

# 1

---

## Introduction

Current technology makes it fairly easy to collect data, but data analysis tends to be slow and expensive. Knowledge discovery in databases (KDD), often called data mining, aims at the discovery of useful information from large collections of data. The motivation for KDD is a suspicion that there might be nuggets of useful information hiding in the masses of unanalyzed or underanalyzed data, and therefore semiautomatic methods for locating interesting information from data would be useful. The discovered knowledge can be rules describing properties of the data, frequently occurring patterns, clusterings of the objects in the database, etc. There are several successful applications of data mining. See [23] for a recent overview of the area.

This introductory chapter gives a short discussion of some of the issues in knowledge discovery. We start in Section 1.1 by looking at the definition of knowledge discovery and some small examples, and we briefly discuss the basic goals of knowledge discovery. The KDD process is considered in Section 1.2. Some applications of KDD are described in Section 1.3. Section 1.4 discusses the role of machine learning and statistics in KDD and data mining, Section 1.5 the role of databases.

### 1.1 What is knowledge discovery

Knowledge discovery in database can be loosely defined as the task of finding interesting and potentially useful knowledge from large masses of data. Examples of types of knowledge that could be discovered are

- rules:
  - ”80 % of customers who buy beer and sausage buy also mustard”
  - ”if Age < 40 then Income < 10”

- trends
- classifiers
- clusterings
- predictions

To illustrate one of these concepts, we consider briefly association rules. An *association rule* [3] about a relation  $r$  over schema  $R$  is an expression of the form  $X \Rightarrow Y$ , where  $X, Y \subseteq R$  and  $X \cap Y = \emptyset$ . The intuitive meaning of the rule is that if a row of the matrix  $r$  has a 1 in each column of  $X$ , then the row also tends to have a 1 in each column of  $Y$ .

Examples of data where association rules might be applicable include the following.

- A student database at a university: rows correspond to students, columns to courses, and a 1 in entry  $(s, c)$  indicates that the student  $s$  has taken course  $c$ .
- Data collected from bar-code readers in supermarkets: columns correspond to products, and each row corresponds to the set of items purchased at one time.
- A database of publications: the rows and columns both correspond to publications, and  $(p, p') = 1$  means that publication  $p$  refers to publication  $p'$ .
- A set of measurements about the behavior of a number of systems, say exchanges in a telephone network. The columns correspond to the presence or absence of certain conditions, and each row corresponds to a measurement: if entry  $(m, c)$  is 1, then at measurement  $m$  condition  $c$  was present.

Given  $X \subseteq R$ , we denote by  $fr(X, r)$  the *frequency* of  $X$  in  $r$ : the fraction of rows of  $r$  that have a 1 in each column of  $X$ . The *frequency* of the rule  $X \Rightarrow Y$  in  $r$  is defined to be  $fr(X \cup Y, r)$ , and the *confidence* of the rule is  $fr(X \cup Y, r)/fr(X, r)$ . The confidence is the observed probability with which a row containing  $X$  also contains  $Y$ .

In the discovery of association rules, the task is to find all rules  $X \Rightarrow Y$  such that the frequency of the rule is at least a given threshold  $min\_fr$  and the confidence of the rule is at least another threshold  $min\_conf$ . In large retailing applications the number of rows might easily be  $10^6$  or  $10^7$ , over thousands of columns. Both the frequency threshold  $min\_fr$  and the confidence threshold  $min\_conf$  can be any percentage. From such databases one might obtain hundreds or thousands of association rules. (Of course, one

has to be careful in assigning any statistical significance to findings obtained with such methods.)

Note that there is no predefined limit on the number of attributes of an association rule  $X \Rightarrow Y$ ; this is important so that unexpected associations are not ruled out before the processing starts. It also means that the search space of the rules has exponential size in the number of attributes of the input relation. Handling this requires some care for the algorithms, but there is a simple way of pruning the search space.

**Example 1.1.** *Discovered in a student enrolment database, the association rule { Distributed Operating Systems, Introduction to Unix }  $\Rightarrow$  Programming in C (frequency = 2 %, confidence = 96 %) states that 96 % of the students that have taken Distributed Operating Systems and Introduction to Unix, also have taken Programming in C, and that 2 % of all the students actually have taken all three courses. Such rules can be useful in obtaining a picture of the combinations of courses actually taken by the students. The acquired knowledge can be applied, e.g., in the design of individual courses and the whole curriculum.*

*In market basket analysis, the number of products being sold can be large and it is more difficult to have a hunch for all the associations in the data. On the other hand, discovered patterns can be very valuable for the business. A well-known story tells how a surprising association was discovered between diapers and beer in late afternoons. With some research a fairly natural cause was discovered: young fathers buying diapers on the way back home from work. This simple but surprising observation could maybe be capitalized by placing beer closer to diapers, in order to make the impulse shopping behaviour stronger, and maybe by placing chips right next to diapers and beer.*

## 1.2 The KDD process

The goal of knowledge discovery is to obtain useful knowledge from large collections of data. Such a task is inherently interactive and iterative: one cannot expect to obtain useful knowledge simply by pushing a lot of data to a black box. The user of a KDD system has to have a solid understanding of the domain in order to select the right subsets of data, suitable classes of patterns, and good criteria for interestingness of the patterns. Thus KDD systems should be seen as interactive tools, not as fully automatic analysis systems.

Discovering knowledge from data should therefore be seen as a process containing several steps:

1. understanding the domain;
2. preparing the data set;

3. discovering patterns (data mining),
4. postprocessing and presenting the discovered patterns, and
5. putting the results into use.

(See [22] for a slightly different process model and excellent discussion.)

Understanding the domain of the data is naturally a prerequisite for extracting anything useful: the user of a KDD system has to have some sort of understanding about the application area before any valuable information can be obtained. As an example, consider the discovered association rule between diapers and beer. Beer probably is purchased in addition to diapers, and the rule indicates a link that can potentially be strengthened. On the other hand, if a strong association is discovered between bread and butter, the actions taken could be opposite: placing bread and butter far away from each other would force customers who really came to pick up both items to spend more time in the store. Critical interpretation of data mining results and the drawing of conclusions typically require good domain knowledge. Data mining does not work by pushing a button.

If very good human experts exist for a domain, it can be hard for semi-automatic tools to obtain any novel information. This can be the case in fairly stable domains, where the humans have had time to achieve expertise even in the details of the data. A possible example occurs in some areas of retailing, where the products and customer profiles can stay about the same for longer periods of time. The easiest application areas for KDD seem to be ones where general human experts can be found, but the actual microlevel properties of the data are changing. This seems to be the case in, e.g., telecommunications, where the operators of the networks have a fairly good overview of the systems characteristics, but changes and updates in equipment and software mean that actual expertise in the details of the data is more difficult to obtain.

Preparation of the data set involves selection of the data sources, integration of heterogeneous data, cleaning the data from errors, assessing noise, dealing with missing values, etc. This step can easily take up most of the time needed for the whole KDD process. This is not surprising: the difficulties in data integration are well known.

The pattern discovery phase in KDD is the step where the interesting and frequently occurring patterns are discovered from the data. In this book we follow the terminology introduced in [22]: data mining refers to the pattern discovery part of knowledge discovery. Elsewhere, especially in industry, data mining is often used as a synonym for KDD.

The data mining step can use various techniques from statistics and machine learning, such as rule learning, decision tree induction, clustering, inductive logic programming, etc. The emphasis in data mining research is mostly on efficient discovery of fairly simple patterns.

The KDD process does not stop when patterns have been discovered. The user has to be able to understand what has been discovered, to view the data and patterns simultaneously, contrast the discovered patterns with background knowledge, etc. Postprocessing of discovered knowledge involves steps such as further selection or ordering of patterns, visualization, etc. Some approaches to KDD methodology put a heavy emphasis on postprocessing.

The KDD process is necessarily iterative: the results of a data mining step can show that some changes should be made to the data set formation step, postprocessing of patterns can cause the user to look for some slightly modified types of patterns, etc. Efficient support for such iteration is one important topic of development in KDD.

## 1.3 Applications of knowledge discovery

### 1.3.1 Scientific applications

Prominent applications of KDD include health care data, financial applications, and scientific data [33, 29, 20].

One of the more spectacular applications is the SKICAT system [21], which operates on 3 terabytes of image data. Image processing of the pixels produces approximately 2 billion objects, basically smudges of light, each with 40 attributes. To be usable, the objects have to be classified into a few classes: stars, galaxies, etc.

The task is obviously impossible to do manually. Using example classifications provided by experts, the system induced decision trees and extracted classification rules for the problem. The results are spectacular: verification shows that the resulting classification is accurate, and the classification has already been used to discover new high-redshift quasars.

Another astronomical application has been done on the radar data produced by the Magellan spacecraft that has surveyed the surface of Venus using radar. The basic problem was to find out which features observed on the surface are volcanos and which are not. The problem is complicated by the fact that finding the “ground truth” is by no means trivial: it is not possible to obtain more accurate information about the surface of Venus than that obtained by the Magellan spacecraft. The approach taken in [15] was to search automatically for volcanos by first training the system using examples provided by geologists.

In biological sciences the ability to analyze the 1-dimensional structure of genes and proteins has made significant advances possible. Finding the structures and functions of proteins is central in molecular biology. The protein and gene data sets yield several important data mining problems, such as the location of recurrent motifs from protein sequences, and similarity searches among large sets of sequences. Several research groups have

recently used sophisticated hidden Markov model (HMM) methods to look for such structural patterns [30, 31].

### 1.3.2 Business applications

In business, the main area of application for KDD techniques has been marketing. A typical problem is targeting of mail advertising: how to determine which products should be offered to customers, on the basis of their past purchase behavior? A good example of this type of work is the system Opportunity Explorer [9].

In many business domains, publication of details of successful systems is rare. Often the reason is simple: if data mining gives a competitive advantage, it is not wise to spread the word to competitors. Successful applications are reported in portfolio management, fraud detection, manufacturing and production, and network management. See [13] for an overview of applications.

### 1.3.3 Data warehousing, OLAP, and knowledge discovery

In industry, the success of KDD is partly related to the rise of the concepts of *data warehousing* and *on-line analytical processing (OLAP)*. These strategies for the storage and processing of the accumulated data in an organization have become popular in recent years. KDD and data mining can be viewed as ways of realizing some of the goals of data warehousing and OLAP.

In principle, a data warehouse aims at the storage and processing of all relevant business data on an enterprise wide level for different analysis tasks. The motivation for data warehousing is that such integrated data storage can be immensely useful for making valuable inferences about the business across the enterprise.

Knowledge discovery is a good way of using the data in the data warehouse. Data warehousing, on the other hand, makes long-term knowledge discovery cheaper, as a data warehouse removes a lot of the effort needed in the early stages of the KDD process. On the other hand, data warehousing often is very expensive: the integration of data from different sources can be quite costly.

On-line analytical processing, OLAP, is a term introduced as a counterpart to *on-line transaction processing*, OLTP, the traditional mode of database usage. In OLAP the emphasis is on producing different types of multidimensional reports for the use of strategic decision making in the organization. A possible way of differentiating between KDD and OLAP is to say that OLAP is oriented towards *verification*, whereas KDD is typically aiming at *discovery*. In OLAP systems, the user typically has a question in

his mind that he wants to be answered, whereas in KDD the user typically has a less clear view of what is to be found from the data.

The boundary of KDD and OLAP is vague, and typically a successful KDD session leads to some more focused questions that could be labeled OLAP-type.

## 1.4 KDD versus machine learning and statistics

Data mining combines methods and tools from at least three areas: machine learning, statistics, and databases. Indeed, one can sometimes hear the following comments.

- Data mining is just machine learning!
- Data mining is just statistics!
- What does data mining have to do with databases?

In this section we discuss briefly the first two points; the next sections are devoted to a discussion on the third point.

The close links between machine learning, statistics, and data mining are fairly obvious. All three areas aim at locating interesting regularities, patterns, or concepts from empirical data. The exact relationships of these areas have been subject to some debate.

*Machine learning* methods form the core of data mining: decision tree learning or rule induction is one of the main components of several data mining algorithms. There are some differences, however.

The emphasis on the *process* of knowledge discovery is one; large parts of machine learning literature concentrate on just the learning or induction step, although exceptions of course exist.

The next difference concerns the relative roles of concepts and data. It seems that most of machine learning research assumes there is *something to be learned*, i.e., that there is an underlying interesting concept or mechanism that produces the data. The data can be corrupted by noise, errors, etc., but still the idea is that there is an interesting concept at the bottom. In knowledge discovery, on the other hand, the data is the primary thing, and one does not necessarily assume that there would be any sensible structure behind the data. For example, in analyzing retail sales data, the data is what it is, and the users are not interested in obtaining a full understanding of it; useful nuggets of information are sufficient. Of course, this difference is not absolute.

A third difference is related to the goals. KDD systems typically have fairly modest aims, in terms of the complexity of the obtained knowledge. Whereas parts of machine learning research aim at learning things that are difficult for humans to do, most of the work in KDD aims at finding

knowledge that a competent data analyst would in principle be able to find, if he had the time. This distinction is particularly evident when one compares the area of machine discovery to knowledge discovery.

An often cited difference between KDD and machine learning is the amount of data. Traditionally, machine learning research has concentrated on looking at data sets containing hundreds or thousands of examples, while KDD applications consider larger data sets. It is not clear how significant this distinction is, however: some machine learning work has been done on huge data sets, and KDD methods can be useful even on small data collections. Furthermore, the essential source of complexity in data mining is typically not the number of objects in the database, but rather the number of attributes: the number of possible patterns typically grows at least exponentially in the number of attributes. This growth is the real source of difficulty, not the number of objects in the database.

Summarizing, machine learning is at the core of KDD, but there are differences between the areas.

In *statistics* the term data mining has been used for a long time, often in slightly derogatory fashion, as referring to data analysis without clearly formulated hypotheses. A more fashionable term is *exploratory data analysis* (EDA) [46], which stresses the supremacy of data as guiding the analysis process. KDD and EDA have very similar aims and methods.

According to the interesting statistical perspective on KDD by Elder and Pregibon [19], the focus of statistics has gradually moved from model estimation to model selection. Instead of looking for the parameter values that make a model fit the data well, also the model structure is part of the search process. This trend fits the goals of KDD nicely: one does not want to fix the model structure in advance. The recent advances in, say Markov chain Monte Carlo (MCMC) methods, make it possible to consider far larger model spaces than previously. In addition to these techniques, the KDD community has lots to learn from statistics, e.g., in the handling of uncertainty.

The main difference between KDD and statistics is perhaps in the extensive use of machine learning methods in KDD, in the volume of data, and in the role of computational complexity issues in KDD. For example, even MCMC methods have difficulties in handling tens of thousands of parameter values; some sort of combinatorial preprocessing is needed to make the model selection task tractable. It seems that such combinations of methods can be useful: combinatorial techniques are used to prune the search space, and statistical methods are used to explore the remaining parts in great detail.

## 1.5 Databases and data mining

What is the role of database management systems in data mining? Normal use of databases can be seen as *deduction*, whereas knowledge discovery aims at *induction*. Furthermore, parts of database technology are not very relevant to knowledge discovery. For example, recovery and transaction management are issues that a KDD application typically does not have to care about.

Nevertheless, database management systems have been especially developed for the storage and flexible retrieval of large masses of structured data, so at least in principle they should be suited for KDD. What database systems have to offer is basically fast access to certain subgroups of a data set and efficient computation of some characteristics of such subgroups.

Usual database systems are not very well suited for such tasks. Implementation of classification algorithms (say, C4.5) or neural networks on top of a large database require tighter coupling with the database system and smart use of indexing techniques. For example, training a classifier on a large training set stored in a database requires possibly multiple passes through the data using different orderings between attributes. This can be implemented by utilizing DBMS support for aggregate operations, indexes and database sorting ('order by'). Clustering may require efficient implementations of nearest neighbor algorithms on the top of large databases. Finally, generation of association rules can be performed in many different ways, depending on the amount of main memory available. There have been a growing number of papers on this subject at recent VLDB and SIGMOD conferences.

Database mining should learn from the general experience of DBMS field and follow one of the key DBMS paradigms [28]: building optimizing compilers for ad hoc queries and embedding queries in application programming interfaces. Thus, the focus should be on increasing *programmer productivity* for KDD application development.

Queries, however have to be much more general than SQL; similarly, the queried objects have to be far more complex than records (tuples) in relational database.



# 2

---

## Discovery of association rules

Frequent sets 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. [4], 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 Association rules

Given a collection of sets of items, association rules describe how likely various combinations of items are to occur together in the same sets. The simple data model we consider is the following.

Row ID	Row
$t_1$	$\{A, B, C, D, G\}$
$t_2$	$\{A, B, E, F\}$
$t_3$	$\{B, I, K\}$
$t_4$	$\{A, B, H\}$
$t_5$	$\{E, G, J\}$

Figure 2.1: An example 0/1 relation  $r$  over the set  $R = \{A, \dots, K\}$ .

**Definition 2.1.** Given a set  $R$ , a 0/1 relation  $r$  over  $R$  is a collection (or multiset) of subsets of  $R$ . The elements of  $R$  are called items, and the elements of  $r$  are called rows. The number of rows in  $r$  is denoted by  $|r|$ , and the size of  $r$  is  $\|r\| = \sum_{t \in r} |t|$ .

In the literature, the rows of such databases are often referred to as *transactions*, with its origin in the initial problem setting. In this chapter, we will also often refer to the used database as a transaction database and to its content as transactions.

**Example 2.2.** In the domain of supermarket basket analysis, items represent products in the stock. There could be items such as beer, chips, diapers, milk, bread, and so on. A row in a basket database then corresponds to the contents of a shopping basket: for each product type in the basket, the corresponding item is in the row. If a customer purchased milk and diapers, then there is a corresponding row  $\{\text{milk}, \text{diapers}\}$  in the database. The quantity or price of items is not considered in this model, only the binary information whether a product was purchased or not.

The number of different items can be in the order of thousands, whereas typical purchases only contain at most dozens of items. When practical storage structures for such sparse databases are used, the physical database size closely corresponds to  $\|r\|$ .

An interesting property of a set  $X \subseteq R$  of items is how many rows contain it. This brings us to the formal definition of the term “frequent”.

**Definition 2.3.** Let  $R$  be a set and  $r$  a 0/1 relation over  $R$ , and let  $X \subseteq R$  be a set of items. The set  $X$  matches a row  $t \in r$ , if  $X \subseteq t$ . The (multi) set of rows in  $r$  matched by  $X$  is denoted by  $\mathcal{M}(X, r)$ , i.e.,  $\mathcal{M}(X, r) = \{t \in r \mid X \subseteq t\}$ , also called the cover of  $X$ . The frequency of  $X$  in  $r$ , denoted by  $fr(X, r)$ , is  $\frac{|\mathcal{M}(X, r)|}{|r|}$ . We write simply  $\mathcal{M}(X)$  and  $fr(X)$  if the database is unambiguous in the context. Given a frequency threshold  $min\_fr \in [0, 1]$ , the set  $X$  is frequent<sup>1</sup> if  $fr(X, r) \geq min\_fr$ .

<sup>1</sup>In the literature, also the terms *large* and *covering* have been used for “frequent”, and the term *support* for “frequency”.

Row ID	A	B	C	D	E	F	G	H	I	J	K
$t_1$	1	1	1	1	0	0	1	0	0	0	0
$t_2$	1	1	0	0	1	1	0	0	0	0	0
$t_3$	0	1	0	0	0	0	0	0	1	0	1
$t_4$	1	1	0	0	0	0	0	1	0	0	0
$t_5$	0	0	0	0	1	0	1	0	0	1	0

Figure 2.2: The example 0/1 relation  $r$  in relational form over 0/1-valued attributes  $\{A, \dots, K\}$ .

**Example 2.4.** Consider the 0/1 relation  $r$  over the set  $R = \{A, \dots, K\}$  in Figure 2.1. We have, for instance,  $\mathcal{M}(\{A, B\}, r) = \{t_1, t_2, t_4\}$  and  $\text{fr}(\{A, B\}, r) = 3/5 = 0.6$ . The database can be viewed as a relational database over the schema  $\{A, \dots, K\}$ , where  $A, \dots, K$  are 0/1-valued attributes, hence the name “0/1 relation”. Figure 2.2 presents the database in this form.

The frequency threshold  $\text{min\_fr}$  is a parameter given by the user and depends on the application. For notational convenience, we next introduce notations for collections of frequent sets.

**Definition 2.5.** Let  $R$  be a set,  $r$  a 0/1 relation over  $R$ , and  $\text{min\_fr}$  a frequency threshold. The collection of frequent sets in  $r$  with respect to  $\text{min\_fr}$  is denoted by  $\mathcal{F}(r, \text{min\_fr})$ ,

$$\mathcal{F}(r, \text{min\_fr}) = \{X \subseteq R \mid \text{fr}(X, r) \geq \text{min\_fr}\},$$

or simply by  $\mathcal{F}(r)$  if the frequency threshold is clear in the context. The collection of frequent sets of size  $l$  is denoted by

$$\mathcal{F}_l(r) = \{X \in \mathcal{F}(r) \mid |X| = l\}.$$

**Example 2.6.** Assume that the frequency threshold is 0.3. The collection of frequent sets in the database  $r$  of Figure 2.1 is then

$$\mathcal{F}(r, 0.3) = \{\{A\}, \{B\}, \{E\}, \{G\}, \{A, B\}\},$$

since no other non-empty set occurs in more than one row. The empty set  $\emptyset$  is trivially frequent in every 0/1 relation; we often ignore the empty set as a non-interesting case.

We now move on and define association rules. An association rule states that a set of items tends to occur in the same row with another set of items. Associated with each rule are two factors: its confidence and frequency.

**Definition 2.7.** Let  $R$  be a set,  $r$  a 0/1 relation over  $R$ , and  $X, Y \subseteq R$  sets of items. Then the expression  $X \Rightarrow Y$  is an association rule over  $r$ . The confidence of  $X \Rightarrow Y$  in  $r$ , denoted by  $\text{conf}(X \Rightarrow Y, r)$ , is  $\frac{\text{fr}(X \cup Y, r)}{\text{fr}(X, r)}$ . The frequency  $\text{fr}(X \Rightarrow Y, r)$  of  $X \Rightarrow Y$  in  $r$  is  $\text{fr}(X \cup Y, r)$ . We write simply  $\text{conf}(X \Rightarrow Y)$  and  $\text{fr}(X \Rightarrow Y)$  if the database is unambiguous in the context. Given a frequency threshold  $\text{min\_fr}$  and a confidence threshold  $\text{min\_conf}$ ,  $X \Rightarrow Y$  holds in  $r$  if and only if  $\text{fr}(X \Rightarrow Y, r) \geq \text{min\_fr}$  and  $\text{conf}(X \Rightarrow Y, r) \geq \text{min\_conf}$ .

In other words, the confidence  $\text{conf}(X \Rightarrow Y, r)$  is the conditional probability that a randomly chosen row from  $r$  that matches  $X$  also matches  $Y$ . The frequency of a rule is the amount of positive evidence for the rule. For a rule to be considered interesting, it must be strong enough and common enough. The association rule discovery task [3] is now the following: given  $R$ ,  $r$ ,  $\text{min\_fr}$ , and  $\text{min\_conf}$ , find all association rules  $X \Rightarrow Y$  that hold in  $r$  with respect to  $\text{min\_fr}$  and  $\text{min\_conf}$ . Note that every frequent set  $X$  in  $r$  also represents the association rule  $X \Rightarrow \{\}$  that holds in  $r$  with respect to  $\text{min\_fr}$  and  $\text{min\_conf}$  (the rule always holds with 100%). In practice, we only consider those rules for which  $X$  and  $Y$  are disjoint and non-empty.

**Example 2.8.** Consider, again, the database in Figure 2.1. Suppose we have frequency threshold  $\text{min\_fr} = 0.3$  and confidence threshold  $\text{min\_conf} = 0.9$ . The only association rule with disjoint and non-empty left and right-hand sides that holds in the database is  $\{A\} \Rightarrow \{B\}$ . The frequency of the rule is  $0.6 \geq \text{min\_fr}$ , and the confidence is  $1 \geq \text{min\_conf}$ . The rule  $\{B\} \Rightarrow \{A\}$  does not hold in the database as its confidence  $0.75$  is below  $\text{min\_conf}$ .

Finding all association rules that hold in a 0/1 relation comprises two main challenges. First, typical databases used for association rule mining tend to be very large and can not be stored main memory. Second, the search space of all association rules contains exactly  $3^{|R|}$  rules, of which  $2^{|R|}$  represent all item sets. Since  $R$  typically contains thousands of attributes, it is infeasible to simply generate and count the frequencies of all possible item sets within reasonable time.

The first algorithm proposed to solve the association rule mining problem was divided into two phases [3]. In the first phase, all frequent sets 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 exists a successful algorithm, called *MagnumOpus*, that uses another strategy to immediately generate a predefined subset of all association rules [47]. We will not discuss this algorithm any further, since our focus is solely on generating all association rules.

Next to the frequency 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 [44]. Here, we only consider algorithms within the frequency-confidence framework as presented before.

## 2.2 Frequent set generation

The task of discovering all frequent sets is quite challenging. The search space is exponential in the number of attributes occurring in the database, although the frequency threshold limits the output to a hopefully reasonable subspace. Additionally, such databases could be massive, containing millions of rows, making frequency counting a hard problem. In this section, we will analyze these two aspects into further detail.

### 2.2.1 The search space

Exhaustive search of frequent sets is obviously infeasible for all but the smallest sets since the search space of potential frequent sets consists of the  $2^{|R|}$  subsets of  $R$ . Instead, we could generate only those sets 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 sets* are generated and their frequencies computed until all frequent sets have been generated. Obviously, the size of a collection of candidate sets must not exceed the size of available main memory. Moreover, it is important to generate as few candidate sets as possible, since computing the frequencies of a collection of sets is a time consuming procedure. In the best case, only the frequent sets are generated and counted. Unfortunately, this ideal is impossible in general, which will be shown later in this chapter.

The main underlying property exploited by most algorithms is that frequency is monotone decreasing with respect to extension of a set.

**Proposition 2.9. (Frequency monotonicity)** *Let  $X, Y \subseteq R$  be two sets. Then,*

$$X \subseteq Y \Rightarrow fr(X) \geq fr(Y).$$

*Proof.* This follows immediately from  $\mathcal{M}(Y) \subseteq \mathcal{M}(X)$ , which follows immediately from  $X \subseteq Y$   $\square$

Hence, if a set 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 sets is closed with respect to set inclusion.

### 2.2.2 The database

To compute the frequencies 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 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 [39, 48]. Note that for both layouts, it is also possible to use the exact bit-strings from the binary matrix [40, 35]. Also a combination of both layouts can be used, as will be explained later in this chapter.

To count the frequency 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 time consuming and I/O intensive operation. In most cases, this is not the major cost of such counting steps. Instead, updating the frequencies 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 frequency counting, but it is by no means the dictating factor.

The vertical database layout has the major advantage that the frequency 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 Rule generation

The search space of all association rules contains exactly  $3^{|R|}$  different rules. However, given all frequent sets, this search space immediately shrinks tremendously. For every frequent set  $X$ , there exist at most  $2^{|X|}$  rules of the form  $X \setminus Y \Rightarrow Y$ , with  $Y \subseteq X \subseteq R$ . 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 sets.

**Proposition 2.10. (Confidence monotonicity)** *Let  $X, Y, Z \subseteq R$ , with  $X \cap Y = \emptyset$ . We have,*

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

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

$$\frac{\text{frequency}(X \cup Y \cup Z)}{\text{frequency}(X \setminus Z)} \leq \frac{\text{frequency}(X \cup Y)}{\text{frequency}(X)}.$$

□

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 the head of an association rule causes the rule to be unconfident, all of the head's supersets must result in unconfident rules.

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

## 2.4 The Apriori Algorithm

The first algorithm to generate all frequent sets and confident association rules was the AIS algorithm by Agrawal et al. [4], 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 frequency of sets and the confidence of association rules [6, 42]. The same technique was independently proposed by Mannila et al. [32]. Both works were combined afterwards [5].

### 2.4.1 Frequent set generation

The set mining phase of the Apriori algorithm is given in Algorithm 2.1. We use the notation  $X[i]$ , to represent the  $i$ th item in  $X$ . The  $k$ -*prefix* of a set  $X$  is the  $k$ -set  $\{X[1], \dots, X[k]\}$ .

---

**Algorithm 2.1** Apriori — Set mining
 

---

**Input:**  $r, \sigma$

**Output:**  $\mathcal{F}(r, \sigma)$

```

1:  $C_1 := \{\{i\} \mid i \in R\}$ 
2:  $k := 1$ 
3: while  $C_k \neq \{\}$  do
4:   // Compute the frequencies of all candidate sets
5:   for all transactions  $(tid, I) \in r$  do
6:     for all candidate sets  $X \in C_k$  do
7:       if  $X \subseteq I$  then
8:         Increment  $X.frequency$  by 1
9:   // Extract all frequent sets
10:   $\mathcal{F}_k := \{X \in C_k \mid X.frequency \geq \sigma\}$ 
11:  // Generate new candidate sets
12:   $C_{k+1} := \{\}$ 
13:  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
14:     $I := X \cup \{Y[k]\}$ 
15:    if  $\forall J \subset I, |J| = k : J \in \mathcal{F}_k$  then
16:      Add  $I$  to  $C_{k+1}$ 
17:  Increment  $k$  by 1

```

---

The algorithm performs a breadth-first search through the search space of all sets by iteratively generating candidate sets  $C_{k+1}$  of size  $k+1$ , starting with  $k=0$ . A set is a candidate if all of its subsets are known to be frequent. More specifically,  $C_1$  consists of all items in  $R$ , and at a certain level  $k$ , all sets 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 sets  $X, Y \in \mathcal{F}_k$  is generated if they have the same  $k-1$ -prefix. In the *prune* step,  $X \cup Y$  is only inserted into  $C_{k+1}$  if all of its  $k$ -subsets occur in  $\mathcal{F}_k$ .

To count the frequencies of all candidate  $k$ -sets, the database, which remains on secondary storage in the horizontal database layout, is scanned one transaction at a time, and the frequencies of all candidate sets that are included in that transaction are incremented. All sets that turn out to be frequent are inserted into  $\mathcal{F}_k$ .

If the number of candidate  $k+1$ -sets is too large to remain into main memory, the algorithm can be modified as follows. The candidate generation procedure stops and the frequencies of all generated candidates is computed as if nothing happened. But then, in the next iteration, instead of generating

candidate sets of size  $k + 2$ , the remaining candidate  $k + 1$ -sets are generated and counted repeatedly until all frequent sets of size  $k + 1$  are generated.

### 2.4.2 Association Rule Mining

Given all frequent sets, we can now generate all frequent and confident association rules. The algorithm is very similar to the frequent set mining algorithm and is given in Algorithm 2.2.

---

#### Algorithm 2.2 Apriori — Association Rule mining

---

**Input:**  $r, \sigma, \gamma$

**Output:**  $\mathcal{R}(r, \sigma, \gamma)$

```

1: Compute  $\mathcal{F}(r, \sigma)$ 
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:      $H_k := \{X \in C_k \mid \text{confidence}(I \setminus X \Rightarrow X, r) \geq \gamma\}$ 
10:    // Generate new candidate heads
11:     $C_{k+1} := \{\}$ 
12:    for all  $X, Y \in H_k, X[i] = Y[i]$  for  $1 \leq i \leq k - 1$ , and  $X[k] < Y[k]$ 
    do
13:       $I = X \cup \{Y[k]\}$ 
14:      if  $\forall J \subset I, |J| = k : J \in H_k$  then
15:        Add  $I$  to  $C_{k+1}$ 
16:      Increment  $k$  by 1
17:    // Cumulate all association rules
18:     $\mathcal{R} := \mathcal{R} \cup \{I \setminus X \Rightarrow X \mid X \in H_1 \cup \dots \cup H_k\}$ 

```

---

First, all frequent sets are generated using Algorithm 2.1. Then, every frequent set  $I$  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 set generation in Algorithm 2.1 (lines 11–16). To compute the confidence of a candidate head  $Y$ , the frequency 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 set  $I$  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 set  $I$ , only if all rules are already generated for all sets  $I' \subset I$ .

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 frequency of at most 2 sets, 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 frequency of an set 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 set generation phase.

### 2.4.3 Data Structures

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

#### Hash-tree

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

Candidate sets are stored in a hash-tree [5]. A node of the hash-tree either contains a list of sets (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  $d + 1$ . Sets are stored in leaves.

When we add a  $k$ -set  $X$  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 set, and following the pointer in the corresponding bucket. All nodes are initially created as leaf nodes. When the number of sets 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-sets that are contained in a transaction  $T$ , we start from the root node. If we are at a leaf, we find which of the sets

in the leaf are contained in  $T$  and increment their frequency. 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, 11, 14, 10]. In a trie, every  $k$ -set has a node associated with it, as does its  $k - 1$ -prefix. The empty set is the root node. All the 1-sets are attached to the root node, and their branches are labelled by the item they represent. Every other  $k$ -set is attached to its  $k - 1$ -prefix. Every node stores the last item in the set it represents, its frequency, 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  $k$ -sets are stored at depth  $k$  in the trie. In order to find the candidate-sets 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 sets 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 sets 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 frequency ascending order [11, 14, 10]. Using this heuristic, we reduce the number of sets 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  $k$ -set 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 sets mining algorithms presented in this thesis are implemented using this trie data structure.

### 2.4.4 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 frequency 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 sets that occur in each transaction.

#### AprioriTid, AprioriHybrid

Together with the proposal of the Apriori algorithm, Agrawal et al. [6, 5] proposed two other algorithms, AprioriTid and AprioriHybrid. The AprioriTid algorithm reduces the time needed for the frequency counting procedure by replacing every transaction in the database by the set of candidate sets 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 2.3.

---

#### Algorithm 2.3 AprioriTid

---

**Input:**  $r, \sigma$

**Output:**  $\mathcal{F}(\nabla, \sigma)$

```

1: Compute  $\mathcal{F}_1$  of all frequent items
2:  $\overline{C}_1 := r$  (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  $k$ -sets
6:    $\overline{C}_k := \{\}$ 
7:   // Compute the frequencies of all candidate sets
8:   for all transactions  $(tid, T) \in \overline{C}_k$  do
9:      $C_T := \{\}$ 
10:    for all  $X \in C_k$  do
11:      if  $\{X[1], \dots, X[k-1]\} \in T \wedge \{X[1], \dots, X[k-2], X[k]\} \in T$  then
12:        Add  $X$  to  $C_T$ 
13:        Increment  $X.frequency$  by 1
14:      if  $C_T \neq \{\}$  then
15:        Add  $(tid, C_T)$  to  $\overline{C}_k$ 
16:   Extract  $\mathcal{F}_k$  of all frequent  $k$ -sets
17:   Increment  $k$  by 1

```

---

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  $k$ -sets, 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, 5] 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 sets, 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-sets

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 sets [36]. During the  $k$ th iteration, when the frequencies of all candidate  $k$ -sets are counted by scanning the database, DHP already gathers information about candidate sets 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 sets have been hashed to that bucket so far. Then, if a candidate set of size  $k + 1$  is generated, the hash function is applied on that set. If the counter of the corresponding bucket in the hash table is below the minimal frequency threshold, the generated set is not added to the set of candidate sets. Also, during the frequency counting phase of iteration  $k$ , every transaction is trimmed in the following way. If a transaction contains a frequent set of size  $k + 1$ , any item contained in that  $k + 1$  set will appear in at least  $k$  of the candidate  $k$ -sets 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  $k$ -sets in  $C_k$ . These techniques result in a significant decrease in the number of candidate sets 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 [41]. Instead of using the hash-tree

to store and count all candidate 2-sets, a triangular array  $C$  is created, in which the frequency counter of a candidate 2-set  $\{i, j\}$  is stored at location  $C[i][j]$ . Using this array, the frequency 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 [34, 35].

Since the number of candidate 2-sets 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-sets do not even occur at all in the database, and hence, their frequency 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-set. 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 frequency of all candidate 2-sets contained in that transaction. However, if the candidate 2-set does not yet exist, we generate the candidate set and initialize its frequency to 1. In this way, only those candidate 2-sets 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-sets. Using this technique, this number was drastically reduced to 1 708 203.

### Frequency lower bounding

As we already mentioned earlier in this chapter, apart from the monotonicity property, it is sometimes possible to derive information on the frequency of an set, given the frequency of all of its subsets. The first algorithm that uses such a technique was proposed by Bayardo in his MaxMiner and Apriori-LB algorithms [10]. The presented technique is based on the following property which gives a lower bound on the frequency of an set.

**Proposition 2.11.** *Let  $X, Y, Z \subseteq R$  be sets.*

$$\text{frequency}(X \cup Y \cup Z) \geq \text{frequency}(X \cup Y) + \text{frequency}(X \cup Z) - \text{frequency}(X)$$

*Proof.*

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

□

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

Additionally, we can exploit a special case of Proposition 2.11 even more.

**Corollary 2.12.** *Let  $X, Y, Z \subseteq R$  be sets.*

$$\text{frequency}(X \cup Y) = \text{frequency}(X) \Rightarrow \text{frequency}(X \cup Y \cup Z) = \text{frequency}(X \cup Z)$$

This specific property was later exploited by Pasquier et al. in order to find a concise representation of all frequent sets [37, 12]. Nevertheless, it can already be used to improve the Apriori algorithm.

Suppose we have generated and counted the frequency of the frequent set  $X \cup \{i\}$  and that its frequency is equal to the frequency of  $X$ . Then we already know that the frequencies of every superset  $X \cup \{i\} \cup Y$  is equal to the frequency 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 frequency of candidate sets [16]. These deduction rules allow for constructing a minimal representation of all frequent sets, but can also be used to efficiently generate the set of all frequent sets. Unfortunately, for a given candidate set, an exponential number of rules in the length of the set 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 sets that can yet be generated after a certain iteration in the Apriori algorithm.

### Dynamic Set 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 [14]. First, all candidate patterns of size 1 are generated. The frequencies of the candidate sets are then counted over the first interval of the database. Based on these frequencies, a new candidate pattern of size 2 is already generated if all of its subsets are already known to be frequent, and its frequency 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 re-ordering all items in frequency ascending order is negligible, this is not true for Apriori in general. Indeed, the reordering used in DIC was based on the frequencies of the 1-sets 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 [26] uses a similar technique, reducing the interval size to 1. More specifically, candidate sets are generated on the fly from every transaction. After reading a transaction, it increments the frequencies of all candidate sets contained in that transaction and it generates a new candidate set 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 sets than DIC or Apriori. (Note that the number of candidate sets generated by DIC is exactly the same as in Apriori.) Additionally, CARMA allows the user to change the minimal frequency 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 sets relative to some threshold which depends on how the user changed the minimal frequency threshold during its execution. However, when the minimal frequency threshold was kept fixed during the complete execution of the algorithm, at least all frequent sets have been generated. To determine the exact frequencies of all generated sets, a second scan of the database is required.

### Sampling

The sampling algorithm, proposed by Toivonen [45], 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 frequencies during a second pass through the database. The probability of such a failure can be kept small by decreasing the minimal frequency 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 [39]. That is, the database is stored in main memory using the vertical database layout and the frequency of an set 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 frequency of a candidate  $k$ -set  $I$ , 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  $I$ .

---

#### Algorithm 2.4 Partition — Local Set Mining

---

**Input:**  $r, \sigma$

**Output:**  $\mathcal{F}(r, \sigma)$

- 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:      $I = \{X[1], \dots, X[k - 1], Y[k - 1]\}$
  - 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:   Increment  $k$  by 1
- 

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 sets that are relatively frequent within that part, using the algorithm described in the previous paragraph and shown in Algorithm 2.4. 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 sets of every part together. This results in a superset of all frequent sets over the complete database, since a set that is frequent in the complete database must be relatively frequent in one of the parts. Then, the actual frequencies of all sets are computed during a second scan through the database. Again, every part is read into main memory using the vertical database layout and the frequency of every set is computed by intersecting the covers of all items occurring in that set. The exact Partition algorithm is given in Algorithm 2.5.

---

**Algorithm 2.5** Partition
 

---

**Input:**  $r, \sigma$

**Output:**  $\mathcal{F}(r, \sigma)$

```

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

```

---

The exact computation of the frequencies of all sets can still be optimized, but we refer to the original article for further implementation details [39].

Although the covers of all items can be stored in main memory, during the generation of all local frequent sets for every part, it is still possible that the covers of all local candidate  $k$ -sets 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 sets, 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.

## 2.5 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 set generation procedure could exceed main memory limits. Nevertheless, it is possible to significantly reduce this total size by generating collections of candidate sets in a depth-first strategy. The first algorithm proposed to generate all frequent sets in a depth-first manner is the Eclat algorithm by Zaki [48, 51]. Later, several other depth-first algorithms have been proposed [1, 2, 25] of which the FP-growth algorithm by Han et al. [25, 24] is the most well known. In this section, we explain both the Eclat and FP-growth algorithms.

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

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.

### 2.5.1 Eclat

Eclat uses the vertical database layout and uses the intersection based approach to compute the frequency of an set. The Eclat algorithm is given in Algorithm 2.6.

---

#### Algorithm 2.6 Eclat

---

**Input:**  $r, \sigma, I \subseteq R$

**Output:**  $\mathcal{F}[I](r, \sigma)$

```

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

```

---

Note that a candidate set is now represented by each set  $I \cup \{i, j\}$  of which the frequency is computed at line 6 of the algorithm. Since the algorithm doesn't fully exploit the monotonicity property, but generates a candidate set based on only two of its subsets, the number of candidate sets 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 sets using only the join step from Apriori, since the sets necessary for the prune step are not available. Again, we can reorder all items in the database in frequency ascending order to reduce the number of candidate sets 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 sets. 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 frequencies of all sets 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  $k$ -sets 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  $k$ -sets 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 [49, 50] proposed a new approach to efficiently compute the frequency of an set using the vertical database layout. Instead of storing the cover of a  $k$ -set  $I$ , 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 frequency of  $I$ , we simply need to subtract the size of the diffset from the frequency of its  $k - 1$ -prefix. Note that this frequency does not need to be stored within each set but can be maintained as a parameter within the recursive function calls of the algorithm. The diffset of an set  $I \cup \{i, j\}$ , given the two diffsets of its subsets  $I \cup \{i\}$  and  $I \cup \{j\}$ , with  $i < j$ , is computed as follows:

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

This technique has experimentally shown to result in significant performance improvements of the algorithm, now designated as *dEclat* [49]. The original database is still stored in the original vertical database layout. Observe an arbitrary recursion path of the algorithm starting from the set  $\{i_1\}$ , up to the  $k$ -set  $I = \{i_1, \dots, i_k\}$ . The set  $\{i_1\}$  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  $|\text{cover}(\{i_1\})|$ . On the other hand, if we generate the cover of each generated set, the total size of all generated covers on that path is at least  $(k - 1) \cdot \sigma$  and can be at most  $(k - 1) \cdot |\text{cover}(\{i_1\})|$ . 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 frequencyed by several experiments [49].

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 frequency of the frequent set  $X \cup \{i\}$  and that its frequency is equal to the frequency of  $X$  (hence, its diffset is empty). Then we already know that the frequency of every superset  $X \cup \{i\} \cup Y$  is equal to the frequency 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 sets are found using Eclat.

Another optimization proposed by Hipp et al. combines Apriori and Eclat into a single Hybrid [27]. More specifically, the algorithm starts generating frequent sets 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-sets if these are generated online as described in Section 2.4.4. Indeed, when a lot of items in the database are frequent, Eclat generates every possible 2-set 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 sets is too high to store into main memory, switching to Eclat might be impossible, while Apriori nicely marches on.

## 2.5.2 FP-growth

In order to count the frequencies of all generated sets, 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 [25]. Again, we order the items in the database in frequency 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, frequency 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

$tid$	$X$
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 2.1: An example preprocessed transaction database.

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 frequency. The reason to store transactions in the FP-tree in frequency 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 2.13.** *Assume we are given a transaction database and a minimal frequency threshold of 2. First, the frequencies of all items is computed, all infrequent items are removed from the database and all transactions are re-ordered according to the frequency descending order resulting in the example transaction database in Table 2.1. The FP-tree for this database is shown in Figure 2.3.*

Given such an FP-tree, the frequencies 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 sets. 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 sets. More specifically, in order to generate for every  $i \in R$  all frequent sets in  $\mathcal{F}[\{i\}](r, \sigma)$ , FP-growth creates the so called  $i$ -projected database of  $r$ . Essentially, the  $r^i$  used in Eclat is the vertical database layout of the  $i$ -projected database considered here. The FP-growth algorithm is given in Algorithm 2.7.

The only difference between Eclat and FP-growth is the way they count the frequencies of every candidate set 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  $r^i$  at lines 6–10,

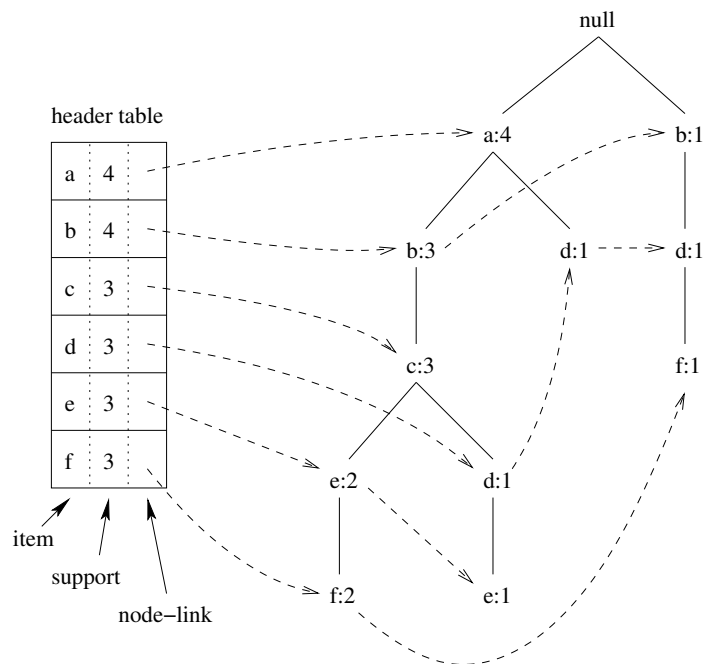


Figure 2.3: An example of an FP-tree.

**Algorithm 2.7** FP-growth**Input:**  $r, \sigma, I \subseteq R$ **Output:**  $\mathcal{F}[I](r, \sigma)$ 


---

```

1:  $\mathcal{F}[I] := \{\}$ 
2: for all  $i \in R$  occurring in  $r$  do
3:    $\mathcal{F}[I] := \mathcal{F}[I] \cup \{I \cup \{i\}\}$ 
4:   // Create  $r^i$ 
5:    $r^i := \{\}$ 
6:    $H := \{\}$ 
7:   for all  $j \in R$  occurring in  $r$  such that  $j > i$  do
8:     if  $\text{frequency}(I \cup \{i, j\}) \geq \sigma$  then
9:        $H := H \cup \{j\}$ 
10:  for all  $(tid, X) \in r$  with  $i \in X$  do
11:     $r^i := r^i \cup \{(tid, X \cap H)\}$ 
12:  // Depth-first recursion
13:  Compute  $\mathcal{F}[I \cup \{i\}](r^i, \sigma)$ 
14:   $\mathcal{F}[I] := \mathcal{F}[I] \cup \mathcal{F}[I \cup \{i\}]$ 

```

---

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 frequency 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  $r$  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 sets 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 frequency set to the minimum of the frequencies of the items in that combination. Essentially, this technique is similar to the technique used by all other algorithms when the frequency of an set is equal to the frequency 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  $r^i$  actually represents the set  $I \cup \{i, j\}$ . In other words, for every frequent item  $i$  occurring in  $r$ , the algorithm recursively finds all frequent 1-sets in the  $i$ -projected database  $r^i$ .

Although the authors of the FP-growth algorithm claim that their algorithm [24, 25] does not generate any candidate sets, we have shown that the algorithm actually generates a lot more candidate sets 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

Data set	$  r  $	$ FP-tree $	$\frac{  r  }{ FP-tree }$
T40I10D100K	3 912 459 : 15 283K	3 514 917 : 68 650K	89% : 449%
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 2.2: Memory usage of Eclat versus FP-growth.

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 2.2 shows for all four used data sets the size of the total length of all arrays in Eclat ( $||r||$ ), 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.

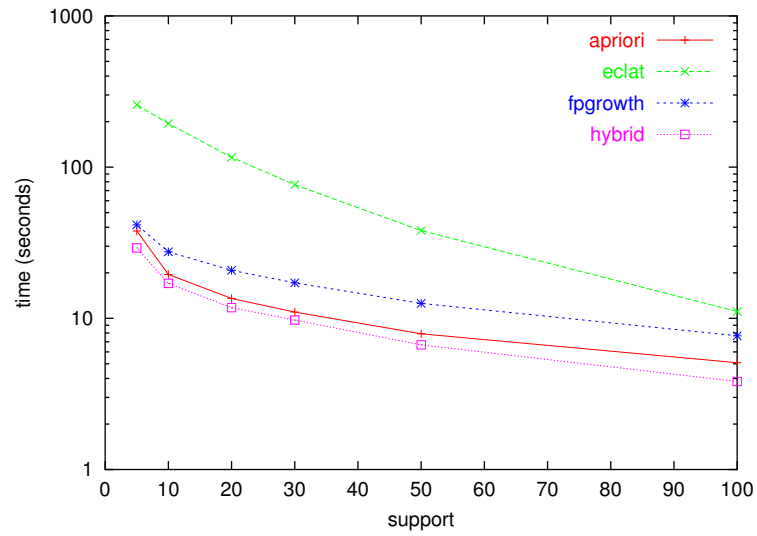
## 2.6 Experimental Evaluation

We implemented the Apriori implementation using the online candidate 2-set 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 [43]. 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.

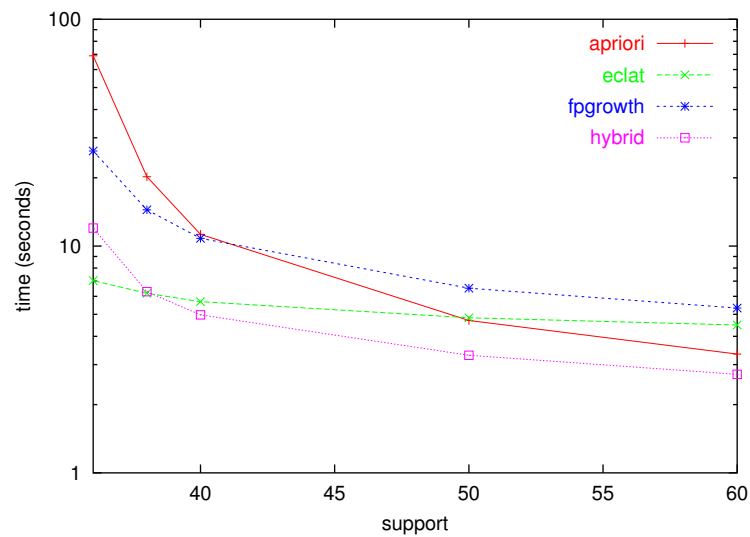
Figure 2.6 shows the performance of the algorithms on several datasets for varying minimal frequency thresholds.

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-sets is generated. The other algorithms all use dynamic candidate generation of 2-sets 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-sets 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

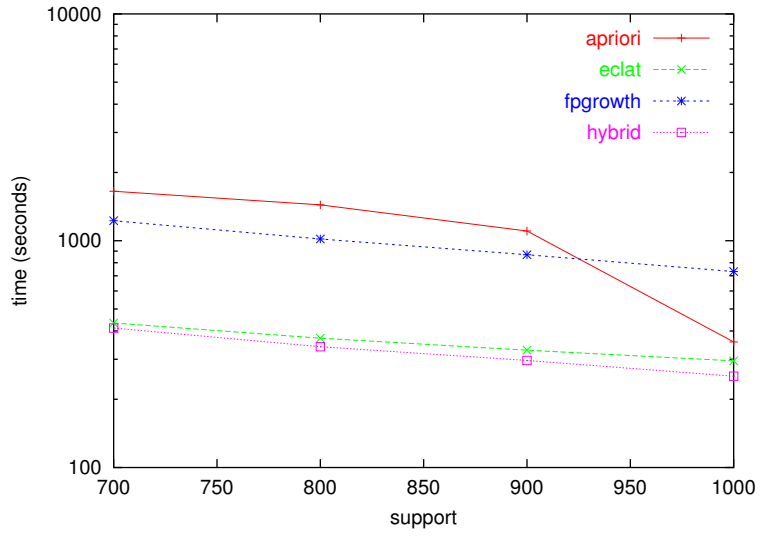


(a) basket

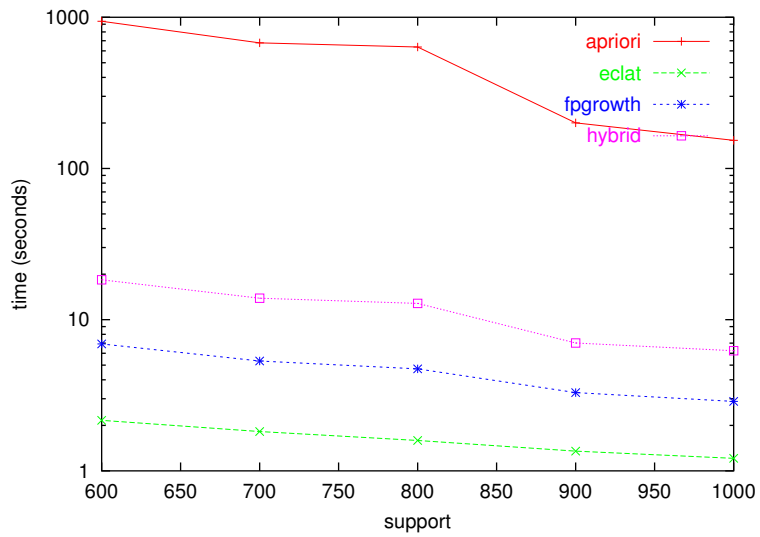


(b) BMS-Webview-1

Figure 2.4: Frequent set mining performance.



(c) T40I10D100K



(d) mushroom

Figure 2.4: Frequent set mining performance.

the maintenance of the FP-tree structure, while updating the frequencies of all candidate sets 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 frequency 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 frequency 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 frequencies of all candidate sets consumes most of the time. To frequency 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 frequency counting of all candidate sets of sized 5 to 10. For example, counting the frequencies of all 7-sets 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 frequencies. 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.

# Bibliography

- [1] R. Agarwal, C. Aggarwal, and V. Prasad. Depth first generation of long patterns. In Ramakrishnan et al. [38], pages 108–118.
- [2] R. Agarwal, C. Aggarwal, and 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. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD’93)*, pages 207 – 216, Washington, D.C., USA, May 1993. ACM.
- [4] R. Agrawal, T. Imielinski, and A. 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.
- [5] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. MIT Press, 1996.
- [6] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. 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] T. Anand. Opportunity explorer: Navigating large databases using knowledge discovery templates. (4):27–37, 1995.
- [10] R. Bayardo, Jr. Efficiently mining long patterns from databases. In L. 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.
- [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] R. J. Brachman, T. Khabaza, W. Kloesgen, G. Piatetsky-Shapiro, and E. Simoudis. Mining business databases. *Communications of the ACM*, pages 42–48, Nov. 1996.
- [14] S. Brin, R. Motwani, J. 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.
- [15] M. C. Burl, U. M. Fayyad, P. Perona, P. Smyth, and M. P. Burl. Automating the hunt for volcanoes on venus. In *Proceedings of the Conference on Computer Vision and Pattern Recognition*, pages 302–309, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.
- [16] 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.
- [17] W. Chen, J. Naughton, and P. Bernstein, editors. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, volume 29(2) of *SIGMOD Record*. ACM Press, 2000.
- [18] U. Dayal, P. Gray, and S. Nishio, editors. *Proceedings 21th International Conference on Very Large Data Bases*. Morgan Kaufmann, 1995.

- [19] J. Elder IV and D. Pregibon. A statistical perspective on knowledge discovery in databases. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 83 – 113. AAAI Press, Menlo Park, CA, 1996.
- [20] U. Fayyad, D. Haussler, and P. Stolorz. Mining scientific data. *Communications of the ACM*, pages 51–57, Nov. 1996.
- [21] U. M. Fayyad, S. G. Djorgovski, and N. Weir. Automating the analysis and cataloging of sky surveys. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 471 – 494. AAAI Press, Menlo Park, CA, 1996.
- [22] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 1 – 34. AAAI Press, Menlo Park, CA, 1996.
- [23] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AAAI Press, Menlo Park, CA, 1996.
- [24] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In Chen et al. [17], pages 1–12.
- [25] 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.
- [26] 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.
- [27] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Mining association rules: Deriving a superior algorithm by analyzing today’s approaches. In D. Zighed, H. Komorowski, and J. 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.
- [28] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of the ACM*, 39(11):58 – 64, Nov. 1996.
- [29] W. Kloesgen. Efficient discovery of interesting statements in databases. *Journal of Intelligent Information Systems*, 4(1):53 – 69, 1995.

- [30] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden Markov models in computational biology: Applications to protein modeling. *Journal of Molecular Biology*, 235:1501–1531, Feb. 1994.
- [31] A. Krogh, I. S. Mian, and D. Haussler. A hidden Markov model that finds genes in *e. coli* DNA. *Nucleic Acids Research*, 22:4768–4778, 1994.
- [32] H. Mannila, H. Toivonen, and A. Verkamo. Efficient algorithms for discovering association rules. In U. Fayyad and R. Uthurusamy, editors, *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, pages 181–192. AAAI Press, 1994.
- [33] C. J. Matheus, G. Piatetsky-Shapiro, and D. McNeill. Selecting and reporting what is interesting. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 495 – 515. AAAI Press, Menlo Park, CA, 1996.
- [34] 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.
- [35] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets. In V. Kumar, S. Tsumoto, P. Yu, and N. Zhong, editors, *Proceedings of the 2002 IEEE International Conference on Data Mining*. IEEE Computer Society, 2002. To appear.
- [36] J. Park, M.-S. Chen, and P. 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.
- [37] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In C. Beerli 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.
- [38] R. Ramakrishnan, S. Stolfo, R. Bayardo, Jr., and I. Parsa, editors. *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2000.
- [39] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In Dayal et al. [18], pages 432–444.

- [40] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In Chen et al. [17], pages 22–33.
- [41] R. Srikant. *Fast algorithms for mining association rules and sequential patterns*. PhD thesis, University of Wisconsin, Madison, 1996.
- [42] R. Srikant and R. Agrawal. Mining generalized association rules. In Dayal et al. [18], pages 407–419.
- [43] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.
- [44] P. Tan, V. Kumar, and J. Srivastava. Selecting the right interestingness measure for association patterns. In D. Hand, D. Keim, and R. Ng, editors, *Proceedings of the Eight ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 32–41. ACM Press, 2002.
- [45] H. Toivonen. Sampling large databases for association rules. In T. Vijayarman, A. Buchmann, C. Mohan, and N. Sarda, editors, *Proceedings 22nd International Conference on Very Large Data Bases*, pages 134–145. Morgan Kaufmann, 1996.
- [46] J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley Publishing Company, Reading, MA, 1977.
- [47] G. Webb. Efficient search for association rules. In Ramakrishnan et al. [38], pages 99–107.
- [48] M. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, May/June 2000.
- [49] M. Zaki. Fast vertical mining using diffsets. Technical Report 01-1, Rensselaer Polytechnic Institute, Troy, New York, 2001.
- [50] M. 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.
- [51] M. 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.