Using SQL primitives and parallel DB servers to speed up knowledge discovery in large relational databases.

Alex Alves Freitas[1] and Simon H. Lavington
{freial,lavis}@essex.ac.uk
University of Essex, Dept. of Computer Science
Wivenhoe Park, Colchester, CO4 3SQ, UK

Abstract

Efficiency is crucial in KDD (Knowledge Discovery in Databases), due to the huge amount of data stored in commercial databases. We argue that high efficiency in KDD can be achieved by combining two approaches, namely mapping KDD functionality onto standard DBMS operations and executing KDD tasks on a parallel SQL server. We propose generic KDD primitives which underly the candidate-rule evaluation procedures of many KDD algorithms, and we evaluate the speed up achieved by a parallel SQL server when executing a decision-tree learner algorithm implemented via these primitives.

1 Introduction

Our approach to Knowledge Discovery in Databases (KDD) is based on Machine Learning (ML) algorithms. However, sequential versions of most ML algorithms are impractical (i.e. take too long to run) on very large data sets. For instance, Catlett [1991] estimated that sequential C4.5 would take several months to learn from 1,000,000 examples by using state-of-the-art hardware at that time. Provost & Aronis [1995] report that a sequential version of the RL algorithm is impractical to run on data sets of more than 70k examples. They also present several reasons why it is desirable that an algorithm learns from very large data sets.

We argue that significantly greater efficiency in KDD tasks can be achieved by combining two approaches. First, the central KDD operations can be mapped to SQL queries submitted to the DBMS, in order to use efficiency-oriented DBMS facilities. Second, existing parallel DB servers such as GIS Teradata, ICL Goldrush, IBM SP2 and White Cross WX9010 [IBC 1994] can be used to speed up KDD operations. Note that, no matter how fast a sequential ML algorithm is, its time complexity is at least $\Omega(N)$, where N is the number of examples. Parallel processing offers the possibility of reducing this lower bound to $\Omega(N/p)$, where p is the number of processors. Hence, the main theme of this paper is the tight *integration* between the areas of ML, relational DBs and parallel DB computers.

This paper is organized as follows. Section 2 presents a primitive-based approach for mapping KDD operations into a parallel DB system. Section 3 describes how to use primitives to implement a decision-tree learner algorithm. Section 4 presents results of running this algorithm on a parallel SQL server. Section 5 reviews related work. Finally, Section 6 presents the conclusions.

2 Mapping KDD operations into a parallel DB system: a primitive-based approach

The use of primitives is relatively common in the area of databases. However, this approach is not very common in the area of AI, probably due to the great difficulty of finding domain-independent primitives in AI (see e.g. [Stolfo et al. 1995]). Fortunately, however, we have identified a ubiquitous, domain-independent operation underlying a number of ML/KDD algorithms. The goal is to create a practical framework for KDD, which takes advantage of high-performance DB servers. Developing generic KDD primitives is important because no single algorithm can be expected to perform well across all domains [King et al. 1995; Schaffer 1994].

First of all, as a pre-processing step for the KDD algorithm, we execute a query which selects - out of the whole DB - the task-relevant data set, i.e. the subset of tuples and attributes to be accessed by the KDD algorithm. The result of this query is stored as a new relation, called the Mine relation. Hence, we avoid the use of computationally-expensive join operations during the KDD algorithm.

A KDD algorithm can be viewed as the iterative process of selecting a candidate rule (CR) according to a CR-evaluation function, expanding it (generating new CRs) and evaluating the just-generated CRs. This process is repeated until a solution(s) - i.e. a satisfactory CR(s) - is(are) found [Holsheimer & Siebes 1995]. We assume that the Mine relation is stored in a parallel DB server,

[1]Supported by Brazilian-government's CNPq, grant 200384/93-7.

which is connected to one or several client workstations. The client selects the next CR to be expanded and expands it. However, in order to carry out CR-evaluation operations, the client sends SQL queries to the server. Finally, we assume that all attributes are categorical. Continuous attributes are discretized in a pre-processing phase [Richeldi & Rossotto 1995]. Although this may result in a small decrease of classification accuracy in some cases, it improves both computational efficiency and comprehensibility of the discovered knowledge.

2.1 A data-parallel KDD primitive for Candidate Rule (CR) evaluation

A primitive should satisfy two requirements, namely generality - i.e., it should find use in a number of KDD algorithms - and significant time saving- i.e., in a given KDD algorithm, a primitive should occur frequently and/or be computationally significant.

We have examined many KDD algorithms [Piatetsky-Shapiro & Frawley 1991; Holsheimer & Siebes 1994], and have come to the conclusion that their central activity can be expressed by a primitive which we will call Count by Group. This primitive consists of counting the number of tuples in each partition (group of tuples with the same value for the *group by* attributes) formed by a SQL *group by* statement. Count by Group has three input parameters, viz. an example-set descriptor, a candidate attribute, and a goal attribute. The output of Count by Group is shown in Figure 1. This is an m x n matrix extended with totals of rows and columns. m is the number of distinct candidate-attribute values and n is the number of distinct goal-attribute values (or classes). Each cell(i,j), i = 1,...,m and j = 1,...,n, of this matrix contains the number of tuples satisfying the example-set descriptor with candidate-attribute value A_i and goal-attribute value G_j. This primitive is implemented in a declarative style by Query 1, followed by a trivial computation of rows and columns totals. Note that Count by Group's set-oriented semantics has a good potential for data parallelism.

	G1 . . . . . Gn	Total
A1 . . . Am		
Total		

Figure 1 - Output of the Count by Group primitive.

```
SELECT Candidate_attribute,Goal_attribute, COUNT(*)
FROM Mine_Relation
WHERE  Example-Set_Descriptor
GROUP BY Candidate_attribute, Goal_attribute
```

Query 1 - SQL query underlying Count by Group.

3 Evaluating the primitive in the context of a DT (Decision-Tree) learner

We chose a DT learner [Quinlan 1993] as the algorithm for our experiments mainly because DT learners are in general the most used and most efficient kind of ML algorithm, and they are implemented in several commercial data mining tools. In Section 3.1. we describe our approach to the issue of pruning in DT learners. In Sections 3.2 and 3.3 we show how to use the primitive Count by Group to implement the candidate rule evaluation procedures of a DT learner.

3.1 DT (Decision-Tree) Pruning

There are essentially two approaches for DT pruning, viz. post-pruning and pre-pruning. In post-pruning the tree is expanded as far as possible by the recursive splitting process, and then the tree is pruned. In pre-pruning splitting can be stopped earlier, depending on some criterion (e.g. a statistical significance test). Post-pruning tends to produce better (i.e. higher classification accuracy) trees, since the larger number of tree nodes provides more information for the pruning procedure. On the other hand, pre-pruning makes the algorithm more efficient, since it expands fewer tree nodes (in our case, executes fewer DB queries). We follow a hybrid approach, using a non-agressive pre-pruning method and then applying a post-pruning method.

As a pre-pruning method, we have implemented Agrawal et al. [1992]'s method. Splitting is stopped if Freq $\geq$ (Depth/Max_Depth)^{Exp}, where Freq is the relative frequency of the prevalent class in the current node, Depth is the depth of the current node, Max_Depth is the maximum possible depth of the tree (i.e. the number of attributes) and the exponent Exp is a user-specified parameter (1 or 2 in our experiments, see Section 4). The larger the value of Exp, the deeper the tree is expanded. Hence, stopping of the recursive splitting process is "incentivated" in deeper levels of the tree. In addition, the method has a dynamic pruning scheme and a lookahead mechanism to avoid pruning too agressively.

As a post-pruning method, we convert the tree to production rules and, for each rule, iteratively remove the most irrelevant condition (the one whose removal most increases the rule's classification accuracy) while this removal process does not reduce the rule's classification accuracy [Quilan 1993]. We have chosen this rule-pruning method for three reasons. First, the conversion to rules usually improves the comprehensibility of the discovered knowledge. Second, the rule-pruning method considers removal of conditions regardless of the order in which conditions were generated, treating a rule as a *set* of conditions. This mitigates the limitations associated with the monothetic nature (i.e. selection of one-attribute-at-a-time) of DT learners. Third, the set-oriented semantics of this rule-pruning method maps naturally into both parallel processing and relational database theory.

3.2 Using Count by Group for selecting the best splitting attribute in a tree node

In each tree node, we need to compute the goodness of each candidate attribute (we use the gain ratio criterion) to select the best splitting attribute. To do that, we specify Count by Group's parameters as follows. The example-set descriptor is a conjunction of attribute-value conditions that represents the path leading from the root to a given node of the decision tree. The candidate and goal attribute parameters are obviously specified as the candidate splitting attribute and the goal (class) attribute.

3.3 Using Count by Group for selecting the most irrelevant rule condition

A rule is an expression of the form "if X then C", where X is a conjunction of attribute-value conditions X_i, i=1,..,c - where c is the number of conditions in the rule; and C is a class (goal attribute's value). Within a given iteration of the rule-pruning method, for each condition X_i, i=1,..,c, we have to evaluate whether X_i is irrelevant. This involves the generation of the table shown in Figure 2. The condition X_i is of the form $A_i = V_{ij}$, where A_i is the i-th attribute and V_{ij} is the j-th value of the domain of A_i. Let S be the set of tuples satisfying all conditions other than X_i in the rule. Each cell (partition) of the table in Figure 2 contains the number of tuples in S satisfying or not X_i and belonging or not to class C. The four cells are correspondingly labelled S_C, nS_C, S_nC and nS_nC. Note that although this rule-pruning method is usually applied only to rules converted from decision trees, it is quite general and can be applied to rules generated by many KDD algorithms. Hence, it is important to map the core operation of this method into an efficient SQL query.

The information necessary to produce the table shown in Figure 2 can be generated by Query 2, which is based on the Count by Group primitive. However, Query 2 returns to the host "too many count values", since it counts the number of tuples for *every* possible *group by* partition, while we are only interested in four *group by* partitions. For example, if both A_i and goal attribute can take on 5 different values, Query 2 would generate a 5x5 table, while we only need a 2x2 table. This query can be simplified, by exploiting a regularity in the count values of partitions S_C and S_nC within a given iteration of the condition-removal procedure, as defined in Theorem 1.

	class C	not class C
satisfies Xi	S_C	S_nC
does not satisfy Xi	nS_C	nS_nC

Figure 2: Table used in the condition-removal procedure.

```
SELECT  A_i, Goal_attribute, COUNT (*)
FROM  Mine_Relation
WHERE  Example_Set_Descriptor_S
GROUP BY  A_i, Goal_attribute
```

Query 2 - SQL query (with 2 grouping attributes) for condition-removal procedure.

```
SELECT  Goal_attribute, COUNT (*)
FROM Mine_Relation
WHERE  Example_Set_Descriptor_S  AND  A_i <> V_ij
GROUP BY  Goal_attribute
```

Query 3 - SQL query (with only 1 grouping attribute) for condition-removal procedure.

Theorem 1: In all the calls of Count by Group in a given iteration of the condition-removal procedure, the *group by* partitions S_C and S_nC in Figure 2 have values that are constant within the iteration. We show below the proof for the partition S_C. The proof for the partition S_nC can be achieved by an analogous argument.

Proof: Assume that the rule to be pruned has c (c > 0) conditions labelled X_i, i=1,..,c, before the beginning of a condition-removal iteration. That iteration calls Count by Group c times. The i-th call (i=1,..,c) of Count by Group has a *where* clause with c-1 conditions, namely X_1,.., X_{i-1},X_{i+1},..X_c. However, the attribute A_i, whose condition X_i was dropped from the *where* clause, is specified as the candidate attribute in Count by Group (see Query 2). The partition S_C consists of all tuples satisfying all conditions specified in the *where* clause of Query 2 (i.e. X_1,..,X_{i-1},X_{i+1},..X_c) *and* having $A_i = V_{ij}$ (i.e. satisfying the condition X_i) and having class C. Hence, in all k calls of Count by Group within an iteration, S_C consistis of tuples satisfying all k conditions X_i, i=1,..,c and having class C. Therefore, all of the c calls of Count by Group compute the same number of tuples in the partition S_C.

In order to implement the selection of the most irrelevant attribute, Theorem 1 allows us to use a primitive simpler (and so potentially faster) than Query 2. Since the count values in partitions S_C and S_nC are constant within a condition-removal iteration, we have to compute them only once in that iteration. (In addition, there is no need to execute a query to compute these values. They can be computed by reusing results of the previous iteration.) Hence, the count values in the other two partitions, viz. nS_C and nS_nC, can be computed by Query 3. Note that this query has only *one* grouping attribute, rather than two as Query 2, so that it returns fewer count values for the host. Moreover, it counts fewer tuples, since its *where* clause is more selective (due to the extra condition A_i <> V_{ij}). We compared the execution time of Queries 2 and 3 in the WX9010 parallel SQL server [Burwen 1994]. We observed that the time saving of Query 3 over Query 2 increases with the number of counted tuples. When Query 2 counts tens of thousands of tuples, Query 3 achieves a time saving of about 18%. When Query 2 counts few (less than one thousand) tuples, Query 3 takes about the same time as Query 2. Hence, we used Query 3 in our experiments.

4 Results on a parallel SQL server

There are now several types of parallel platform commonly available for commercial data processing. At a recent conference [IBC 1994], case-studies were given of ten parallel machines successfully used for revenue-earning applications. The most common architectural category was MIMD. We did experiments comparing a MIMD machine, viz. the White Cross WX9010 parallel SQL server, with a Sun IPC running at 25 MHz (15 MIPS) with 24 MBytes RAM. The White Cross WX9010 entry-level system (release 3.2.1.2) is based upon 12 T425 transputers, each with 16 Mbytes RAM, each rated at about 12 MIPS and running at 25 MHz [Burwen 1994]. (Note that each transputer belongs to the same technology generation and has approximately the same MIP rate as the Sun workstation). The WX9010 uses Data-in-Memory techniques for fast processing. Data is loaded into the 192 MBytes of RAM before processing begins. The WX9010 is a back-end SQL server attached to an Ethernet LAN. This machine is interesting because it has a very high rate of scanning tuples: 3 million tuples/sec. We did experiments with synthetic and real-world DBs, as discussed below.

4.1 Experiments with synthetic databases

We did experiments with two synthetic DBs, where attribute values were randomly generated according to a uniform probability distribution. In both DBs the predicting attributes' domain cardinalities varied from 2 to 10 (with average cardinality 6), and the goal attribute had domain cardinality 2. The first DB has 9 predicting attributes and a relatively simple classification function, denoted F1. F1 consists of 3 disjuncts, each of them with 2 conditions. The default accuracy is 68%. The second DB has 14 predicting attributes and a relatively complex classification function, denoted F2. F2 consists of five disjuncts, out of which one has 5 conditions, one has 4 conditions and three have 3 conditions. The default accuracy is 87.4%. For every experiment, we first generated a tuple and then applied a classification function on the tuple to determine its class. To introduce some noise, the wrong class is assigned to a tuple with a probability of 5%. The resulting data set was randomly partitioned into a training set and a 10,000-tuple test set. The classification functions are defined as follows (A_i denotes the value of the i-th attribute):

Function F1:
if (((A_1<3) and (A_3=1))
or ((A_1=3) and (A_3=2))
or ((A_1>3) and(A_2=1)))
then Goal = 1;
else Goal = 2;

Function F2:
if (((A_1<3) and (A_2>8) and (A_3=1) and (A_4=3) and (A_5=8))
or ((A_1=6) and (A_6=1) and (A_8<5) and (A_{10}=4))
or ((A_7<3) and (A_8>8) and (A_9=1)) or ((A_4=4) and (A_6=1) and (A_9=1))
or ((A_2<3) and (A_5>8) and (A_7=4)))
then Goal = 1;
else Goal = 2;

Table 1 - Results for synthetic data set with simple classification function F1.

PrePrun	Tuples	Accur.	Queries	Tup/Qu	Sun	WX	Sp
strong	100,000	94.97	131	22145	76.1	10.1	7.5
strong	200,000	95.06	131	44288	153.7	11.0	14.0
weak	100,000	95.01	945	6658	321.5	63.4	5.4
weak	200,000	95.03	948	13474	709.0*	66.8	10.6*

* Estimated figure

Table 2 - Results for synthetic data set with complex classification function F2.

PrePrun	Tuples	Accur.	Queries	Tup/Qu	Sun	WX	Sp
strong	100,000	90.94	65	48870	73.0	6.3	11.6
strong	200,000	91.05	65	97763	156.4	7.1	21.9
weak	100,000	93.85	2945	2804	985.5*	225.1	4.4*
weak	200,000	94.09	2690	5981	1800.0*	212.4	8.5*

*Estimated figure

Table 3 - Results for (real-world) Labour Force Survey data set.

PrePrun	Tuples	Accur.	Queries	Tup/Qu	Sun	WX	Sp
strong	103,219	85.41	421	16544	238.2	31.9	7.5
weak	103,219	85.19	4029	5675	1522.0*	282.5	5.4*

*Estimated figure

The results for the two synthetic DBs are shown in Tables 1 and 2. The first column in these tables indicates the "pre-pruning strength", viz. "strong" (if parameter Exp defined in Section 3 equals 1) or "weak" (if Exp equals 2). The other columns indicate respectively the number of tuples in the training set, the classification accuracy (in %), the number of DB queries done by the algorithm, the average number of tuples counted per query, the elapsed time on the Sun workstation (in min.), the elapsed time on the WX9010 (in min.), and the speed up of the WX9010 over the Sun. The results are averaged over 5 runs, except in rows marked with '*' in Tables 1 and 2, where figures were estimated due to very long elapsed times.

In both Tables 1 and 2, for a given pre-pruning method the Sun's elapsed time is roughly linear in the number of tuples counted per query, but the WX9010's elapsed time is much less sensitive to this number. Thus, the Sp increases with a larger number of tuples counted per query. For a fixed number of tuples, the Sp for strong pre-pruning is greater than the Sp for weak pre-pruning, since in the former the number of tuples counted per query is much larger. This occurs because strong pre-pruning implies a much smaller number of queries, and these queries are associated with higher levels of the tree, where the number of examples in a tree node is larger. When the classification function is simple (Table 1) the time saving associated with strong pre-pruning is achieved without loss of classification accuracy. However, when the classification function is complex (Table 2) that method results in some decrease in classification accuracy. (Ideally, the classification accuracy would be 95%, since both training and test set have 5% of noise.)

The small sensitivity of theWX9010 to an increase in the number of tuples counted per query indicates that the communication overhead between client and server represents a significant part of the query processing time (for the data set sizes used in the experiments). For instance, to select the best splitting attribute, Count by Group is called k times in a given tree node, where k is the number of candidate attributes. These k calls of Count by Group incur a repeated communication overhead that could be avoided. Indeed, the goal attribute is always the same and the k queries *select* (and *count*) exactly the same set of examples (i.e. they have the same *where* clause), so that candidate attribute is the only input parameter of Count by Group that varies among the k queries. Thus, the k queries could be transformed into a single, coarser-grained primitive, say a 'Multiple Count by Group' primitive, where the candidate attribute parameter is replaced with a *list* of candidate attributes. Hence, several query-execution steps (e.g. send query to server, compile query, and select tuples satisfying *where* clause) would be executed just once (rather than k times) per tree node. This idea is important in the case of the WX9010, since the server is connected to the client through a LAN network (rather than a dedicated channel) and the query-compilation time is about 0.8 secs. We estimate that, for the WX9010, the time saving achieved with 'Multiple Count by Group' would be about 20%.

4.2 Experiments with a real-world database

The Labour Force Survey (LFS) data is produced by the UK's Department of Employment. The data subset used in our experiments consists of employee records with 12 categorical attributes (with domain cardinality varying between 2 and 10), as follows: Sex, Age, Ethnic, Full/Part-time job, Permanent/Temporary job, Time in the company, Educational level, Public/Private job, Industry-area code, Occupation code, Managerial status, No. of employees at workplace. The goal attribute is Training Status, with two classes, indicating whether or not the employee had training in the past four weeks. These attributes were selected by the user among 350 attributes (most of them clearly irrelevant and/or redundant). The default accuracy is 85.3%.

The results are presented in Table 3, where column names are defined as in Section 4.1. Like the results for synthetic DBs, the WX9010 achieves a Sp of about one order of magnitude over a Sun workstation, and the Sp is larger for strong pre-pruning. Unfortunately, the classification accuracy is no better than the default accuracy. (This may be due to a high degree of noise in the data or simply due to an inadequacy of the DT learner's bias to this domain - see e.g. [Schaffer 1994] for a discussion of why classification accuracy below the default one is relatively common in practice. Anyway, note that the main contribution of this paper is to significantly speed up KDD algorithms.)

5 Related work

Houtsma & Swami [1995] have mapped a KDD algorithm into SQL, but their work does not address the issue of parallel processing. In addition, they do not propose any ubiquitous KDD primitive. Anand et al. [1995] use evidence-theory operators as primitives for parallel KDD. Their work exploits both data- and control-parallelism, but there is no interface with the DBMS during the execution of the algorithm (i.e. the algorithm is not implemented through DB queries).

Provost & Aronis [1995] have implemented a parallel version of the RL algorithm on the CM-2 with 8k processors. They achieved a speed up of about 1200 over sequential RL on a data set with 1,000,000 examples. However, the CM-2 is a very costly machine of limited commercial availability, and it is not clear whether parallel RL has a good *cost/benefit* ratio. In contrast, our approach is oriented towards more affordable, *cost-effective* parallel SQL servers. In addition, the CM-2 (like most SIMD computers) has boundary problems: a small increase in the size of the data set may significantly increase processing time. Furthermore, there is no interface with the DBMS.

Holsheimer et al. [1996] implemented a beam-search KDD algorithm on a parallel DB server. Like our work, candidate rules (CRs) are evaluated by submitting queries to the DB server. However, they exploit *inter*-query parallelism (to improve throughput), while our work exploits *intra*-query parallelism (to improve response-time). In addition, the evaluation of CRs is based on *semi-join* operations (although they are not expressed in SQL) and vertical partitioning of DB relations. We have avoided semi-joins and vertical partitioning, since these generally lead to inefficiencies in popular distributed-memory MIMD parallel servers. Furthermore, a horizontal partitioning is more suitable for parallel servers, since a tuple is the logical unit of data for most relational primitives.

There is extensive research about intra-query parallelism in decision support systems, already implemented in some commercial products [Zeller 1994], but this approach is not very popular in AI research.

None of the above work addresses the question of using parallel SQL servers and intra-query parallelism to speed up a wide range of KDD algorithms.

6 Conclusions and future research

Mapping KDD primitives into a parallel SQL server can be regarded as a form of extended DB functionality and automatic parallelization, increasing application-programmer productivity. Besides, a significant speed up can be achieved by processing the query on a parallel DB machine. Thus, a larger number of tuples may be considered by the KDD algorithm in a given time.

We have shown that a parallel SQL server achieves a speed up of about one order of magnitude (i.e. approximately linear in the number of processors) over a workstation of same technology generation running Ingres, when executing the central operations of a decision-tree learner. We also proposed two variations in our basic Count by Group primitive, to reduce query-response time. One variation consists of carrying out the *group by* operation based on a single (rather than on two) grouping attribute and the other one consists of using a coarser-grained Multiple Count by Group primitive to reduce the communication overhead between client and server.

Future research will involve the evaluation of algorithms using relational KDD primitives on other parallel SQL servers (especially to analyze communication overheads - see Section 4.1). We also intend to use other large real-world data sets.

Acknowledgements

We thank D.R. Thoen, N.E.J. Dewhurst, and R.A. Gamble for valuable discussions and Steve Hassan for his help in using the WX9010.

References

[Agrawal et al. 1992] R. Agrawal, S. Ghosh, T. Imielinski, B. Yer and A. Swami. An interval classifier for database mining applications. Proc. 18th Int. Conf. Very Large Databases, 560-573.

[Anand et al. 1995] S.S. Anand, C.M. Shapcott, D.A. Bell and J.G. Hughes. Data mining in parallel. Proc. World Occam and Transputer User Group Conf., Apr. 1995. Manchester, UK.

[Burwen 1994] M.P. Burwen. The White Cross parallel database servers. The Superperformance Computing Service. Product/Technology Review No. 145, 1994.

[Catlett 1991] J. Catlett. Megainduction: a test flight. Proc. 8th Int. Workshop on Machine Learning, 596-599, 1991.

[Holsheimer & Siebes 1994] M. Holsheimer and A. Siebes. Data mining: the search for knowledge in databases. Report CS-R9406. Amsterda: CWI, 1994.

[Holsheimer et al. 1996] M. Holsheimer, M. Kersten and A. Siebes. Data surveyor: searching the nuggets in parallel. To appear in: Advances in Knowledge Discovery and Data Mining. AAAI, 1996.

[Houtsma & Swami 1995] M. Houtsma and A. Swami. Set-oriented mining for association rules in relational databases. Proc. 11th IEEE Int. Conf. Data Engineering, 25-34. 1995.

[IBC 1994] IBC Ltd. Proc. Conf. on Commercial Parallel Processing, London, Nov. 1994.

[King et al. 1995] R.D. King, C. Feng and A. Shutherland. STATLOG: comparison of classification algorithms on large real-world problems. Applied Artificial Intelligence, 9(3). May/June 1995, 259-287.

[Piatetsky-Shapiro & Frawley 1991] G. Piatetsky-Shapiro and W.J. Frawley (Eds.) Knowledge Discovery in Databases. Menlo Park: AAAI, 1991.

[Provost & Aronis 1995] F.J. Provost & J.M. Aronis. Scaling up inductive learning with massive parallelism. To appear in Machine Learning.

[Quinlan 1993] J.R. Quinlan. C4.5: Programs for Machine Learning. Morgan Kaufmann, 1993.

[Richeldi & Rossotto 1995] M. Richeldi and M. Rossotto. Class-driven statistical discretization of continuous attributes. Proc. 8th European Conf. Machine Learning, 1995. (LNAI-912, 335-338).

[Schaffer 1994] C. Schaffer. A conservation law for generalization performance. Proc. 11th Int. Conf. Machine Learning, 259-265. 1994.

[Stolfo et al. 1995] S.J. Stolfo, H.M. Dewan, D. Ohsie and M. Hernandez. A parallel and distributed environment for database rule processing: open problems and future directions. In M. Abdelguerfi & S. Lavington (Ed) Emerging Trends in Database and Knowledge-Base Machine, 225-253. IEEE, 1995.

[Zeller 1994] H. Zeller. A new hash-based join algorithm in non-stop SQL. Tandem Systems Review, July 1994.