Mining Association Rules: Deriving a Superior Algorithm by Analyzing Today's Approaches

Jochen Hipp[1], Ulrich Güntzer[1], and Gholamreza Nakhaeizadeh[2]

[1] Wilhelm Schickard-Institute, University of Tübingen, 72076 Tübingen, Germany
`{jochen.hipp, guentzer}@informatik.uni-tuebingen.de`

[2] DaimlerChrysler AG, Research & Technology FT3/AD, 89081 Ulm, Germany
`rheza.nakhaeizadeh@daimlerchrysler.com`

Abstract. Since the introduction of association rules, many algorithms have been developed to perform the computationally very intensive task of association rule mining. During recent years there has been the tendency in research to concentrate on developing algorithms for specialized tasks, e.g. for mining optimized rules or incrementally updating rule sets. Here we return to the "classic" problem, namely the efficient generation of *all* association rules that exist in a given set of transactions with respect to minimum support and minimum confidence. From our point of view, the performance problem concerning this task is still not adequately solved.
In this paper we address two topics: First of all, today there is no satisfying comparison of the common algorithms. Therefore we identify the fundamental strategies of association rule mining and present a general framework that is independent of any particular approach and its implementation. Based on this we carefully analyze the algorithms. We explain differences and similarities in performance behavior and complete our theoretic insights by runtime experiments. Second, the results are quite surprising and enable us to derive a new algorithm. This approach avoids the identified pitfalls and at the same time profits from the strengths of known approaches. It turns out that it achieves remarkably better runtimes than the previous algorithms.

1 Introduction

Association rules were introduced in [1]. The intuitive meaning of such rules $X \Rightarrow Y$, where X, Y are sets of items, is that a transaction containing X is likely to also contain Y. The prototypical application is the analysis of basket data where rules like "$p\%$ of all customers who buy $\{x_1, x_2, \dots\}$ also buy $\{y_1, y_2, \dots\}$" are found. Our paper deals with the "classic" mining of associations. That is, the mining of *all* rules that exist in a given database with respect to thresholds on support and confidence. We assume transactions that are typical for retail applications, i.e. transactions containing between 10 to 20 items on average and items out of $\mathcal{I}$ with $|\mathcal{I}| \approx 1,000 - 100,000$. Several algorithms for association rule mining have been developed, e.g. Apriori [2], Partition [9], DIC [4], and

Eclat [11], but finally from our point of view the performance problem is still not satisfyingly solved. Especially in the context of an interactive KDD-process, c.f. [5], performance is still a remaining problem.

Although the algorithms share basic ideas they fundamentally differ in certain aspects and this is also true for their runtime performance. Unfortunately today there is no exhaustive comparison of at least the most important of the algorithms concerning the fundamental ideas behind them or their different runtime behavior. Both topics are addressed in this paper. Moreover based on the insights of our study we derive a new algorithm that remarkably reduces runtimes compared to the previous approaches.

The paper is structured as follows: In Section 2 we identify the general strategies of association rule mining. The overview given is independent of any particular algorithm and its implementation. In Section 3, we systematize the most common algorithms by putting them into the general framework developed in the section before. In addition we show several performance evaluations that, from our point of view, are quite surprising. After that we come to the main part of this section, the detailed explanation of the different runtimes. In Section 4, we deploy the insights from the previous sections and introduce a hybrid algorithm. This algorithm avoids the identified pitfalls and at the same time profits from the recognized strengths of the fundamental strategies. We conclude with performance studies that show the efficiency of this new algorithm. Finally we give a short summary and close with interesting topics for future research.

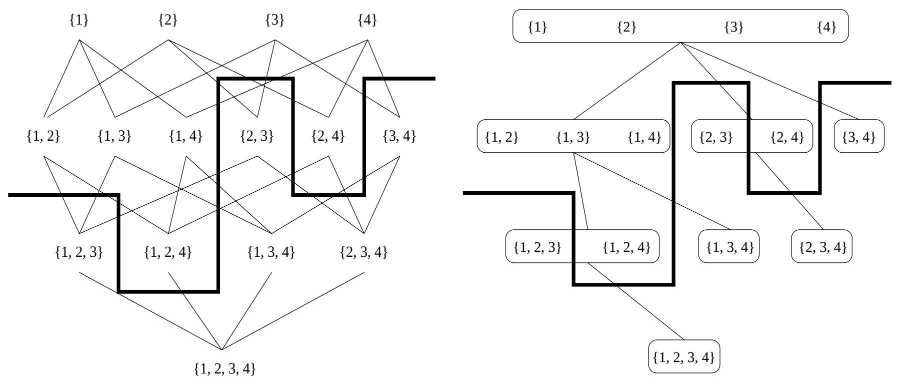
2 Identifying Fundamental Strategies of Association Rule Mining

2.1 Problem Description

Let $\mathcal{I} = \{x_1, \dots, x_n\}$ be a set of distinct items. A set $X \subseteq \mathcal{I}$ with $|X| = k$ is said to be a k-itemset or just an itemset. Let $\mathcal{D}$ be a multi-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 a confidence $\mathsf{conf}(X \Rightarrow Y) = \mathsf{supp}(X \cup Y)/\mathsf{supp}(X)$, c.f. [2]. An itemset X is called frequent if $\mathsf{supp}(X) \geq \mathsf{minsupp}$. For the purpose of association rule generation it suffices to find all frequent itemsets, c.f. [2].

2.2 Search Strategies

Except the empty set the lattice of all $2^{|\mathcal{I}|}$ subsets of the itemset $\mathcal{I}$ are shown in Figure 1(a). 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. Obviously the search space is exponentially growing with $|\mathcal{I}|$. It is therefore not practicable to determine the support of each of the subsets of $\mathcal{I}$ in order to



(a) Lattice (b) Tree

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

decide whether it is frequent or not. Instead, the idea 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 downward closure property of itemset support is employed: All subsets of a frequent itemset are also frequent.

In Figure 1(b) a tree on the itemsets is shown: The nodes are the classes $E(P), P \subseteq \mathcal{I}$, $E(P) = \{H \subseteq \mathcal{I} \mid |H| = |P| + 1$ and P is a “prefix” of $H\}$, where the sets are represented as ordered lists, c.f. [6] for precise formalization. Two nodes are connected by an edge, if all itemsets of the lower class can be generated by joining two itemsets of the upper class. A class E and its descendants only contain frequent itemsets if the parent class E' of E contains at least two frequent itemsets. That is, whenever we encounter a class E that contains less than two frequent itemsets than we are allowed to prune all branches starting in node E without accidentally missing any of the frequent itemsets.

With that pruning, we drastically reduce the search space and moreover we are free to choose the strategy – typically breadth-first search (BFS) or depth-first search (DFS) – to traverse the classes of the lattice, respectively the nodes of the tree.

2.3 Counting Strategies

When traversing the search space, i.e. the subsets of $\mathcal{I}$, we always have to check whether or not potential frequent itemsets achieve minsupp. We call these itemsets candidates and counting their support is done in two fundamentally different ways:

The first strategy is to *count the occurrences* of the candidates by setting up counters and then pass over all transactions. Whenever one of the candidates is contained in a transaction, we increment its counter.

The second strategy indirectly counts the support of candidates by *intersecting sets*. This requires for every itemset that the set of transactions containing this itemset is provided. Such tid-sets 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 $|C.\mathsf{tids}|$.

3 Analysis of the Algorithms

3.1 Algorithms

In this subsection we put the most common algorithms into the general framework that we identified in the previous section:

Apriori [2] is the very first efficient algorithm to mine association rules. Basically it combines BFS with counting of occurrences of candidates. In addition, it employs the following: When using BFS all frequent itemsets at level s of the tree (c.f. Figure 1(b)) are known when starting to count candidates at level $s+1$. Together with the downward closure property of itemset support, this enables Apriori to do additional pruning: Look at all the subsets of size $|C| - 1$ of candidate C and whenever there is at least one of those infrequent, then prune C without counting its support.

Partition [9] combines the Apriori approach with set intersections instead of counting occurrences. That is, Partition also checks the subsets of each candidate for frequency before actually determining its support. In addition, Partition "partitions" the database in several chunks. This is necessary because otherwise the memory usage of the tid-sets to be held simultaneously in main memory would easily grow beyond the physical limitations of common machines.

DIC [4] is an extension of Apriori that aims at minimizing the number of database passes. The idea is to relax the strict separation between generating and counting of candidates: During counting DIC looks for candidates that already achieve minsupp though their final support may still not be determined. Based on these DIC generates new candidates and immediately starts counting them.

In contrast, **Eclat** [11] relies on DFS instead of BFS. With that, Eclat does not need to partition even huge databases although it counts the support values by intersections. The reason is that only the tid-sets of the itemsets on one path from the root down to one of the leaves have to be kept in memory simultaneously. But there is a draw back not described in [11]: DFS implies that in general Eclat cannot prune candidates by looking at their subsets. That is, Eclat does not fully realize the so called apriori_gen()-function. The reason is that basic DFS descends from the root to the leaves of the tree without caring about any subset relation among the itemsets. [11] introduces an important optimization they call "fast intersections". In brief only tid-sets that achieve a size greater than minimum support are of relevance. The idea is to immediately stop an intersection as soon as it is foreseeable that it will never reach this threshold. Eclat as introduced in [11] presumes the frequent itemsets of size 1 and 2 to be known. I.e. Eclat starts at level 3 in the tree. In order to make a fair runtime comparison of Eclat with the other algorithms, we modify Eclat to also start at level 1. This is straight forward to implement by simply calling Eclat on the class consisting of all frequent 1-itemsets.

None of the above algorithms combines counting occurrences with DFS. The reason is evident when considering how counting occurrences actually works: First, a set of candidates is set up, second, the algorithm passes all transactions and increments the counters of the candidates. That is, for each set of candidates one pass over all transactions is necessary. With BFS such a pass is needed for each of level s of the tree as long as there exists at least one candidate of size s. In contrast, DFS would make a separate pass necessary for each class that contains at least one candidate. The costs for each pass over the whole database would contrast with the relatively small number of candidates counted in the pass.

3.2 Performance Studies

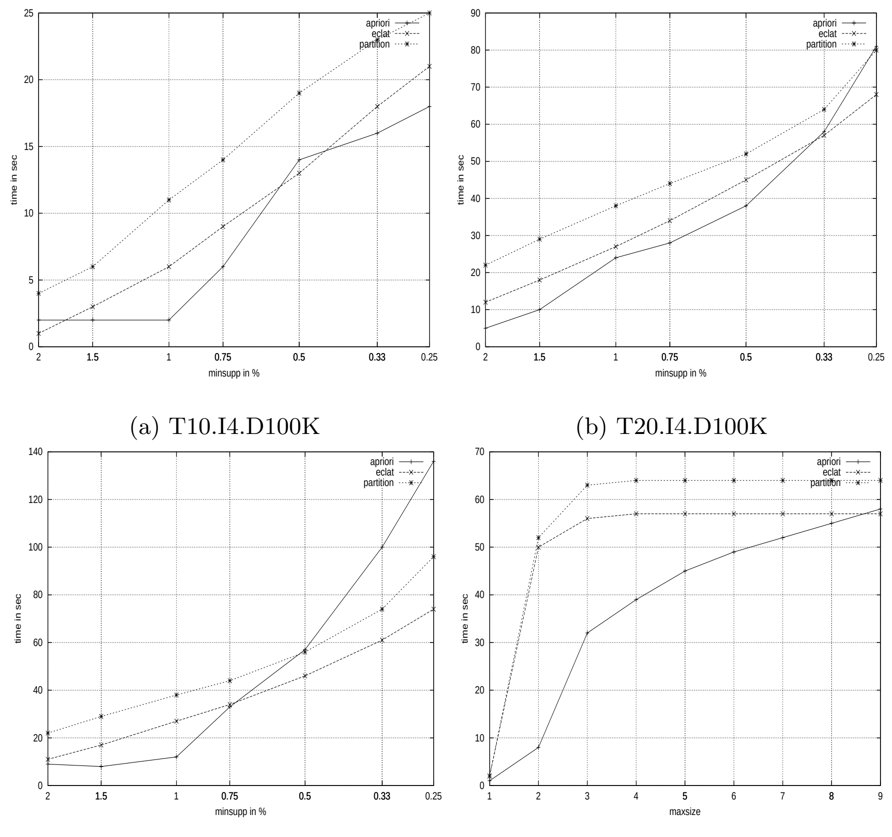
We performed the experiments on an Pentium III Linux machine, running at 500Mhz, and C++-implementations of the algorithms. The experiments and datasets were taken from [2,9]. The naming convention of the datasets reflects their basic characteristics, e.g. "T20.I4.D100K" is a dataset with average transaction size of 20, average frequent itemset size of 4, and consisting of $100,000$ transactions. The test sets were generated with the dataset generator from [8]. The number of items was always set to $1,000$ and the number patterns always to $2,000$. In addition to [2,9], we experimented with different restrictions on the maximal size of the generated frequent itemsets based on the dataset "T20.I4.D100K" at minsupp 0.33%.

3.3 Comparing the Algorithms

In Figure 2 the algorithms show quite similar runtime behaviors. At least there is no algorithm fundamentally beating out the other ones. This is quite surprising, especially with regard to former publications. To explain this, we start with some general thoughts on performance issues concerning the strategies described in Section 2.

The influence of the search strategy is relatively small. Only the fact that DFS does not allow proper candidate pruning by subset checking makes BFS somewhat superior to DFS. But we must keep in mind that checking subsets might be costly, especially for larger itemsets. In addition, it makes only sense for itemsets of a size greater than 2. But as Figure 2(d) and [7] show, the time spent with the itemsets of size 2 may dominate the whole generation process.

Counting occurrences is usually done by using a hashtree, c.f. [2]. Counting a candidate that occurs rather infrequently is quite cheap. Costs are only caused by the actual occurrences of the candidate in the transactions. In contrast, whenever incrementing the candidate size by one the hashtree grows one level. I.e., especially for larger candidate sizes – caused by the characteristics of the dataset or by smaller values for minsupp – counting can get fairly expensive. At the



(a) T10.I4.D100K (b) T20.I4.D100K

(c) T20.I6.D100K (d) Maximal Size of Itemsets

Fig. 2. Execution Times on Synthetic Data

same time, the size of the candidates has no influence when using intersections. No matter how long the candidates are only the sizes of their tid-sets count. Of course, there is also a drawback: The costs for an intersection are at least $\mathsf{min}\{|X.\mathsf{tids}|, |Y.\mathsf{tids}|\}$ operations regardless of the actual number of occurrences of the candidate $X \bigcup Y$.[1]

As to be expected Partition[2] and Eclat show very similar runtimes, with Eclat beating Partition by a fairly constant factor. The reason is that Partition does not employ "fast intersections". The effect of the additional candidate pruning employed by Partition is not able to compensate this disadvantage. In fact,

[1] "Fast intersections" reduce the costs but are also not directly bound by the number of occurrences of a candidate.

[2] We were always able to skip the partitioning step, because our machine is equipped with sufficient main memory.

when enhancing Partition with "fast intersections" we experienced that both algorithms reach about the same runtimes.

As a surprise the behavior of Apriori is also very similar to that of the tids-intersecting algorithms. On first sight this seems to contradict to the results in [9] where Partition is shown to outperform Apriori. Actually, in [9] at minsupp $\approx$ 0.75% the runtimes of Apriori start to grow fundamentally whereas the behavior of Partition does not change. Our Apriori implementation, that uses an optimized structure to count candidate 2-itemsets does not show this behavior.[3] Partition clearly outperforms Apriori only on "T20.I6.D100K". This is due to the higher average size of candidate itemsets found in this dataset. Higher average sizes are also caused by lowering minsupp. Due to this the runtime of Apriori compared to Partition suffers at very low support thresholds.

The above holds also for Eclat that in addition profits from the "fast intersections". But most of the time Apriori still outperforms Eclat. This seems to contradict the experiences in [11], but in [11] only itemsets of size ≥ 3 are generated. As justified before, for our experiments we modified Eclat to mine also frequent 1- and 2-itemsets.

We left out DIC in the charts, because our very first experiments were quite discouraging. Even DIC that passes all transactions before generating candidates, that is DIC that "should be" Apriori, performed badly. We finally realized that replacing the hashtree with the structure from [4] has two draw backs: First, considering only the frequent items when setting up the hashtables in nodes of the hashtree is no longer possible. Second, in contrast to the hashtree used in [2] a prefix tree does not group itemsets sharing a common prefix in its leaves but each itemset and each of its subsets is represented by a node of its own. Both properties lead to a tremendous growth of memory usage. In addition, when counting candidates with the modified hashtree each of the already counted frequent itemsets causes overhead no matter whether it yields to a candidate or not. In contrast Apriori keeps the candidates separated. Actually even when overcoming both, DIC only showed an improvement of less than $\approx 30\%$ over Apriori but no fundamental different behavior on basket data in [4].

The surprisingly similar behavior of the considered algorithms shows that the advantages and disadvantages identified in the beginning of this subsection balance out on market basket-like data. This is also supported by Figure 2(d).

4 New Approach

4.1 Hybrid Approach

The performance studies and explanations of the results in the previous section suggest the development of a hybrid approach. The idea is to count occurrences whenever determining the support values of relatively small candidates and to rely on tid-set intersections for the remaining candidates. Of course this implies

[3] Replacing the hashtree for candidates of size 2 with an array is suggested in [10]

additional costs for generating the tid-sets when switching between the two counting strategies. For this purpose we use a hashtree-like structure that contains pointers to tid-sets instead of counters.

On the one hand with BFS the hybrid algorithm would suffer from memory problems when using tid-sets intersections. At least a costly mechanism like partitioning the database would be needed. On the other hand using DFS would elegantly solve this problem but when starting to count occurrences the runtime of our algorithm would suffer substantially, c.f. Section 3.1. The solution is to switch from BFS to DFS when switching from counting occurrences to intersections.

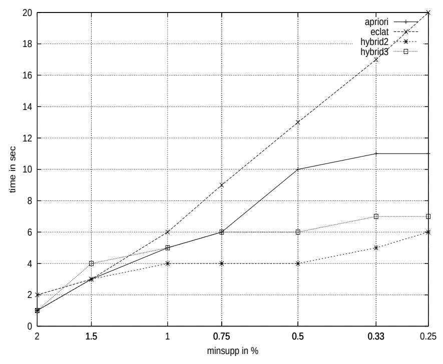
```
algorithm hybrid(transactions, sw, minsupp)
{
    frequent_itemsets[1] = get_frequent_items(transactions, minsupp);
    // BFS together with counting occurrences:
    for($s = 2; s \leq$ sw;$++s$)
    {
        candidates = generate_candidates(frequent_itemsets[$s - 1$]);
        count_candidates(candidates, transactions);
        frequent_itemsets[$s$] = get_frequent_itemsets(candidates, minsupp);
    }
    // DFS together with tid-set intersections:
    for each class $\in$ frequent_itemsets[s-1] do
    {
        class_with_tid-sets = generate_tid-sets(class, transactions);
        depth-first_search(class_with_tid-sets, minsupp);
    }
}
```

Fig. 3. Hybrid Algorithm

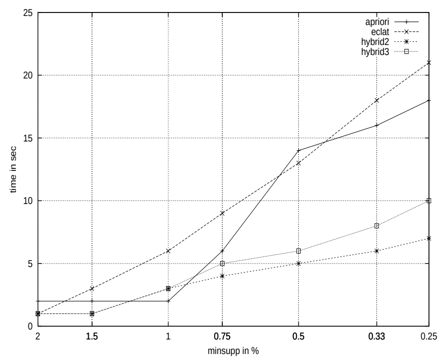
The finally resulting algorithm **Hybrid** is sketched in Figure 3. The argument sw determines when to change the counting strategy. As explained basic DFS does not allow candidate pruning by infrequent subsets. To overcome this we employ right-most DFS from [6].

4.2 Evaluation

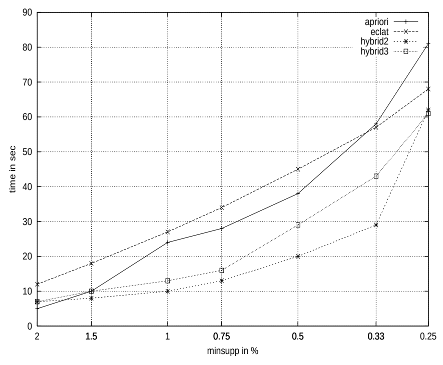
We repeated our experiments with two versions of our hybrid algorithm. One switching at candidate size 2 and the other at size 3, c.f. Figure 4. Moreover we made experiments on real-world data from a supermarket with about 60,000 items in roughly 70,000 transactions. The new algorithm performs best in nearly all cases and shows the anticipated behavior. This is most obvious in Figure 4(e) where Hybrid shows the smooth rise for the 2-, 3- and to some extend 4-itemsets of Apriori combined with the ability of Eclat to mine frequentent itemsets of size ≥ 4 at hardly any additional effort. The generation of the tid-sets that takes place at sw is efficiently solved by the modified hashtree. Nevertheless a very small average size of frequent itemsets let the algorithm suffer.



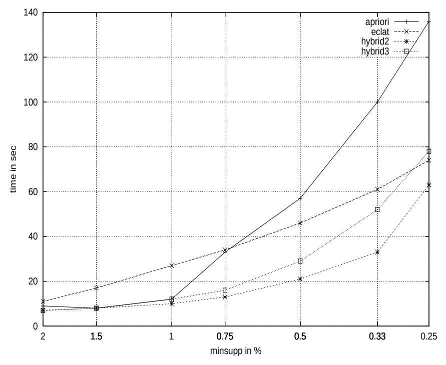
(a) T10.I2.D100K



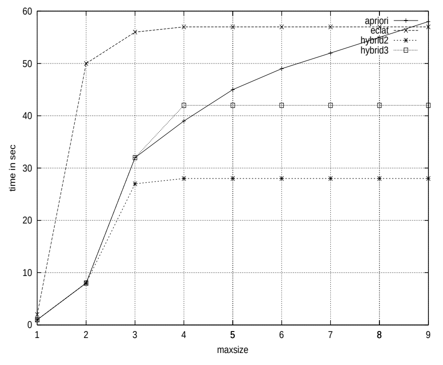
(b) T10.I4.D100K



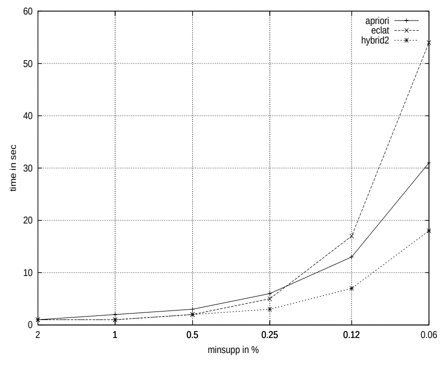
(c) T20.I4.D100K



(d) T20.I6.D100K



(e) Maximal Size of Itemsets



(f) Supermarket

Fig. 4. Execution Times on Synthetic Data and Real-World Data

5 Summary

In this paper we addressed the "classic" association rule problem, i.e. the generation of *all* association rules that exist in a given set of transactions with regard to minsupp and minconf. We identified the fundamental strategies of association rule mining and derived a general framework that is independent of any particular algorithm. Based on this we analyzed the performance of todays approaches both theoretically and by carrying out experiments. The results were quite surprising. In addition, our insights lead to the development of a new approach. The resulting algorithm Hybrid exploits the strengths of the known approaches and at the same time avoids their weaknesses. It turns out that in general for the "classic" association rule problem our algorithm achieves remarkably better runtimes than the previous approaches.

Our future research we will focus on the following: We want to explore how the approaches behave when mining databases that fundamentally differ from retail databases, e.g. dense datasets [3]. In addition, the aspects of memory usage are still not exhaustively studied. In fact memory usage is closely related to runtime because all the described algorithms suffer substantially when physical memory is exhausted and parts of the memory are paged out to disk.

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 '93*, 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)*, Chile, June 1994.
3. 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.
4. 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 '97*, 1997.
5. CRISP. The CRISP-DM process model. `http://www.crisp-dm.org`.
6. 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 Symp. on Principles of Data Mining and Knowledge Discovery (PKDD '98)*, France, Sept. 1998.
7. M. Holsheimer, M. Kersten, H. Mannila, and H. Toivonen. A perspective on databases and data mining. In *Proc. of the 1st Int'l Conf. on Knowlegde Discovery and Data Mining (KDD '95)*, Montreal, Canada, August 1995.
8. IBM. QUEST Data Mining Project. `http://www.almaden.ibm.com/cs/quest`.
9. 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)*, Switzerland, September 1995.
10. R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. of the 21st Conf. on Very Large Databases (VLDB '95)*, Switzerland, September 1995.
11. 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.