A Perspective on Databases and Data Mining

Marcel Holsheimer Martin Kersten Heikki Mannila Hannu Toivonen

CWI
Database Research group
P.O.Box 94079
NL-1090 GB Amsterdam
The Netherlands
{marcel,mk}@cwi.nl

University of Helsinki
Department of Computer Science
P.O.Box 26
FIN-00014 University of Helsinki
Finland
{Heikki.Mannila,Hannu.Toivonen}@cs.Helsinki.FI

Abstract

We discuss the use of database methods for data mining. Recently impressive results have been achieved for some data mining problems using highly specialized and clever data structures. We study how well one can manage by using general purpose database management systems.

We illustrate our ideas by investigating the use of a dbms for a well-researched area: the discovery of association rules. We present a simple algorithm, consisting of only union and intersection operations, and show that it achieves quite good performance on an efficient dbms.

Our method can incorporate inheritance hierarchies to the association rule algorithm easily. We also present a technique that effectively reduces the number of database operations when searching large search spaces that contain only few interesting items.

Our work shows that database techniques are promising for data mining: general architectures can achieve reasonable results.

Introduction

Data mining is an area in the intersection of machine learning, statistics, and databases. How similar or different data mining is from machine learning and statistics is an interesting question. As to databases, there has been some discussion on the importance of database methods in data mining: are they useful at all, or is data mining just machine learning for larger sets of examples?

In this paper we address this question by looking at a well-researched and prototypical problem in data mining, the discovery of association rules. Association rules are a simple form of knowledge that can be used to express relationships between attributes in binary data. In recent years, several efficient algorithms have been developed for finding association rules, and there are also some theoretical results in this area (Agrawal *et al.* 1995; Agrawal & Srikant 1994; Mannila, Toivonen, & Verkamo 1994). The algorithms are specialized, and use clever data structures to speed up the search.

We study how one can efficiently find such rules using only a general-purpose database management system and the operations of relational algebra that it supports. Our goal is to see how well simple and general methods compare with other, specialized, techniques.

We show that a simple algorithm using an efficient relational dbms can achieve quite good performance on the problem of finding association rules. The algorithm uses only union and intersection operations, and constructs new relations. Additionally, the method can incorporate inheritance hierarchies to the association rule framework quite easily.

We also present a relational technique that can be used to efficiently prune large search spaces with only few interesting items.

Our work shows that the potential of general dbms techniques is high for data mining applications; general architectures can compete with specialized methods.

In more detail, the paper is organized as follows. Association rules and a general algorithm for their discovery are discussed in Section 2. Section 3 describes our implementation of this algorithm, where the data is stored in a general purpose database. As we will see, the search space can be very large, so in Section 4, we outline a technique to assemble global information on this space. Experiments in Section 5 show that this technique can reduce execution time by 50% and the number of database operations by up to 90%. Section 6 is a short conclusion.

Association rules

Association rules are a class of regularities in binary databases (Agrawal, Imielinski, & Swami 1993). An association rule is an expression $X \Rightarrow Y$, where X and Y are sets of attributes, meaning that in the rows of the database where the attributes in X have value true, also the attributes in Y tend to have value true.

Application areas are numerous. We have applied association rules e.g. in telecommunications alarm correlation, university course enrollment analysis, and discovery of product sets often ordered together from a manufacturer. A prototypical application area –

also the domain of our examples – is customer behavior analysis in retailing, the so-called *basket analysis*: which items do customers often buy together in a supermarket?

Such data can be viewed as a relation with binary attributes: each transaction is a row in the database, and contains 1's in the attributes corresponding to the items bought in this transaction. Retailers are interested in which items are often bought together, the so-called *itemsets.* Given an itemset X, the *support* $s(X)$ of X is the number of transactions that contain all items in X[1]. Given a *support threshold* σ, we say that an itemset X is *large* if $s(X) \geq \sigma$. The support threshold σ is specified by the user, as the minimum fraction of the database that is still interesting. The *confidence* of an association rule $X \Rightarrow Y$ is $\frac{s(Y)}{s(XY)}$, i.e., the probability that a transaction with items X also contains items Y. An itemset consisting of s items is called an *s-itemset.*

All association rules $X \Rightarrow Y$ with $s(XY) \geq \sigma$ can be found in two phases (Agrawal, Imielinski, & Swami 1993). In the first, expensive phase the database is searched for all large itemsets, i.e., sets of items that occur together at least in σ transactions in the database. In the second – and easy – phase, association rules are generated from these large itemsets. In this paper, we focus on the first phase: the discovery of large itemsets. Details on the construction of association rules can be found in (Agrawal, Imielinski, & Swami 1993).

Most algorithms for the discovery of large itemsets work as follows (Agrawal *et al.* 1995; Agrawal & Srikant 1994; Mannila, Toivonen, & Verkamo 1994). First, the supports for single items are computed and large 1-itemsets are found. Then, iteratively for sizes $s = 2, 3, \ldots$, *candidate* s-itemsets are generated from the large $(s-1)$-itemsets of the previous pass. Supports for the candidates are then computed from the database, and those candidates that turned out to be large are used in the next pass to generate candidates of size $s+1$.

The specification of candidate itemsets is based on the observation that for a large s-itemset, all its $(s-1)$-subsets are large; accordingly, for sizes $s > 1$, candidate itemsets are those s-itemsets whose all $(s-1)$-itemsets are large. This simple condition effectively *prunes* the potentially large search space.

Hierarchies

In retailing, much domain-knowledge is available in the form of hierarchies: items belong to categories of a generalization hierarchy. For example, *Budweiser* and *Heineken* are both *beer*; *beer*, *lemonade*, and *juice* are *beverages*, etc. Rules expressed in terms of such general categories provide very useful high-level information. Also, generalization may be necessary for having supports larger then the support threshold: the combination of Heineken and chips may not be large, put the more general 'beer and chips' probably is.

The items of a category need not be disjunct. I.e., a customer can buy both Heineken and Budweiser. Accordingly, to compute the support for beer, we have to take the union of the rows with Heineken and the rows with Budweiser, rather than simply add the supports for Heineken and Budweiser.

Algorithms for discovering large sets do not directly support item hierarchies. Hierarchies can, of course, be accounted for by generating derived attributes, but then effort is wasted on the discovery of redundant large sets. We will show in Section 4 how item hierarchies can be supported architecturally.

Database support

The expensive activity in the above described association rule algorithm is in computing the supports for itemsets, i.e., operations on the data. We now describe the use of the general purpose database system Monet (Kersten August 1991; van den Berg & Kersten 1994) for that task. Monet offers the necessary storage structures and operations, and takes care of optimizing the database activity.

Data representation

The database is stored as a *decomposed storage structure* (Khoshafian *et al.* 1987). Normally one would store the data as a set of transactions (rows), and for each transaction enumerate the items that are members of this transaction. In a decomposed storage structure, each transaction has a unique transaction identifier (TID), and the database is stored as a set of items (columns), where for each item the TIDs of the transactions that contain this item are enumerated. For example, a database with 100,000 transactions, each containing on average 10 items out of a choice of 500, is stored as a set of 500 columns, where each column contains on average 2000 TIDs.

Operations

The advantage of a decomposed storage structure is that each candidate (itemset) in the search space has its counterpart in the database, such that its support can be computed by a few simple database operations, rather than a full scan over the database. The support of an 1-itemset A is simply the size of column A in the database. So in pass 1 of the large set discovery algorithm, we only have to select 1-itemsets whose columns have size above the support threshold σ.

The support of a 2-itemset AB is the number of transactions that contain both items A and B. Since we stored the TIDs for the transactions for A and B in separate database columns, we need to know how many TIDs appear in both A and B. So we compute

[1]We use a notion of support slightly different from (Agrawal, Imielinski, & Swami 1993), where $s(X)$ is defined as the *fraction* of the database that contains X.

the *intersection* A ∩ B, using the Monet intersect command:

```
AB=intersect(A,B);
```

The result of this intersection is a new column AB that contains the TIDs that are in both A and B[2]. This column is stored in the database system or destroyed upon user demand. The size of this column is the support for the 2-itemset AB. If all AB, BC and AC are large, then in the third pass the support for the 3-itemset ABC must be computed. Since intersection is a binary operation, we can first take the intersection of A and B, and intersect the result with column C, as in

```
ABC=intersect(intersect(A,B),C);
```

The intersection AB has already been computed in the previous pass, and the result can be reused:

```
ABC=intersect(AB,C);
```

By retaining all columns for large itemsets of the previous pass, we can reduce the number of intersections in each pass to exactly one intersection per candidate itemset. A further optimization can be achieved by rewriting the intersection to take only results of the previous pass as arguments. That is

```
ABC=intersect(AB,AC);
```

These intersections will be faster, because the size of their arguments decreases. Moreover, there is no need to access the columns A, B, and C from the original database anymore. Hence these columns can be removed from memory, thereby decreasing memory requirements. By reusing results we actually manipulate the database itself, such that it always reflects the information need of the association algorithm.

Optimization: A bird's eye view of the search space

Although the methods described above are efficient, the problem is that especially in the second pass many candidates are generated, but only very few prove to be large. As an example take the database that we will present in Section 5: in the first pass 600 out of 1000 1-itemsets are large. In the second pass, these large sets generate $600^2/2 = 180{,}000$ candidates, of which only 44 (!) are actually large. To find these, we need 180,000 intersections; they consume over 99% of total processing time.

Because of the sparseness of the databases one can reduce the number of database operations by exploring the candidate space using a coarser granularity. That is, we assemble aggregate information on sets of candidates, rather than on single candidates. This information allows us to infer that the candidate collection under investigation either does not contain any large itemsets, or that the collection *might* contain large itemsets. The first case allows us to discard the whole candidate collection; in the latter case we have to do some computation on this collection that has be done in the naive method. The extra investment consists of assembling global information, and zooming in on suspect subsets. However, this extra investment pays if a small fraction of the candidates is actually large.

Aggregate information

The idea of assembling aggregate information is simple. Assume that $A_1, A_2, \ldots, A_n$ are large 1-itemsets. In pass 2, the naive method would compute the $(n(n-1)/2)$ intersections A_1A_2, $A_1A_3, \ldots,$ $A_{n-1}A_n$. If the size of intersection A_1A_2 is larger than the support threshold σ, A_1A_2 is a large set. The union $A_1 \cup A_2$ contains all TIDs of transactions that are either in A_1 or A_2. If we take the *aggregate* intersection

$$(A_1 \cup A_2) \ \cap \ (A_3 \cup A_4)$$

and this intersection is *small* (i.e., not large), then this allows us to infer that none of A_1A_3, A_1A_4, A_2A_3, A_2A_4 is large. Correctness is easily verified: if for example A_1A_3 is large, then there are at least σ TIDs that are both in A_1 and A_3. These TIDs are also in the unions $(A_1 \cup A_2)$ and $(A_3 \cup A_4)$, so their intersection has to be large as well.

If the aggregate intersection is large, we have to compute all intersections A_1A_3, A_1A_4, A_2A_3, A_2A_4 to determine which of these are large. If none of these intersections is large, we did some superfluous work and the aggregate is said to be a *false alarm.*

By computing the union $A_1 \cup A_2$ we also know the size of intersection A_1A_2, since this is the sum of the sizes of both operands minus the size of their union. So by taking the union, we can determine whether A_1A_2 is a large itemset.

Taking the aggregate intersection costs three operations. If the result is small, no further computation is needed as we established that none of the 6 candidates are large. If the result is large, we have to compute 4 additional intersections. So we either win 3 operations, or lose 1, compared to the naive approach, where all 6 intersections are needed.

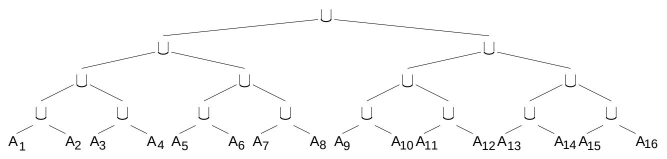
So in this approach we split the set $A_1, A_2, \ldots, A_n$ into $n/2$ pairs, compute the $n/2$ unions and the $n^2/8$ intersections between the pairs. At best, i.e., when all aggregate intersections are small, this saves about 3/4 of the operations. So for our example, we reduced the number of database operations from $180,000$ to $45,000$. The worst case is that all aggregate intersections are large so all $n^2/2$ intersections have to be made as well. In total this would be 1/4 more operations than in the naive approach.

If we take the aggregate union instead of the intersection, we can *reuse* the resulting column (i.e., the union of A_1, A_2, A_3 and A_4) and compute the aggregate intersection with another union:

[2]With AB we denote both the 2-itemset $\{A, B\}$ and the result of the Monet intersection $A \cap B$.

$$[(A_1 \cup A_2) \cup (A_3 \cup A_4)] \;\cap\; [(A_5 \cup A_6) \cup (A_7 \cup A_8)]$$

If this aggregate intersection is small, each of the 16 combinations $A_1A_5, A_1A_6, \ldots, A_4A_8$ is small as well. By again taking the union instead of the intersection, we can reuse this result to compute the intersection of the union of $A_1, \ldots, A_8$ and $A_9, \ldots, A_{16}$. If this intersection is small, we can rule out another 64 candidates. So finally we construct the following tree:

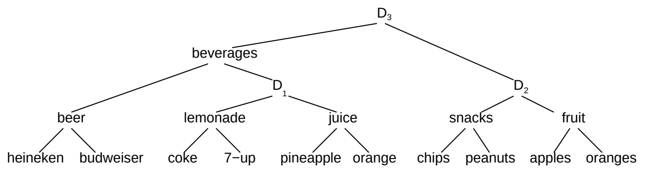


Each node D_i in this tree is a newly generated column in the database, formed by the union of its children. During tree construction, we compute the size of the intersection for each node, using the size of the union and the sizes of its children. When the size of an intersection D_iD_j exceeds the support threshold σ, then D_iD_j possibly contains large 2-itemsets, and is called an *alarm*.

Since the size of the unions in this tree increases, and hence also the probability of false alarms, it is not useful to compute the tree up to the highest level. It may be better to *cut-off* the tree-construction at a particular level, and compute all remaining $n'(n'-1)/2$ for the n' nodes at this level.

We wish to compute the level in which false alarms start to dominate. For brevity, we present the results only in an extremely simple model. Assume the support of all large 1-itemsets is 2σ, twice the support threshold, and that occurrences of such itemsets are independent. Thus there are no large 2-itemsets in this model. Then the expected size of the set D_i at level k is approximately $2^{k+1}\sigma$, and for the expected support E of the intersection $D_i \cap D_{i+1}$ we have $E \approx 2^{k+1}\sigma 2^{k+1}\sigma = 2^{2k+2}\sigma^2$. This is greater than or equal to σ in the case $k \geq \frac{1}{2}\log(1/\sigma) - 1$, for example for $\sigma = 0.001$ for about $k \geq 4$. Thus in this model from about the fourth level upwards the false alarms become quite frequent.

One may observe that internal nodes in this tree correspond to higher-level concepts, e.g., 'beer *or* wine'. If we construct the tree such that it contains the generalization hierarchies, we can label some of the internal nodes with category names. Once we have computed the tree, we also know the support for these categories. Hierarchies need not be binary trees, so we may have to include intermediate nodes, e.g., D_1 in the figure below.



Solving Alarms

If D_iD_j at level l is an alarm, then the intersection of D_i and D_j is large. These columns are unions of nodes at level $l-1$, respectively, e.g., $D_1 \cup D_2$ and $D_3 \cup D_4$, so we have to check the four remaining intersections of these children, i.e., $D_1D_3, D_1D_4, D_2D_3, D_2D_4$. If one or more of these intersections is large, then we must find out which of the children in level $l-2$ caused this intersection to be large, i.e., recursively repeat the above activities.

We work our way down the tree and when we finally find a large intersection where both arguments are either items (leaves) or categories, we have located a large 2-itemset. When one of the arguments is a category (as in 'beer and chips'), we continue with its children ('Heineken and chips', 'Budweiser and chips'). If, on the other hand, at level $l-k$ no large intersections can be found, then the alarm was *false*, and dissolved at level $l-k$.

In the following, we give the algorithm for solving alarms in pseudo-code. As input it takes the two nodes D_i and D_j whose intersection is large. The output consists of the discovered large itemsets; if the alarm was false, then the algorithm returns an empty set. With I, we denote the set of all items and category names.

```
**procedure** solve-alarm($D_i, D_j$)
  **if** $D_i \in I, D_j \in I$ **then** Large := $\{D_iD_j\}$
                          **else** Large := $\emptyset$
  **if** $D_i \in I$ **then** Next := $D_i$ × children($D_j$)              (1)
  **if** $D_j \in I$ **then** Next := Next ∪ children($D_i$) × $D_j$     (2)
  **if** $D_i \notin I, D_j \notin I$ **then**                                   (3)
        Next := children($D_i$) × children($D_j$)
  **forall** $D_i'D_j'$ ∈ Next **do**
        compute-intersection($D_i', D_j'$)
        **if** intersection is large **then**
              Large := Large ∪ solve-alarm($D_i', D_j'$)
**return** Large
```

When D_i is a leaf, the set children(D_i) is empty. The set $A \times B$ denotes the Cartesian product of sets A and B, i.e., $\{ab \mid a \in A, b \in B\}$.

Example 1 Assume that during the construction of the tree in the above figure, we discover that beverages-D_2 is an alarm. Since beverages is a category, we apply rule 1 of the algorithm and compute the two intersections beverages-snacks and beverages-fruit.

Beverages-snacks is a large 2-itemset. Both beverages and snacks are categories, so we apply rules 1 and 2, and compute the intersections beverages-chips, beverages-peanuts, beer-snacks and D_1-snacks, of whom the first three are large. Next, we solve beverages-chips and discover that beer-chips is large and D_1-chips is small. All combinations in beer-chips (Heineken-chips and Budweiser-chips) are small, just as the combinations in beverages-peanuts (beer-peanuts and D_1-peanuts).

So finally we discovered the large sets: beverages-snacks, beverages-chips, beverages-peanuts, beer-snacks and beer-chips. Likewise, the alarm beverages-fruit is solved, discovering that also beverages-apples, juice-fruit and juice-apples are large 2-itemsets. ▮

Experimental results

To verify our theoretical results, and assess the relative reduction of database operations, we implemented our algorithm on top of the Monet database server (Kersten August 1991; van den Berg & Kersten 1994). Monet uses a vertically partitioned database model, which is very well suited for a decomposed storage structure. It supports SQL and ODMG interfaces, and is used for another data mining tool, Data Surveyor (Holsheimer, Kersten, & Siebes forthcoming; Holsheimer & Kersten 1994).

Although Monet can execute operations in parallel, we ran our experiments in sequential mode on an SGI Challenge with 150 Mhz processors and 256 Mbytes of memory (performance results on parallel database mining can be found in (Holsheimer, Kersten, & Siebes forthcoming)). As a test-database, we used the T10.I4.D100K and the T5.I2.D100K databases, used in (Agrawal *et al.* 1995; Agrawal & Srikant 1994). These databases contain 100,000 transactions and the average number of items per transaction is 10 and 5 respectively.

Number of database operations

In the first test, we measured the number of database operations for different databases, support levels and cutoff-levels. Figure 1 depicts the number of database operations (unions and intersections) as a function of the cut-off level. A cut-off level of 1 corresponds to the naive approach, where all $n(n-1)/2$ intersections are computed. These test-results show that our technique effectively reduces the number of database queries with up to 90% if we construct at least three levels of the tree.

Elementary database operations

The previous experiment suggests that performance is stable for cut-off level ≥ 3. However, database activity, and hence the execution time, is not only determined by the number of database operations, but also by the size of the database relations.

In the following experiment, we assess the influence of the cut-off level on the database activity. To obtain implementation and machine independent results, the amount of activity is measured as the number of elementary operations, i.e., comparisons between database objects (TIDs) in the union and intersect operations.

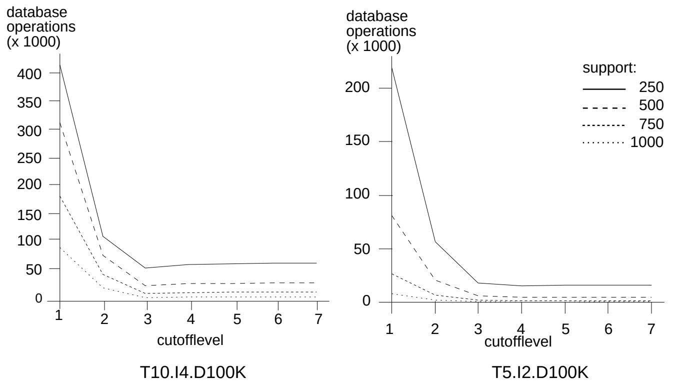


Figure 1: Number of database operations

The results for the T10.I4.D100K database in Figure 2 show that the cost of tree construction (a) is linear in the height of the tree: although the number of nodes halves at each level, the average size of each node doubles, since it is nearly the sum of the size of its children. The costs of computing intersections (b) decreases, since fewer intersections have to be computed, but their arguments grow in size. For higher cut-off levels, the costs for solving alarms (c) grow very fast, because more false alarms are encountered. Alarms in the higher levels in the tree are also more expensive to solve, since arguments for the intersections are larger.

The costs of solving alarms start to dominate from level 3 onwards. So we may expect that for this database an optimal performance is achieved by cutting the tree construction at level 3. This also matches our theoretical analysis in Section 4, that suggested that false alarms dominate from level 4 on. Figure 3 shows that our assumption is correct, the total execution time for both the T10.I4.D100K and the T5.I2.D100K databases is minimal at cut-off level 3.

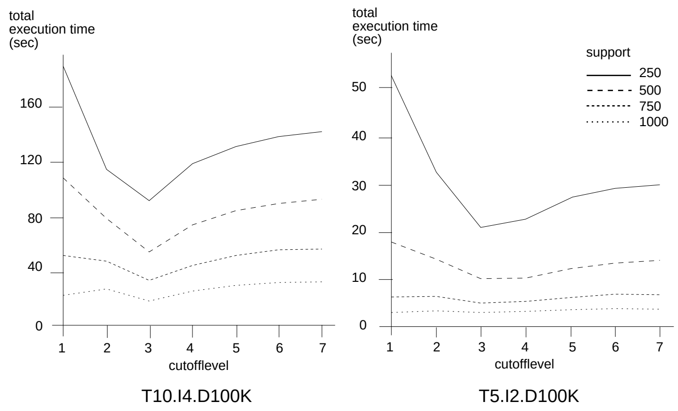


Figure 3: Total execution time.

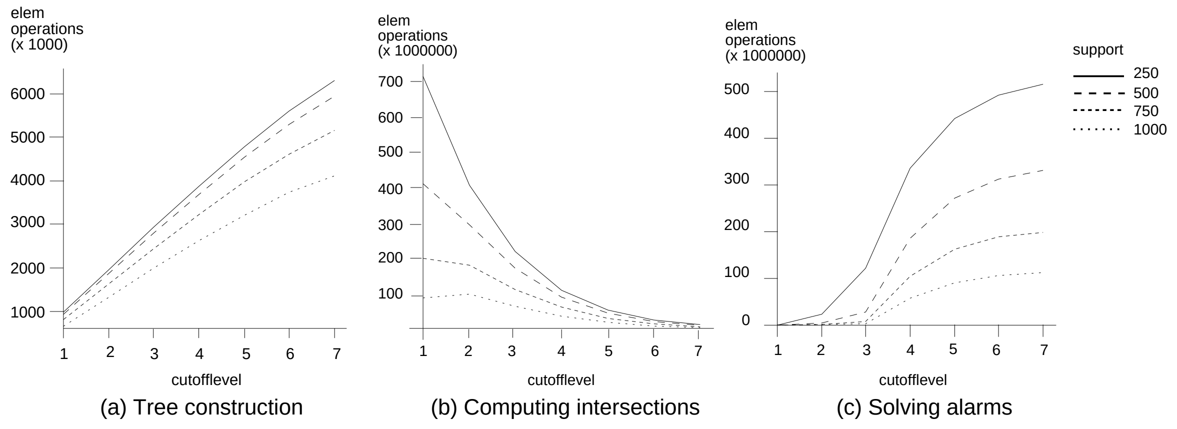


Figure 2: Elementary operations for different phases.

Conclusions

We have considered finding association rules by using a general-purpose database management system. The resulting algorithm is extremely easy to implement and reasonably fast: while it does not compete with the fastest methods, it is quite usable on all but the largest data sets and the smallest support thresholds.

Our results support the notion that dbms techniques can be used profitably in building data mining tools (Holsheimer *et al.* 1995). We are currently investigating how this approach works on other topics, e.g., for finding integrity constraints on databases (Mannila & Räihä 1994).

While our goal was not to develop a yet faster association rule finding method, the approach described above gives some possibilities even for that. For example, if the construction of the tree in Section 4 succeeds in an optimal way, there will be very few alarms. While an optimal construction is difficult, one can approximate it quite well either by looking at the supports of the large 1-itemsets, or by taking a sample, finding the large 2-itemsets from it and using that information to build the tree. Moreover, parallel database techniques (Holsheimer, Kersten, & Siebes forthcoming) can be exploited to even further speed up search.

References

Agrawal, R., and Srikant, R. 1994. Fast algorithms for mining association rules in large databases. In *VLDB '94*.

Agrawal, R.; Mannila, H.; Srikant, R.; Toivonen, H.; and Verkamo, A. I. 1995. Fast discovery of association rules. In Fayyad, U. M.; Piatetsky-Shapiro, G.; Smyth, P.; and Uthurusamy, R., eds., *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press. To appear.

Agrawal, R.; Imielinski, T.; and Swami, A. 1993. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 International Conference on Management of Data (SIGMOD 93)*, 207 – 216.

Fayyad, U. M., and Uthurusamy, R., eds. 1994. *AAAI-94 Workshop Knowledge Discovery in Databases.*

Holsheimer, M., and Kersten, M. L. 1994. Architectural support for data mining. In Fayyad and Uthurusamy (1994), 217 – 228.

Holsheimer, M.; Klösgen, W.; Mannila, H.; and Siebes, A. 1995. A data mining architecture. In preparation.

Holsheimer, M.; Kersten, M.; and Siebes, A. forthcoming. Data Surveyor: Searching the nuggets in parallel. In Fayyad, U. M.; Piatetsky-Shapiro, G.; Smyth, P.; and Uthurusamy, R., eds., *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press.

Kersten, M. L. August 1991. Goblin: A DBPL designed for Advanced Database Applications. In *2nd Int. Conf. on Database and Expert Systems Applications, DEXA'91.*

Khoshafian, S.; Copeland, G.; Jadodits, T.; Boral, H.; and Valduriez, P. 1987. A query processing strategy for the decomposed storage model. In *Proc. IEEE Data Engineering Conf*, 636–643.

Mannila, H., and Räihä, K.-J. 1994. Algorithms for inferring functional dependencies. *Data & Knowledge Engineering* 12(1):83–99.

Mannila, H.; Toivonen, H.; and Verkamo, A. I. 1994. Efficient algorithms for discovering association rules. In Fayyad and Uthurusamy (1994), 181 – 192.

van den Berg, C. A., and Kersten, M. L. 1994. An analysis of a dynamic query optimisation scheme for different data distributions. In Freytag, J.; Maier, D.; and Vossen, G., eds., *Advances in Query Processing.* Morgan-Kaufmann. 449 – 470.