Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

On the symbiosis of a data mining environment and a DBMS

M.L. Kersten and M. Holsheimer

Computer Science/Department of Algorithmics and Architecture

CS-R9521 1995

Report CS-R9521
ISSN 0169-118X

On the Symbiosis of a Data Mining Environment and a DBMS

Martin L. Kersten Marcel Holsheimer
{mk,marcel}@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 outline a two-level architecture, consisting of a mining tool and a database server. Key elements in its success are a clear separation of concerns: the mining tool organizes and controls the search process, while all data-handling is performed by the parallel main memory DBMS. Data is stored as a set of binary tables. The interaction consists of queries for statistical information. Properties of the DBMS and the search algorithm are exploited for optimization of the data handling. In particular, results of previous computations are re-used, and I/O activity is reduced by keeping a small hot-set of binary tables in main-memory. As test results show, this system handles large datasets at a competitive performance.

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*

Keywords & Phrases: data mining, parallel databases, knowledge discovery in databases

1. Introduction

In recent years, the scientific community has shown a rapidly growing interest in the discovery of hidden information in databases, also known as Data Mining [5, 8, 14]. The motivation for Data Mining is that databases contain more information than the stored facts themselves. For example, in commercial databases, global changes over time (*trends*), or (unknown) dependencies among attributes provide a valuable source of higher-level – strategic – information. These trends and dependencies are hidden among the vast amounts of data, and conventional DBMSs do not support their discovery, making specialized data mining tools a must. Successful application of these tools to real-world databases requires efficient techniques to handle large data-sets. Several techniques have been proposed:

1.0.1 Sampling techniques where only a small subset of the database is used, such that e.g. neural-net techniques [12] and traditional machine learning algorithms [13, 15] become tractable. A disadvantage of sampling is the inherent loss of information.

1.0.2 Incremental techniques which focus on discovering the hidden information with a single pass through the database [1]. These techniques pre-suppose a sizeable memory to maintain the learning or classification tree. Otherwise, it has to be decided on the fly what knowledge can be discarded from further consideration.

1.0.3 Iterative techniques where an initial set of rules is stepwise refined by retrieving additional information from the database.

The latter technique is used in our Data Surveyor/Monet system, a client-server architecture consisting of the *Data Surveyor* mining tool and the *Monet* parallel database server. The Data Surveyor front-end is a search engine to formulate and direct the mining activities. During the search process it queries the Monet server to retrieve statistical information. All massive data handling is performed by the Monet back-end. It has been developed around a binary storage model and provides an extensible system for studies into novel DBMS techniques considered essential for advanced applications [17].

The focal point of this paper is the interaction between Data Surveyor and Monet and the performance of the database server. Further details on the data mining algorithms and techniques to determine the quality of the relationships are presented in [6, 7, 16]. We will show that a high performance on the database is achieved by efficient use of main memory and reduction of I/O activity. Monet's binary storage model allows us to keep only a limited *hot-set* of currently relevant database attributes in memory. The size of this set decreases during the search process, this *zooming* property allows for considerable speed-up. Furthermore, the iterative nature of the search process is exploited by re-using results of previous computations.

The paper is organized as follows. Section 2 and 3 introduce Data Surveyor and Monet. Section 4 describes their interaction. Section 5 discusses the performance of a sequential and parallel back-end.

2. Data Surveyor

The Data Surveyor system is designed for the discovery of rules. These rules predict the probability that a database attribute (the *target* variable) has a particular value, given the values of other attributes.

Example 1 For a car insurance company it would be helpful if they could predict whether a particular insurant will have an accident. Clearly, predicting this for each and every client is impossible. What insurance companies do, is identifying groups of clients with the same accident *probability*. This probability is clearly not the same for all insurants, but depends on factors such as age, gender, etc. For example, young men are known to have a higher probability of having an accident than other clients.

So given last year's insurance database, containing information such as age, gender, hometown, carprice and a binary attribute accident, the data mine tool will search for rules that predict the probability that 'accident = yes'. ∎

2.1 Mining

The database is a set of complex objects, each consisting of attributes $A, B, C, \ldots$. The user defines one of these attributes as the *target* variable Z, e.g. accident. In this paper, we assume that the target is a binary attribute.[1]

Data Surveyor searches for rules that estimate the probability that the target attribute has a particular value, e.g.

[1]This form of learning is known as single class supervised learning [3].

$$\text{gender} = \text{m}, \text{age} \in [19, 24] \quad \rightarrow \quad \text{accident} = \text{yes} \qquad\qquad \text{probability} = 61.3\%$$

stating that male persons, aged between 19 and 24, will have an accident. Since not all young men will have an accident, we are merely interested in the probability that they have an accident. This probability is estimated as the percentage of correct predictions made with respect to the database. In our database, 61.3% of all young men has had an accident, hence the probability or *quality* of this rule is 61.3%. In fact, the Data Surveyor systems employs more sophisticated tests to guarantee that this probability is significantly different from the accident-probability of other clients, see [16].

In general, an insurance company won't be interested in a large set of rules predicting the accident-probability for all its insurants, but only in the n rules with the most deviating accident-probabilities, i.e. the most careful and the most dangerous drivers. To discover these n rules, the system uses an iterative search strategy. The initial rule is the rule with an empty condition

$$\langle \text{true} \rangle \quad \rightarrow \quad \text{accident} = \text{yes} \qquad\qquad \text{probability} = 50\%$$

stating that every insurant has had an accident. The quality of this rule is simply the percentage of accidental drivers in the database, i.e. the probability that an arbitrary insurant in the database will have an accident. In the first phase of the mining process, this rule is extended with an attribute-value condition to obtain a new rule, e.g.

$$\text{age} \in [19{,}24] \quad \rightarrow \quad \text{accident} = \text{yes} \qquad\qquad \text{probability} = 55.4\%$$

and this new rule is called an *extension* of the initial rule. It applies to only a subset of the database, i.e. all insurants aged 19 to 24. This subset is called the *cover*. In following phases, the rule is repeatedly extended with conditions. Data Surveyor uses a *hill-climber* technique to discover rules of high quality: at each phase the quality of *all* possible extensions of the current rule are computed. The extension with the most deviating quality is added to the rule. For example, adding the condition 'gender = m' results in the rule at the top of this page. This process continues until no further improvement is possible, or until the length of the rule exceeds a user defined threshold (the search depth d).[2]

The system searches the n-best rules in a single run, i.e. using multiple hill-climbers in parallel (called a *beam-search*). Instead of a single rule, the system maintains a set of rules (the *beamset*). At each phase, the quality of all extensions of these rules is computed, and the n-best are selected to form the new beamset.

EXAMPLE 2 The illustrate the concepts, we use a fictive 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 target variable is accident.

A search tree, constructed using a beam search is shown in Figure 1. Each node is annotated with the rule quality, the difference with the initial rule and the size of the cover. The initial condition covers 100K tuples, the quality is 50%, i.e. half of the insurants had an accident. The rule with condition 'gender = m, age ∈ [19,24]' covers 8130 objects from which 61.3% has had an accident.

[2]Note that since a hill-climber explores only part of the search space, it is not guaranteed to find the 'best' rule, but only a good one. As we show in [8], an exhaustive search of all rules is far too expensive.

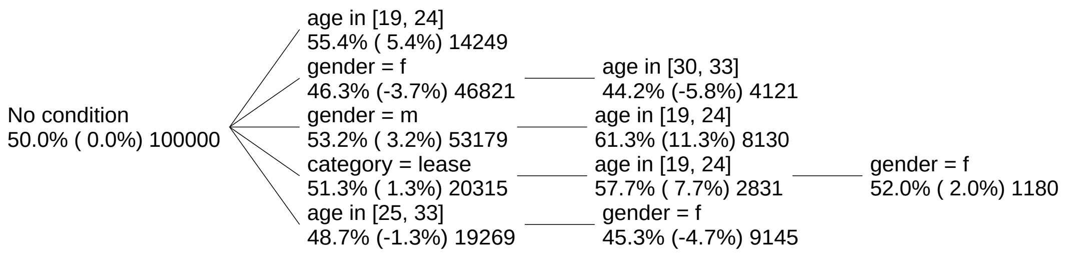

Figure 1: A search tree.

The first phase consists of a single – initial – rule, the second phase consists of 5 rules. Data Surveyor discovered only 4 extensions of this rule, forming the third phase, and a single extension in the fourth phase. ∎

3. The Monet database server

The DBMS used for our data mining tool is the *Monet* database server developed at CWI as a vehicle to support advanced applications and as an experimentation platform for novel database techniques (see [9, 10, 18]). Besides Data Surveyor it supports an SQL and ODMG compiler. Its salient features include:

3.0.1 Binary relation model Amongst its novelties are a fully decomposed storage scheme and adaptive indexing to speed-up query processing [18]. That is, complex objects are decomposed into a set of binary relationships using an OID to relate their components. Hence only a vertical subset of the database, consisting of the currently relevant attributes, can be loaded into main memory. Such a storage scheme is very effective in situations where the database *hot-set* can be kept memory resident. This design feature has proved crucial in the efficient support of our data mining tool.

Search accelerators are introduced as soon as an operator would benefit from their existence. They exists as long as the table is kept in memory and are not stored on disk.

3.0.2 Inter-operator parallelism The system exploits shared-store and all-cache architectures. A distributed shared-nothing approach as described in [17] is under development. Unlike the implementations of PRISMA [2] and GAMMA [4], Monet does not use tuple or segment pipelining. Instead, the algebraic operators are the units for scheduling and parallel execution. Their result is completely materialized before being used in the next phase. This approach improves throughput at a slight reduction of response time.

3.0.3 Simplicity Given the limited resources to construct a full-fledged DBMS and the complexity of such a system, a decisive factor in its development has been to choose the simplest solutions. In the design of its algorithms and data structures, Monet respects the limitations of its environment. This means that no code is introduced to 'improve' or 'replace' the operating system facilities for memory/buffer management. Instead, it looks for facilities to delegate as much as possible or to advice the lower level OS-primitives on the intended behavior, such as memory-mapped files and scheduling advice.

The Monet development platform has been the Silicon Graphics R3000 and R4000 workstations with the IRIX operating system. Furthermore, a shared-memory multi-processor of 6 150Mhz CPUs and 256 Mbyte of main memory is used to exploit parallelism.

The mode of interaction is that of client/server, where a single Monet server takes care of all interactions with a specific physical database. The clients are either application programs linked with Monet application programing interface library or to use a textual interface. The latter is used to interface with Data Surveyor. Although this implies some parsing overhead, it also provides a clean and traceable interface for debugging and to cross hardware implementation schemes.

3.1 Data storage and manipulation

A relational table is broken down into multiple Binary Association Tables (or BAT), where each database attribute is stored in a separate table. The first column, the *head*, of each BAT contains a unique object identifier (oid), the second column, *tail*, contains the attribute values. A similar vertical partitioning of the database is used by the Explora system [11], where only attributes that are relevant for the current mining task are loaded into main memory.

EXAMPLE 3 To illustrate, the table 'test100k(age, carprice, category, gender, town)' introduced in Example 2 is stored in 5 BATs: 'test100k_age', 'test100k_carprice', etc. It includes both string values and integers. The age and category domains are dense; all values within min(age) and max(age) are represented in the database. The carprice domain is sparse. This artificial database enables us to predict the statistical properties of queries. The binary target variable accident is not stored as a single BAT, but as two BATs, 'pos_accident' and 'neg_accident', whose heads contain the oid's of objects with accident = yes and accident = no respectively.

The database of 100K objects consumes about 2.6 Mbyte of disk space. It is stored in compressed form. The BATs are automatically decompressed and loaded upon need. The tables are not sorted, nor do they carry an index to speed-up selection on the tail of the association. ▮

The Monet kernel provides a rich set of algebraic operations on BATs, the subset relevant to the Data Mine system is summarized in Table 1.

BAT operator	result
AB.select(Tl,Th)	$\{ab \mid ab \in AB \wedge b >= Tl \wedge b <= Th\}$
AB.select(T)	$\{ab \mid ab \in AB \wedge b = T\}$
semijoin(AB,CD)	$\{ab \mid ab \in AB, \exists cd \in CD \wedge a = c\}$
AB.histogram	$\{bf \mid ab \in AB, f$ is frequency of b $\}$

Table 1: A subset of the Monet operators.

4. BRIDGING THE GAP

As we outlined in Section 2, processing takes place in different *phases*. During each phase, the Data Surveyor system computes the quality of all extensions of the rules in the current beamset, i.e. all possible conditions on each of the attributes. For example, on the attribute 'gender', conditions would be 'gender = m' and 'gender = f', conditions on age would be 'age $\in$ [19,20]', 'age $\in$ [19,21]', etc.

Note that only the attributes that are not yet present in the rule have to be taken into account. Extending the rule 'gender = m' with a condition on gender is useless. This is the *vertical zooming* property: the set of interesting attributes decreases in size during the search process.

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 for which the extension predict the target variable correctly, and another to compute the incorrect predictions. Although intuitively appealing, this technique is far too expensive. In Example 2, the quality of 1000 extensions have been computed, which would have resulted in 2000 database server interactions.

4.1 Retrieving statistics

We propose an alternative architecture, where the quality of all possible extensions of a rule is computed using only a few simple database operations. Data Surveyor queries the DBMS for information on the frequency of *combinations* of an attribute and the target variable in the database.

EXAMPLE 4 The combinations of the (binary) attributes 'gender' and 'accident' are: male drivers that had an accident, males with no accident, females with an accident and females without an accident.

Knowing the number of database-objects in these four classes, Data Surveyor can compute the quality of any extensions on gender, e.g. the quality of the rule 'gender = m' is the number of male drivers that had an accident, divided by the total number of males (with or without an accident). ∎

To compute the quality of extensions on an n-ary variable, such as hometown, Monet computes the sizes of each of the $2n$ classes. The algorithm, used to compute the quality of extensions on a numerical attribute, e.g. age, falls outside the scope of this paper.

The Data Surveyor system generates a Monet script to compute sizes of classes. First the semijoin of test100k_gender and pos_accident is computed, resulting in a BAT containing the ⟨oid, gender⟩ pairs of accidental drivers. From this BAT, the histogram is computed, i.e. a BAT containing the frequency of each value (f or m). A similar BAT is computed for the non-accident drivers, and for the other attributes:

```
t0_pos = semijoin(test100k_gender, pos_accident).histogram;
t0_neg = semijoin(test100k_gender, neg_accident).histogram;
t1_pos = semijoin(test100k_carprice, pos_accident).histogram;
t1_neg = semijoin(test100k_carprice, neg_accident).histogram;
t2_pos = semijoin(test100k_category, pos_accident).histogram;
...
```

The (temporary) bats tX_pos and tX_neg store the histograms and are printed after the above code is executed (the *delivery* phase). These results are shipped back to Data Surveyor.

4.2 Selection

The above computations are used to compute the extensions of the initial rule, i.e. the rule that covers the entire database. However, in the second phase of the beam search, the quality of extensions of rules such as 'age ∈ [19,24]' have to be computed (see Figure 1). So we have to compute the sizes of the classes *within* the cover of this rule. First we compute the database selection, corresponding to the cover, i.e. a BAT containing the oid's of young people. Next we compute the BATs pos_accident0 and neg_accident0 containing respectively the oid's of the accident and non-accident drivers in the cover. The code is repeated for the other rules in the beamset, i.e. 'gender = f', etc.

```
{tmp0 = test100k_age.select(19,24);
                 pos_accident0 = semijoin(tmp0, pos_accident);
                 neg_accident0 = semijoin(tmp0, neg_accident);
}
{tmp1 = test100k_gender.select("f");
                 pos_accident1 = semijoin(tmp1, pos_accident);
                 neg_accident1 = semijoin(tmp1, neg_accident);
}
{tmp2 = test100k_gender.select("m");
                 ...
```

Since 'pos_accidentX' contains the oid's of the accidental drivers in the cover of Xth rule in the beamset, it simply replaces 'pos_accident' in computing the quality of possible extensions. The code, executed by Monet to compute the quality of the extensions of the rules in the beamset is

```
t0_pos = semijoin(test100k_gender, pos_accident0).histogram;
t0_neg = semijoin(test100k_gender, neg_accident0).histogram;
t1_pos = semijoin(test100k_gender, pos_accident3).histogram;
t1_neg = semijoin(test100k_gender, neg_accident3).histogram;
t2_pos = semijoin(test100k_gender, pos_accident4).histogram;
t2_neg = semijoin(test100k_gender, neg_accident4).histogram;
t3_pos = semijoin(test100k_carprice, pos_accident0).histogram;
t3_neg = semijoin(test100k_carprice, neg_accident0).histogram;
t4_pos = semijoin(test100k_carprice, pos_accident1).histogram;
.......
```

4.3 Session optimization

In the third phase, we first have to compute the cover of the rules in the new beamset, e.g. the cover of 'gender = f, age ∈ [30,33]', by taking selections on both gender and age. At this point, we can re-use results of a previous computation, because the selection 'gender = f' is already made, and stored in 'pos_accident0' and 'neg_accident0'. The code to compute the new cover is:

```
{tmp0 = test100k_age.select(30,33);
                 pos_accident5 = semijoin(tmp0, pos_accident0);
                 neg_accident5 = semijoin(tmp0, neg_accident0);
}
```

This code contains no references to the gender BAT. The BATS 'pos_accident0' and 'neg_-

accident0' are no longer needed and destroyed.

We already mentioned the vertical zooming behavior of this algorithm, the number of columns that are of interest decreases during the search process. Something similar happens for 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.

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 name or using explicit destroy operations. Attributes that are no longer used are automatically flushed to disk.

4.4 Parallelism

The above program fragments illustrate the potential use of parallelism. Each cycle in the beam search can be decomposed into a *selection-*, *statistics-*, and *delivery-* phase. The computation of covers in the first phase, and the computation of statistics in the second can be ran in parallel. The last phase forces sequential processing, because the front-end itself has not been parallelised.

The Monet kernel largely relies on the scheduling of processes over the available processors. This is the default mode of operation on the SGI multi-processors and assures that resources are divided over multiple users. The experiments were ran on a lightly-loaded system and we have assured ourself that work was divided over the processors evenly.

5. Performance results

The current implementation has been subject to extensive performance assessment to isolate the bottlenecks and to direct research in both development of mining algorithms and the database kernel. The results presented here are focussed on the database activity.

In Section 5.1 we summarize the interaction of Data Surveyor and Monet in terms of database instructions, data exchanged and overall performance. Section 5.2 provides more detailed information on the cost of the database operations, while Section 5.3 illustrates the effect of parallel execution.

5.1 Sequential database mining

The first experiment illustrates the global division of labor between Data Surveyor and Monet. 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 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.

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.

These experiments indicate that the performance of Data Surveyor is largely independent of the database size. It is primarily determined by the properties of the search algorithm. Moreover, the front-end is relatively expensive compared to the back-end processing. This is a result of our implementation strategy for Data Surveyor, which is programmed in Prolog and XPCE, a windowing system for Prolog applications. It 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.

A second observation is that the database processing is fast for small scale databases (<100K). Therefore, we have enlarged the database size to gain more insight in the point where performance break-down occurs.

5.2 Breakdown of database processing

Each stage in the beam search algorithm involves a few heavily used database operations. In the loading phase, the hot-set is obtained from disk and decompressed. Subsequently, the histograms are computed for all database tables to find a starting point for further exploration.

The subsequent stages involve selections to determine the database cover of interest, followed by exploring the possible extensions. For each branch in the search tree, we require several semijoins followed by histogram construction.

The time components of a 5×3 beam search are illustrated in Figure 3 for several database sizes. These experiments where ran on a single processor SGI of 100Mhz and 32 Mbyte of main memory. The memory limitation helped to determine the performance break-down point, because the system is not able to keep everything in main-memory anymore. This is reflected in the increased system time for larger tables.

The table clearly illustrates the zooming behavior of the algorithm. Moreover, the performance of the key database operations is linear in the size of the database. The main-memory

Stage	Operation	50K cpu	50K sys	100K cpu	100K sys	200K cpu	200K sys
1	loading	2980	560	7960	990	12820	2320
	histogram	1690	160	3380	660	6900	1500
	printing	60	380	60	800	60	540
2	select	690	70	1370	520	2720	810
	statistics	580	90	1140	500	2280	720
	print	80	470	90	820	90	730
	statistics	600	70	1160	690	2370	800
	print	50	340	60	570	60	690
	statistics	470	40	930	260	1850	310
	print	50	320	50	650	50	310
	semijoin	710	50	1400	290	2770	320
	print	60	350	60	720	60	330
3	select	390	340	760	800	1460	710
	semijoin	160	100	290	470	550	520
	print	50	330	50	440	50	320
	statistics	180	40	330	370	680	400
	print	60	400	60	550	60	390
4	select	40	10	190	170	150	50
	statistics	30	140	50	390	20	130
	print	90	10	20	360	360	70
	statistics	30	110	90	200	90	290
	print	30	120	20	310	20	140

Table 3: Database processing for 5×3 beam search search per stage.

scans on columns of 100K takes about 70 milliseconds. Construction of a histogram on the age column is ca. 200 milliseconds. Printing is more or less constant, the same amount of information is returned to the front-end.

5.3 Parallel database mining

As indicated in Section 4.4, our search strategy includes two processing phases 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 Figure 2.

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.

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

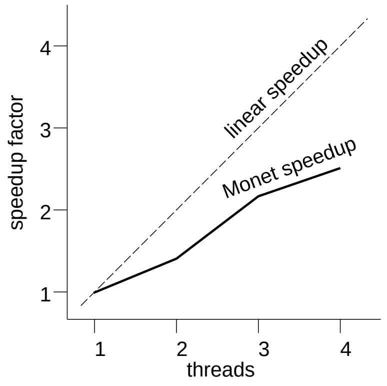

Figure 2: Results for parallel processing.

6. Conclusions

The two-level architecture of our data mining environment provides a performance-effective solution to data mining against real-life databases based on an iterative technique.

The Monet DBMS facilities lead to a solution, where classification of large databases can be handled within seconds on state-of-the-art multiprocessors. The repetitive nature and overlap of successive queries calls for automated support for browsing session optimization in the database server [9]. Thereby further off loading 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.

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

Due to the simplicity of the algorithm it is possible to distribute the work over replicated versions of database and the Monet server.

Acknowledgements
We wish to thank Arno Siebes for his remarks on the paper, and the members of the CWI Database Research Group 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. Peter M.G. Apers, Carel A. van den Berg, Jan Flokstra, Paul W.P.J. Grefen, Martin L. Kersten, and Anita N. Wilschut. PRISMA/DB: A parallel, main-memory relational dbms. *IEEE KDE, special issue on Main-Memory DBMS*, Dec, 1992.

3. 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.
4. D. J. DeWitt, S. Ghadeharizadeh, D.A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The GAMMA Database Machine Project. *IEEE Journal on Data and Knowledge Engineering*, 2(1):44–51, 1990.
5. Usama M. Fayyad and Ramasamy Uthurusamy, editors. *AAAI-94 Workshop Knowledge Discovery in Databases*, Seattle, Washington, 1994.
6. Marcel Holsheimer, Martin Kersten, and Arno Siebes. Data Surveyor: Searching the nuggets in parallel. In Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors, *Knowledge Discovery in Databases*, volume II. AAAI/MIT Press, forthcoming.
7. Marcel Holsheimer and Martin L. Kersten. Architectural support for data mining. In Fayyad and Uthurusamy [5], pages 217 – 228.
8. Marcel Holsheimer and Arno P.J.M. Siebes. Data mining: the search for knowledge in databases. Technical Report CS-R9406, CWI, January 1994.
9. 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.
10. Martin L. Kersten, Carel A. van den Berg, and Sander Plomp. Object storage management in goblin. In *Distributed Object Management*. Morgan Kaufman, Edmonton, Canada, 1992.
11. Willi Klösgen. Efficient discovery of interesting statements in databases. Technical report, GMD, 1993.
12. Richard P. Lippmann. An introduction to computating with neural nets. *IEEE ASSP Magazine*, April:4 – 22, 1987.
13. Ryszard S. Michalski, Igor Mozetic, Jiarong Hong, and Nada Lavrac. The AQ15 inductive learning system: an overview and experiments. Technical Report UIUCDCS-R-86-1260, University of Illinois, July 1986.
14. Gregory Piatetsky-Shapiro and William J. Frawley, editors. *Knowledge Discovery in Databases*. AAAI Press, Menlo Park, California, 1991.
15. J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1:81 – 106, 1986.
16. Arno Siebes. Homogeneous discoveries contain no surprises: Inferring risk-profiles from large databases. In Fayyad and Uthurusamy [5], pages 97 – 108.
17. Carel A. van den Berg and Martin L. Kersten. A dynamic parallel query processing architecture. In *COMAD'92*, Bangalore, India, 1992.
18. 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.