A New Algorithm for Faster Mining of Generalized Association Rules

Jochen Hipp[1], Andreas Myka[1], Rüdiger Wirth[2], and Ulrich Güntzer[1]

[1] Wilhelm Schickard Institute, University of Tübingen, 72076 Tübingen, Germany
{hippj, myka, guentzer}@informatik.uni-tuebingen.de
[2] Daimler Benz AG, Research & Technology FT3/KL, 89081 Ulm, Germany
wirth@dbag.ulm.daimlerbenz.com

Abstract. Generalized association rules are a very important extension of boolean association rules, but with current approaches mining generalized rules is computationally very expensive. Especially when considering the rule generation as being part of an interactive KDD-process this becomes annoying. In this paper we discuss strengths and weaknesses of known approaches to generate frequent itemsets. Based on the insights we derive a new algorithm, called Prutax, to mine generalized frequent itemsets. The basic ideas of the algorithm and further optimisation are described. Experiments with both synthetic and real-life data show that Prutax is an order of magnitude faster than previous approaches.

1 Introduction

Association rules were introduced in [1] and today the mining of such rules can be seen as one of the key tasks of KDD. The intuitive meaning of an association rule $X \Rightarrow Y$, where X and Y are sets of items, is that a transaction containing X is likely to also contain Y. The prototypical application is the analysis of supermarket basket data where rules like "34% of all customers who buy fish also buy white wine" may be found. Obviously, association rules can be quite useful in business applications. But even the above example shows a severe shortcoming: Rather than containing an item like "white wine" the transactions will typically contain item identifiers derived from a barcode which distinguishes the items quite accurately. So instead of finding a few useful rules a huge set of rules like "Smoked Irish Salmon, 500g $\Rightarrow$ Augey, Bordeaux White, 1993" will be generated.

One approach is to substitute all items with their generalizations, but this leads to a loss of information. A more elaborate solution are the generalized association rules introduced in [3,4]. These rules extend the simple ones to contain items from arbitrary levels of a taxonomy. However this leads to an enormous increase in itemsets that have to be evaluated, because many more generalized items are frequent than simple ones. At the same time when considering the rule generation as being part of an interactive KDD-process performance becomes very important. To address this problem, we present Prutax, a new algorithm for fast mining of generalized association rules.

The problem of mining association rules is formally described in Section 2. In Section 3 the common approaches to generate boolean frequent itemsets are

sketched. In addition we explain how the performance of each of the approaches is differently affected by the characteristics of the database. We show that, when mining boolean rules, choosing a superior algorithm is not as straightforward as supposed in literature. In Section 4 we present the algorithm Prutax to mine generalized frequent itemsets. Based on the insights of Section 3 we conclude that in the presence of a taxonomy a considerable performance gain can be expected when determining supports by tid-intersections instead of counting actual occurrences. In order to avoid the overhead of partitioning the database, as done eg in [5], we use a special kind of depth-first search. Our approach differs from [7] in so far as it is able to prune candidates that have an infrequent subset and is not restricted to mine frequent k-itemsets only for $k \geq 3$. In addition our modified depth-first search makes it possible to add further optimisations to the basic algorithm which are described in the remainder of Section 4. In Section 5 the performance of the new algorithm is compared to former algorithms on both synthetic and real-life data. The paper ends with a short summary in Section 6.

2 Problem Description

Let $\mathcal{I} = \{x_1, \ldots, x_n\}$ be a set of distinct literals, called items. A set $X \subseteq \mathcal{I}$ that contains k items is said to be a k-itemset or just an itemset. Let $\mathcal{D}$ be a set of transactions T, $T \subseteq \mathcal{I}$. A transaction T supports an itemset X if $X \subseteq T$. The fraction of transactions from $\mathcal{D}$ that support X is called the support of X, denoted by $\mathsf{supp}(X)$. An association rule is an implication $X \Rightarrow Y$, where $X, Y \subseteq \mathcal{I}$ and $X \cap Y = \emptyset$. In addition to $\mathsf{supp}(X \Rightarrow Y) = \mathsf{supp}(X \cup Y)$ every rule is assigned the confidence $\mathsf{conf}(X \Rightarrow Y) = \mathsf{supp}(X \cup Y)/\mathsf{supp}(X)$, cf [2].

A set of taxonomies $\mathcal{T}$ is coded as an acyclic directed graph with the items as nodes. An edge (x, y) means that x "is-a" y. x is the child and y the parent. $\mathsf{ancestors}(x)$ denotes the set of all items $\hat{x}$ for which an edge $(x, \hat{x})$ exists in the transitive closure of $\mathcal{T}$. A non-leaf item is called a generalized item. Accordingly, a transaction T supports an item $x \in \mathcal{I}$ if $x \in T$ or $\exists y \in T : x \in \mathsf{ancestors}(y)$. T supports an itemset $X \subseteq \mathcal{I}$ if T supports every item in X. A generalized rule may contain items from arbitrary levels of the taxonomy, cf [4].

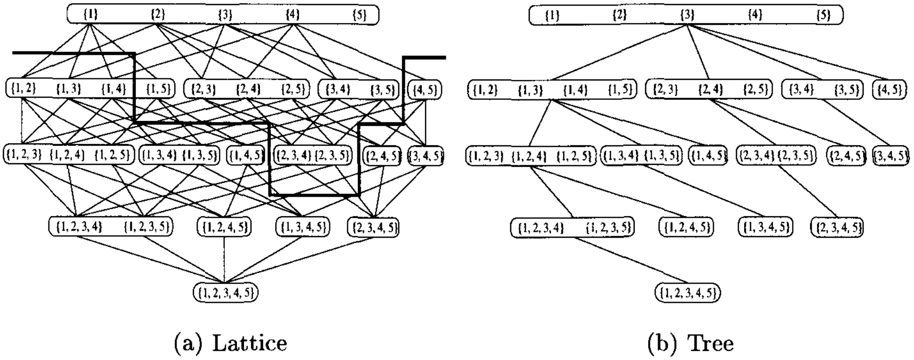
To obtain all association rules that achieve minimal thresholds for support and confidence, $\mathsf{minsupp}$ and $\mathsf{minconf}$ respectively, it suffices to generate the set of all frequent itemsets, cf [2].

3 Generation of Frequent Itemsets

Since the introduction of association rules in [1], several algorithms for the generation of frequent itemsets have been developed, eg Apriori [2], Partition [5] or Eclat [7]. In this section we give a general survey and discuss the pros and cons.

3.1 Basics

Except for the empty set all $2^{|\mathcal{I}|}$ subsets of the itemset $\mathcal{I} = \{1, 2, 3, 4, 5\}$ are shown as a lattice in Figure 1(a). The thin lines indicate the subset relations. The bold line is an example of actual itemset support and separates the frequent itemsets in the upper part from the infrequent ones in the lower part. The goal



(a) Lattice (b) Tree

Fig. 1. Subsets of $\mathcal{I} = \{1, 2, 3, 4, 5\}$

is to traverse the lattice in such a way that all frequent itemsets are found but as few infrequent itemsets as possible are visited. To achieve this the algorithms use the downward closure property of itemset support: All subsets of a frequent itemset must also be frequent.

Let map: $\mathcal{I} \to \{1, \ldots, |\mathcal{I}|\}$ be a mapping that maps all items $x \in \mathcal{I}$ one-to-one onto natural numbers. Now the items are totally ordered by the usual relation "$<$". In addition, for $X \subseteq \mathcal{I}$ let $X.\mathsf{item} : \{1, \ldots, |X|\} \to \mathcal{I} : n \mapsto X.\mathsf{item}_n$ be a mapping with $X.\mathsf{item}_n$ denoting the n-th item of the items $x \in X$ increasingly sorted by "$<$". The n-prefix of an itemset X with $n \leq |X|$ is then given by $P = \{X.\mathsf{item}_m \mid 1 \leq m \leq n\}$. The common strategy of the recent algorithms is to join every two frequent $(k-1)$-itemsets which have a $(k-2)$-prefix in common. Such a join results in a candidate k-itemset. In Figure 1(a), eg , $\{2, 3, 4\}$ and $\{2, 3, 5\}$ form the candidate $\{2, 3, 4, 5\}$. After the support of a candidate is counted, it will by pruned or will be added to the set of frequent itemsets. This approach starts with the 1-itemsets as the set of candidate 1-itemsets. Whenever a candidate turns out to be frequent, it may be used for further candidate generation. This strategy ensures that all frequent itemsets are visited. At the same time the number of infrequent itemsets that are visited is reduced.

3.2 Lattice Traversal

Let the classes $E(P), P \subseteq \mathcal{I}$ with $E(P) = \{H \subseteq \mathcal{I} \mid |H| = |P| + 1 \text{ and } P \text{ is a prefix of } H\}$ be the nodes of a tree. Two classes are connected by an edge, if all itemsets of the first class can be generated by joining two itemsets of the second class, eg Figure 1(b).

When traversing the tree by breadth-first search – BFS – as done by Apriori and Partition, all frequent $(k-1)$-itemsets are known when generating the candidate k-itemsets. Therefore both algorithms improve performance by pruning those candidates that have an infrequent subset before counting supports.

The algorithms introduced in [7] use depth-first search – DFS – to traverse the tree but are restricted to mine only frequent k-itemsets with $k \geq 3$. Furthermore, a fundamental drawback is not mentioned in [7]: Arbitrary DFS does not guarantee that the infrequent $(|C| - 1)$-subsets of a candidate C are known at

the time the support of C has to be determined. Therefore, candidates having infrequent subsets cannot be pruned by the algorithms from [7] and usually must be counted at large expenses instead. Especially when mining generalized rules this problem becomes impeding because the pruning is required by an important optimization. In Section 4 we show how to integrate the pruning into DFS.

3.3 Support Counting

Counting actual occurrences of candidates as done by Apriori relies on a hashtree structure, cf [4]. Obviously counting candidates that occur quite infrequently is fairly cheap. But with growing candidate sizes, this approach gets more and more expensive because the number of levels of the hashtree increases.

Counting support by intersecting tid-sets as done in Partition and Eclat requires for every item in $\mathcal{I}$ the tid-set, i.e. the set of transactions containing this item, to be provided. Tid-sets also exist for every itemset X and are denoted by $X.\mathsf{tids}$. The support of a candidate $C = X \cup Y$ is obtained by the intersection $C.\mathsf{tids} = X.\mathsf{tids} \cap Y.\mathsf{tids}$ and evaluating $|C.\mathsf{tids}|$. Intersecting tid-sets does not suffer from large candidate sizes. Yet, there is another problem: Regardless of the actual support of a candidate, the cost of an intersection is at least $\mathsf{min}(\{|X.\mathsf{tids}|, |Y.\mathsf{tids}|\})$ operations. In addition, memory usage may become critical but solutions to this problem are given in [5, 7].

3.4 Conclusion

Obviously the performance of the algorithms is affected differently by the characteristics of the database. Whereas the inability to prune candidates that have an infrequent subset is an obvious disadvantage of DFS, the performance studies in literature seem to be contradictory concerning the different approaches of support counting: According to [5] the algorithm Partition that relies on tid-intersections achieves a much better performance than Apriori that counts actual occurrences (both use BFS). On the other hand in [6] it is shown that tid-intersections usually are more expensive than counting actual occurences. Our own experiments support [6]: Even when extending Eclat to prune candidates that have an infrequent subset – cf Subsection 4.2 – Eclat does not perform better than Apriori for k-itemsets with $k \geq 1$ on datasets comparable to those in [5].

4 Algorithm Prutax

Based on the insights from the boolean case described in Section 3, the basic approach to mine generalized association rules is derived in Section 4.1. The resulting algorithm is then further optimised in Sections 4.2 - 4.4.

4.1 Basic Idea

One perception of Section 3 is that determining supports by tid-intersection instead of counting actual occurrences is favoured under certain conditions:

(a) A shrinking average gap between the number of actual occurrences of a candidate $C = X \cup Y$ and $\mathsf{min}(\{|X.\mathsf{tids}|, |Y.\mathsf{tids}|\})$.
(b) A growing average size of candidates and frequent itemsets.

Experiments showed that both conditions typically become true when introducing a taxonomy. The reason is that, usually, the more general an item is, the higher is its support. Therefore our algorithm Prutax uses tid-intersections and combines them with DFS for two reasons: With DFS only the tid-sets of the frequent 1-itemsets that need roughly the same amount of memory as the original transactions must be maintained in memory permanently and partitioning, as done in [5], is not necessary under normal conditions. In addition it allows to prune candidates by taxonomy information as described later.

4.2 Optimisation i: Prune Candidates with infrequent subsets

As noticed in Section 3.2 arbitrary DFS does not allow the pruning of candidates by their infrequent subsets. This is due to the fact that in general not all infrequent $(|C|-1)$-subsets of a candidate C are already known at the time its support must be determined. In order to cope with this problem, the following relation is defined for all pairs of itemsets $X, Y \in K_m = \{H \subseteq \mathcal{I} \mid |H| = m\}$:

$$\begin{aligned} X < Y :\Leftrightarrow\ & \exists n, n \in \mathbb{N}, n \leq m : X.\mathsf{item}_n < Y.\mathsf{item}_n \wedge \\ & \forall n', n' \in \mathbb{N} \wedge n' < n : X.\mathsf{item}_{n'} = Y.\mathsf{item}_{n'} \end{aligned}$$

"$<$" imposes the lexicographic order on the itemsets of each K_m. Consequently for every subset K' of K_m there exists exactly one largest itemset. Let $C = \{c_1, \ldots, c_n\}$ with $c_i = C.\mathsf{item}_i$ be a subset of $\mathcal{I}$ and let $P = \{c_1, \ldots, c_{n-1}\}$ be the $(|C|-1)$-prefix of C. For all $S \subseteq C$ with $|S| = |C| - 1$ follows:

$$\begin{aligned} S \neq P \Rightarrow S = & \{c_1, \ldots, c_{j-1}, c_{j+1}, \ldots, c_n\} \\ & \Rightarrow P.\mathsf{item}_j < S.\mathsf{item}_j \wedge \forall n', n' \in \mathbb{N}, n' < j : P.\mathsf{item}_{n'} = S.\mathsf{item}_{n'} \\ & \Rightarrow P < S. \end{aligned}$$

When generating a candidate $C = X \cup Y$ with $X < Y$ as supposed in Section 3, the $(|C|-1)$-prefix of C is X. Consequently, all $S \subseteq C, |S| = |C|-1, S \neq X$ have the property $S > X$. To assure that all infrequent itemsets $S > X$ are known at the time C is generated, it suffices to realize a right-most DFS by choosing the largest itemset according to "$<$" whenever there is the choice of:

(a) different candidates to be counted,
(b) different prefixes P that determine the next class $E(P)$ to descend to.

4.3 Optimisation ii: Avoid counting redundant supports

In [4] the following property of itemsets containing generalized items is described:

> The support of an itemset X that contains both an item x and its ancestor $\hat{x}$ will be the same as the support of the itemset $X \setminus \{\hat{x}\}$.

As a consequence, counting the support of an itemset that contains both an item and an ancestor of this item is redundant. If our optimisation i) is applied to DFS, the only thing that has to be done in order to avoid determining redundant supports is to treat redundant 2-itemsets as infrequent, cf [4].

4.4 Optimisation iii: Prune by taxonomy

The itemset $\hat{X}$ is an ancestor of X if $|\hat{X}| = |X|$ and $\hat{X}$ can be generated by replacing one or more items from X with one of their ancestors. $\hat{X}$ is a parent of X if there is no X' with X' being an ancestor of X and $\hat{X}$ being an ancestor of X'. Obviously those candidates can be pruned that have an infrequent parent. In the context of BFS the use of this approach is rather limited, cf [4], but not when using our optimised right-most DFS lattice traversal: As a prerequisite, it has to be guaranteed that at the time the support of a candidate is to be determined all its infrequent ancestors are known. Accordingly, a certain ordering among itemsets of the same size has to be followed when traversing the subsets of $\mathcal{I}$. This can be realized by introducing the depth of each itemset:

$$\mathsf{depth} : 2^{\mathcal{I}} \to \mathbb{N} : X \mapsto \begin{cases} 0, & \text{if } \{\hat{X} \mid \hat{X} \text{ is ancestor of } X\} = \emptyset \\ \max(\{\mathsf{depth}(\hat{X} \mid \hat{X} \text{ is parent of } X\}) + 1, & \text{else} \end{cases}$$

For all ancestors $\hat{C}$ of a candidate C $\mathsf{depth}(\hat{C}) < \mathsf{depth}(C)$ holds. Consequently determining the support of all candidates C with $\mathsf{depth}(C) = i$ before counting the support of a candidate C' with $\mathsf{depth}(C') = i + 1$ ensures that all infrequent ancestors of a candidate are known when its support has to be counted.

Pruning candidates by their infrequent subsets requires to follow the order imposed by the mapping map, cf optimisation i). On first sight, this order may seem to be contradictory to the one imposed by depth. But whereas depth is determined by the taxonomy, map only serves as a common base for the relation "$<$" no matter what specific kind of order it actually implies. Let x_n be the n-th element of the list generated by sorting all items $x \in \mathcal{I}$ according to descending depth. Now map is chosen so that $\mathsf{map}(x_n) = n$ holds for all x_n. It follows:

$$\mathsf{map}(x) < \mathsf{map}(\hat{x}) \Leftrightarrow \mathsf{depth}(\{x\}) \geq \mathsf{depth}(\{\hat{x}\})$$

For arbitrarily chosen $X, \hat{X} \subseteq \mathcal{I}$ with $|X| = |\hat{X}|$ that means:

$$\begin{aligned} X < \hat{X} &\Rightarrow \exists n, n \in \mathbb{N} : X.\mathsf{item}_n < \hat{X}.\mathsf{item}_n \\ &\Rightarrow \mathsf{depth}(\{X.\mathsf{item}_n\}) \geq \mathsf{depth}(\{\hat{X}.\mathsf{item}_n\}) \\ &\Rightarrow X.\mathsf{item}_n \text{ is not an ancestor of } \hat{X}.\mathsf{item}_n \\ &\Rightarrow X \text{ is not an ancestor of } \hat{X} \end{aligned}$$

In other words, $\hat{C} > C$ holds for all ancestors $\hat{C}$ of C. Accordingly, if choosing map as indicated above, applying optimization i) will ensure that all infrequent ancestors are known when processing a candidate.

4.5 Algorithm

The algorithm Prutax that incorporates the described ideas is given in Figure 2. The parameters $H_1, \ldots, H_n$ are initialized with the frequent 1-itemsets, considering $H_i < H_{i+1}, \forall H_i,\ 1 \leq i < n$. The part to enrich the tid-sets of the generalized items with the tid-sets of their children has been left out but its implementation is straightforward as a recursive function. The set of frequent itemsets, F, should be implemented as a hashtree [2] in order to allow fast lookup.

```
(1)  function prutax(H_1, ..., H_n)
(2)    for i = n - 1 down to 1 do
(3)      E.ClearArray();
(4)      for j = n down to i + 1 do
(5)        C = H_i ∪ H_j;
(6)        if not (|C| = 2 ∧ C.item_2 ∈ ancestors(C.item_1)) then
(7)          if not (|C| ≠ 2 ∧ ∃S, S ⊆ C, |S| = |C| - 1 : S ∉ F) then
(8)            if not (∃Ĉ, Ĉ ⊆ I, Ĉ is parent of C : Ĉ ∉ F) then do
(9)              C.tids = H_i.tids ∩ H_j.tids;
(10)             if|C.tids| ≥ minsupp · |D| then do
(11)               E = C.Append(E);
(12)               F = F ∪ {C};
(13)               endif;
(14)             endif;
(15)       endfor;
(16)     prutax(E);
(17)     endfor;
(18)   end.
```

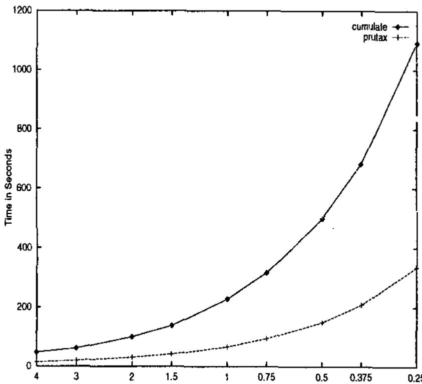
Fig. 2. Algorithm Prutax

5 Performance Study

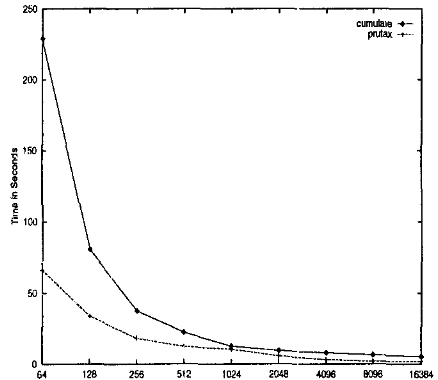
Prutax is evaluated and compared with the algorithm Cumulate, cf [4], that uses BFS and counting of actual occurences. For EstMerge, a variation of Cumulate that uses sampling, only a performance gain up to 30% and no fundamental different performance behaviour is detected in [4]. So due to the difficulties in duplicating the circumstances of sampling only "pure" Cumulate was taken into account. In addition the algorithms from the ML-family, cf [3], are not considered: They perform badly because of the extra pass over the database done for every level of the taxonomy and excessive data pre-processing. To make the comparison fair the time to generate the tid-sets of the generalized items is added to the total time when generating frequent itemsets with Prutax.

The first part of the performance evaluation relies on synthetic datasets similar to those from [4]. They were generated by the tool gen but with slightly modified default values. We decreased the number of items on level 2 of the taxonomy – the number of roots in terms of gen – from 250 down to 64. Even this seems to be quite large, eg if thinking of level 2 representing the different departments of a supermarket. Yet, this number could only be slightly decreased because of Cumulate performing badly on lower values as shown in Figure 3(b). In addition we decided to double the default value of the minimal support from 0.5% to 1% in order to decrease the gap between the algorithms. Furthermore the number of transactions has been decreased from 1000K to 100K. This does not affect the overall results because the needed time grows linearly with the number of transactions for both algorithms.

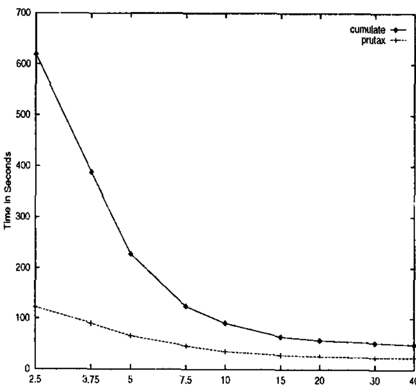
According to our evaluation Prutax is more than 3 times faster than Cumulate at minsupp=0.25%. As shown in Figure 3(a), the gap is even increasing with decreasing support. This is due to the fact that, when lowering minsupp, the average size of the candidates and frequent itemsets increases and the average gap between the number of actual occurrences of a candidate $C = X \cup Y$ and $\min(\{|X.\text{tids}|, |Y.\text{tids}|\})$ shrinks. Fewer items at level 2 of the taxonomy mean



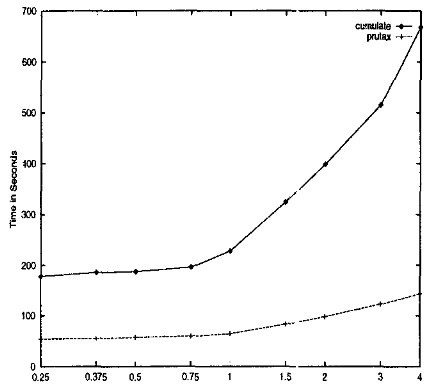
(a) Minimum Support in %



(b) Number of Items on Level 2



(c) Fanout



(d) Depth-Ratio

Fig. 3. Experiments on Synthetic Data

that the support of these items increases. The effects are the same as when lowering minsupp, cf Figure 3(b). Decreasing the parameter fanout, cf Figure 3(c), corresponds to increasing the number of taxonomy levels. At lower values Prutax is able to prune quite a lot of candidates by their infrequent parents and at fanout=2.5 performs nearly 6 times faster than Cumulate. With higher depth-ratio the support of frequent itemsets that contain items from the lower levels of the taxonomy increases and more candidate counting is done on lower levels. Again Prutax is able to prune quite many candidates by their infrequent parents. At depth-ratio=4 Prutax achieves a performance gain of almost factor 5 over Cumulate. In addition to the experiments from Figure 3 both algorithms scale linearly with the number of transactions and showed to be independent from the number of items when the number of frequent itemsets stays roughly constant.

A further evaluation is done on a real-life dataset. It consists of about 70,000 customer transactions from a supermarket. There is a total number of about 60,000 different items with an average of 10.5 items per transaction. Again Prutax outperforms Cumulate being even 10 times faster at minsupp = 0.6%.

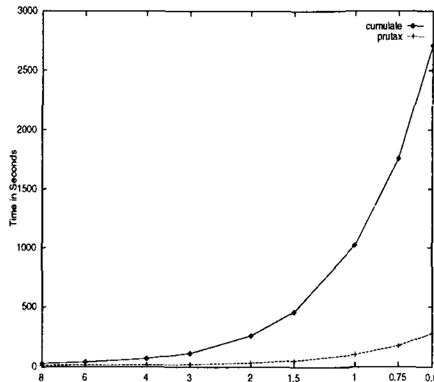


Fig. 4. Comparison on a Real-life Dataset (Minimum Support in %)

6 Summary

In this paper we described a new algorithm for fast mining of generalized association rules. First we discussed todays commonly known approaches to mine frequent itemsets. Based on this, support counting by tid-set intersections was recognised as the best approach to mine generalized association rules. In order to avoid the overhead of partitioning the database, as it is proposed in [5], we introduced right-most DFS to traverse the lattice. In contrast to DFS, as described in [7], the resulting algorithm is able to prune candidates that have an infrequent subset and in addition is not restricted to mine frequent k-itemsets only for $k \geq 3$. Furthermore we were able to add further optimisations that use the taxonomy to improve candidate pruning. The resulting algorithm Prutax achieves an order of magnitude better performance than former approaches. This performance gain was shown on both synthetic data and on a real-life dataset. The overall benefit from Prutax is the enhanced consideration of taxonomic relationships: the stronger the taxonomy dominates rule generation the more significant is the performance gain, especially at lower values of minimum support.

References

1. R. Agrawal, T. Imielinski, A. Swami: Mining Association Rules between Sets of Items in Large Databases, In *Proc. of ACM SIGMOD '93*, 1993, Washington, USA.
2. R. Agrawal, R. Srikant, Fast Algorithms for Mining Association Rules, In *Proc. of the VLDB '94*, 1994, Santiago, Chile.
3. J. Han, Y. Fu, Discovery of Multiple-Level Association Rules from Large Databases, In *Proc. of the VLDB '95*, 1995, Zürich, Switzerland.
4. R. Srikant, R. Agrawal, Mining Generalized Association Rules, In *Proc. of the VLDB '95*, 1995, Zürich, Switzerland.
5. A. Savasere, E. Omiecinski, S. Navathe, An Efficient Algorithm for Mining Association Rules in Large databases, In *Proc. of the VLDB '95*, 1995, Zürich, Switzerland.
6. M. Holsheimer, M. Kersten, Heikki Mannila, Hannu Toivonen, A Perspective on Databases and Data Mining, In *Proc. of the KDD '95*, 1995, Montreal, Canada.
7. M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li, New Algorithms for Fast Discovery of Association Rules, In *Proc. of the KDD '97*, 1997, Newport Beach, California.