Towards Large-Scale Knowledge Discovery in Databases (KDD) by Exploiting Parallelism in Generic KDD Primitives.

Alex A. Freitas
University of Essex, Dept. of Computer Science
Wivenhoe Park, Colchester, CO4 3SQ, UK
freial@essex.ac.uk

Abstract

Efficiency and scalability are crucial issues in Knowledge Discovery in Databases (KDD). Our approach to these challenging issues is to devise generic, set-based KDD primitives which are insensitive to the order in which data elements are processed. Such primitives facilitate the exploitation of parallelism. Furthermore, these primitives are generic in that they support a wide selection of rule-induction and instance-based learning KDD algorithms. We present the results of running the primitives on two commercially-available parallel database platforms. We show that scalable parallel performance is possible on large databases.

1. Introduction.

Knowledge Discovery in Databases (KDD), or Data Mining, consists of extracting interesting, useful knowledge from real-world databases [8]. Despite the great demand for KDD in large databases, "conventional" KDD algorithms have been applied mainly to relatively small samples of data (typically less than 5,000 tuples) and do not have any integration at all with Relational DBMSs. Next-generation KDD systems will face the challenge of coping with huge amounts of data stored in data warehouses and of being tightly integrated with database systems in a corporate-wide scale.

However, the application of KDD algorithms to large databases faces serious scalability problems, particularly in respect of problem of excessive processing time. For instance, Cohen [7] remarks that the processing time of the rule-pruning method of C4.5, a well-known decision-tree-based KDD algorithm, scales roughly as the cube of the number of tuples. As a result, Cohen estimates that this method would take 79 years on a 150-MHz processor in order to process 500,000 tuples. As another example, Provost & Aronis [20] report that a sequential version of the RL algorithm is impractical (i.e. takes too long to run) on data sets of more than 70,000 tuples, and that a massively-parallel version of the RL algorithm is usually faster than its sequential version when the number of tuples is greater than 10,000.

It should be emphasized that, no matter how fast a sequential KDD algorithm is, its time complexity is at least $\Omega(N)$, where N is the number of tuples. Parallel processing offers the possibility of reducing this lower bound to $\Omega(N/p)$, where p is the number of processors. Hence, parallelism seems to be the great hope to scale up next-generation KDD systems.

Our approach to improved efficiency (shorter processing time) in KDD consists of exploiting data parallelism in KDD via carefully-designed primitives. In this paper we propose generic, set-oriented primitives to support the data-intensive operations of algorithms belonging to two major KDD paradigms, namely Rule Induction (RI) and Instance-Based Learning (IBL). A discussion of the pros and cons of these paradigms in the context of KDD is beyond the scope of this paper. Here we briefly remark that these two paradigms have complementary strengths and weaknesses, so that each of them tends to achieve good results in domains where the other might not do very well [18]. Indeed, the integration of RI and IBL is an emergent trend in KDD [24], [1]. The scope of this paper is restricted to relational databases.

This paper is organized as follows. Section 2 introduces our set-oriented, primitive-based framework for KDD. Section 3 proposes a primitive for the Rule Induction paradigm, while Section 4 proposes a primitive for the Instance-Based Learning paradigm. Both Sections also report results of experiments evaluating the efficiency in the exploitation of data parallelism on parallel database servers. Finally, Section 5 presents the conclusions.

2. A Set-Oriented, Primitive-Based Framework for Knowledge Discovery in Databases.

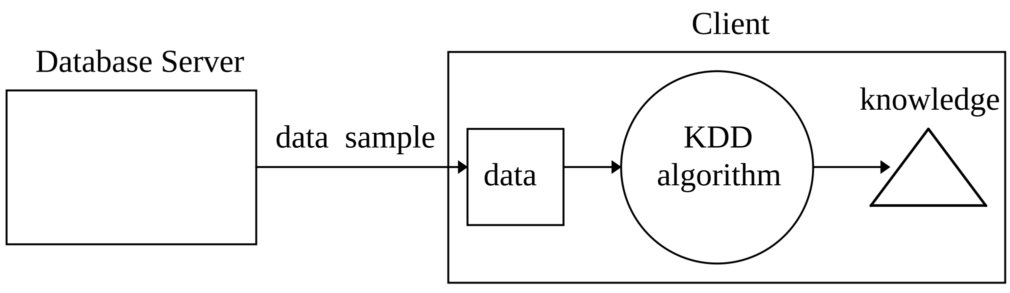
A central theme of this paper is the development of generic, well-defined, context-free primitives that capture the core operations underlying a number of KDD algorithms, as it will be shown later. We stress that developing *generic* KDD primitives is important because no single algorithm can be expected to perform well across all domains [18], [23].

We assume that the database system has a client-server architecture - adopted by most current systems. Figure 1 illustrates the main differences between the "conventional" framework for KDD and our set-oriented architectural framework. In this Figure the relative size of the squares, circles and triangles (representing respectively data, KDD algorithm and discovered knowledge) roughly indicate the size of the corresponding object.

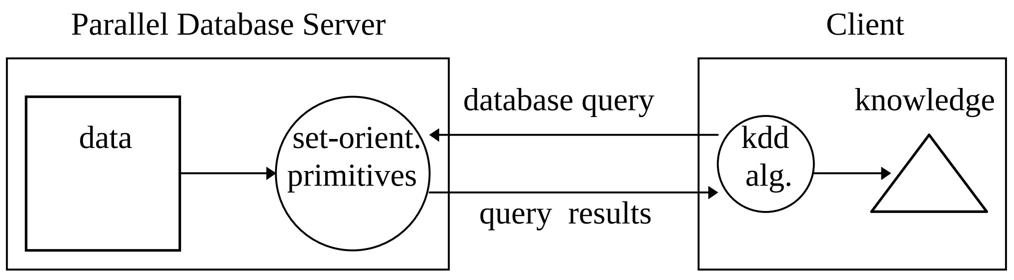
In the "conventional" framework for KDD - Figure 1(a) - the square representing data is small and the circle representing the KDD algorithm (running only on the client) is big. In other words, a small data sample is downloaded from the Server to the Client, and *all* the procedures of the KDD algorithm are executed on the Client.

In contrast, in our framework - Figure 1(b) - the data is kept on a Parallel Database Server. Hence, the square representing data is big, indicating that a larger database can be mined, and the circle representing the KDD algorithm is now split into two circles, the largest of them representing set-oriented primitives running on the Server (performing the most time-consuming operations) and the smallest of them representing KDD supervisory procedures running on the Client. Finally, the triangle representing the discovered knowledge is the same in both Figures 1(a) and 1(b), since a data-parallel version of a KDD algorithm discovers the same knowledge as its sequential counterpart.

Note that in our framework the KDD algorithm does not have direct access to the data. That algorithm simply sends database queries to the Parallel Database Server, which uses automatic parallel-query-optimization methods to efficiently access the data and returns the query results to the Client. The database queries submitted by the Client are actually requests for the execution of set-oriented primitives. We map these primitives into SQL. Despite its limitations, SQL is the industry-standard query language of Relational DBMS. Furthermore, it is a declarative interface, effectively decoupling applications programmers from low-level computational and architectural details. The result is an efficient, easy-to-use framework for KDD that takes advantage of high-performance parallel database servers.



(a) The "conventional" KDD framework.



(b) Our framework for KDD.

Figure 1. The "conventional" KDD framework vs. our set-oriented framework.

For the sake the efficiency, as a pre-processing step for the KDD algorithm we execute a query which selects - out of the whole database - 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. The Mine relation can be stored as a snapshot, as a new base relation or as a materialized view. We stress that this approach is compatible with the concept of the data warehouse. Actually, a data warehouse can be seen as a materialized view over multiple, autonomous data sources [30].

To summarize, in our framework a KDD primitive should satisfy four requirements, as mentioned below. The next two Sections propose primitives satisfying these requirements.

(1) Well-defined specification - Its input, output and processing should be precisely defined.

(2) Generality - It should find use in a number of KDD algorithms;

(3) Computational significance - In a given KDD algorithm it should occur frequently and/or take a significant part of the total processing time of the algorithm.

(4) Set-oriented nature - It should process many-tuples-at-a-time, independently of the order of the tuples.

3. A Primitive for Rule Induction (RI) Algorithms.

A rule is a knowledge-representation structure of the form: “if P then Q”, where P is a conjunction of attribute-value conditions and Q is a goal-attribute-value pair indicating the class predicted by the rule. In the Rule Induction (RI) paradigm the KDD algorithm can be cast as a heuristic search in the space of candidate rules (CRs). In essence, a RI algorithm can be viewed as the iterative process of selecting the “best” CR according to a CR-evaluation function, expanding it (generating new CRs) and evaluating the just-generated CRs. The expansion of the selected CR involves the application of specialization and/or generalization operations to the CR. (In general a conjunctive CR is specialized by adding conditions to it, and it is generalized by removing conditions from it). This process is repeated until a satisfactory set of CRs is found [15].

As discussed above, in our KDD framework the Mine relation (i.e. the subset of tuples and attributes to be accessed by the KDD algorithm) is stored on a Parallel Database Server. The Client selects the next CR to be expanded, and then expands it. However, in order to carry out CR-evaluation operations, the Client sends SQL queries to the server. This is the kind of operation for which the primitive proposed in this Section has been developed.

3.1 Count by Group: a Primitive for Candidate Rule (CR) Evaluation.

This Section introduces Count by Group, a generic primitive to support Candidate Rule (CR) evaluation operations, which constitute the principal data-intensive, time-consuming activity of Rule Induction algorithms. This primitive was developed to process categorical attributes - i.e. attributes whose domain consist of a small set of discrete values, or categories. Continuous attributes would be discretized in a pre-processing phase. See [14] for a modification of this primitive developed to cope with continuous attributes and for a discussion of the pros and cons of discretization in the context of KDD.

Firstly we specify the primitive in terms of its input parameters, its output and its processing, as follows. Count by Group has three input parameters, namely:

(a) A tuple-set descriptor - a logical conjunction of attribute-value pairs describing the tuples covered by the current CR.

(b) A candidate attribute - the attribute whose values will be used to expand the current CR if the candidate attribute is selected as the “best” one by a given evaluation function;

(c) A goal attribute - the attribute whose value must be predicted. This attribute is fixed during the execution of the KDD algorithm.

The output of Count by Group is shown in Figure 2(a). 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 tuple-set descriptor with candidate-attribute value A_i and goal-attribute value G_j.

Figure 2(b) shows a simple example of the output of Count by Group, based on two attributes of a company's database: Training and Job-status. It is assumed that the goal attribute *Training* can take on two values - whether or not a given employee has had some training in the company - and the candidate attribute *Job-status* can also take on two values - employee has part-time or full-time job. In this example, 100 employees were counted - i.e. 100 tuples satisfied the tuple-set descriptor. Intuitively, Figure 2(b) says that Job-status - in particular the part-time value of Job-status - is relevant to discriminate among employees with and without training. Hence, Job-status values (or at least its part-time value) could be selected (depending on the goodness of other candidate attributes) to compose the antecedent of classification rules which have the Training attribute as its consequent.

The processing of Count by Group essentially 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. This processing 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 computationally significant, since the construction of the matrix shown in Figure 2(a) is the bottleneck of KDD algorithms analyzing very large DBMSs. Hence, it is crucial to speed up the execution of this primitive. Moreover, Count by Group has a set-oriented semantics, since it operates on many tuples at a time in an order-independent fashion.

	G_1 . . . . . G_n	Total
A_1	C_{11} . . . . . C_{1n}	C_{1+}
.	. . . .	.
.	. . . .	.
.	. . . .	.
A_m	C_{m1} . . . . C_{mn}	C_{m+}
Total	C_{+1} . . . . . C_{+n}	C_{++}

(a) General structure of the output of the Count by Group primitive.

	some training	no training	Total
part-time	0	30	30
full-time	60	10	70
Total	60	40	100

(b) An example of the output of the Count by Group primitive.

Figure 2. Structure of the output of the Count by Group primitive.

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

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

Table 1. List of some CR-evaluation measures, as well as KDD algorithms computing them, that are supported by the primitive Count by Group.

Candidate-Rule evaluation measure	**KDD algorithm or system**
Information Gain	ICET [26]
Information Gain Ratio	C4.5 [21]
Reduction of Gini Diversity Index	CART [5]
Orthogonality between class vectors	O-BTree [9]
J-measure	CUPID [17]
Resubstitution Error	SWAP-1 [27]
Chi-squared	49er [31]
Cramer's Coefficient	regularity-based clustering [25]
Tau measure of association	KDW [19]
Category Utility	COBWEB [10]

Count by Group is generic, in the sense that it can be used to measure the quality of a Candidate Rule (CR) in a number of KDD algorithms. To show this generality, we list in Table 1 ten major quality measures of CRs that can be computed via this primitive. For each CR's quality measure we mention one KDD algorithm or system computing that measure. Due to space limitations, we do not discuss these measures here - see Table 1's references. For a discussion about how the Count values shown in Figure 2(a) are used to implement each of the CR-evaluation measures listed in Table 1, see [11].

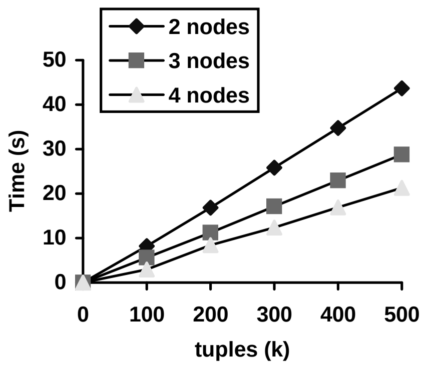
3.2. Computational Results.

In order to evaluate the efficiency in the exploitation of data parallelism when running Count by Group, we have done several experiments. We first describe the experiments running the primitive Count by Group alone, regardless of any KDD algorithm. Later in this Section we mention results of applying Count by Group to a full KDD algorithm.

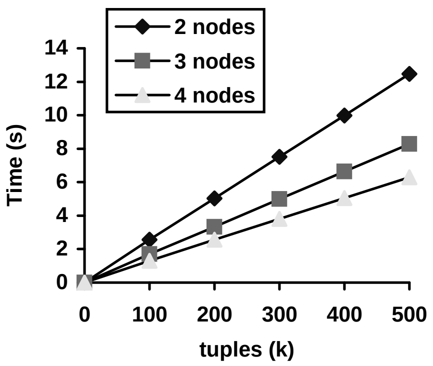
To evaluate the exploitation of data parallelism in Count by Group alone, this primitive was run on synthetic databases, which were randomly generated according to a uniform probability distribution. The number of tuples in the Mine relation varied from 100k tuples to 500k tuples. The number of attributes was fixed at ten (including the goal attribute). In all the experiments the candidate attribute had domain cardinality of 10 and the goal attribute had domain cardinality of 2 (common values in practice), so that the output of Count by Group was a matrix with 20 cells.

The experiments were done on an IBM SP2 running DB2 Parallel Edition [4]. Each SP2 node runs at about 250 Mflops and has 256 Mbytes of memory. We varied both the number of SP2 processor nodes used to process the primitive and the number of conditions in the *Where* clause of Query 1. The results are shown in Figures 3 and 4, where the horizontal axis shows the number of k tuples in the Mine relation and the vertical axis shows the processing time (measured in seconds). Figure 3 shows the time taken by Count by Group on the SP2 varying the number of processor nodes (2, 3 and 4 nodes). More precisely, Figure 3(a) shows the results for 0 conditions in the *Where* clause of Query 1 - i.e. the *Where* clause is void and all tuples of the Mine relation are counted by the primitive. Figure 3(b) shows the corresponding results for 2 conditions in the *Where* clause of Query 1. In both graphs the processing time scales linearly with the number of tuples.

Figure 4 shows the running time of Count by Group on the SP2 for a fixed number of processor nodes - 2 nodes in Figure 4(a) and 4 nodes in Figure 4(b) - varying the number of conditions in the *Where* clause of Query 1. The running times in the case of 2 and 4 conditions are almost co-linear, and they are significantly shorter than the running time in the case of 0 conditions. This is due to the fact that in the case of 2 and 4 conditions the number of tuples *counted* by the primitive is much smaller than the number of tuples in the Mine relation. In general, the larger the number of conditions in the *Where* clause of Query 1, the smaller the number of counted tuples, since that clause is a conjunction of conditions.

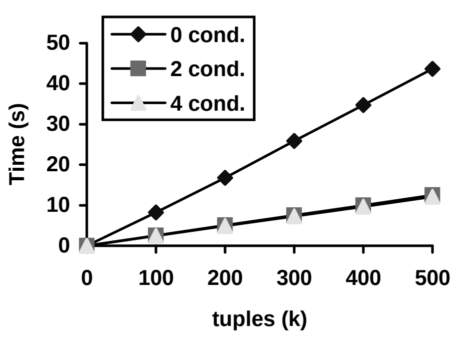


(a) 0 conditions on the *Where* clause.

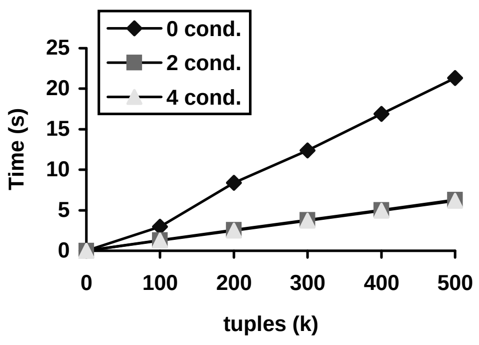


(b) 2 conditions on the *Where* clause.

Figure 3. Time on SP2, for a fixed number of conditions on the *Where* clause.



(a) 2 processor nodes.



(b) 4 processor nodes.

Figure 4. Time on SP2, for a fixed number of processor nodes.

We have also done some experiments to evaluate the efficiency in the exploitation of data parallelism in the context of a full TDIDT (Top-Down Induction of Decision-Tree) algorithm. The results of these experiments are described in detail in [13]. Overall the results of these experiments were consistent with the results reported above for the primitive Count by Group alone. In particular the speed up (Sp) achieved by the use of parallelism varied significantly across both the number of tuples in the Mine relation and properties of different versions of the TDIDT algorithm, but in general a roughly linear Sp was achieved. As expected, the Sp increased with the number of tuples, since the Client/Server communication overhead becomes proportionally smaller in this case.

4. A Primitive for Instance-Based Learning (IBL) Algorithms.

In the Instance-Based Learning (IBL) paradigm the KDD algorithm does not induce an explicit classification model. It simply stores the data set, or a subset of it, and uses the data rather than an induced model to classify new tuples (instances) [2].

Note that although conventional algorithms of the IBL paradigm do not induce any explicit model, they *do* provides an explanation about how the classification of a new tuple is done, which is important in the context of KDD. When a new tuple is classified, the system can show the user the classifying tuple, i.e. the most similar stored tuple (the "nearest neighbor") retrieved by the system. (Furthermore, IBL can also be used to generalize from individual tuples, e.g. by producing hyperrectangles in the instance space that are easily interpretable by a human user [22], or to extract highly summarized information in the form of prototypes [29].)

In essence, when a new tuple has to be classified, an IBL algorithm compares that tuple with all stored instances and retrieves the "nearest" (most similar) - or the k nearest, where k is a user-specified parameter - tuple(s) to the new one, as determined by a distance metric. Then the class of the retrieved tuple, or the prevalent class in the k retrieved tuples, is assigned to the new tuple. IBL is also known as the "(k-)Nearest-Neighbor" algorithm in Statistics.

4.1. Compute Tuple Distances: a Primitive to Compute Distance Metrics.

This Section introduces Compute Tuple Distances, a generic primitive to support the computation of a distance metric between a new tuple (to be classified) and all stored tuples. This is the primary data-intensive, time-consuming operation of IBL algorithms.

Let Dist(X,Y) be the distance between a stored tuple X and a new tuple Y. Let X_i and Y_i be the value of the i-th attribute of the corresponding tuple, i=1...M, where M is the number of attributes. In essence, the primitive computes the core of the distance metric Dist(X,Y) used by most IBL algorithms, as expressed by the formula (1), where W_X denotes the weight of the stored tuple X, W_i denotes the weight of the i-th attribute, and $dist_i(X_i,Y_i)$ denotes the distance between the values X_i and Y_i (see below). The tuple weight W_X usually indicates the quality of the stored tuple X as a classifying tuple - see e.g. [29] or [1]. The attribute weights W_i, i=1...M, indicate the relevance of each attribute for predicting the class of a tuple [28]. The exponent Exp in formula (1) is usually a small integer, typically set to 1 or 2. When Exp = 1 we have the Manhattan ("city-block") distance, and when Exp = 2 we have the Euclidean distance.

$$\mathrm{Dist}(X,Y) = W_X \left(\sum_{i=1}^{M} W_i \, \mathrm{dist}_i(X_i,Y_i)^{Exp} \right)^{1/Exp} \qquad (1)$$

for continuous attributes:

$$\mathrm{dist}_i(X_i,Y_i) = \mathrm{abs}(X_i - Y_i) \qquad (2)$$

for categorical attributes:

$$\mathrm{dist}_i(X_i,Y_i) = 0 \text{ if } X_i = Y_i \text{ ; or } \mathrm{dist}_i(X_i,Y_i) = 1 \text{ if } X_i \neq Y_i \quad (3)$$

Now we specify the primitive in terms of its input, output and processing. The primitive Compute Tuple Distances has four input parameters, namely:

(a) The attribute values of the new tuple - i.e. Y_i , i=1...M;

(b) The exponent Exp in formula (1).

(c) The attribute weights W_i , i=1...M.

(d) The tuples weights W_X for each tuple X stored in the Mine relation.

The above parameters (c) and (d) are optional, i.e. they are not used by several IBL algorithms. However, parameters (a) and (b) are essential in any IBL algorithm supported by the primitive. The output of Compute Tuple Distances is simply the set of distance values between the new tuple Y and each stored tuple X.

The processing of this primitive consists of computing formula (1). To compute $\mathrm{dist}_i(X_i,Y_i)$ we consider two cases. If the i-th attribute is continuous (ordinal), we use formula (2) where abs(x) denotes the absolute (unsigned) value of x. We assume that continuous attributes are normalized to avoid an attribute having a weight much larger than others in the distance metric just because its absolute values happen to be much larger. We use the well-known linear normalization method, i.e. $X_i = |X_i - X_{min}| / (X_{max} - X_{min})$, where X_{min} and X_{max} are respectively the minimum and the maximum values of attribute X_i's domain.

If the i-th attribute is categorical (non-ordinal), we use the overlap distance metric, which essentially counts the number of distinct attribute values, as given by formula (3). However, the mapping of formula (3) into SQL is not trivial, since we cannot specify a conditional command (*if*) within an SQL query. Our solution is as follows.

We assume, without loss of generality, that the attribute values are stored in the database as small integer numbers, i.e. alphanumeric values such as "low" and "high" are converted to numerical codes such as 1 and 2 as a pre-processing step. Then we compute $\mathrm{dist}_i(X_i,Y_i)$ as follows: $\mathrm{dist}_i(X_i,Y_i) = \mathrm{ceiling}(\mathrm{abs}(X_i - Y_i)/c)$, where c is any constant equal to or greater than the cardinality of the i-th attribute's domain (for practical purposes, say c = 100). The function ceiling(x) returns the smallest integer number which is equal to or greater than x. Note that whenever $X_i \neq Y_i$ the function call ceiling(abs(X_i - Y_i)/c) will return the value of its parameter rounded up to 1. Both ceiling() and abs() functions are available in major DBMSs such as Oracle 7.X.

Query 2 shows the general structure of a set-oriented IBL algorithm implemented via the primitive Compute Tuple Distances. Conceptually speaking, Query 2 can be divided into two sequential steps. First, the SELECT within the WHERE...IN clause is executed to select the nearest stored tuple to the new tuple. Then, the class of the selected tuple is retrieved by the outer SELECT. A detailed example of the application of Compute Tuple Distances to support an IBL algorithm (without tuple weighting nor attribute weights) is shown in Query 3. In this example the Mine relation has two predicting attributes: the first one categorical and the second one continuous. The Manhattan distance is used.

```
SELECT    class
FROM      Mine_relation
WHERE     Dist(X,Y)   IN
(SELECT  MIN(Dist(X,Y))  FROM Mine_relation)
```

Query 2. The general structure of a set-oriented IBL algorithm mapped into SQL.

```
SELECT    class
FROM      Mine_relation
WHERE     ceiling(abs(X1 - Y1)/k) + abs(X2 - Y2)  IN
(SELECT  MIN(ceiling(abs(X1 - Y1)/k) +
          abs(X2 - Y2))  FROM  Mine_relation)
```

Query 3. Example of a set-oriented IBL algorithm mapped into SQL.

Note that the primitive Compute Tuple Distances is very computationally significant, since it accesses all the attributes and all the tuples of the Mine relation. In this sense, it has an ideal potential for data parallelism. Moreover, it obviously has a set-oriented semantics, since it operates on many tuples at a time in an order-independent fashion.

To show the generality of Compute Tuple Distances, we list in Table 2 some IBL or hybrid IBL/RI algorithms whose data-intensive, distance-metric computation is supported by this primitive. Due to space limitations, for each algorithm we only mention whether or not it uses attribute weights or tuple weights in the distance metric - the optional input parameters of the primitive. Refer to the original references for details of these algorithms. A more detailed discussion about how Compute Tuple Distances supports these algorithms can be found in [11]. The last two systems in Table 2 are actually general methodologies for integrating RI and IBL algorithms, so that the underlying IBL algorithm can use or ignore attribute weights or tuple weights in their distance metric.

Table 2. List of some IBL algorithms supported by the primitive Compute Tuple Distances.

Algorithm or system.	weighted tuples?	weighted attributes?
IB1, IB2, IB3 [2]	no	no
GA-WKNN [16]	no	yes
TIBL [29]	yes	no
CoRCase [1]	yes	yes
Integrated rule/case base [24]	possibly	possibly
Integrated decision-tree/CBR [3]	possibly	possibly

4.2. Computational Results.

Experiments were performed with three databases. Two of them were obtained from the LIACC at the University of Porto and were used in the Esprit project Statlog [18], viz. the Shuttle and the Letter data sets. In the former the predicting attributes concern the position of radiators in a NASA space shuttle and the classes are the appropriate actions to be taken during a space shuttle flight, whereas in the latter the goal is to recognize a letter as one of the 26 letters of the alphabet. The third database used in our experiments was obtained from the Labour Force Survey (LFS) data, produced by the UK's Department of Employment. The goal attribute is Managerial Status, which can take on two values (or classes), namely manager/supervisor or employee. The number of [tuples; attributes] in the Mine relation for each of these databases is respectively [43,500; 9], [15,000; 16] and [113,432; 11]. Although these data sets are not huge, they are large enough for testing purposes and they are one order of magnitude larger than the data sets reported in the majority of the IBL literature (typically less than 5,000 tuples).

We did experiments comparing a MIMD machine, namely the White Cross WX9010 parallel database server, against an Ingres 6.4 DBMS running on a 25-MHz, 24-MBytes-RAM Sun IPC. In all experiments, our results refer to main memory databases (i.e. disk 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 12 MIPS and 25 MHz [6]. Note that each transputer belongs to the same technology generation and has roughly the same MIP rate as the Sun IPC workstation. Ten out of the 12 transputers are actually used to process the query in parallel. The WX9010 is a main-memory shared-nothing machine. It is a back-end SQL server attached to an Ethernet LAN. Although this machine is a relatively small, entry-level system, it is interesting for our experiments for two reasons. First, it has a very high rate of scanning tuples: 3 million tuples/sec. Second, it was specifically designed for Decision Support Systems applications (including KDD), rather than for OLTP applications.

In all the experiments we used the Manhattan Distance. The results are shown in Tables 3 and 4. Table 3 presents speed up results for a simple IBL algorithm (called IBL-1), which classifies test tuples according to Query 2. Table 4 presents speed up results for a more elaborate IBL algorithm (called IBL-2), which classifies test tuples according to Query 2 extended to consider attribute weights. In both Tables the columns have the following meaning. The first column indicates the database. The second and third columns show the average time (in seconds) taken by the corresponding SQL query in Sun/Ingres and on the WX9010, respectively. Unfortunately, the current version of the WX9010 system does not allow the direct execution of Query 2 as a single query. Hence, we estimated the time that Query 2 would take (if it was implemented as a single query) on the WX9010 by executing simpler queries on this machine. The fourth column shows the speed up (Sp) of the WX9010 over Sun/Ingres.

We are aware that it is common for papers on experiments with parallelization to show the behavior of the algorithm on a range of processor numbers. However, in our case the WX9010 is a commercially-available parallel database server to whose firmware we do not have access, so that we cannot control the number of processors used by the machine. Hence, our experiments investigate the behaviour of the speed up for different databases. This is important because it allows us to study the behavior of the speed up for different numbers of categorical and continuous attributes and different number of tuples.

Table 3. Speed up for (unweighted) IBL-1.

Database	Sun (s)	WX (s)	Sp
Letter	45.1	7.1	6.3
Shuttle	59.8	6.4	9.3
LFS	519.4	27.4	19.9

Table 4. Speed up for (weighted) IBL-2.

Database	Sun (s)	WX (s)	Sp
Letter	56.3	19.0	3.0
Shuttle	79.9	24.8	3.2
LFS	506.3	50.8	9.9

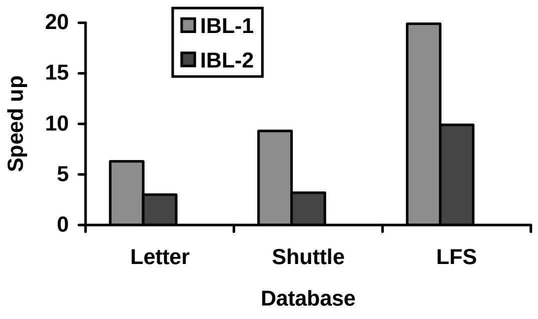


Figure 5. Speed up of IBL-1 and IBL-2

In both Tables 3 and 4, as expected the Sp is smaller in the case of the Letter database. Due to the small size of this database, the communication overhead between the client and the WX9010 server represents a significant part of the total query processing time, so reducing the Sp. However, the Sp is larger in the Shuttle database and is particularly large in the LFS database, the largest database used in the experiments.

In general attribute weighting turned out to significantly increase query processing time, particularly for the WX9010. This can be seen in Figure 5, which compares the Sp associated with the two IBL algorithms - i.e. it compares the Sp shown in the last column of Table 3 against the Sp shown in the last column of Table 4, for each database. The Sp is significantly smaller in the case of the attribute-weighting algorithm IBL-2 (for all the three databases). The large drop in the Sp (e.g. from 9.3 for IBL-1 to 3.2 for IBL-2 in the case of the Shuttle database) is somewhat surprising. The reason for this Sp drop seems to be that the SQL query of IBL-2 is more complex, since each term $dist_i$ in Equation (1) must be multiplied by the corresponding weight W_i. In theory, this should not significantly reduce the Sp, since the introduction of attribute weights in Equation (1) does not reduce its potential for the exploitation of data parallelism. However, the WX9010, unlike Ingres, is very sensitive to the arithmetic complexity of Equation (1). Hence, although the Data-in-Memory techniques of the WX9010 are very effective for *relational selection* operations, they seem not to be so effective for “complex” arithmetic operations.

5. Conclusions.

We have proposed generic, set-oriented primitives for two important KDD paradigms, namely Rule Induction (RI) and Instance-Based Learning (IBL). This has allowed us to create a set-oriented, data-parallel framework for KDD which improves the scalability of KDD algorithms and uses Parallel Database Servers to significantly reduce the processing time of KDD algorithms. We have demonstrated the generality of the proposed primitives and evaluated the efficiency in the exploitation of data parallelism. Note that due to their generality, a significant speed up in the execution of the proposed primitives (by exploiting parallelism) will lead to a significant speed up in a number of different KDD algorithms.

To measure the efficiency in the exploitation of data parallelism we did several experiments. The processing-time results reported in this paper can be summarized as follows. The running time of the RI primitive on an IBM SP2 scales linearly with the number of tuples, at least for a small number of processor nodes (Figures 3 and 4). This is consistent with the results reported in [13] for a decision-tree-building algorithm.

When running a simple IBL algorithm (without attribute weighting), the 12-node WX9010 achieves a speed up of about one order of magnitude over the Sun uniprocessor (Table 3). When running a more elaborate IBL algorithm (with attribute weighting) the speed up is reduced by a factor of about two (Table 4 and Figure 5), indicating that the WX9010 is quite sensitive to the complexity of the arithmetic expression used to compute the distance metric.

Several directions for future work are possible. It would be interesting to extend these experiments for other databases, other Parallel Database Servers (PDS) and other KDD algorithms, to generalize the results reported above. In addition, so far we have focused on a generic framework to integrate KDD algorithms, relational databases and parallel processing. Our studies could be extended to a deeper investigation about how to optimize the database queries underlying the proposed primitives (e.g. what is the best data partitioning strategy to minimize inter-processor communication costs). Note that conclusions stemming from this kind of investigation tend to be dependent on a specific PDS architecture. In practice, however, since a given organization usually has just one or a couple of different PDS, it would make sense for that organization to make a deeper study of inter-processor communication costs when executing the proposed primitives on its particular PDS. Finally, the proposed set-oriented, primitive-based framework might be extended for other KDD paradigms, such as genetic algorithms and neural networks. A preliminary work about a set-oriented, generic primitive for the genetic programming paradigm is discussed in [12].

Acknowledgments.

I thank Simon H. Lavington, for valuable discussions; Dominicus R. Thoen, for doing the experiments reported in Section 3.2 and for helping doing the experiments reported in Section 4.2; Steve O'Connel, from the University of Southampton's Dept. Of Electronics and Computer Science, for his assistance in the use of the IBM SP2; and Steve Hassan, for his assistance in the use of the White Cross WX9010. During this research I was financially supported by the Brazilian government's CNPq, process number 200384/93-7.

References.

[1] G. Agre. Knowledge-base maintenance as learning two-tiered domain representation. *Proc. 1st Int. Conf. CBR (ICCBR-95). LNAI 1010*, 109-120. 1995.

[2] D.W. Aha, D. Kibler and M.K. Albert. Instance-Based Learning Algorithms. *Machine Learning* 6, 1991, 37-66.

[3] E. Auriol, M. Manago, K.-D. Althoff, S. Wess and S. Dittrich. Integrating induction and case-based reasoning: methodological approach and first evaluations. *Proc. 2nd European Workshop on CBR (EWCBR-94)*. LNAI 984, 18-32. 1994.

[4] C.K. Baru, G. Fecteau, A. Goyal, H. Hsiao, A. Jhingran, S. Padmanabhan, G.P. Copeland and W.G. Wilson. DB2 Parallel Edition. *IBM Systems Journal*, 34(2), 292-322, 1995.

[5] L. Breiman, J.H. Friedman, R.A. Olshen and C.J. Stone. *Classification and Regression Trees*. Pacific Groves, CA: Wadsworth, 1984.

[6] 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, 940443.)

[7] W.W. Cohen. Fast effective rule induction. *Proc. 12th Int. Conf. Machine Learning (ML-95)*, 115-123. 1995.

[8] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth and R. Uthurusamy. (Eds.) *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press. 1996.

[9] U.M. Fayyad, S.G. Djorgovski and N. Weir. Automating the analysis and cataloging of sky surveys. In [8], 471-493. 1996.

[10] D.H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2, 1987, 139-172.

[11] A.A. Freitas. *Generic, set-oriented primitives to support data-parallel knowledge discovery in relational databases systems.* Ph.D. thesis. University of Essex, UK. July 1997.

[12] A.A. Freitas. A genetic programming framework for two data mining tasks: classification and generalized rule induction. To appear in: *1997 Genetic Programming Conf. Proc.* Stanford University, CA, USA. July 1997.

[13] A.A. Freitas and S.H. Lavington. Using SQL primitives and parallel DB servers to speed up knowledge discovery in large relational databases. In: R. Trappl. (Ed.) *Cybernetics and Systems'96: Proc. 13th European Meeting on Cybernetics and Systems Research*, 955-960. Vienna: Austrian Society for Cybernetics Studies, 1996.

[14] A.A. Freitas and S.H. Lavington. Speeding up knowledge discovery in large relational databases by means of a new discretization algorithm. R. Morrison & J. Kennedy (Eds.) Advances in Databases: *Proc. 14th British Nat. Conf. on Databases (BNCOD-14)*. Edinburgh, UK. July 1996. LNCS 1094, 124-133.

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

[16] J.D. Kelly Jr. and L. Davis. A hybrid genetic algorithm for classification. *Proc. 12th Int. Joint Conf. on AI (IJCAI-91)*, 645-650. 1991.

[17] J. Mallen and M. Bramer. CUPID - an iterative knowledge discovery framework. *Expert Systems'94.* 1994.

[18] D. Michie, D.J. Spiegelhalter and C.C. Taylor. *Machine Learning, Neural and Statistical Classification*. New York: Ellis Horwood, 1994.

[19] G. Piatetsky-Shapiro and C.J. Matheus. Measuring data dependencies in large databases. *Proc. AAAI-93 Workshop on Knowledge Discovery in Databases*, 162-173. Washington, July 1993.

[20] F.J. Provost and J.M. Aronis. Scaling up inductive learning with massive parallelism. *Machine Learning* 23(1), Apr./96, 33-46.

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

[22] S. Salzberg. A nearest hyperrectangle learning method. *Machine Learning* 6, 1991, 251-276.

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

[24] J. Surma and K. Vanhoof. Integrating rules and cases for the classification task. *Proc. 1st Int. Conf. CBR (ICCBR-95). LNAI* 1010, 325-334. 1995.

[25] M. Troxel, K. Swarm, R. Zembowicz and J.M. Zytkow. From law-like knowledge to concept hierarchies in data. *Proc. AAAI-94 Workshop on Knowledge Discovery in Databases (KDD-94)*, 193-204. 1994.

[26] P.D. Turney. Cost-sensitive classification: empirical evaluation of a hybrid genetic decision tree induction algorithm. *Journal of Artificial Intelligent Research,* 2, Mar. 1995, 369-409.

[27] S.M. Weiss and N. Indurkhya. Optimized rule induction. *IEEE Expert,* Dec. 1993, 61-69.

[28] D. Wettschereck and D.W. Aha. Weighting features. *Proc. 1st Int. Conf. CBR. (ICCBR-95). LNAI* 1010, 347-358.

[29] J. Zhang. Selecting typical instances in instance-based learning. *Proc. 9th Int. Workshop on ML (ML-92),* 470-479. 1992.

[30] Y. Zhuge, H. Garcia-Molina, J. Hammer and J. Widom. View maintenance in a warehousing environment. *Proc. 1995 ACM SIGMOD Int. Conf. Management of Data*, 316-327. 1995.

[31] J. Zytkow and R. Zembowicz. Database exploration in search of regularities. *Journal of Intelligent Information Systems*, 2(1), Mar. 1993, 39-81.