Algorithms for Association Rule Mining – A General Survey and Comparison

Jochen Hipp
Wilhelm Schickard-Institute
University of Tübingen
72076 Tübingen, Germany
jochen.hipp@informatik.uni-tuebingen.de

Ulrich Güntzer
Wilhelm Schickard-Institute
University of Tübingen
72076 Tübingen, Germany
guentzer@informatik.uni-tuebingen.de

Gholamreza Nakhaeizadeh
DaimlerChrysler AG
Research & Technology
FT3/AD
89081 Ulm, Germany
rheza.nakhaeizadeh@daimlerchrysler.com

ABSTRACT

Today there are several efficient algorithms that cope with the popular and computationally expensive task of association rule mining. Actually, these algorithms are more or less described on their own. In this paper we explain the fundamentals of association rule mining and moreover derive a general framework. Based on this we describe today's approaches in context by pointing out common aspects and differences. After that we thoroughly investigate their strengths and weaknesses and carry out several runtime experiments. It turns out that the runtime behavior of the algorithms is much more similar as to be expected.

1. INTRODUCTION

1.1 Association Rules

Since its introduction in 1993 [1] the task of association rule mining has received a great deal of attention. Today the mining of such rules is still one of the most popular pattern-discovery methods in KDD.

In brief, an association rule is an expression $X \Rightarrow Y$, where X and Y are sets of items. The meaning of such rules is quite intuitive: Given a database $\mathcal{D}$ of transactions – where each transaction $T \in \mathcal{D}$ is a set of items –, $X \Rightarrow Y$ expresses that whenever a transaction T contains X than T probably contains Y also. The probability or rule confidence is defined as the percentage of transactions containing Y in addition to X with regard to the overall number of transactions containing X. That is, the rule confidence can be understood as the conditional probability $p(Y \subseteq T | X \subseteq T)$.

The idea of mining association rules originates from the analysis of market-basket data where rules like "A customer who buys products x_1 and x_2 will also buy product y with probability c%." are found. Their direct applicability to business problems together with their inherent understandability – even for non data mining experts – made association rules a popular mining method. Moreover it became clear that association rules are not restricted to dependency analysis in the context of retail applications, but are successfully applicable to a wide range of business problems.

When mining association rules there are mainly two problems to deal with: First of all there is the algorithmic complexity. The number of rules grows exponentially with the number of items. Fortunately today's algorithms are able to efficiently prune this immense search space based on minimal thresholds for quality measures on the rules. Second, interesting rules must be picked from the set of generated rules. This might be quite costly because the generated rule sets normally are quite large – e.g. more than $100,000$ rules are not uncommon – and in contrast the percentage of useful rules is typically only a very small fraction. The work concerning the second problem mainly focuses on supporting the user when browsing the rule set, e.g. [14] and the development of further useful quality measures on the rules, e.g. [7; 6; 22].

1.2 Outline of the Paper

In this paper we deal with the algorithmic aspects of association rule mining. In fact, a broad variety of efficient algorithms to mine association rules have been developed during the last years. These approaches are more or less described separately in the corresponding literature. To overcome this situation we give a general survey of the basic ideas behind association rule mining. In Section 2 we identify the basic strategies and describe them in detail. The resulting framework is used in Section 3 to systematize and present today's most common approaches in context. Furthermore we show the common principles and differences between the algorithms. Finally in Section 4 we complete our overview with a comparison of the algorithms concerning efficiency. This comparison is based on theoretic considerations and concrete runtime experiments. In Section 5 we conclude with a short summary of our results.

1.3 Related Work

In our work we mainly restrict ourselves to what we call the "classic association rule problem". That is, the mining of *all* rules existing in a database $\mathcal{D}$ with respect to minimal thresholds on certain quality measures. $\mathcal{D}$ in this case consists of market-basket like data, that is, transactions containing $10-20$ items in the average out of a total set of $1,000-100,000$ items.

Although the "classic problem" is still topic of further research, during recent years many algorithms for specialized tasks have been developed: First of all, there are the approaches that enhance the association rules itself. E.g. quantitative association rules, e.g. [24], generalized association rules, e.g. [23; 12] and to some extent the work on sequential patterns, e.g. [3; 15]. Moreover there are several generalizations of the rule problem, e.g. [16; 27].

In addition algorithms were developed that mine well defined subsets of the rule set according to specified items or quality measures etc, e.g. general constraints [17; 25], optimized rules [8; 20], maximal frequent itemsets [28], and frequent closed itemsets [18; 19]. Moreover there are algorithms to mine dense databases [5]. These approaches are supplemented by algorithms for online mining of association rules, e.g. [10] and incremental algorithms, e.g. [26; 4].

2. BASIC PRINCIPLES

2.1 Formal Problem Description

Let $\mathcal{I} = \{x_1, \ldots, x_n\}$ be a set of distinct literals, called items. A set $X \subseteq \mathcal{I}$ with $k = |X|$ is called a k-itemset or simply an itemset. Let a database $\mathcal{D}$ be a multi-set of subsets of $\mathcal{I}$. Each $T \in \mathcal{D}$ is called a transaction. We say that a transaction $T \in \mathcal{D}$ supports an itemset $X \subseteq \mathcal{I}$ if $X \subseteq T$ holds. An association rule is an expression $X \Rightarrow Y$, where X, Y are itemsets and $X \cap Y = \emptyset$ holds. The fraction of transactions T supporting an itemset X with respect to database $\mathcal{D}$ is called the support of X, $\mathsf{supp}(X) = |\{T \in \mathcal{D} \mid X \subseteq T\}|/|\mathcal{D}|$. The support of a rule $X \Rightarrow Y$ is defined as $\mathsf{supp}(X \Rightarrow Y) = \mathsf{supp}(X \cup Y)$. The confidence of this rule is defined as $\mathsf{conf}(X \Rightarrow Y) = \mathsf{supp}(X \cup Y)/\mathsf{supp}(X)$, c.f. [2].

As mentioned before the main challenge when mining association rules is the immense number of rules that theoretically must be considered. In fact the number of rules grows exponentially with $|\mathcal{I}|$. Since it is neither practical nor desirable to mine such a huge set of rules, the rule sets are typically restricted by minimal thresholds for the quality measures support and confidence, minsupp and minconf respectively.

This restriction allows us to split the problem into two separate parts [2]: An itemset X is frequent if $\mathsf{supp}(X) \geq$ minsupp. Once, $\mathcal{F} = \{X \subseteq \mathcal{I} \mid X \text{ frequent}\}$, the set of all frequent itemsets together with their support values is known, deriving the desired association rules is straight forward (See [2] for minor enhancements.): For every $X \in \mathcal{F}$ check the confidence of all rules $X \setminus Y \Rightarrow Y$, $Y \subseteq X, \emptyset \neq Y \neq X$ and drop those that do not achieve minconf. According to its definition above, it suffices to know all support values of the subsets of X to determine the confidence of each rule. The knowledge about the support values of all subsets of X is ensured by the downward closure property of itemset support: All subsets of a frequent itemset must also be frequent, c.f. [2].

With that in mind the task of association rule mining can be reduced to the problem of finding all itemsets that are frequent with respect to a given minimal threshold minsupp. The rest of this paper and most of the literature on association rule mining addresses exactly this topic.

2.2 Traversing the Search Space

As explained we need to find all itemsets that satisfy minsupp. For practical applications looking at all subsets of $\mathcal{I}$ is doomed to failure by the huge search space. In fact, a linearly growing number of items still implies an exponential growing number of itemsets that need to be considered.

For the special case $\mathcal{I} = \{1, 2, 3, 4\}$ we visualize the search space that forms a lattice in Figure 1, c.f. [28]. The frequent

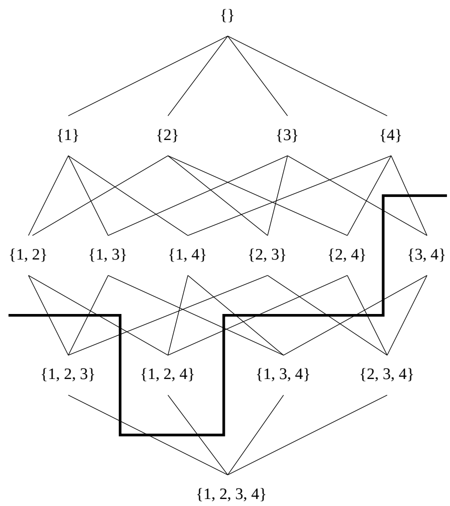

Figure 1: Lattice for $\mathcal{I} = \{1, 2, 3, 4\}$

itemsets are located in the upper part of the figure whereas the infrequent ones are located in the lower part. Although we do not explicitly specify support values for each of the itemsets, we assume that the bold border separates the frequent from the infrequent itemsets. The existence of such a border is independent of any particular database $\mathcal{D}$ and minsupp. Its existence is solely guaranteed by the downward closure property of itemset support.

The basic principle of the common algorithms is to employ this border to efficiently prune the search space. As soon as the border is found, we are able to restrict ourselves on determining the support values of the itemsets above the border and to ignore the itemsets below.

Let $\mathsf{map}\colon \mathcal{I} \rightarrow \{1, \ldots, |\mathcal{I}|\}$ be a mapping that maps all items $x \in \mathcal{I}$ one-to-one onto natural numbers. Now the items can be seen as totally ordered by the relation "$<$" between natural numbers. In addition, for $X \subseteq \mathcal{I}$ let $X.\mathsf{item} : \{1, \ldots, |X|\} \rightarrow \mathcal{I} : n \mapsto X.\mathsf{item}_n$ 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 defined by $P = \{X.\mathsf{item}_m \mid 1 \leq m \leq n\}$, c.f. [12].

Let the classes $E(P), P \subseteq \mathcal{I}$ with $E(P) = \{X \subseteq \mathcal{I} \mid |X| = |P| + 1 \text{ and } P \text{ is a prefix of } X\}$ be the nodes of a tree. Two nodes are connected by an edge, if all itemsets of a class E can be generated by joining two itemsets of the parent class E', e.g. Figure 2.

Together with the downward closure property of itemset support this implies the following: If the parent class E' of a class E does not contain at least two frequent itemsets than E must also not contain any frequent itemset. If we encounter such a class E' on our way down the tree, then we have reached the border separating the infrequent from the frequent itemsets. We do not need to go behind this border so we prune E and all descendants of E from the search space.

The latter procedure allows us to efficiently restrict the number of itemsets to investigate. We simply determine the

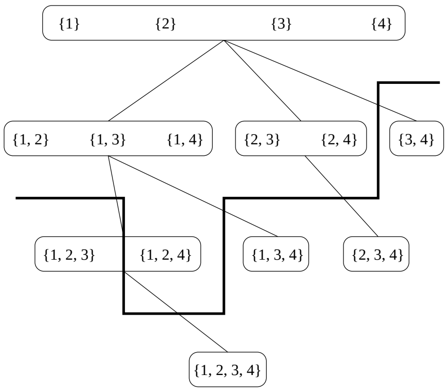

Figure 2: Tree for $\mathcal{I} = \{1, 2, 3, 4\}$

support values only of those itemsets that we "visit" on our search for the border between frequent and infrequent itemsets. Finally, the actual strategy to search for the border is at our own choice. Today's common approaches employ both breadth-first search (BFS) or depth-first search (DFS). With BFS the support values of all $(k-1)$-itemsets are determined before counting the support values of the k-itemsets. In contrast, DFS recursively descends following the tree structure defined above.

2.3 Determine Itemset Supports

In the following an itemset that is potentially frequent and for which we decide to determine its support during lattice traversal is called a candidate itemset or simply a candidate.
One common approach to determine the support value of an itemset is to directly *count* its *occurrences* in the database. For that purpose a counter is set up and initialized to zero for each itemset that is currently under investigation. Then all transactions are scanned and whenever one of the candidates is recognized as a subset of a transaction, its counter is incremented. Typically subset generation and candidate lookup is integrated and implemented on a hashtree or a similar data structure. In brief, not all subsets of each transaction are generated but only those that are contained in the candidates or that have a prefix in common with at least one of the candidates, c.f. [2] for further details.
Another approach is to determine the support values of candidates by *set intersections*. A tid is a unique transaction identifier. For a single item the tidlist is the set of identifiers that correspond to the transactions containing this item. Accordingly tidlists also exist for every itemset X and are denoted by $X.\mathsf{tidlist}$. The tidlist of a candidate $C = X \cup Y$ is obtained by $C.\mathsf{tidlist} = X.\mathsf{tidlist} \bigcap Y.\mathsf{tidlist}$. The tidlists are sorted in ascending order to allow efficient intersections.
Note that by buffering the tidlists of frequent candidates as intermediate results, we remarkably speedup the generation of the tidlists of the following candidates. Finally the actual support of a candidate is obtained by determining $|C.\mathsf{tlist}|$.

3. COMMON ALGORITHMS

In this section we briefly describe and systematize the most common algorithms. We do this by referring to the fundamentals of frequent itemset generation that we identified in the previous section. Our goal is not to go to much into detail but to show the basic principles and the differences between the approaches.

3.1 Systematization

The algorithms that we consider in this paper are systematized in Figure 3. We characterize each of the algorithms a) by its strategy to traverse the search space and b) by its strategy to determine the support values of the itemsets. In

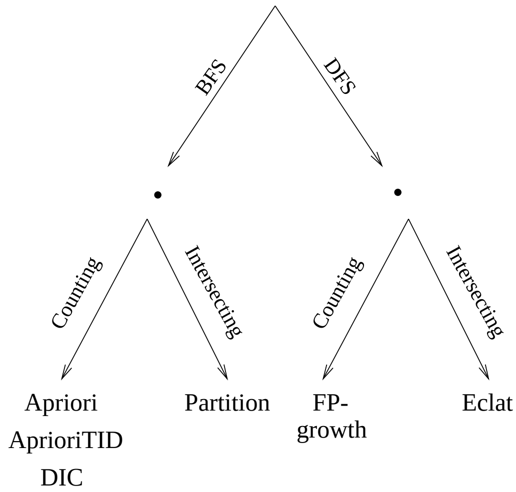

Figure 3: Systematization of the Algorithms

addition an algorithm may employ specific optimizations for further speedup.

3.2 BFS and Counting Occurrences

The most popular algorithm of this type is **Apriori** [2] where also the downward closure property of itemset support was introduced. Apriori makes additional use of this property by pruning those candidates that have an infrequent subset before counting their supports. This optimization becomes possible because BFS ensures that the support values of all subsets of a candidate are known in advance.
Apriori counts all candidates of a cardinality k together in one scan over the database. The critical part is looking up the candidates in each of the transactions. For this purpose [2] introduces a so called hashtree structure. The items in each transaction are used to descend in the hashtree. Whenever we reach one of its leafs, we find a set of candidates having a common prefix that is contained in the transaction. Then these candidates are searched in the transaction that has been encoded as a bitmap before. In the case of success the counter of the candidate in the tree is incremented.
AprioriTID [2] is an extension of the basic Apriori approach. Instead of relying on the raw database AprioriTID internally represents each transaction by the current candidates it contains. With **AprioriHybrid** both approaches are combined, c.f. [2]. To some extent also **SETM** [13] is an Apriori(TID)-like algorithm which is intended to be implemented directly in SQL.
DIC is a further variation of the Apriori-Algorithm [7]. DIC softens the strict separation between counting and generating candidates. Whenever a candidate reaches $\mathsf{minsupp}$, that is even when this candidate has not yet "seen" all transactions, DIC starts generating additional candidates based

on it. For that purpose a prefix-tree is employed. In contrast to the hashtree, each node – leaf node or inner node – of the prefix-tree is assigned to exactly one candidate respectively frequent itemset. In contrast to the usage of a hashtree that means whenever we reach a node we can be sure that the itemset associated with this node is contained in the transaction. Furthermore interlocking support determination and candidate generation decreases the number of database scans.

3.3 BFS and TID-List Intersections

The **Partition**-Algorithm [21] is an Apriori-like algorithm that uses set intersections to determine support values. As described above Apriori determines the support values of all $(k-1)$-candidates before counting the k-candidates. The problem is that Partition of course wants to use the tidlists of the frequent $(k-1)$-itemsets to generate the tidlists of the k-candidates. Obviously the size of those intermediate results easily grows beyond the physical memory limitations of common machines. To overcome this Partition splits the database into several chunks that are treated independently. The size of each chunk is chosen in such a way that all intermediate tidlists fit into main memory. After determining the frequent itemsets for each database chunk, an extra scan is necessary to ensure that the locally frequent itemsets are also globally frequent.

3.4 DFS and Counting Occurrences

Counting occurrences assumes candidate sets of a reasonable size. For each of those candidate sets a database scan is performed. E.g. Apriori that relies on BFS scans the database once for every candidate size k. When using DFS the candidate sets consist only of the itemsets of one of the nodes of the tree from Section 2.2. Obviously scanning the database for every node results in tremendous overhead. The simple combination of DFS with counting occurrences is therefore of no practical relevance, c.f.[11].

Recently in [9] a fundamentally new approach called **FP-growth** was introduced. In a preprocessing step FP-growth derives a highly condensed representation of the transaction data, the so called FP-tree. The generation of the FP-tree is done by counting occurrences and DFS. In contrast to former DFS-approaches, FP-growth does not follow the nodes of the tree from Subsection 2.2, but directly descends to *some part* of the itemsets in the search space. In a second step FP-growth uses the FP-tree to derive the support values of all frequent itemsets.

3.5 DFS and TID-List Intersections

In [28] the algorithm **Eclat** is introduced, that combines DFS with tidlist intersections. When using DFS it suffices to keep the tidlists on the path from the root down to the class currently investigated in memory. That is, splitting the database as done by Partition is no longer needed.

Eclat employs an optimization called "fast intersections". Whenever we intersect two tidlists then we are only interested in the resulting tidlist if its cardinality reaches minsupp. In other words, we should break off each intersection as soon as it is sure that it will not achieve this threshold.

Eclat originally generates only frequent itemsets of size ≥ 3. We modified Eclat to mine also the frequent 1- and 2-itemsets by calling it on the class that contains the 1-itemsets together with their tidlists.

In addition in [28] algorithms that mine only the maximal frequent itemsets are introduced, e.g. **MaxEclat**. An itemset X is maximal frequent if for every frequent itemset Y $X \subseteq Y \Rightarrow Y = X$ holds. We do not consider these algorithms because although it is straight forward to derive the set of all frequent itemsets from the maximal frequent itemsets this does not hold for the corresponding support values. Without those, we are not able to derive rule confidences and therefore we cannot generate association rules.

4. COMPARISON OF THE ALGORITHMS

In this section we compare the algorithms and explain the observed differences in performance behavior.

4.1 Experiments

To carry out performance studies we implemented the most common algorithms to mine frequent itemsets, namely Apriori, DIC, Partition, and Eclat, in C++. Actually we had to leave out DIC in the charts for reasons explained later. In addition we did not consider AprioriTID and FP-growth because these algorithms were designed to mine data that is not typical for retail environments, that is data containing quite long patterns.

The experiments were performed on a SUN UltraSPARC-II workstation clocked at 248Mhz. The experiments in Figures 4 – 11 were carried out on synthetic datasets from [2; 21]. These datasets were generated with a data generator [2] that simulates the buying behavior of customers in retail business. Dataset "T10.I4.D100K" means an average transaction size of 10, an average size of the maximal potentially frequent itemsets of 4 and 100,000 generated transactions. The number of patterns was set to $2,000$ and the number of items to $1,000$.

In addition to the experiments from [2; 21], we restricted the maximal length of generated itemsets from 1 up to 9 on the dataset "T20.I4.D100K" at minsupp $= 0.33\%$, c.f. Figure 9. Figures 10 and 11 show the behavior of the algorithms on real-world applications. The basket data consists of about $70,000$ customer transactions with approximately $60,000$ different items. The average transaction size is ≈ 10.5 items. The car equipment data contains information about $700,000$ cars with about $6,000$ items. In the average ≈ 20 items are assigned to each car.

It is important to say that all algorithms scale linearly with the database size.

4.2 Counting Occurrences vs. Intersecting Sets

The basic question concerning the runtime of the algorithms is whether counting occurrences or intersecting tidlists shows better performance results. The advantage of counting is that only candidates that actually occur in the transactions cause any effort. In contrast, an intersection means at least passing through all tids of the smaller of the two tidlists, even if the candidate is not contained in the database at all. ("Fast Intersections" save some costs but we still need to pass a substantial number of tids.) But intersections also have their benefits. Counting implies looking up the candidates in the transactions. Of course this can get quite expensive for candidates of higher cardinality. On the contrary when using intersections the size of the candidate under investigation does not have any influence.

In practice both effects seem to balance out on the basket-

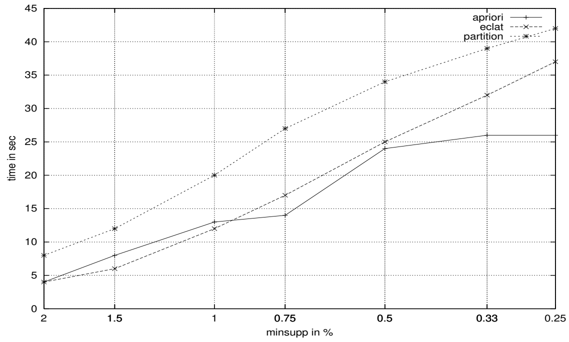

Figure 4: T10.I2.D100K

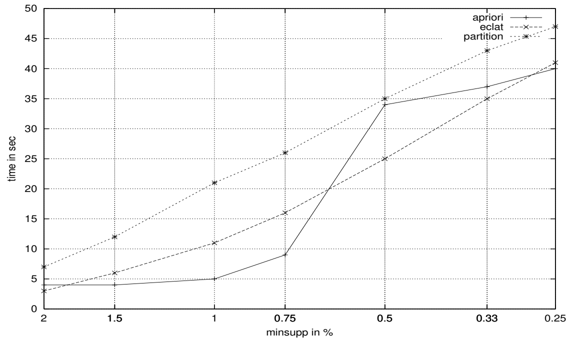

Figure 5: T10.I4.D100K

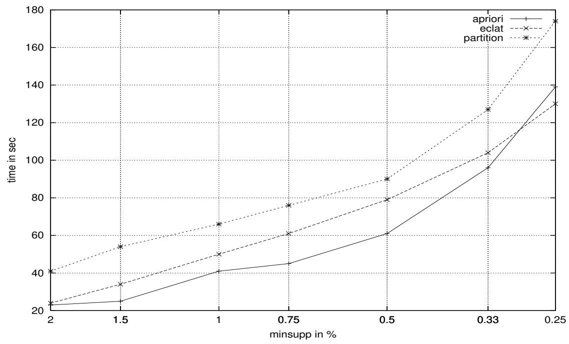

Figure 6: T20.I2.D100K

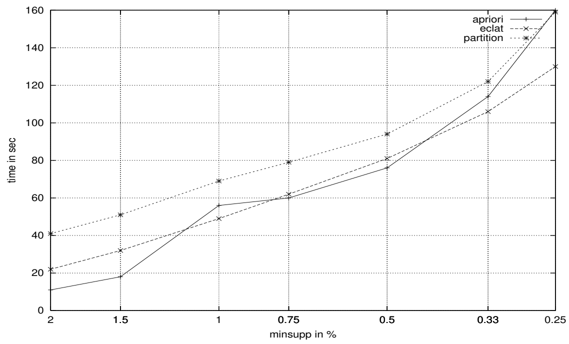

Figure 7: T20.I4.D100K

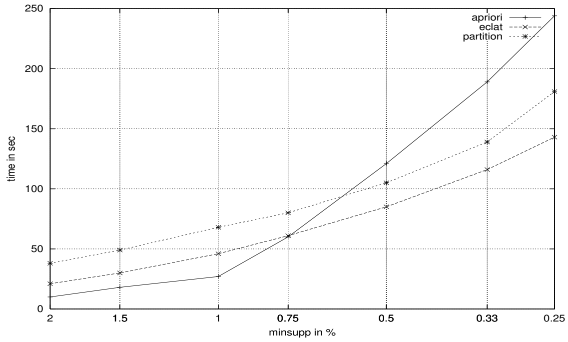

Figure 8: T20.I6.D100K

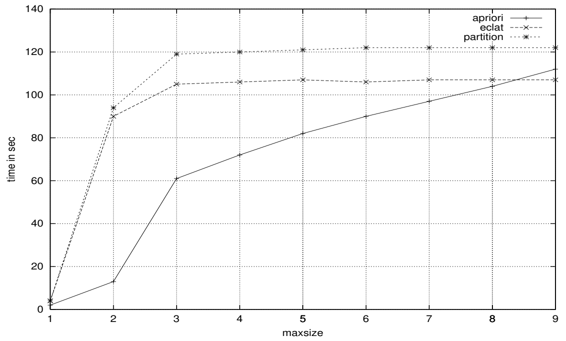

Figure 9: Maximal Frequent Itemset Size

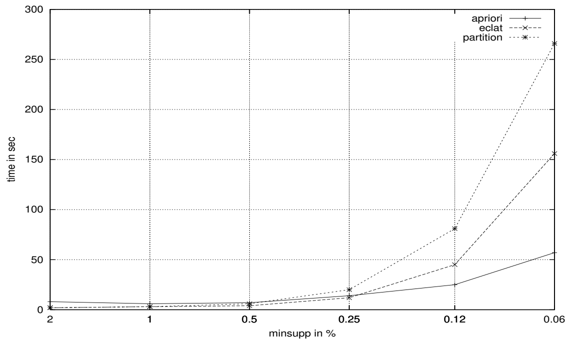

Figure 10: Basket Data

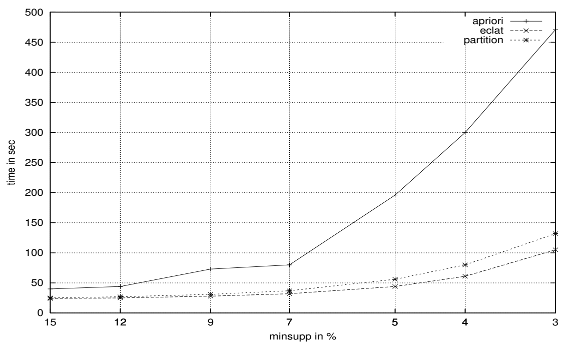

Figure 11: Car Equipment

like data. The runtime behavior in Figures 4 – 8 does not show any substantial differences between the algorithm Apriori that counts occurrences and the tidlists intersecting algorithms Partition and Eclat. Only at quite low average size of the maximal potentially frequent itemsets, e.g. "T10.I2.D100K" in Figure 4, Apriori is somehow superior whereas at larger average size of the maximal potentially frequent itemsets, e.g. "T20.I6.D100K" in Figure 8, Partition and Eclat perform better. The same explanation holds for the real-world experiments. With an average size of ≈ 2.2 items at $\mathsf{minsupp} = 0.06\%$ the frequent itemsets found in the basket data were rather short compared to the frequent itemsets from the car equipment database, that contained ≈ 4.1 items in the average at $\mathsf{minsupp} = 3\%$.

In Figure 9 it becomes clear what happens behind the scenes: Eclat and Partition spend most of their time with determining the support values of the 2- and 3-candidates whereas Apriori is efficiently handling such small itemsets. In contrast for itemsets with size ≥ 4 the additional effort caused for Eclat and Partition is to be neglected whereas this does not hold for Apriori.

4.3 Relaxing the Separation between Candidate Generation and Support Counting

On the one hand the introduction of the prefix-tree with DIC allows relaxing the separation between candidate generation and support counting and therefore reduces the number of database scans. Moreover each node is assigned to precisely one itemset. Consequently looking up candidates in bitmap-encoded transactions is no longer necessary. On the other hand the memory usage of the prefix-tree caused problems, when we experimented with our own implementation of DIC: The prefix-tree is already setup before all frequent 1-itemsets are known. That means a mapping to frequent items as described in [2] to keep the memory usage of hash tables in each node of the tree small is not possible. Moreover this effect is strengthened by the fact that every candidate is stored in its own node and that in addition a separate node for each prefix of the candidate exists. Another drawback of the prefix-tree is that the frequent itemsets are stored in the same tree as the candidates. Each frequent k-itemset that is not prefix of any k'-candidate with $k' > k$ imposes overhead when counting that is avoided by the hashtree-approach of Apriori.

Actually we did not come to a final result concerning the efficiency of DIC. But what we want to say is that DIC's advantage of reducing the number of database scans should not be overestimated in a retail environment. In [7] a performance gain of only about 30% compared with basic Apriori is detected for data with quite small average size of the maximal potentially frequent itemsets.

4.4 Additional Candidate Pruning

Typically the main task of the algorithms is determining support values. That is, the time spent with candidate generation – and candidate pruning – can be neglected. Consequently Apriori's candidate pruning step helps to reduce the candidates to be counted but does not add any substantial overhead. It is important to note that basic DFS, as employed by Eclat does not allow proper subset pruning. Only right-most DFS as introduced in [12] for the mining of generalized association rules allows transferring the additional prune step of Apriori to algorithms using DFS.

Our experiments suggest that the effect of additional candidate pruning is rather small for the datasets we took into consideration. Additional pruning does not help Partition to compensate Eclat's advantage based on the "fast intersections". This impression was also supported by further studies with an enhanced version of Eclat that incorporates additional candidates pruning by using right-most DFS.

4.5 Splitting the Database

Partition needs to split the database. Whereas this optimization helps to cope with large databases it adds the additional overhead of an extra pass to determine the globally frequent itemsets. In our experiments the size of the transaction sets were always small enough that we could employ Partition without splitting. In [21] especially at lower values for $\mathsf{minsupp}$ Partition that splits suffers strongly. The reason is the increasing number of locally frequent itemsets that finally turn out to be globally infrequent.

4.6 "Fast Intersections"

Eclat's fast intersections are obviously an advantage. The overhead caused by checking whether $\mathsf{minsupp}$ is still reachable is clearly outweighed by breaking off unnecessary intersections. As a result in all our experiments Eclat beats Partition with a nearly constant factor.

5. CONCLUSION

In this paper we dealt with the algorithmic aspects of association rule mining. We restricted ourselves to the "classic" association rule problem, that is the generation of *all* association rules that exist in market basket-like data with respect to minimal thresholds for support and confidence.

From the broad variety of efficient algorithms that have been developed we compared the most important ones. We systematized the algorithms and analyzed their performance based on both runtime experiments and theoretic considerations. The results were quite surprising: Although we identified fundamental differences concerning the employed strategies, the algorithms show quite similar runtime behavior in our experiments. At least there is no algorithm that is fundamentally beating out the other ones. In fact our experiments showed that the advantages and disadvantages we identified concerning the strategy to determine the support values of the frequent itemsets nearly balance out on market basket-like data.

In a forthcoming paper we pursue the development of a hybrid approach that efficiently combines counting occurrences and tidlist intersections [11].

6. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (ACM SIGMOD '93)*, Washington, USA, May 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Int'l Conf. on Very Large Databases (VLDB '94)*, Santiago, Chile, June 1994.

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the Int'l Conf. on Data Engineering (ICDE)*, Taipei, Taiwan, March 1995.

[4] N. F. Ayan, A. U. Tansel, and E. Arkun. An efficient algorithm to update large itemsets with early pruning. In *Proc. of the 5th Int'l Conf. on Knowledge Discovery and Data Mining (KDD '99)*, San Diego, California, USA, August 1999.

[5] R. J. Bayardo Jr., R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *Proc. of the 15th Int'l Conf. on Data Engineering*, Sydney, Australia, March 1999.

[6] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (ACM SIGMOD '97)*, 1997.

[7] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, 1997.

[8] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Mining optimized association rules for numeric attributes. In *Proc. of the 15th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS '96)*, Montreal, Canada, June 1996.

[9] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of the 2000 ACM-SIGMOD Int'l Conf. on Management of Data*, Dallas, Texas, USA, May 2000.

[10] C. Hidber. Online association rule mining. In *Proc. of the 1999 ACM SIGMOD Conf. on Management of Data*, 1999.

[11] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Mining association rules: Deriving a superior algorithm by analysing today's approaches. In *Proc. of the 4th European Conf. on Principles and Practice of Knowledge Discovery*, Lyon, France, September 2000. to appear.

[12] J. Hipp, A. Myka, R. Wirth, and U. Güntzer. A new algorithm for faster mining of generalized association rules. In *Proc. of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD '98)*, Nantes, France, September 1998.

[13] M. Houtsma and A. Swami. Set-oriented mining for association rules in relational databases. Technical Report RJ 9567, IBM Almaden Research Center, San Jose, California, Oktober 1993.

[14] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proc. of the 3rd Int'l Conf. on Information and Knowledge Management*, Gaithersburg, Maryland, 29. Nov - 2. Dez 1994.

[15] H. Mannila, H. Toivonen, and I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3), November 1997.

[16] R. Motwani, E. Cohen, M. Datar, S. Fujiware, A. Gionis, P. Indyk, J. D. Ullman, and C. Yang. Finding interesting associations without support pruning. In *Proc. of the 16th Int'l Conf. on Data engineering (ICDE)*. IEEE, 2000.

[17] R. Ng, L. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. of 1998 ACM SIGMOD Int'l Conf. on Management of Data*, Seattle, Washington, USA, June 1998.

[18] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. of the 7th Int'l Conf. on Database Theory (ICDT'99)*, Jerusalem, Israel, January 1999.

[19] J. Pei, J. Han, and R. Mao. An efficient algorithm for mining frequent closed itemsets. In *Proc. of the 2000 ACM-SIGMOD Int'l Conf. on Management of Data*, Dallas, Texas, USA, May 2000.

[20] R. Rastogi and K. Shim. Mining optimized support rules for numeric attributes. In *Proc. of the 15th Int'l Conf. on Data Engineering*, Sydney, Australia, March 1999. IEEE Computer Society Press.

[21] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21st Conf. on Very Large Databases (VLDB '95)*, Zürich, Switzerland, September 1995.

[22] C. Silverstein, S. Brin, R. Motwani, and J. D. Ullman. Scalable techniques for mining causal structures. In *Proc. of 1998 ACM SIGMOD Int'l Conf. on Management of Data*, Seattle, Washington, USA, June 1998.

[23] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. of the 21st Conf. on Very Large Databases (VLDB '95)*, Zürich, Switzerland, September 1995.

[24] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proc. of the 1996 ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, June 1996.

[25] R. Srikant, Q. Vu, and R. Agrawal. Mining association-rules with item constraints. In *Proc. of the 3rd Int'l Conf. on KDD and Data Mining (KDD '97)*, Newport Beach, California, August 1997.

[26] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Proc. of the 3rd Int'l Conf. on KDD and Data Mining (KDD '97)*, Newport Beach, California, August 1997.

[27] D. Tsur, J. D. Ullman, S. Abitboul, C. Clifton, R. Motwani, S. Nestorov, and A. Rosenthal. Query flocks: A generalization of association-rule mining. In *Proc. of 1998 ACM SIGMOD Int'l Conf. on Management of Data*, Seattle, Washington, USA, June 1998.

[28] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proc. of the 3rd Int'l Conf. on KDD and Data Mining (KDD '97)*, Newport Beach, California, August 1997.