Chapter 17

FREQUENT SET MINING

Bart Goethals

Departement of Mathemati1cs and Computer Science, University of Antwerp, Belgium

bart.goethals@ua.ac.be

Abstract Frequent sets lie at the basis of many Data Mining algorithms. As a result, hundreds of algorithms have been proposed in order to solve the frequent set mining problem. In this chapter, we attempt to survey the most successful algorithms and techniques that try to solve this problem efficiently.

Keywords: Frequent Set Mining, Association Rule, Support, Cover, Apriori

Introduction

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, as explained in Chapter 16 in this volume, is one of the most popular problems. The identification of sets of items, products, symptoms, characteristics, and so forth, that often occur together in the given database, can be seen as one of the most basic tasks in Data Mining.

Since its introduction in 1993 by Agrawal et al. (1993), the frequent set mining problem has received a great deal of attention. Hundreds of research papers have been published, presenting new algorithms or improvements to solve this mining problem more efficiently.

In this chapter, we explain the frequent set mining problem, some of its variations, and the main techniques to solve them. Obviously, given the huge amount of work on this topic, it is impossible to explain or even mention all proposed algorithms or optimizations. Instead, we attempt to give a comprehensive survey of the most influential algorithms and results.

1. Problem Description

The original motivation for searching frequent sets came from the need to analyze so called supermarket transaction data, that is, to examine customer behavior in terms of the purchased products (Agrawal et al., 1993). Frequent sets of products describe how often items are purchased together.

Formally, let $\mathcal{I}$ be a set of items.

A *transaction* over $\mathcal{I}$ is a couple $T = (tid, I)$ where tid is the transaction identifier and I is a set of items from $\mathcal{I}$.

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

A transaction $T = (tid, I)$ is said to *support* a set X, if $X \subseteq I$. The *cover* of a set X in $\mathcal{D}$ consists of the set of transaction identifiers of transactions in $\mathcal{D}$ that support X. The *support* of a set X in $\mathcal{D}$ is the number of transactions in the cover of X in $\mathcal{D}$. The *frequency* of a set X in $\mathcal{D}$ is the probability that X occurs in a transaction, or in other words, the support of X divided by the total number of transactions in the database. We omit $\mathcal{D}$ whenever it is clear from the context.

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

DEFINITION 17.1 *Let $\mathcal{D}$ be a database of transactions over a set of items $\mathcal{I}$, and σ a minimal support threshold. The collection of frequent sets in $\mathcal{D}$ with respect to σ is denoted by*

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

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

PROBLEM 1 **(Frequent Set Mining)** *Given a set of items $\mathcal{I}$, a database of transactions $\mathcal{D}$ over $\mathcal{I}$, and minimal support threshold σ, find $\mathcal{F}(\mathcal{D}, \sigma)$.*

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

For example, consider the database shown in Table 17.1 over the set of items $\mathcal{I} = \{\text{beer}, \text{chips}, \text{pizza}, \text{wine}\}$.

Table 17.2 shows all frequent sets in $\mathcal{D}$ with respect to a minimal support threshold equal to 1, their cover in $\mathcal{D}$, plus their support and frequency.

Note that the Set Mining problem is actually a special case of the Association Rule Mining problem explained in Chapter 16 in this volume. Indeed, if

Table 17.1. An example database $\mathcal{D}$.

tid	set of items
100	$\{\text{beer, chips, wine}\}$
200	$\{\text{beer, chips}\}$
300	$\{\text{pizza, wine}\}$
400	$\{\text{chips, pizza}\}$

Table 17.2. Frequent sets, their cover, support, and frequency in $\mathcal{D}$.

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

we are given the support threshold σ, then every frequent set X also represents the trivial rule $X \Rightarrow \{\}$ which holds with 100% confidence.

Nevertheless, the task of discovering all frequent sets is quite challenging. The search space is exponential in the number of items occurring in the database and the targeted databases tend to be massive, containing millions of transactions. Both these characteristics make it a worthwhile effort to seek the most efficient techniques to solve this task.

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

Instead, we could limit ourselves to those sets that occur at least once in the database by generating only all subsets of all transactions in the database. Of course, for large transactions, this number could still be too large. As an optimization, we could generate only those subsets of at most a given maximum size. This technique, however, suffers from massive meory requirements for any but a database with only very small transactions (Amir et al., 1997). Most other efficient solutions perform a more directed search through the search space. During such a search, several collections of *candidate sets* are generated and their supports 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 supports 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 section.

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

PROPERTY 1 **(Support monotonicity)** *Given a database of transactions $\mathcal{D}$ over $\mathcal{I}$, and two sets $X, Y \subseteq \mathcal{I}$. Then,*

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

Hence, if a set is infrequent, all of its supersets must be infrequent, and vice versa, if a set is frequent, all of its subsets must be frequent too. In the literature, this monotonicity property is also called the downward closure property, since the set of frequent sets is downward closed with respect to set inclusion. Similarly, the set of infrequent sets is upward closed.

Database Issues. To compute the supports of a collection of sets, 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 two-dimensional binary 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 so called *horizontal 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 layout*, in which the database consists of a set of items, each followed by its cover (Holsheimer et al., 1995; Savasere et al., 1995; Zaki, 2000).

To count the support of a candidate set X using the horizontal 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 sets at once. Although scanning the database is an I/O intensive operation, in most cases, this is not the major cost of such counting steps. Instead, updating the supports of all candidate sets 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 set whether it is included in that transaction, or otherwise, we need to check for every subset of that transaction whether it is in the set of candidate sets.

The vertical database layout on the other hand, has the major advantage that the support of a set X can be easily computed by simply intersecting the covers of any two subsets $Y, Z \subseteq X$, such that $Y \cup Z = X$ (Holsheimer et al., 1995; Savasere et al., 1995). Given a set of candidate sets, however, this technique requires that the covers of a lot of sets are available in main memory, which is evidently not always possible. Indeed, the covers of all singleton sets already represent the complete database.

In the next two sections, we will describe the standard algorithm for mining all frequent sets using the horizontal layout and the vertical database layout. After that, we consider several optimizations and variations of both approaches.

2. Apriori

Together with the introduction of the frequent set mining problem, also the first algorithm to solve it was proposed, later denoted as *AIS* (Agrawal et al., 1993). Shortly after that, the algorithm was improved and called *Apriori*. The main improvement was to exploit the monotonicity property of the support of sets (Agrawal and Srikant, 1994; Srikant and Agrawal, 1995). The same technique was independently proposed by Mannila et al. (1994). Both works were combined afterwards (Agrawal et al., 1996). Note that the Apriori algorithm actually solves the complete association rule mining problem, of which mining all frequent sets was only the first, but most difficult phase.

From now on, we assume for simplicity that items in transactions and sets are kept sorted in their lexicographic order, unless stated otherwise.

The set mining phase of the Apriori algorithm is given in Algorithm 17.1. We use the notation $X[i]$ to represent the ith item in X; the k*-prefix* of a set X is the k-set $\{X[1], \ldots, X[k]\}$, and $\mathcal{F}_k$ denotes the frequent k-sets.

The algorithm performs a breadth-first (levelwise) search through the search space of all sets by iteratively generating and counting a collection of candidate sets. More specifically, a set is candidate if all of its subsets are counted and frequent. In each iteration, the collection C_{k+1} of candidate sets of size $k+1$ is generated, starting with $k = 0$. Obviously, the initial set C_1 consists of all items in $\mathcal{I}$ (line 1). At a certain level k, all candidate sets of size $k+1$ are generated. This is done in two steps. First, in the *join* step, the union $X \cup Y$ of sets $X, Y \in \mathcal{F}_k$ is generated if they have the same $k-1$-prefix (lines 10–11). In the *prune* step, $X \cup Y$ is inserted into C_{k+1} only if all of its k-subsets are frequent and thus, must occur in $\mathcal{F}_k$ (lines 12–13).

To count the supports of all candidate k-sets, the database, which remains on secondary storage in the horizontal layout, is scanned one transaction at a

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

Figure 17.1. Apriori

time, and the supports of all candidate sets that are included in that transaction are incremented (lines 4–7). All sets that turn out to be frequent are inserted into $\mathcal{F}_k$ (line 8).

If the number of candidate sets is too large to remain into main memory, the algorithm can be easily modified as follows. The candidate generation procedure stops and the supports of all generated candidates is counted. 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 and counted.

Although this is a very efficient and robust algorithm, its main drawback lies in its inefficient support counting mechanism. As already explained, for each transaction, we need to check for every candidate set whether it is included in that transaction, or otherwise, we need to check for every subset of that transaction whether it is in the set of candidate sets.

When transactions are large, generating all k-subsets of a transaction and testing for each of them whether it is a candidate set, can take a prohibitive amount of time. For example, suppose we are counting candidate sets of size 5 in single transaction containing only 20 items. Then, we have to do already more than $15\,000$ set equality tests. Of course, this can be somewhat optimized since many of these sets have large intersections and hence, can be tested at the same time (Brin et al., 1997). Nevertheless, transactions can be much larger causing this method to become a significant bottleneck.

On the other hand, testing for each candidate set whether it is contained in the given transaction can also take to much time when the collection of candidate sets is large. For example, consider the case in which we have $1\,000$ frequent items. This means there are almost $500\,000$ candidate 2-sets. Obviously, testing whether all of them occur in a single transaction, for every transaction, could take an immense amount of time. Fortunately, a lot of counting optimizations have been proposed for many different situations (Park et al., 1995; Srikant, 1996; Brin et al., 1997; Orlando et al., 2002). To reduce the number of iterations that are needed to go through the the database, it is also possible to combine the last few iterations of the algorithm. That is, generate every candidate set of size $k+\ell$ if all of its k-subsets are known to be frequent, for all possible $\ell > 1$. Of course, it is of crucial importance not to do this too early, since that could cause an exponential blowup in the number of generated candidate sets. It is possible, however, to bound the remaining number of candidate sets very accurately using a combinatorial technique proposed by Geerts et al. (2001). Given this bound, a combinatorial explosion can be avoided.

Another important aspect of the Apriori algorithm is the data structure used to store the candidate and frequent sets for the candidate generation and the support counting processes. Indeed, they both 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. The two most successful data structures are the hash-tree and the trie. We refer the interested reader to other literature describing these data structures in more detail, e.g. (Srikant, 1996; Brin et al., 1997; Borgelt and Kruse, 2002).

3. Eclat

As explained earlier, when the database is stored in the vertical layout, the support of a set can be counted much easier by simply intersecting the covers of two of its subsets that together give the set itself. The original Eclat algorithm essentially used this technique inside the Apriori algorithm (Zaki, 2000). This is, however, 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. Fortunately, it is possible to significantly reduce this total size by generating collections of candidate sets in a depth-first strategy. Also, in stead of using the intersection based technique already from the start, it is usually more efficient to first find the frequent items and frequent 2-sets separately and use the Eclat algorithm only for all larger sets (Zaki, 2000).

Given a database of transactions $\mathcal{D}$ and a minimal support threshold σ, denote the set of all frequent sets with the same prefix $I \subseteq \mathcal{I}$ by $\mathcal{F}[I](\mathcal{D}, \sigma)$. (Note that $\mathcal{F}[\{\}](\mathcal{D}, \sigma) = \mathcal{F}(\mathcal{D}, \sigma)$.) The main idea of the search strategy is that all sets containing item $i \in \mathcal{I}$, but not containing any item smaller than i, can be found in the so called *i-conditional database* (Han et al., 2004), denoted by $\mathcal{D}^i$. That is, $\mathcal{D}^i$ consists of those transactions from $\mathcal{D}$ that contain i, and from which all items before i, and i itself are removed. In general, for a given set I, we can create the I-conditional database, $\mathcal{D}^I$, consisting of all transactions that contain I, but from which all items before the last item in I and that item itself have been removed. Then, for every frequent set found in $\mathcal{D}^I$, we add I to it, and thus, we found exactly all large tiles containing I, but not any item before the last item in I which is not in I, in the original database, $\mathcal{D}$. Finally, Eclat recursively generates for every item $i \in \mathcal{I}$ the set $\mathcal{F}[\{i\}](\mathcal{D}^i, \sigma)$.

For simplicity of presentation, we assume that all items that occur in the 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.

The final Eclat algorithm is given in Algorithm 17.2.

Note that a candidate set is now represented by each set $I \cup \{i, j\}$ of which the support is computed at line 6 and 7 of the algorithm. Since the algorithm doesn't fully exploit the monotonicity property, but generates a candidate set based on the frequency of only two of its subsets, the number of candidate sets that are generated is much larger as compared to Apriori's breadth-first approach. As a comparison, Eclat essentially generates candidate sets using

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

Figure 17.2. Eclat

only the join step from Apriori. The sets that are needed for the prune step are simply not available.

Recently, Zaki et al. proposed a significant improvement to this algorithm to reduce the amount of necessary memory and to compute the support of a set even faster using the vertical database layout (Zaki and Gouda, 2003). 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, called the *diffset* of I. To compute the support of I, we simply need to subtract the size of the diffset from the support of its $k-1$-prefix. This support can be provided as a parameter within the recursive function calls of the algorithm. The diffset of a 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:

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

This technique has experimentally shown to result in significant performance improvements of the algorithm, now designated as *dEclat* (Zaki and Gouda, 2003). 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 $I = \{i_1, \ldots, i_k\}$. 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 $|cover(\{i_1\})|$. 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 $(k-1)\cdot|cover(\{i_1\})|$. This observation indicates that the total size of all diffsets that are stored in main memory at a certain point in the algorithm is much less than the total size of all covers. These predictions were supported by several experiments (Zaki and Gouda, 2003).

4. Optimizations

A lot of other algorithms proposed after the introduction of Apriori retain the same general structure, adding several techniques to optimize certain steps within the algorithm. Since the performance of the Apriori algorithm is almost completely dictated by its support counting procedure, most research has focused on that aspect of the Apriori algorithm.

The Eclat algorithm was not the first of its kind when considering the intersection based counting mechanism (Holsheimer et al., 1995; Savasere et al., 1995). Also, its original design did not pursue a depth-first traversal of the search space, although this is only a simple but effective change, which was later corrected in extensions of the algorithm (Zaki and Gouda, 2003; Zaki and Hsiao, 2002). The effectiveness of this change mainly shows in the amount of memory that is consumed. Indeed, the amount and total size of all covers or diffsets stored within a depth-first recursion is usually much smaller than compared to this amount during a breadth-first recursion (Goethals, 2004).

4.1 Item reordering

One of the most important optimizations which can be effectively exploited by almost any frequent set mining algorithm, is the reordering of items.

The underlying intuition is to assume statistical independence of all items. Then, items with high frequency tend to occur in more frequent sets, while low frequent items are more likely to occur in only very few sets.

For example, in the case of Apriori, sorting the items in support ascending order improves the distribution of the candidate sets within the used data structure (Borgelt and Kruse, 2002). Also, the number of candidate sets generated during the join step can be reduced in this way. Also in Eclat the number of candidate sets that is generated is reduced using this order, and hence, the number of intersections that need to be computed and the total size of the covers of all generated sets is reduced accordingly. In fact, in Eclat, such reordering can be performed at every recursion step of the algorithm.

Unfortunately, until now, no results have been presented on an optimal ordering of all items for any given algorithm and only vague intuitions and heuristics are given supported by practical experiments.

4.2 Partition

As the main drawback of Apriori is its slow and iterative support counting mechanism, Eclat has the drawback that it requires large parts of the (vertical) database to fit in main memory. To solve these issues, Savasere et al. proposed the Partition algorithm (Savasere et al., 1995). (Note, however, that this algorithm was already presented before Eclat and its relatives.)

The main difference in the Partition algorithm, compared to Apriori and Eclat, is that the database is partitioned into several disjoint parts and the algorithm generates for every part all sets that are relatively frequent within that part. This is can be done very efficiently by using the Eclat algorithm (originally, a slightly different algorithm was presented). The parts of the database are chosen in such a way that each part fits into main memory. Then, 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. Finally, the actual supports of all sets are computed during a second scan through the database.

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 the algorithm essentially comes down to Eclat.

4.3 Sampling

Another technique to solve Apriori's slow counting and Eclat's large memory requirements is to use sampling as proposed by Toivonen (Toivonen, 1996).

The presented Sampling algorithm picks a random sample from the database, then finds all relatively frequent patterns in that sample, and then verifies the results with the rest of the database. In the cases where the sampling method does not produce all frequent sets, the missing sets can be found by generating all remaining potentially frequent sets and verifying their supports during a second pass through the database. The probability of such a failure can be kept small by decreasing the minimal support threshold. However, for a reasonably small probability of failure, the threshold must be drastically decreased, which can cause a combinatorial explosion of the number of candidate patterns. Nevertheless, in practice, finding all frequent patterns within a small sample of the database can be done very fast using Eclat or any other efficient frequent set mining algorithm. In the next step, all true supports of these patterns must be counted after which the standard levelwise algorithm could finish finding all other frequent patterns by generating and counting all candidate patterns iteratively. It has been shown that this technique usually needs only one more scan resulting in a significant performance improvement (Toivonen, 1996).

4.4 FP-tree

One of the most cited algorithms proposed after Apriori and Eclat is the FP-growth algorithm by Han et al. (2004). Like Eclat, it performs a depth-first search through all candidate sets and also recursively generates the so called i-conditional database $\mathcal{D}^i$, but in stead of counting the support of a candidate set using the intersection based approach, it uses a more advanced technique.

This technique is based on the so-called *FP-tree*. The main idea is to store all transactions in the database in a trie based structure. In this way, in stead of storing the cover of every frequent item, the transactions themselves are stored and each item has a linked list linking all transactions in which it occurs together. By using the trie structure, a prefix that is shared by several transactions is stored only once. Nevertheless, the amount of consumed memory is usually much more as compared to Eclat (Goethals, 2004).

The main advantage of this technique is that it can exploit the so-called *single prefix path* case. That is, when it seems that all transactions in the currently observed conditional database share the same prefix, the prefix can be removed, and all subsets of that prefix can afterwards be added to all frequent sets that can still be found (Han et al., 2004), resulting in significant performance improvements. As we will see later, however, an almost equally effective technique can be used in Eclat, based on the notion of closure of a set.

5. Concise representations

If the number of frequent sets for a given database is large, it could become infeasible to generate them all. Moreover, if the database is dense, or the minimal support threshold is set too low, then there could exist a lot of very large frequent sets, which would make sending them all to the output infeasible to begin with. Indeed, a frequent set of size k includes the existence of at least $2^k - 1$ frequent sets, i.e. all of its subsets. To overcome this problem, several proposals have been made to generate only a concise representation of all frequent sets for a given database such that, if necessary, the frequency of a set, or the support of a set not in that representation can be efficiently determined or estimated (Gunopulos et al., 2003; Bayardo, 1998; Mannila, 1997; Pasquier et al., 1999; Boulicaut et al., 2003; Bykowski and Rigotti, 2001; Calders and

Goethals, 2002; Calders and Goethals, 2003). In this section, we address the most popular.

5.1 Maximal Frequent Sets

Since the collection of all frequent sets is downward closed, it can be represented by its maximal elements, the so called *maximal frequent sets*. Most algorithms that have been proposed to find the maximal frequent sets rely on the same general structure as the Apriori and Eclat algorithm. The main additions are the use of several lookahead techniques and efficient subset checking.

The Max-Miner algorithm, proposed by Bayardo (1998), is an adapted version of the Apriori algorithm to which two lookahead techniques are added. Initially, all candidate $k+1$-sets are partitioned such that all sets sharing the same k-prefix are in a single part. Hence, in one such part, corresponding to a prefix set X, each candidate set adds exactly one item to X. Denote this set of 'added' items by I. When a superset of $X \cup I$ is already known to be frequent, this part of candidate sets can already be removed, since they can never belong to the maximal frequent sets anymore, and hence, also their supports don't need to be counted anymore. This subset checking procedure is done using a similar hash-tree as is used to store all frequent and candidate sets in Apriori.

First, during the support counting procedure, for each part, not only the support of all candidate sets is counted, but also the support of $X \cup I$. If it turns out that this set it frequent, again none of its subsets need to be generated anymore, since they can never belong to the maximal frequent sets. All other $k{+}1$-sets that turn out to be frequent are added to the collection of maximal sets unless a superset is already known to be frequent, and all subsets are removed from the collection, since, obviously, they are not maximal.

A second technique is the so called *support lower bounding* technique. That is, after counting the support of every candidate set $X \cup \{i\}$, it is possible to compute a lower bound on the support its supersets using the following inequality:

$$support(X \cup J) \geq support(X) - \sum_{i \in J} support(X) - support(X \cup \{i\}).$$

For every part with prefix set X, this bound is computed starting with J containing the most frequent item, after which items are added in frequency decreasing order as long as the total sum remains above the minimum support threshold. Finally, $X \cup J$ is added to the maximal frequent sets and all its subsets are removed.

Obviously, these techniques result in additional pruning power on top of the Apriori algorithm, when only maximal frequent sets are needed. Later, several other algorithms used similar lookahead techniques on top of depth-first algorithms such as Eclat. Among them, the most popular are GenMax (Gouda and Zaki, 2001) and MAFIA (Burdick *et al*., 2001), which also use more advanced techniques to check whether a superset of a candidate set was already found to be frequent. Also the FP-tree approach has shown to be effective for maximal frequent set mining (G. Grahne, 2003; Liu et al., 2003).

A completely different approach, called *Dualize and Advance*, was proposed by Gunopulos et al. (2003). Here, a randomized algorithm finds a few maximal frequent sets by simply adding items to a frequent set until no extension is possible anymore. Then, all other maximal frequent sets can be found similarly by adding items to sets which are so called minimal hypergraph transversals of the complements of all already found maximal frequent sets. Although the algorithm has been theoretically shown to be better than all other proposed algorithms, until now, extensive experiments have only shown otherwise (Uno and Satoh, 2003; Goethals and Zaki, 2003).

5.2 Closed Frequent Sets

Another very popular concise representation of all frequent sets are the so called *closed frequent sets*, proposed by Pasquier et al (1999). A set is called closed if its support is different from the supports of its supersets. Although all frequent sets can essentially be closed, in practice, it shows that a lot of sets are not. Also here, several different algorithms, based on those described earlier, have been proposed to find only the closed frequent sets. The main added pruning technique simply checks for each set whether its support is the same as any of its subsets. If this is the case, the item can immediately be added to all frequent supersets of that subset, and does not need to be considered separately anymore as it can never result in a closed frequent set. Again, efficient subset checking techniques are necessary to make sure that a generated frequent has no closed superset with the same support that was generated earlier. Efficient algorithms include CHARM (Zaki and Hsiao, 2002) and CLOSET+ (Wang et al., 2003), and many of their improvements (G. Grahne, 2003; Liu et al., 2003).

5.3 Non Derivable Frequent Sets

Although the support monotonicity property is very simple and easy, it is possible to derive much better bounds on the support of a candidate set I, by using the inclusion-exclusion principle, given the supports of all subsets of I (Calders and Goethals, 2002). More specifically, for any subset $J \subseteq I$, we obtain a lower or an upper bound on the support of I using one of the following formulas.

If $|I \setminus J|$ is odd, then

$$support(I) \leq \sum_{J \subseteq X} (-1)^{|I \setminus X|+1} support(X). \tag{17.1}$$

If $|I \setminus J|$ is even, then

$$support(I) \geq \sum_{J \subseteq X} (-1)^{|I \setminus X|+1} support(X). \tag{17.2}$$

Then, when the smallest upper bound is less than the minimal support threshold, the set does not need to be counted anymore, but more interestingly, if the largest lower bound is equal to the smallest upper bound of the support of the set, then it also does not need to be counted anymore since these bounds are necessarily equal to support itself. Such a set is called *derivable* as its support can be derived from the supports of its subsets, or *non-derivable* otherwise. A nice property of the collection of non-derivable frequent sets is that it is downward closed. That is, every subset of a non-derivable set is non-derivable. An additional interesting property is that the size of the largest non-derivable set is at most $1 + \log |\mathcal{D}|$ where $|\mathcal{D}|$ denotes the total number of transactions in the database.

As a result, it makes sense to generate only the non-derivable frequent sets as its derivable counterparts essentially give no new information about the database. Also, the Apriori algorithm can easily be adapted to generate only the non-derivable frequent sets by implementing the inclusion-exclusion formulas as stated above. The resulting algorithm is called NDI (Calders and Goethals, 2002).

6. Theoretical Aspects

Already in the first section of this chapter, we made clear how hard the problem of frequent set mining is. More specifically, the search space of all possible frequent sets is exponential in the number of items and the number of transactions in the database tends to be huge such that the number of scans through it should be minimized. Of course, we can make it all sound as hard as we want, but fortunately, also some theoretical results have been presented, proving the hardness of the frequent set mining problems.

First, Gunupolos et al. studied the problem of counting the number of frequent sets and have proven it to be #P-hard (Gunopulos et al., 2003). Additionally, it was shown that deciding whether there is a maximal frequent set of size k, is NP-complete (Gunopulos et al., 2003). After that, Yang has shown that even counting the number of maximal frequent sets is #P-hard (Yang, 2004).

Ramesh et al. presented several results on the size distributions of frequent sets and their feasibility (G. Ramesh, 2003). Mielikäinen introduced and studied the *inverse frequent set mining problem*, i.e., given all frequent sets, what is the computational complexity of finding a database consistent with the collection of frequent sets (Mielikäinen, 2003). It is shown that this problem is NP-hard and its enumeration conterpart, counting the number of compatible databases, also #P-hard. Similarly, Calders introduced and studied the FREQSAT problem, i.e. given some set-interval pairs, does there exist a database such that for every pair, the support of the set falls in the interval? Again, it is shown that this problem is NP-complete (Calders, 2004).

7. Further Reading

During the first ten years after the proposal of the frequent set mining problem, several hundreds of scientific papers were written on the topic and it seems that this trend is keeping its pace. For a fair comparison of all these algorithms, a contest is organized to find the best implementations in order to to understand precisely why and under what conditions one algorithm would outperform another (Goethals and Zaki, 2003).

Of course, many articles also study variations of the frequent set mining problem. In this section, we list the most prominent, but refer the interested reader to the original articles.

Another interesting issue is how to effectively exploit more contraints next to the frequency constraint (Srikant et al., 1997). For example, find all sets contained in a specific set or containing a specific set, or boolean combinations of those (Goethals and den Bussche, 2000). Ng et al. have listed a large collection of constraints and classified them into several classes for which different optimization techniques could be used (Ng et al., 1998). The most studied classes or the class of so-called *anti-monotone* constraints, as is the minimal support threshold, and the *monotone constraints*, such as the minimum length constraint (Bonchi et al., 2003).

Combining the exploitation of constraints with the notion of concise representations for the collection of frequent sets has been widely studied within the *inductive database* framework (Mannila, 1997) as they are both crucial steps towards an effective optimization of so called *Data Mining queries*.

When databases contain only a small number of transactions, but a huge number of different items, then it is best to focus on only the closed frequent sets, and a slightly different approach might be benificial (Pan et al., 2003; Rioult et al., 2003). More specifically, as a closed set is essentially the intersection of transactions of the given database (while a non-closed set is not), these approaches perform a search traversal through all combinations of transactions in stead of all combinations of items.

Since privacy in Data Mining presents several important issues, also private frequent set mining has been studied (Vaidya and Clifton, 2002). Also from a theoretical point of view, several problems closely related to frequent set mining remain unsolved (Mannila, 2002).

References

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

Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., and Verkamo, A. (1996). Fast discovery of association rules. In Fayyad, U., Piatetsky-Shapiro, G., Smyth, P., and Uthurusamy, R., editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. MIT Press.

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

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

Bayardo, Jr., R. (1998). Efficiently mining long patterns from databases. In (Haas and Tiwary, 1998), pages 85–93.

Bonchi, F., Giannotti, F., Mazzanti, A., and Pedreschi, D. (2003). Exante: Anticipated data reduction in constrained pattern mining. In (Lavrac et al., 2003).

Borgelt, C. and Kruse, R. (2002). Induction of association rules: Apriori implementation. In Härdle, W. and Rönz, B., editors, *Proceedings of the 15th Conference on Computational Statistics*, pages 395–400. Physica-Verlag.

Boulicaut, J.-F., Bykowski, A., and Rigotti, C. (2003). Free-sets: A condensed representation of boolean data for the approximation of frequency queries. *Data Mining and Knowledge Discovery*, 7(1):5–22.

Brin, S., Motwani, R., Ullman, J., and Tsur, S. (1997). 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.

Burdick, D., Calimlim, M., and Gehrke, J. (2001). MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 443–452. IEEE Computer Society.

Bykowski, A. and Rigotti, C. (2001). A condensed representation to find frequent patterns. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-*

SIGART Symposium on Principles of Database Systems, pages 267–273. ACM Press.

Calders, T. (2004). Computational complexity of itemset frequency satisfiability. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 143–154. ACM Press.

Calders, T. and Goethals, B. (2002). Mining all non-derivable frequent itemsets. In Elomaa, T., Mannila, H., and Toivonen, H., 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.

Calders, T. and Goethals, B. (2003). Minimal k-free representations of frequent sets. In (Lavrac et al., 2003), pages 71–82.

Cercone, N., Lin, T., and Wu, X., editors (2001). *Proceedings of the 2001 IEEE International Conference on Data Mining*. IEEE Computer Society.

Dayal, U., Gray, P., and Nishio, S., editors (1995). *Proceedings 21th International Conference on Very Large Data Bases*. Morgan Kaufmann.

G. Grahne, J. Z. (2003). Efficiently using prefix-trees in mining frequent itemset. In (Goethals and Zaki, 2003).

G. Ramesh, W. Maniatty, M. Z. (2003). Feasible itemset distributions in Data Mining: theory and application. In *Proceedings of the Twenty-second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 284–295. ACM Press.

Geerts, F., Goethals, B., and den Bussche, J. V. (2001). A tight upper bound on the number of candidate patterns. In (Cercone et al., 2001), pages 155–162.

Getoor, L., Senator, T., Domingos, P., and Faloutsos, C., editors (2003). *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press.

Goethals, B. (2004). Memory issues in frequent itemset mining. In Haddad, H., Omicini, A., Wainwright, R., and Liebrock, L., editors, *Proceedings of the 2004 ACM symposium on Applied computing*, pages 530–534. ACM Press.

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

Goethals, B. and Zaki, M., editors (2003). *Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, volume 90 of *CEUR Workshop Proceedings*.

Gouda, K. and Zaki, M. (2001). Efficiently mining maximal frequent itemset. In (Cercone et al., 2001), pages 163–170.

Gunopulos, D., Khardon, R., Mannila, H., Saluja, S., Toivonen, H., and Sharma, R. (2003). Discovering all most specific sentences. *ACM Transactions on Database Systems*, 28(2):140–174.

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

Han, J., Pei, J., Yin, Y., and Mao, R. (2004). Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87.

Holsheimer, M., Kersten, M., Mannila, H., and Toivonen, H. (1995). A perspective on databases and Data Mining. In Fayyad, U. and Uthurusamy, R., editors, *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*, pages 150–155. AAAI Press.

Lavrac, N., Gamberger, D., Blockeel, H., and Todorovski, L., editors (2003). *Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases*, volume 2838 of *Lecture Notes in Computer Science*. Springer.

Liu, G., Lu, H., Yu, J., Wei, W., and Xiao, X. (2003). AFOPT: An efficient implementation of pattern growth approach. In (Goethals and Zaki, 2003).

Mannila, H. (1997). Inductive databases and condensed representations for Data Mining. In Maluszynski, J., editor, *Proceedings of the 1997 International Symposium on Logic Programming*, pages 21–30. MIT Press.

Mannila, H. (2002). Local and global methods in Data Mining: Basic techniques and open problems. In Widmayer, P., Ruiz, F., Morales, R., Hennessy, M., Eidenbenz, S., and Conejo, R., editors, *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 57–68. Springer.

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

Mielikäinen, T. (2003). On inverse frequent set mining. In Du, W. and Clifton, C., editors, *2nd Workshop on Privacy Preserving Data Mining*, pages 18–23.

Ng, R., Lakshmanan, L., Han, J., and Pang, A. (1998). Exploratory mining and pruning optimizations of constrained association rules. In (Haas and Tiwary, 1998), pages 13–24.

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

Pan, F., Cong, G., and A.K.H. Tung, J. Yang, M. Z. (2003). Carpenter: finding closed patterns in long biological datasets. In (Getoor et al., 2003), pages 637–642.

Park, J., Chen, M.-S., and Yu, P. (1995). 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.

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

Rioult, F., Boulicaut, J.-F., and B. Crémilleux, J. B. (2003). Using transposition for pattern discovery from microarray data. In Zaki, M. and Aggarwal, C., editors, *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 73–79. ACM Press.

Savasere, A., Omiecinski, E., and Navathe, S. (1995). An efficient algorithm for mining association rules in large databases. In (Dayal et al., 1995), pages 432–444.

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

Srikant, R. and Agrawal, R. (1995). Mining generalized association rules. In (Dayal et al., 1995), pages 407–419.

Srikant, R., Vu, Q., and Agrawal, R. (1997). Mining association rules with item constraints. In Heckerman, D., Mannila, H., and Pregibon, D., editors, *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, pages 66–73. AAAI Press.

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

Uno, T. and Satoh, K. (2003). Detailed description of an algorithm for enumeration of maximal frequent sets with irredundant dualization. In (Goethals and Zaki, 2003).

Vaidya, J. and Clifton, C. (2002). Privacy preserving association rule mining in vertically partitioned data. In Hand, D., Keim, D., and Ng, R., editors, *Proceedings of the Eight ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 639–644. ACM Press.

Wang, J., Han, J., and Pei, J. (2003). CLOSET+: searching for the best strategies for mining frequent closed itemsets. In (Getoor et al., 2003), pages 236–245.

Yang, G. (2004). The complexity of mining maximal frequent itemsets and maximal frequent patterns. In DuMouchel, W., Gehrke, J., Ghosh, J., and Kohavi, R., editors, *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press.

Zaki, M. (2000). Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390.

Zaki, M. and Gouda, K. (2003). Fast vertical mining using diffsets. In (Getoor et al., 2003), pages 326–335.

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