Parallel data mining for very 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

Data mining, or Knowledge Discovery in Databases (KDD), is of little benefit to commercial enterprises unless it can be carried out efficiently on realistic volumes of data. Operational factors also dictate that KDD should be performed within the context of standard DBMS. Fortunately, relational DBMS have a declarative query interface (SQL) that has allowed designers of parallel hardware to exploit data parallelism efficiently. Thus, an effective approach to the problem of efficient KDD consists of arranging that KDD tasks execute on a parallel SQL server. In this paper we devise generic KDD primitives, map these to SQL and present some results of running these primitives on a commercially-available parallel SQL server.

1 Introduction.

In very broad terms, Knowledge Discovery in Databases (KDD), or Data Mining, is the application of Machine Learning (ML) and/or Statistics to databases (DBs) [3], [7]. However, the word “databases” has been interpreted in a rather loose sense by many KDD researchers. The “databases” reported in the ML/KDD literature usually consist of relatively small (less than 10,000 tuples) datasets and are not stored in any DB Management System (DBMS). In contrast, we take the word “databases” as refering to large datasets (at least tens of thousands of tuples) maintained in a DBMS (e.g. Oracle or Ingres). Due to the very large amount of data, these databases *must* be accessed through efficiently-executed DB queries (expressed in SQL in the case of relational DBs). This issue is ignored in most of the KDD literature, which assumes that a relatively small sample is extracted from the DB and processed by a KDD algorithm with no interface with the DBMS. (Of course, there are a few exceptions, such as [4].) See [8] for a discussion of why it is desirable that a ML algorithm learns from very large datasets and for some results showing that sequential versions of ML algorithms are impractical (i.e. take too long to run) on very large datasets.

Our approach to KDD consists of *integrating* (symbolic) ML algorithms, relational DBs and parallel DB servers into a simple, practical framework. We map the central, time-consuming operations of KDD algorithms to SQL queries submitted to the DBMS, in order to use efficiency-oriented DBMS facilities. In addition, we use parallel DB servers [5] to speed up these KDD operations.

This paper is organized as follows. Section 2 presents an approach based on generic primitives to map KDD operations into a parallel DB system. Section 3 presents computational results on a parallel SQL server. Finally, Section 4 presents the conclusions.

---

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

2 A Generic KDD Primitive for Candidate-Rule Evaluation Procedures.

A large part of the published algorithms for KDD have originated from the Machine Learning community, itself a branch of AI. Although domain-independent AI primitives are hard to devise [13], we have identified a ubiquitous, domain-independent operation underlying a number of ML/KDD algorithms. Developing generic KDD primitives is important because no single algorithm can be expected to perform well across all domains [6], [11].

In common with much contemporary data-warehousing practice, we first assume that the whole DB has been subjected to prior queries that select a 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.

Most KDD algorithms 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 [3]. We assume that the Mine relation is stored in a parallel DB server, which is connected to one or several client workstations. The client is in charge of selecting the next CR to be expanded and expanding 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 converted to categorical ones in a pre-processing phase [10].

We have studied many ML/KDD algorithms (see e.g. [7], [3], [6]) and have come to the conclusion that their central, time-consuming 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 relational *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 is inherently data-parallel, due to its set-oriented semantics. Some recent advances in the optimization of parallel *group by* queries are described in [12].

The output of Count by Group can be directly used to compute many CR goodness measures, such as information gain or entropy (used by algorithms like C4.5 and CN2), J-measure (used by ITRULE), conditional probabilities (used e.g. by Bayesian Classifiers), and statistical significance tests (e.g. Chi-Square) - see e.g. [6] for an overview of these algorithms. It is interesting to note that, although Count by Group was designed for ML algorithms, it finds use in some statistical classification algorithms as well.

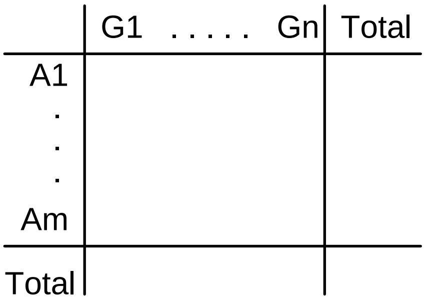

Fig. 1. Structure of the matrix produced by the Count by Group primitive.

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

Query 1. SQL query underlying the Count by Group primitive .

3 Computational Results on a Parallel SQL Server.

We did experiments comparing a MIMD machine, viz. the White Cross WX9010 parallel SQL server, with a Sun IPC running at 25 MHz with 24 MBytes RAM. In all experiments, our results refer to main memory databases (i.e. disc activity is excluded). The White Cross WX9010 (release 3.2.1.2) has 12 T425 transputers, each with 16 Mbytes RAM, each rated at about 10 MIPS and 25 MHz [2]. (Note that each transputer belongs to the same technology generation and has roughly the same MIP rate as the Sun workstation). The WX9010 is a main-memory shared-nothing DB machine with a very high rate of scanning tuples: 3 million tuples/sec. It is a back-end SQL server attached to an Ethernet LAN.

We compared the time taken to execute Count by Group alone (regardless of any particular algorithm) on synthetic datasets, where 100,000 10-attribute tuples were randomly generated according to a uniform probability distribution. In all experiments, the *group by* attributes had a domain cardinality of 2, so that the output of Count by Group was a 2x2 contingency table - in general the contingency-table size most encountered in practice. The results are presented in Table 1.

Table 1. Results of executing Count by Group on two DB servers.

Where Cond.	Counted Tup.	Sun (s)	WX9010 (s)	Speed up
0	100,000	127.5	5.8	22.0
2	8,341	32.2	4.9	6.6
4	137	21.7	5.0	4.3

Each row of results in Table 1 is averaged over 30 experiments. The first column of this Table gives the number of attribute-value conditions in the *Where* clause of Query 1. The second column gives the number of *counted* tuples (the more conditions the *Where* clause has, the smaller the number of tuples *selected* and *counted*, since the *Where* clause is a conjunction of conditions). The third and fourth columns give the time (in secs.) taken by Count by Group on the Sun and on the WX9010. The last column gives the speed up of the WX9010 over the Sun.

As expected, the speed up increases with a larger number of counted tuples (fewer conditions specified in the *Where* clause). This begs the question of how many tuples are *selected/counted* per query, on the average, when executing a KDD algorithm. To take one step towards answering this question, we used Count by Group to implement a D.T. learner [9], which is in general the most used and most efficient kind of ML algorithm (being available in several commercial KDD tools). In each node of the tree, the output of Count by Group was used to compute the Gain Ratio measure for candidate splitting attributes. Concerning tree pruning, we implemented a hybrid pruning method, by using the pre-pruning method proposed by [1] and the (post-pruning) rule-pruning method described in [9]. We believe this represents a good trade-off between the computational efficiency of pre-pruning (which executes fewer DB queries) and the effectiveness of post-pruning (which tends to generate better rules, but it is very time-consuming on large data sets). A variant of Count by Group (to be discussed in another paper) was used in the rule-pruning procedure.

We did experiments with two synthetic datasets and a real-world dataset. In the synthetic datasets, attribute values were randomly generated according to a uniform probability distribution. The first dataset (called DS1) has 9 predicting attributes and a relatively simple classification function (used to generate the values of the goal attribute based on the values of the predicting attributes), which consists of 3 disjuncts, each of them with 2 conjuncts of attribute-value conditions. The second dataset (called DS2) has 14 predicting attributes and a relatively complex classification function consisting of five disjuncts, out of which one has 5 conjunctive conditions, one has 4 conjunctive conditions and three have 3 conjunctive conditions. The real-world data set was the Labour Force Survey (LFS) data, produced by the UK's Department of Employment. We used a subset of the LFS data consisting of 12 predicting attributes. The goal attribute was Training Status, indicating whether or not an employee has had training.

Table 2. Results of running a decision-tree learner on 3 datasets.

Dataset	PrePrun	Tuples	Queries	Tup/Qu	Sun	WX	Sp
DS1	strong	1000,00	131	22145	76.1	10.1	7.5
DS1	strong	200,000	131	44288	153.7	11.0	14.0
DS1	weak	100,000	945	6658	321.5	63.4	5.4
DS1	weak	200,000	948	13474	709.0*	66.8	10.6*
DS2	strong	100,000	65	48870	73.0	6.3	11.6
DS2	strong	200,000	65	97763	156.4	7.1	21.9
DS2	weak	100,000	2945	2804	985.5*	225.1	4.4*
DS2	weak	200,000	2690	5981	1800.0*	212.4	8.5*
LFS	strong	103,219	421	16544	238.2	31.9	7.5
LFS	weak	103,219	4029	5675	1522.0*	282.5	5.4*

* Estimated figure.

The results are shown in Table 2. (Due to limitations of space, we discuss only the results about the execution time - whose minimization is the goal of our research - and ignore the quality of the discovered knowledge.) The first column shows the name of the dataset. The second column shows the pre-pruning "strength", viz. "strong" - if the tree expansion is stopped relatively early, trading effectiveness for efficiency - or "weak" - if the tree expansion is stopped relatively late. (The pre-pruning "strength" is specified by the user.) The other columns show respectively the number of tuples in the training set, 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 (Sp) of the WX9010 over the Sun. The results are averaged over 5 runs (except for experiments marked with '*' in Table 2, where figures were estimated due to the very long elapsed times).

In both datasets DS1 and DS2, 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. As a result, 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 method the number of tuples counted per query is much larger. This occurs because strong pre-pruning implies a much fewer number of queries, which are associated with higher levels of the tree - where the number of examples in a tree node is larger.

The small sensitivity of the WX9010 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 Conclusions and Future Research.

We have derived a generic KDD primitive and mapped it into SQL. We have implemented a Machine Learning algorithm via this primitive, and measured its performance on a parallel platform. This approach simplifies applications software (therefore increasing programmer productivity), whilst reducing data mining run-times. Thus, a larger number of tuples may be considered by the KDD algorithm in a given time, which is desirable for a number of reasons [8].

We have shown that a 12-processor parallel SQL server achieves a speed up of about one order of magnitude over a workstation of same technology generation running Ingres, when executing both a KDD primitive and a decision-tree learner implemented via that primitive. We also proposed a variation in our basic Count by Group primitive, to reduce query-response time. This variation consists of using a coarser-grained Multiple Count by Group primitive to reduce the communication overhead between client and server.

We intend to implement at least a restricted version of the Multiple Count by Group by using the mechanism of stored procedures offered by modern DBMS, such as Oracle 7.X. Future research will also involve the evaluation of algorithms using relational KDD primitives on other parallel SQL servers, such as the ICL Goldrush (especially to analyze communication overheads - see Section 3). We also intend to use other large real-world datasets.

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.

[1] R. Agrawal et al. An interval classifier for database mining applications. Proc. 18th Int. Conf. Very Large Databases, 560-573. Vancouver, 1992.

[2] M.P. Burwen. The White Cross parallel database servers. The Superperformance Computing Service. Product/Technology Review No. 145. (Available from 2685 Marine Way, Suite 1212, Mountain View, CA, USA, 94043.)

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

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

[5] IBC Ltd. Proc. Conf. on Commercial Parallel Processing, London, Oct. 1995. (Available from IBC Technical Services Ltd., 57-61 Mortimer Street, London.)

[6] D. Michie et al. (Ed.) Machine Learning, Neural and Statistical Classification. New York: Ellis Horwood, 1995.

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

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

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

[10] M. Richeldi and M. Rossotto. Class-Driven statistical discretization of continuous attributes. Proc. 8th ECML-95. LNAI-912, 335-338.

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

[12] A. Shatdal and J.F. Naughton. Adaptive parallel aggregation algorithms. Proc. 1995 ACM SIGMOD Int. Conf. Management of Data, 104-114.

[13] S.J. Stolfo et al. 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. IEEE Computer Science Press, 1995.