Architectural support for data mining

Marcel Holsheimer Martin L. Kersten
{marcel,mk}@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

One of the main obstacles in applying data mining techniques to large, real-world databases is the lack of efficient data management. In this paper, we present the design and implementation of an effective two-level architecture for a data mining environment. It consists of a mining tool and a parallel DBMS server. The mining tool organizes and controls the search process, while the DBMS provides optimal response times for the few query types being used by the tool. Key elements of our architecture are its use of fast and simple database operations, its re-use of results obtained by previous queries, its maximal use of main-memory to keep the database hot-set resident, and its parallel computation of queries.

Apart from a clear separation of responsibilities, we show that this architecture leads to competitive performance on large data sets. Moreover, this architecture provides a flexible experimentation platform for further studies in optimization of repetitive database queries and quality driven rule discovery schemes.

CR Subject Classification (1991): Data storage representations (E.2), Database systems (H.2.4) *parallel systems, query processing*, Information search and retrieval (H.3.3), Learning (I.2.6) *induction, knowledge acquisition*

1. Introduction

Recent years have shown an increased interest in data mining techniques, especially in their application to real-world databases. These databases tend to be very large (> 100K objects) and contain objects with many attributes. These vast amounts of data hide many interesting relationships, but their discovery is obstructed by two problems.

First of all, the number of potential relationships in a database is very large. Efficient search algorithms (e.g. Quinlan's ID3 algorithm [10]) are needed to discover the most interesting ones. However, simply applying these algorithms to large databases causes another optimization problem, because candidate relationships have to be validated against the database. For many techniques, this results in a repetitive, expensive scan of the entire database.

A straight-forward solution would be to reduce the amount of data by taking a sample from the database. In such a sample, small differences among sets of objects in the database will become less clear, and interesting relationships can easily be overlooked. So one would prefer to mine on the entire database, or at least on a very large sample.

In this paper, we outline a two-level architecture that enables mining on large databases. The front-end consists of a data mining tool that provides a graphical user interface to formulate and direct the mining activities. All data handling is performed by the back-end, the Monet database server, a parallel DBMS developed within our group to support a variety of

advanced applications [13]. The focal point of this paper is on the gross architecture and its performance. Details on the data mining algorithms and the internals of the DBMS are given elsewhere.

Although the idea of using a DBMS is not new, e.g. systems such as SKICAT use a relational DBMS [4], our system offers some novel optimization features like the parallel computation of column oriented operations, the reuse of results of previous queries and fast data access by keeping only a limited *hot-set* of data in main memory. This hot-set is changed dynamically during the search process to reflect the information requirements of the datamine tool.

These optimizations are equally applicable to a range of datamine systems. Moreover, other modern extendible DBMSs (e.g. Postgress [12], Starburst [9], EXODUS [3]) support the inclusion of specialized routines, and can thus be used as a back-end.

The paper is organized as follows. Section 2 gives an introduction to the classification problem. Section 3 illustrates a general search strategy for this problem. Section 4 contains a brief introduction to the Monet DBMS and derives the interface requirements. Section 5 gives the performance figures and their analysis for this architecture. We conclude with a short indication of future and related research issues addressed within our group.

2. Mining for classification rules

The input to our data mining system consists of a relational table. This table is both broad and large, i.e. many attributes and several hundred thousand objects. In particular, we are currently mining on a real-life table of 180 K objects with 90 attributes obtained from an insurance company.

In this database, the system has to discover rules that define whether an object belongs to a particular subset, called a *class*. This class is defined by the user, hence this form of learning is known as supervised learning [2].

In practice, the user partitions the database into two sets, a set P containing the positive examples of the class, and a set N containing the negative examples. The system searches for classification rules, i.e. rules that predict whether a particular object will be in P or N, based on values of its attributes.

The rules considered are of the form $\langle D \rangle \rightarrow \langle C \rangle$, where C is the class, and D is a condition formed of a conjunction of attribute-value conditions. These attribute-value conditions are either equality conditions, e.g. 'city = Denver' when the domain of the attribute contains categorical values, or inequalities, such as 'age < 27', in case of a numerical attribute. For example, the rule

$$R: \langle \text{city} = \text{Denver} \wedge \text{age} < 27 \rangle \rightarrow \langle \text{dangerous driver} \rangle$$

states that persons living in Denver and under 27 are dangerous drivers.

The condition D is in fact a database selection, which returns all satisfying objects when applied directly. The partitioning of the database into P and N implies that each condition D correspond to two database selections $\sigma_D(P)$ and $\sigma_D(N)$: the sets of all positive and all negative examples, covered by the condition.

A rule is correct with respect to the database when its condition covers all positive and none of the negative examples, that is, $\sigma_D(P) = P$ and $\sigma_D(N) = \emptyset$. In general only few

correct rules will be found in a database, because class membership is often determined by information not represented in the database. For example, whether a person is a dangerous driver is not uniquely determined by age and home-town. Hence, it makes sense to search for probabilistic rules. With each rule R, we associate a *quality* or strength $Q(R)$ to indicate how well it performs against the database. The quality is the ratio of the number of positive examples, covered by the rule, to the number of positive and negative examples, covered by the rule, i.e.

$$Q(R) = \frac{|\sigma_D(P)|}{|\sigma_D(P)| + |\sigma_D(N)|}$$

Hence a correct rule has quality 1, while quality equals 0 implies that the negation holds.

3. Search strategy

To discover probabilistic rules of a high quality[1], the system uses an iterative search strategy controlled by the quality metric. The initial rule R_0 is the rule with an empty condition

$$R_0 : \langle \text{true} \rangle \rightarrow \langle C \rangle$$

assigning all objects in the database to class C. The quality of this rule is simply the number of positive objects, divided by the size of the database, i.e. the probability that an arbitrary object in the database belongs to class P. During data mining, this rule is extended with an attribute-value condition to obtain a new rule, e.g.

$$R_1 : \langle \text{Age} < 27 \rangle \rightarrow \langle C \rangle$$

and this new rule, called an *extension* of R_0, is further extended with conjunctions. The heuristic to reach a satisfactory classification is to choose rule extensions with a perceived high quality. This algorithm is generally applicable and basically underlies many machine learning systems [5]. Hence, we expect that a wide variety of data mine systems may benefit from the optimizations that we discuss in the following section.

To select the extensions with the highest quality, we compute the quality of *all* possible rule extensions. The combinatorial explosion of candidate rules is controlled using a beam search algorithm where the best w (the beam width) extensions are selected for further exploration. Of these new rules, all possible extensions are computed, and again, the best w are selected. This process continues until no further improvement is possible, or until the length of the rules exceeds a user defined threshold (the search depth d).

To compute the quality of all extensions of a rule R_i, we only need to look at the cover of R_i and not at the entire database. This is caused by the 'zooming' behavior of our algorithm: the cover of an extension of R_i is always a subset of the cover of R_i. Hence, at every stage we compute the cover, and use it for the computation of the quality of the newly generated rules. This algorithm is described in pseudo-code as

[1] Actually, rules of a very low quality can be of interest as well, because these are rules for the opposite class, e.g. safe drivers. Our tool searches for these rules as well, but for reasons of simplicity, we ignore low quality rules in this paper.

```
Beamset := {initial rule R_0},
while improvement and depth < d
    All_extensions := ∅,
    For each R_i in Beamset do
        C_i := cover(R_i),
        Extensions := extend(R_i),
        compute_quality(Extensions, C_i),
        All_extensions := All_extensions ∪ Extensions,
    Beamset := best(All_extensions, w)
```

EXAMPLE 1 Consider a car insurance database storing information about clients, such as age, gender, home-town, and information about their cars, such as price and whether it is a lease car or not. The user defines a class 'dangerous driver' and assigns any client who has had a car accident to this class. The other clients serve as counter examples.

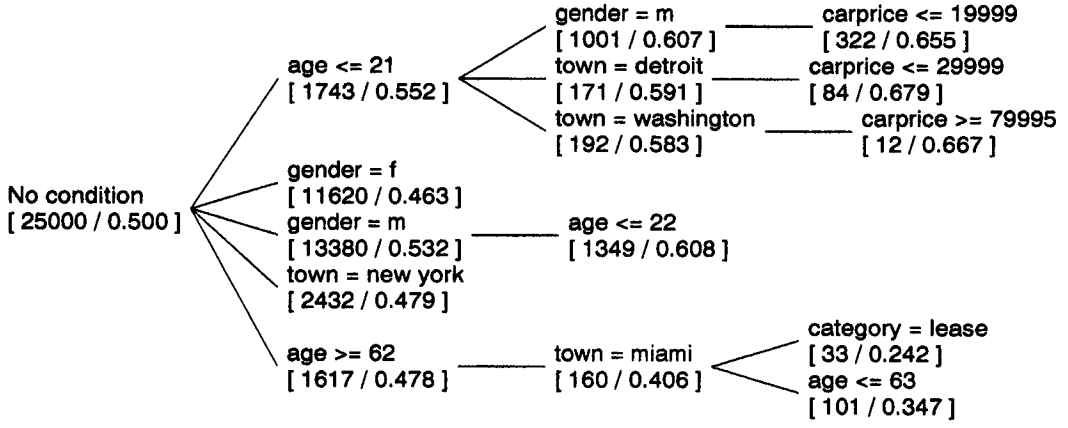


Figure 1: A 5 × 3 search tree.

A search tree, constructed using a 5 × 3 beam search is shown in Figure 1. In each node, the condition is stated together with the size of the cover and the rule quality. The initial condition covers 25K objects, half of which belongs to the class. For example, the rule with condition (age <= 21) ∧ (gender = m) covers 1001 objects and 60.7% belongs to class 'dangerous driver'. ∎

The main problem to be tackled is to compute efficiently the quality of all possible extensions. A straight-forward technique is to compute the quality of each extension separately. This involves issuing two database queries, one to compute the number of objects in the database selection $\sigma_D(P)$ and another to compute the size of $\sigma_D(N)$. Although intuitively appealing, this technique is far too expensive. In Example 1, the quality of 1000 extensions have been computed.

A more sophisticated technique, proposed by Agrawal *et al.* [1], is to incrementally compute the quality of a set of rules in a single pass over the database by updating counters for each extension. To avoid excessive memory consumption the rule-set under consideration is pruned along the way. This approach works under the assumption that the database scan sequence

does not influence the pruning of the classification rules.

We propose an alternative architecture, where the quality of all possible extensions of a rule is computed using only a few simple database operations. Moreover, our architecture heavily uses the zooming characteristic of the mining algorithm to reduce database activity. In particular, the answers to both $\sigma_D(P)$ and $\sigma_D(N)$ are cached and reused wherever possible.

4. Database support for the search algorithm

The DBMS used for our data mining tool is the *Monet* database server developed at CWI as an experimentation vehicle to support advanced applications (see [6, 13]). It is geared towards exploitation of large main-memories and shared store multi-processors. Amongst its novelties are a fully decomposed storage scheme and adaptive indexing to speed-up query processing. The system runs in parallel on a Silicon Graphics shared-memory multi-processor.

The data mining application sketched above requires database support in the following areas: efficient data storage and access, statistics gathering, computation of selections, and handling intermediate results. The support provided by Monet in each of these areas is shortly indicated.

4.1 Data storage and access

As argued in [13], it generally pays to decompose complex objects and relational tables into binary associations tables. Although such a storage scheme leads to a poor performance in record-oriented applications using disk-based databases, it is far more effective in situations where the database *hot-set* can be kept memory resident or where only a small fraction of the objects are returned to the application. This design feature has proved crucial in the efficient support of our data mining tool.

The database table is broken down into multiple Binary Association Tables (or BAT), where each database attribute is stored in a separate table[2]. The first column, the *head*, of each BAT contains a unique object identifier (oid), the second column, *tail*, contains the attribute values.

To illustrate, assume that the table has attributes $\{A, B, C, \ldots\}$, and is partitioned into two sets P and N. In Monet, each object is assigned a unique oid. The table is represented as a set of BATs $\{A, B, C, \ldots\}$, where each BAT stores the object's oid and its value for this attribute. The class information is represented in two BATs P_0 and N_0, whose heads contain the oid's of objects in P and N respectively. This storage scheme improves database access for our data mining tool, because access is largely column oriented. In particular, the hot attributes, i.e. those under consideration for extending the classification rules, are easily kept in main-memory.

Although this scheme would seem to double the disk space requirement, prudent implementation of the underlying software leads to an overhead per object which compares favorably with other modern DBMSs (e.g. Starburst). Likewise, the additional cpu overhead incurred in object reconstruction is marginal, because the system automatically creates and maintains search accelerators to speed-up its operations.

A similar vertical partitioning of the database is used by the Explora system [8], where

[2]In the following, we will identify an attribute with its associated BAT table, and use the same symbol for both. The meaning will be clear from the context.

only attributes that are relevant for the current mining task are loaded into main memory. The main difference is that in our architecture, the front-end need not make a distinction between active and passive attributes. Hence, active attributes need not be defined a-priori by the user, because data management (i.e. dynamically loading and removing data from main memory) is performed by the DBMS.

4.2 Statistics gathering

The quality of all extensions of a rule is computed using histograms. A histogram H for BAT A is a BAT containing ⟨value, frequency⟩ pairs

$$H(A) = \{\langle v_1, f_1\rangle, \langle v_2, f_2\rangle, \ldots, \langle v_n, f_n\rangle\}$$

where frequency f_i denotes the number of occurrences of value v_i in A. Although histograms can be easily constructed with a SQL aggregate query, we have used the hooks for extending the DBMS with new functionality. In particular, it was sufficient to implement a half-page C program to produce efficiently the histogram for any BAT. Its performance is linear in the size of the underlying table.

The quality of all extensions with a condition on attribute A of the empty rule R_0 : $\langle \text{true}\rangle \rightarrow \langle C\rangle$ is computed by first computing the subsets A_{p_0} and A_{n_0}. The BAT A_{p_0} contains the values for A for objects in P_0 and is the semi-join of A and P_0. By looking at the frequencies in the histograms $H(A_{p_0})$ and $H(A_{n_0})$ for a particular value v, we can compute the quality of the rule R_1 : $\langle A = v\rangle \rightarrow \langle C\rangle$. This quality is given by $p/(p+n)$, where $\langle v, p\rangle \in H(A_{p_0})$ and $\langle v, n\rangle \in H(A_{n_0})$, that is, p is the frequency of v in the set of positive examples P, and n is the frequency of v in N.

Computing the quality for conditions on numerical attributes is slightly more complicated. We use the same technique as in C4.5 (an extension of the ID3 system, see [11]), where the quality of a rule $\langle A < m\rangle \rightarrow \langle C\rangle$ is computed by summing the frequencies for all values, smaller than m, in both histograms $H(A_{p_0})$ and $H(A_{n_0})$. The same holds for rules of the form $\langle A > m\rangle \rightarrow \langle C\rangle$, where all values greater than m are summed. This may seem rather expensive, but the quality of all possible extensions can be computed in one pass over the histograms.

For the initial rule, the data mining tool requests histograms for all attributes[3]. They are used to compute the quality of all extensions, and the w best are selected. As described in Section 3, the search process is iterative, so each newly generated rule has to be extended as well. Now, the histograms have to be computed over the cover of condition D of rule R_0. So we have to compute *selections* over the BAT tables P_0 and N_0.

4.3 Computing selections

To determine the set of objects covered by rule R_1, i.e. all objects where $(A = v)$, two tables P_1 and N_1 are constructed that contain the oid's of all covered objects in respectively P and N. First, the selection $A = v$ is computed from table A, which is a fast, column oriented operation. The head of the resulting table T contains the oid's of all objects where $A = v$. To compute P_1 and N_1, the semijoin of P_0 (respectively N_0) and T is computed.

[3]This initial statistics gathering can be integrated with the database loading phase, because its results are also relevant for the query optimizer.

Now assume that rule R_1 is again extended to rule R_2 : $\langle (A = v) \wedge (B = w) \rangle \rightarrow \langle C \rangle$. Instead of computing the complex selection $(A = v) \wedge (B = w)$ from the database, we *reuse* the tables P_1 and N_1 and compute the selection $(B = w)$ using these tables. This is again a column oriented operation, thus using the advantages of Monet's storage structures.

Note that to compute the quality of extensions of R_1, it is not necessary to compute the histogram for table A, since it will only contain the value v. This is what we intend with the *zooming* behavior: the number of columns that are of interest decreases during the search process (vertical zooming), just as the number of objects, covered by a rule (horizontal zooming). Due to this zooming behavior, a smaller and smaller portion of the database is relevant for exploration. This means that the main-memory buffer requirements for retaining intermediate results stabilizes after a few iterations.

4.4 Temporary management

The database operations sketched above lead to an abundance of intermediate results. Although the Monet server automatically flushes the least recently used temporaries to disk, our data mining algorithm can determine precisely the subset of interest for the remainder of the session. Therefore, after each phase it releases temporaries by re-use of their external name or using explicit destroy operations. Attributes that are no longer used are automatically flushed to disk.

4.5 Parallelism

The data mining architecture offers opportunities for parallelization. The BATs $\{A_{p_i}, A_{n_i}, B_{p_i}, B_{n_i}, \ldots\}$ and their histograms can be computed in parallel. Furthermore, the beam search algorithm being employed calls for a parallel computation of its branches, i.e. the selections and histograms for each of the w rules can be computed in parallel.

5. Performance experiments

In this section we summarize the performance results of our two-level architecture against a hypothetical database. In Section 5.1 we show the result of loading the database from an **ascii** data file. Section 5.2 and 5.3 illustrate the performance for sequential and parallel execution of our algorithm, respectively.

5.1 Database loading

Since most of our data mining activities are focused on databases provided by clients, it is necessary to quickly load them into Monet. Therefore, the Monet loading utility takes an **ascii** representation of a table produced by an Oracle or Ingres database and constructs the binary association tables. Loading includes compression of the binary tables using Lempel-Ziv coding to trade disk space against a marginal cpu overhead. Figure 1 illustrates the linear behavior of loading tables to build an experimentation environment. The loading speed is about 1 Mb/s.

5.2 Sequential database mining

The performance of our architecture was analysed against the dummy insurance database introduced in Section 3. A screen dump of the user interface is included in Appendix A. Amongst others it illustrates the controllable parameters, such as beam width and tree depth,

ascii size	# of objects	cpu	sys
230 K	5 K	160	130
460 K	10 K	320	160
1.1 M	25 K	910	380
2.3 M	50 K	1850	960
4.6 M	100 K	3830	1290

Table 1: Loading ascii tables into Monet, times are in ms.

and options such as parallelism and simulated annealing (the latter is not discussed in this paper).

Table 2 shows the performance results for 5×5 and 7×7 beam searches. The database size ranges from 5K to 100K records, which covered a spectrum hardly reported in the literature. In these experiments, the code produced by the mining tool was executed in sequential mode by Monet. The experimentation platform was a 6-node SGI machine of 150Mhz processors and 256 Mbytes of main memory. The experiments were run in competition with other users on the system.

The table is read as follows. The column marked *miner* contains the time involved in the mining algorithm and management of the graphical user interface. The column *# ext.* shows the number of tested extensions. The columns marked *Monet cpu* and *Monet sys* describe the processing times as measured by the database back-end. All times are in milliseconds (!). The results indicate the constant processing cost within the user interface of about 12 and 24 seconds, respectively. The processing time in the DBMS back-end is largely linear in the database size and the number of tested extensions.

w × d	size	# ext.	miner	Monet cpu	Monet sys
5 × 5	5K	1549	12970	1400	1920
5 × 5	10K	1552	12380	2200	2075
5 × 5	25K	1559	13020	4350	2450
5 × 5	50K	1617	13000	7500	2500
5 × 5	75K	1668	13810	11100	3040
5 × 5	100K	1639	11970	13640	3320
7 × 7	5K	2320	22750	2470	4980
7 × 7	10K	2408	24790	4010	5402
7 × 7	25K	2671	25900	7350	5300
7 × 7	50K	2795	24230	11330	6000
7 × 7	75K	2756	27480	17460	8430
7 × 7	100K	2668	24610	21930	9890

Table 2: Performance results (in ms.) for different databases and beam search sizes.

As far as we have come across performance data in the literature, we can conclude that the set-up and the implementation of the DBMS is highly competitive. The mining tool itself has been designed with emphasis on simplicity and extensibility, so we expect that reasonable speed-up can be achieved by partly switching from Prolog to C, and optimization

of its algorithms.

5.3 Parallel database mining

As indicated in Section 4.5, our search strategy includes a processing phase that can be easily parallelized. The next experiments were set up to 1) test the implementation of the parallelization features in the database back-end and 2) assess the speed-up factor. The algorithms were modified to generate parallel code upon request by the user.

For this experiment we repeated the 5×5 case for a 25K database by turning on the parallelization switch. We varied the number of threads from 1 to 4 to determine their global contribution. The processing time in the user interface remained the same, because it is independent of the parallelization within Monet. The results of these experiments are shown in Table 3.

threads	Monet cpu	Monet sys
1	4350	2450
2	2630	2310
3	1730	1400
4	1465	1260

Table 3: Results in parallel mode.

We may conclude that considerable speed-up is achieved when multiple threads are used, although this speed-up is not linear. This is caused by the locking and synchronisation mechanisms in Monet, and by the fact that not all code can be run in parallel. In particular, the communication with the data mining tool is still serial. Figure 2 depicts the number of extensions per seconds as a function of the number of threads.

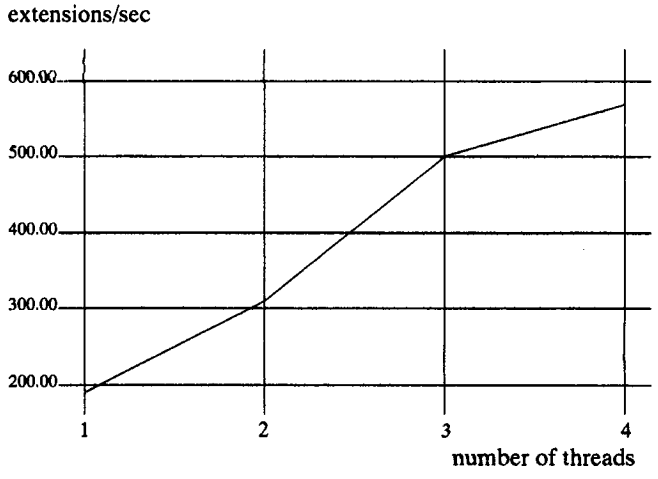


Figure 2: Computed extensions per second.

6. Conclusions

The two-level architecture of our data mining tool provides a performance-effective solution to data mining against real-life databases. The storage and access mechanism of the underlying general purpose DBMS architecture leads to a solution, where classification of large databases can be handled within seconds on a state-of-the-art multiprocessor.

We are currently investigating refinements of the quality control primitives to explore rule enhancements. Other points of interest at the CWI are 1) the use of domain knowledge to speed up the search process and distinguish interesting rules from trivial or already known knowledge; 2) the determination of sample sizes; and 3) alternative search strategies, such as simulated annealing or genetic algorithms.

The repetitive nature and overlap of successive queries call for automated support for browsing sessions in the database server [7], thereby further offloading parts of the mining activity to the DBMS kernel. Moreover, the policy to retain most information in main memory during a transaction leads to a high demand on the available store. A more prudent memory management scheme may be required to avoid clashes with other users and to scale beyond 10M objects easily.

Acknowledgements

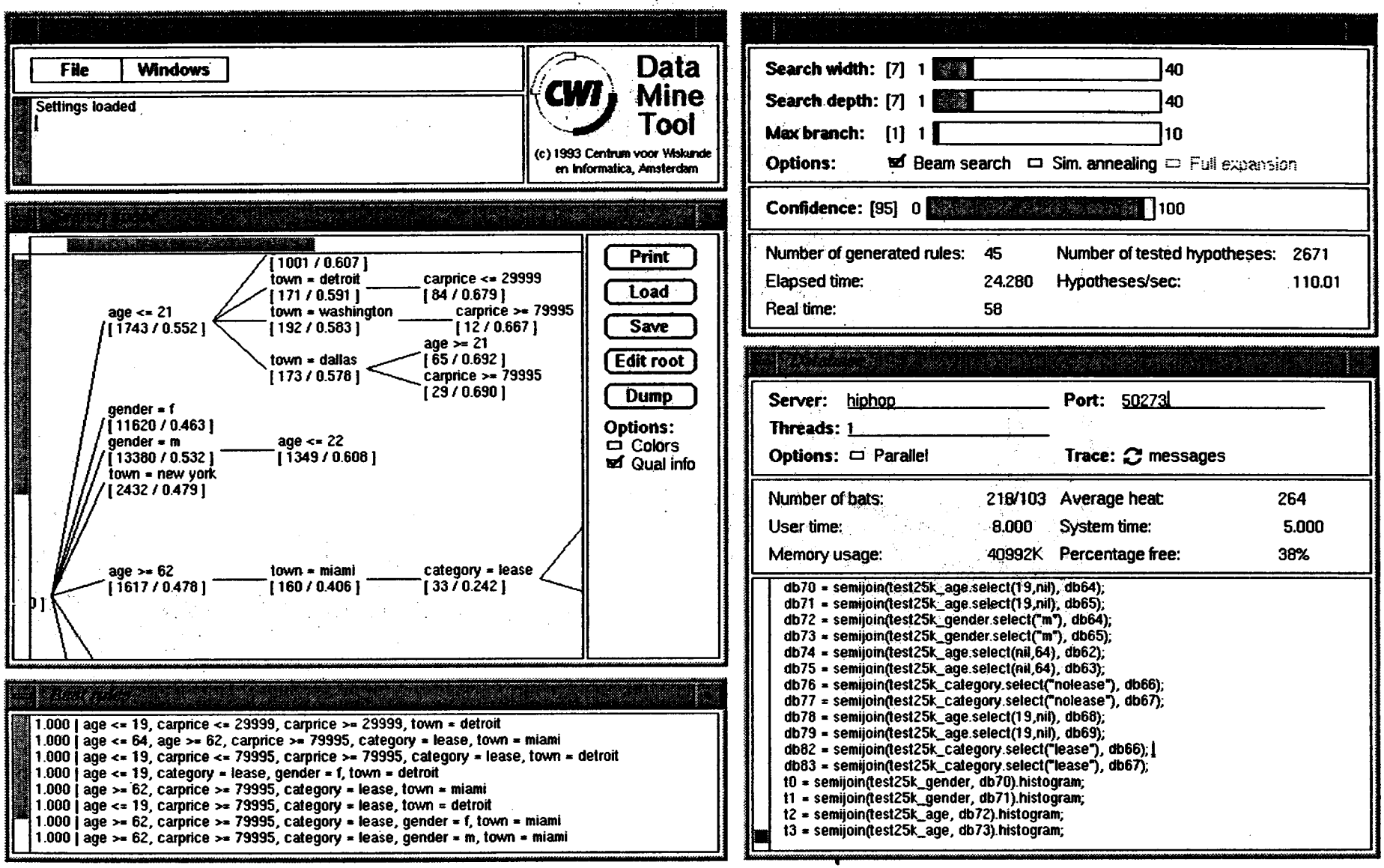
We wish to thank Carel van den Berg for his remarks on the paper, and Arno Siebes, Frank van Dijk, and Fred Kwakkel for continual support in the realisation of the CWI data mining tool.

References

1. Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: a performance perspective. *IEEE transactions on knowledge and data engineering*, Vol. 5, No. 6, December 1993:914 – 925, 1993.
2. Jaime G. Carbonell, Ryszard S. Michalski, and Tom M. Mitchell. An overview of machine learning. In Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors, *Machine Learning, an Artificial Intelligence approach*, volume 1, pages 3 – 24. Morgan Kaufmann, San Mateo, California, 1983.
3. M. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the Twelfth International Conference on Data Bases*, pages 91–100, August 1986. Kyoto.
4. Usama M. Fayyad, Nicholas Weir, and S. Djorgovski. Automated cataloging and analysis of ski survey image databases: the SKICAT system. In *Proc. of the second Int. Conf. on Information and Knowledge Management*, pages 527–536, Washington DC, 1993.
5. Marcel Holsheimer and Arno P.J.M. Siebes. Data mining: the search for knowledge in databases. Technical Report CS-R9406, CWI, January 1994.
6. Martin L. Kersten. Goblin: A DBPL designed for Advanced Database Applications. In *2nd Int. Conf. on Database and Expert Systems Applications, DEXA '91*, Berlin, Germany, August 1991.
7. Martin L. Kersten and Michiel de Boer. Query optimization strategies for browsing sessions. In *Proc. IEEE Int. Conf. on Data Engineering*, Houston, 1994.
8. Willi Klösgen. Efficient discovery of interesting statements in databases. Technical report,

GMD, 1993.

9. T.J. Lehman, E.J. Shekita, and L.F. Cabrera. An Evaluation of Starburst's Memory Resident Storage Component. *Journal of Data and Knowledge Engineering*, 4(6):555–567, December 1992.
10. J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81 – 106, 1986.
11. J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1992.
12. M. Stonebraker, L. Rowe, and M. Hirohama. The Implementation of POSTGRES. *Journal of Data and Knowledge Engineering*, 2(1):125–142, March 1990.
13. Carel A. van den Berg and Martin L. Kersten. An analysis of a dynamic query optimisation scheme for different data distributions. In J. Freytag, D. Maier, and G. Vossen, editors, *Advances in Query Processing*, pages 449 – 470. Morgan-Kaufmann, 1994.


File
Windows
Settings loaded
Data Mine Tool
(c) 1993 Centrum voor Wiskunde en Informatica, Amsterdam
Search width: [7] 1 40
Search depth: [7] 1 40
Max branch: [1] 1 10
Options: Beam search Sim. annealing Full expansion
Confidence: [95] 0 100
Number of generated rules: 45
Number of tested hypotheses: 2671
Elapsed time: 24.280
Hypotheses/sec: 110.01
Real time: 58
Print
Load
Save
Edit root
Dump
Options:
Colors
Qual info
[1001 / 0.607]
age <= 21
[1743 / 0.552]
town = detroit
[171 / 0.591]
carprice <= 29999
[84 / 0.679]
town = washington
[192 / 0.583]
carprice >= 79995
[12 / 0.667]
town = dallas
[173 / 0.578]
age >= 21
[65 / 0.692]
carprice >= 79995
[29 / 0.690]
gender = f
[11620 / 0.463]
gender = m
[13380 / 0.532]
age <= 22
[1349 / 0.608]
town = new york
[2432 / 0.479]
age >= 62
[1617 / 0.478]
town = miami
[160 / 0.406]
category = lease
[33 / 0.242]
Server: hiphop
Port: 50273
Threads: 1
Options: Parallel
Trace: messages
Number of bats: 218/103
Average heat: 264
User time: 8.000
System time: 5.000
Memory usage: 40992K
Percentage free: 38%
db70 = semijoin(test25k_age.select(19,nil), db64);
db71 = semijoin(test25k_age.select(19,nil), db65);
db72 = semijoin(test25k_gender.select("m"), db64);
db73 = semijoin(test25k_gender.select("m"), db65);
db74 = semijoin(test25k_age.select(nil,64), db62);
db75 = semijoin(test25k_age.select(nil,64), db63);
db76 = semijoin(test25k_category.select("nolease"), db66);
db77 = semijoin(test25k_category.select("nolease"), db67);
db78 = semijoin(test25k_age.select(19,nil), db68);
db79 = semijoin(test25k_age.select(19,nil), db69);
db82 = semijoin(test25k_category.select("lease"), db66);
db83 = semijoin(test25k_category.select("lease"), db67);
t0 = semijoin(test25k_gender, db70).histogram;
t1 = semijoin(test25k_gender, db71).histogram;
t2 = semijoin(test25k_age, db72).histogram;
t3 = semijoin(test25k_age, db73).histogram;
1.000 | age <= 19, carprice <= 29999, carprice >= 29999, town = detroit
1.000 | age <= 64, age >= 62, carprice >= 79995, category = lease, town = miami
1.000 | age <= 19, carprice <= 79995, carprice >= 79995, category = lease, town = detroit
1.000 | age <= 19, category = lease, gender = f, town = detroit
1.000 | age >= 62, carprice >= 79995, category = lease, town = miami
1.000 | age <= 19, carprice >= 79995, category = lease, town = detroit
1.000 | age >= 62, carprice >= 79995, category = lease, gender = f, town = miami
1.000 | age >= 62, carprice >= 79995, category = lease, gender = m, town = miami