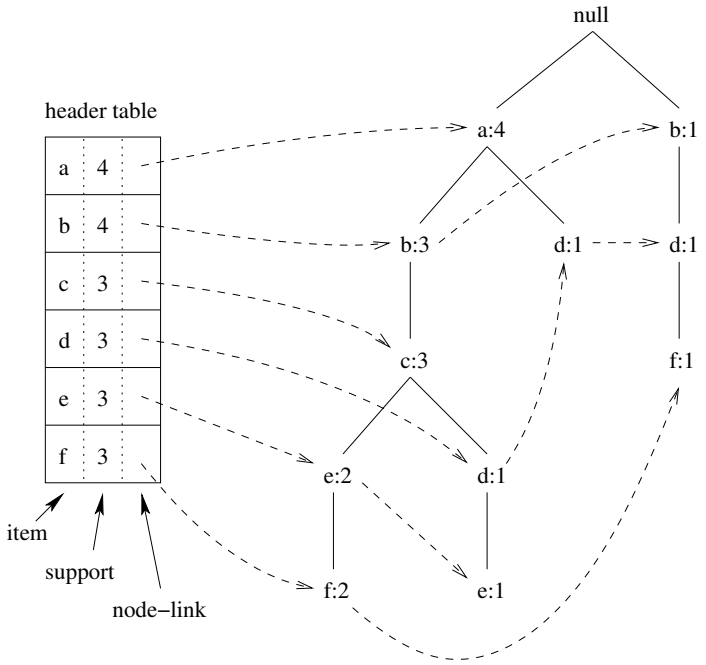


Figure 3: An example of an FP-tree.

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

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 8: Memory usage of Eclat versus FP-growth.

i-projected database $\mathcal{D}^i$.

Although the authors of the FP-growth algorithm claim that their algorithm [19, 20] 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 8 shows for all four used data sets 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.

8 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 [37]. 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.

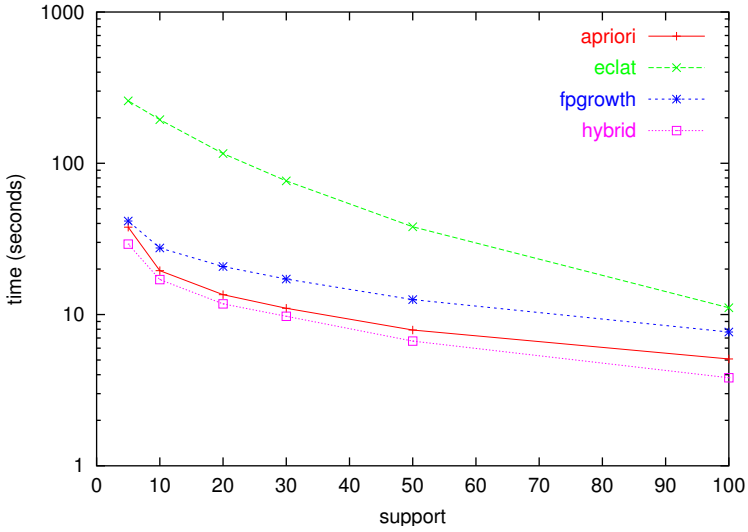
Figure 8 shows the performance of the algorithms on each of the data sets described in Section 5 for varying minimal support thresholds.

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.

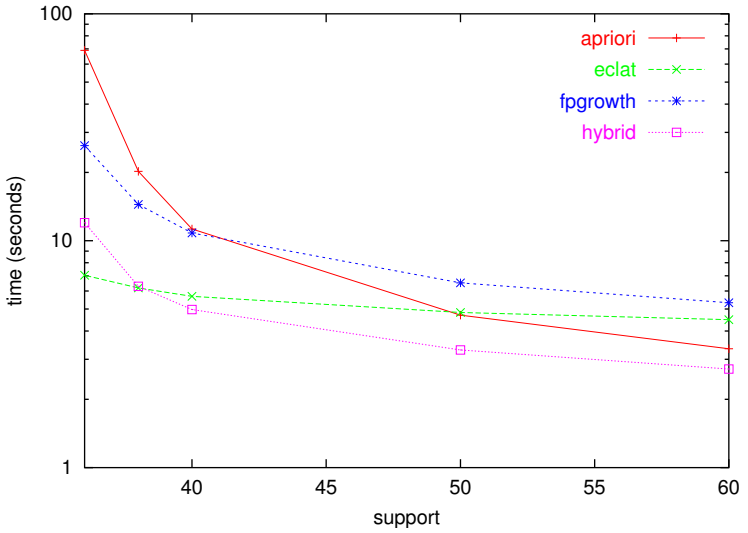
Another remarkable result is that Apriori performs better than FP-growth for the basket data set. This result is due to the overhead created by the maintenance of the FP-tree structure, while updating the supports of all candidate itemsets contained in each transaction is performed very fast due to the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

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

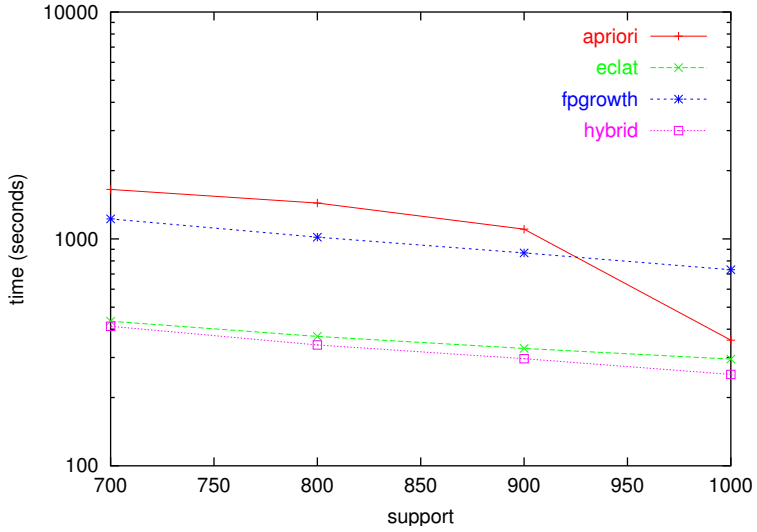


(a) basket

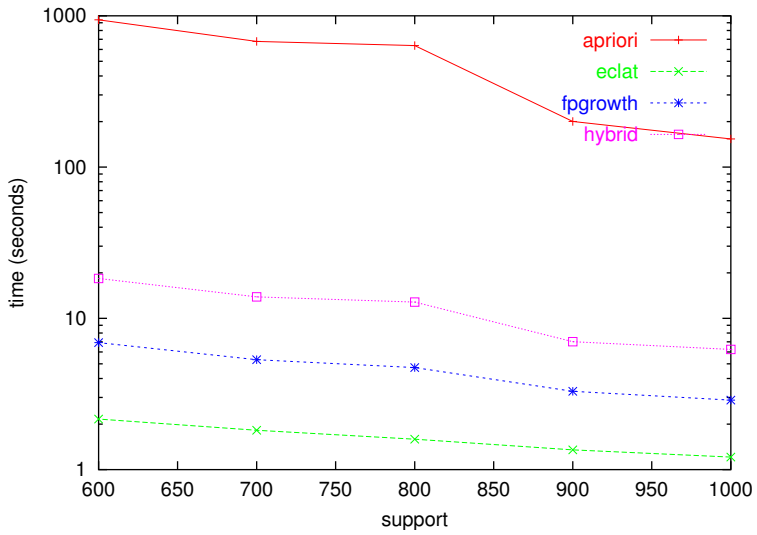


(b) BMS-Webview-1

Figure 4: Frequent itemset mining performance.



(c) T40I10D100K



(d) mushroom

Figure 4: Frequent itemset mining performance.

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.

9 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 [11]. 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 [45] 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.

References

[1] R.C. Agarwal, C.C. Aggarwal, and V.V.V. Prasad. Depth first generation of long patterns. In Ramakrishnan et al. [32], 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 U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. MIT Press, 1996.

[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 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*, pages 85–93. ACM Press, 1998.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] T. Calders and B. Goethals. Mining all non-derivable frequent itemsets. In 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*, pages 74–85. Springer, 2002.

[16] 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.

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

[18] 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.

[19] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In Chen et al. [16], pages 1–12.

[20] 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.

[21] C. Hidber. Online association rule mining. In 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*, pages 145–156. ACM Press, 1999.

[22] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Mining association rules: Deriving a superior algorithm by analyzing today's approaches. In 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*, pages 159–168. Springer, 2000.

[23] 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`.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In Dayal et al. [17], pages 432–444.

[34] 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. [16], pages 22–33.

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

[36] R. Srikant and R. Agrawal. Mining generalized association rules. In Dayal et al. [17], pages 407–419.

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

[38] P. Tan, V. Kumar, and J. Srivastava. Selecting the right interestingness measure for association patterns. In D. Hand, D. Keim, and R.T. Ng, editors, *Proceedings of the Eight ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 32–41. ACM Press, 2002.

[39] H. Toivonen. Sampling large databases for association rules. In T.M. Vijayaraman, A.P. Buchmann, C. Mohan, and N.L. Sarda, editors, *Proceedings 22nd International Conference on Very Large Data Bases*, pages 134–145. Morgan Kaufmann, 1996.

[40] G.I. Webb. Efficient search for association rules. In Ramakrishnan et al. [32], pages 99–107.

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

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

[43] 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.

[44] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In D. Heckerman, H. Mannila, and D. Pregibon, editors, *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, pages 283–286. AAAI Press, 1997.

[45] 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.