A data-parallel primitive for high-performance knowledge discovery in large databases

A A Freitas
and
S H Lavington

Internal Report CSM-242, May 1995

Department of Computer Science
University of Essex
Colchester CO4 3SQ
UK

Tel: 01206 872677
e-mail: lavington@essex.ac.uk

A data-parallel primitive for high-performance knowledge discovery in large databases.

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

Abstract

Efficiency is crucial in KDD (Knowledge Discovery in Databases), due to the huge amount of data stored in current databases. We argue that high efficiency in KDD can be achieved by combining two approaches, namely encapsulating KDD functionality within standard DBMS operations and using parallel processing. Hence, KDD tasks can be executed on a back-end SQL server, e.g. a parallel DB machine. We propose a KDD primitive (a set of basic operations) which underlies the candidate-rule evaluation procedures of many KDD algorithms. We compare and analyze the time required to carry out this primitive on three different computational architecture, viz. a conventional workstation and two parallel DB machines. The main advantages of encapsulating a KDD primitive in a parallel DB server are automatic parallelization and the run-time speed which can be achieved through parallel processing.

May 1995.

1. Introduction.

In broad terms, Knowledge Discovery in Databases (KDD) is the "non-trivial extraction of implicit, previously unknown and potentially useful information from databases [5]." This paper addresses the question of computational efficiency in KDD. Efficiency is crucial in KDD, since next-generation DBs (databases) are estimated to have on the order of 10^{12} bytes [18]. Already, organizations such as large banks and department stores wish to carry out KDD activities on several gigabytes of data.

The large amount of data stored in a DB introduces several difficulties for a KDD system, such as the need for efficient access to data, existence of many irrelevant attributes, etc. On the other hand, DB technology offers several means to efficiently access large amounts of data, such as query optimization techniques, attribute indexing, domain knowledge available in the form of integrity constraints, statistical information about the data, etc. [3]. These facilities are ignored by many AI-based KDD systems, which simply extract a data sample from the DB and apply a machine learning or statistics algorithm to that data set. In principle, the KDD process would be more efficient if there was a tighter integration between the KDD algorithm and the DBMS. Moreover, this approach has the advantage of compatibility with the support usually given to commercial DBMSs.

We argue that significantly greater efficiency in KDD tasks can be achieved by combining two approaches. First, some KDD operations can be encapsulated within the DBMS, in order to use efficiency-oriented DBMS facilities. Second, parallel processing can be used to speed up KDD operations. These two approaches can be combined by storing the DB on a parallel DB machine and carrying out KDD tasks on that machine, via standard DBMS commands.

The scope of this paper is limited to relational DBs, since the relational model has gained increased popularity in the commercial world. Moreover, this model is suitable for parallelization, due mainly to the set-theoretic concept of relation - which allows the processing of many (unordered) tuples at a time.

This paper is organized as follows. Section 2 presents a review of some KDD concepts relevant for an understanding of the paper. Section 3 proposes a data-parallel KDD primitive

for the candidate-rule evaluation operation of KDD algorithms. Section 4 presents results about the implementation of the primitive in parallel DB computers. Section 5 discusses some ways of improving the efficiency in the execution of that KDD primitive. Finally, Section 6 presents the conclusions.

2 Knowledge Discovery in Databases - the essential components.

Matheus et al. [10] present an idealized KDD model consisting of six basic components, namely DB Interface, Knowledge Base, Focus (determining which tuples or attributes should be analyzed), Pattern Extraction Algorithm, Evaluation (determining the relevance of the extracted patterns and filtering out uninteresting patterns) and Controller (controlling the Focus, Pattern Extraction and Evaluation components). In actual KDD systems, some of these components are usually mixed. For example, the Evaluation component - and, less often, the Focus component - is(are) often embedded in the Pattern Extraction Algorithm. This latter is the core of a KDD system and will be called the KDD algorithm. We now briefly discuss some KDD-model components necessary for an understanding of this paper.

Concerning the DB Interface, there are two general approaches. The first one is to extract a relatively small sample of data from the original, large DB and to apply a KDD algorithm directly on that unstructured data sample, without interfacing with the DB. The second one is to interface with the DB, so that the KDD algorithm is implemented through DB queries, such as the algorithm described in [7]. Most KDD systems follow the first approach, which is simpler. However, we argue that the second approach is more promising, since it allows the KDD algorithm to use several facilities offered by a DBMS, such as an effective query optimizer, efficient data access methods, statistical information about the data, etc. [3].

We assume that at least part of the Focus component is independent of the KDD algorithm. Hence, 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. Ideally, the result of this query should be stored as a snapshot. If the DBMS does not support snapshots (which are not supported by

standard SQL), that result is simply stored as a new relation, called the Mine relation. We use this term throughout this paper, for generality.

The question of how the task-relevant data set is specified - either by the user or by a (partially) autonomous Focus algorithm - is beyond the scope of this paper. The important point is that the task-relevant data set is physically stored as a *single* relation, i.e. the Mine relation. Note that this relation can contain information redundancy (corresponding to a week degree of normalization), since it may have been generated through joins of relations in the whole DB. It is well-known that redundancy can lead to undesirable update anomalies. However, this is not a serious concern in our case, once the Mine relation is "read-only", i.e. it will not be updated during the whole KDD algorithm. Thus, storing the task-relevant data set as a single relation, we can achieve a significant gain in performance, since we avoid the use of computationally-expensive join operations during the KDD algorithm. This approach also offers the advantage that the Mine relation can be created with access methods tailored for the KDD task, rather than with access methods used in the current relations of the DB (designed for general DB applications). This idea will be further discussed in Section 5. Note that we are implicitly assuming that the KDD algorithm is relatively complex, so that a given tuple may be accessed several times by the algorithm. Clearly, the overhead of separately storing the Mine relation is not cost-effective if the algorithm involves a single scan of all task-relevant data set.

Unless mentioned otherwise, we assume that a KDD algorithm discovers classification rules of the form "if D then C", where D is a description formed by a conjunction of attribute-value conditions and C is a class (value of a goal attribute). KDD algorithms can be cast as a state-space search, where problem states correspond to a candidate rule (CR) and operators (state-transition rules) correspond to generalization/specialization operations [6]. Hence, a KDD algorithm can be viewed as the iterative process of selecting a CR (according to a CR-evaluation function), expanding it (applying generalization/specialization operators to it, so generating new CRs) and evaluating the extended (just-generated) CRs. This process is repeated until a solution(s) - i.e. a satisfactory CR(s) - is(are) found. A CR can be

evaluated along several dimensions. The most used criteria are predictive accuracy and statistical significance.

3 A data-parallel KDD primitive for Candidate Rule evaluation.

We distinguish between control parallelism and data parallelism. Note that the exploitation of data parallelism in search is particularly interesting in KDD applications, where CR (Candidate Rule) evaluation has a computationally-expensive nature since it involves accessing a large number of tuples.

In this Section, we propose a KDD primitive (a set of basic operations) to exploit the data parallelism inherent in CR-evaluation operations. We assume that the Mine relation is stored in a back-end parallel DB machine (or DB server), which is connected to a host processor, or to several client workstations. The host (or workstation) is in charge of selecting the next CR to be expanded and expanding it (generating new CRs). However, in order to carry out operations involving access to the DB, such as CR evaluation, the host sends queries to the DB server. We assume that the DB server offers an SQL or SQL-like interface, since SQL is the de facto standard of relational DB query languages. We also assume that all attributes are symbolic, i.e. they can take on a relatively small number of discrete values. Continuous attributes would be converted to symbolic ones in a pre-processing phase [17], [20].

3.1 Satisfying the requirements.

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, as indicated in Section 3.3, and have come to the conclusion that their central activity can be expressed by a primitive (a set of basic operations) which we will call Count by Group. This primitive 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. Count by Group has three input parameters, namely an example-set descriptor, a candidate attribute, and a goal

attribute. The output of Count by Group is shown in Figure 1(a). This is essentially an m x n matrix extended with totals of rows and columns, which is called a contingency table in statistics. 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 Ai and goal-attribute value Gj. It is important to note that this primitive can be implemented in a declarative style by the SQL query shown in Figure 2, followed by a trivial computation of rows and columns totals.

It remains to be seen whether the Count by Group primitive is capable of efficient parallel execution. We describe some experiments on parallel SQL implementations in Section 4, which lead us to suggest a modification to the primitive. However, before proceeding to Section 4, it is first necessary to describe our basic primitive in more detail.

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

1(a)

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

1(b)

Figure 1 - Structure of the output of the Count by Group primitive.

1(a) General structure. 1(b) Example of one output.

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

Figure 2 - SQL query underlying the Count by Group primitive .

3.2 An example.

Figure 1(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 example-set descriptor. Intuitively, Figure 1(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.

3.3 Generality.

Count by Group underpins the CR-evaluation operation of many KDD algorithms, since its output can be used to compute many different kinds of correlation measures between a candidate attribute and a goal attribute. Typically, each iteration of a KDD algorithm would execute this primitive several times, computing the goodness of several candidate attributes, to select the best one. For example, in decision-tree learners [15], Count by Group can be used to compute several splitting-evaluation measures[12]. Note that decision-tree learners are used by many KDD systems, e.g. KDW [14] and INLEN [11], and they are the very core of many KDD algorithms, e.g. CDP [1] and Carper [16]. In decision-tree learners, the example-set descriptor is a conjunction of attribute-value conditions that represents the path leading from the root node to a given node of the decision tree. Hence, the example-set descriptor is an intensional (rather than extensional) description of the set of examples belonging to a given node. As another example, Count by Group can be used in the CN2 algorithm [2] to compute an information-theoretic measure of attribute goodness. Count by Group can also be used as a building block in the design of a new algorithm. Moreover, this primitive can be used by many statistical methods, such as Bayesian classifiers [19] and statistical significance tests - particularly chi-square tests, which are popular in KDD systems, e.g. 49er [20].

4 Evaluating the primitive on parallel DB computers.

In this Section, we first discuss some classes of parallel platforms. Then we show how the Count by Group primitive can be executed on two parallel DB machines, the White Cross Server WX9010 and the IFS/2 research prototype. We present results comparing the performance of these two machines with a Sun IPC workstation. All three platforms have the same clock-rate (25 MHz) and represent the same technololgy generation.

4.1 Classes of parallel platform.

There are now several types of parallel platform commonly available for commercial data processing. At a recent conference [8], case-studies were given of ten parallel machines successfully used for revenue-earning applications. The most common architectural category was MIMD (e.g. IBM SP2, ICL Goldrush, Meiko, White Cross, Teradata), followed by SMP (e.g. Cray CS6400, Tandem, Sequent) and then SIMD (e.g. MasPar, DAP). At Essex we have a White Cross WX9010 parallel dataserver (see below), so this formed the basis for our MIMD experiments. This machine is interesting because it has a very high rate of scanning tuples - the entry-level WX9010 scanning 2.4 million tuples/sec. We compared the WX9010 with two other plataforms of similar technology generation. One was a uniprocessor, namely a Sun IPC running at 25 MHz with 24 MBytes RAM; the other was an experimental server, called the IFS/2, as described below.

4.2 The machines used for measurement.

4.2.1 White Cross. The White Cross WX9010 entry-level system (release 3.0) is based upon 12 T425 transputers, each with 16 Mbytes RAM, each rated at about 10 MIPS and running at 25 MHz [13]. The WX9010 uses Data-in-Memory techniques for fast processing. Data is loaded into the 192 MBytes of RAM of the entry-level configuration before processing begins. (The 192 MBytes RAM is backed by four 2 GBytes discs, contained within the White Cross cabinet.) The White Cross is a back-end SQL server attached to an Ethernet LAN. The SQL front-end resides on a Sun Workstation client.

In the WX9010, each processor independently *selects* its local tuples satisfying the example-set descriptor, and the *Group by* and *Count* operations are computed in two steps. First, each processor counts the selected tuples in its local memory for each partition (i.e. each distinct pair of grouping-attribute values) and orders the local results by the two grouping attributes. Second, the partial results are merged as they are sent to a query manager processor.

Note that the second step above is performed in a sequential fashion. This step could be done in a (potentially faster) parallel fashion, by using a scheme similar to the one used in the Gamma DB machine [4]. In Gamma, processors redistribute the partial results - by hashing on the grouping-attribute values - to collect the partial results for each partition at a single processor and compute the final result for each partition.

4.2.2 The IFS/2 research prototype - This is a back-end parallel database server [9], linked by a SCSI channel to a Sun Workstation. The IFS/2 architecture consists of two levels. At the higher level, called inter-node level, the IFS/2 consists of N1 nodes interconnected via transputer links. At the lower level, called intra-node level, each node consists of a node controller (essentially a Transputer T425 and RAM) and an array of N2 SEs (Search Engines) operating in SIMD mode. On the prototype, N1 =3 and N2 = 9. The IFS/2 uses hardware hashing to implement a pseudo Associative (i.e. Content Addressable) Memory. The IFS/2 memory is horizontally partitioned into bins or rows. For KDD applications the data stored on each bin are distributed across all SEs, and each bin is conceptually accessed as a single logical unit (simultaneously accessed by all SEs). Each SE has a single hardware comparator to carry out the match between the interrogand and addressed tuples.

When a tuple is inserted into the IFS/2, all hash-bit values are known - so that a single bin address is produced. However, in later accesses, some of the hash-bit values will probably be unknown (i.e. the corresponding attributes are not specified in the query), so that a set of bin addresses wil be produced. This obviously increases the associative-search time, since all those bins will have to be scanned.

In the case of the Count by Group primitive, the selection of tuples satisfying the example-set descriptor is done in parallel by all SEs. The operation of grouping together tuples with the same value of the grouping attributes can also be done in parallel by all SEs, by hashing on the grouping-attribute values. However, the operation of counting the number of tuples in each group cannot be done at the SE level, and must be done at the node-controller (Transputer) level, which limits the degree of parallelism (recall that the IFS/2 prototype has only three node-controllers).

4.3 Results.

The results are presented in Tables 1, 2 and 3. The number of tuples in the Mine relation was varied from 20000 to 100000. The number of attributes was fixed at 10. Each row of Tables 1,2 and 3 contains results averaged over 30 experiments, i.e. 30 randomly-generated (with a uniform probability distribution) data sets. The cardinality of the domain of the attributes - i.e. the number of distinct values that an attribute can take on - varied from two to ten distinct values. More precisely, out of the ten attributes, four attributes had a domain cardinality of 2, four attributes had a domain cardinality of 6, and two attributes had a domain cardinality of 10. In all experiments, the *Group by* attributes were attributes with cardinality of 2, so the output of the primitive was a 2x2 contingency table, which is in general the contingency-table size most encountered in practice. For each data set, we measured the time taken by Count by Group with 0, 2 and 4 "*Where* attributes", i.e. attributes specified in the *Where* clause of Figure 2. Thus, corresponding rows (i.e., rows with the same number of tuples) in Tables 1, 2 and 3 refer to the same set of 30 data sets.

In these tables, the first two columns show the number of tuples randomly generated, and the total number of tuples counted in all *Group by* partitions - i.e. the number of tuples satisfying the example-set descriptor. In general, the larger the number of *Where* attributes, the smaller the number of counted tuples, since fewer tuples satisfy the example-set descriptor. In particular, when the *Where* clause is empty (Table 1), all tuples are counted, so that the first two columns of Table 1 are equal. The other three columns in Tables 1, 2 and 3 show the elapsed time (in seconds) to execute Count by Group in the corresponding

machine. Unfortunately, we ran out of memory on the IFS/2 prototype when the number of generated tuples was greater than 40000. All experiments were done when a single user was using the system. Moreover, in all experiments, disc activity is excluded from our measurements - i.e. our results refer to main-memory databases.

Table 1 - Results for 0 attributes specified in *Where* clause.

Generated	Counted	Sun	WX9010	IFS/2
20000	20000	25.7	4.53	3.6
40000	40000	52.6	4.78	7.0
60000	60000	78.2	5.19	
80000	80000	100.3	5.39	
100000	100000	127.5	5.80	

Table 2 - Results for 2 attributes specified in *Where* clause.

Generated	Counted	Sun	WX9010	IFS/2
20000	1675	6.1	4.59	0.55
40000	3337	12.1	4.59	0.99
60000	4974	21.6	4.84	
80000	6657	26.3	4.74	
100000	8341	32.2	4.90	

Table 3 - Results for 4 attributes specified in *Where* clause.

Generated	Counted	Sun	WX9010	IFS/2
20000	28	3.8	4.69	0.217
40000	56	7.8	4.69	0.224
60000	83	15.9	4.95	
80000	110	18.7	4.81	
100000	137	21.7	4.97	

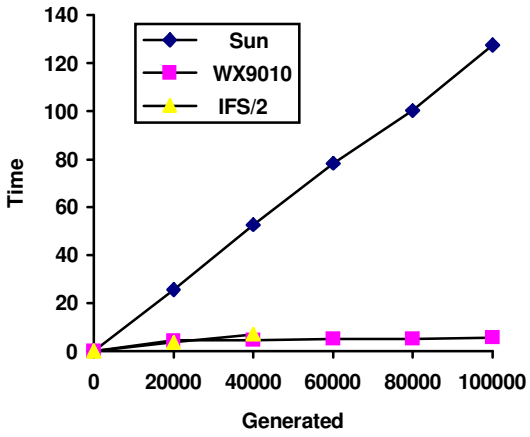

Figure 3 - Results for 0 Where attributes.

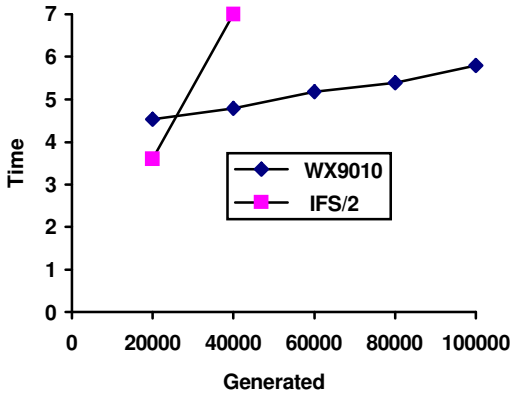

Figure 4 - Results for 0 Where attributes, comparing WX9010 and IFS/2.

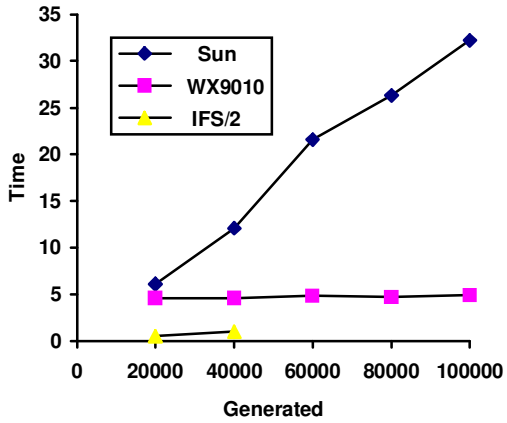

Figure 5 - Results for 2 Where attributes.

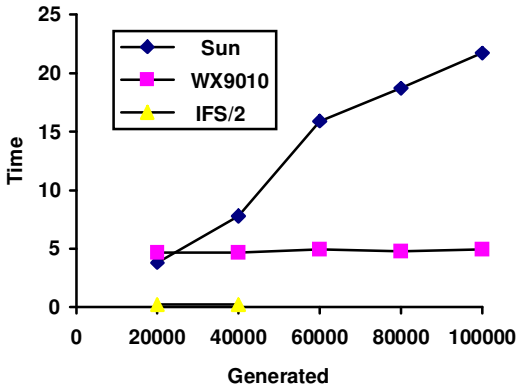

Figure 6 - Results for 4 Where attributes.

The results presented in Table 1 are also presented as graphs in Figures 3 and 4. Figure 4 is simply a "zoom" of the results for the WX9010 and IFS/2, since the lines representing the results for these two machines cannot easily be distinguished in the graph of Figure 3. The results of Tables 2 and 3 are also presented as graphs in Figures 5 and 6, respectively.

In general, both the WX9010 and the IFS/2 machines obtain a speed up of about one order of magnitude over the Sun workstation. An analysis of the computational activity on the WX9010 and the IFS/2 indicates that performance should scale reasonably well with increased number of processors (or nodes, in the case of the IFS/2) [13], [9]. It is interesting to compare the performance of the WX9010 and the IFS/2, as follows.

The WX9010 is particularly fast in the case of an empty *Where* clause (corresponding to queries which count all tuples of the Mine relation), as shown in Table 1. In this table, the WX9010's time is linear on the number of tuples. Note that, in Table 1, the WX9010 is faster than the IFS/2 when the number of tuples is over 20000 tuples (the break-even point is about 25000 tuples).

However, in the case of 2 and 4 *Where* attributes, shown in Tables 2 and 3, the IFS/2 is faster than the WX9010. Although we lack results for the IFS/2 with more than 40000 tuples, the partial results in Tables 2 and 3 suggest that the WX9010 would be faster than the IFS/2 only when the number of generated tuples is far larger than 100000. This occurs because in Tables 2 and 3 the number of *counted* tuples is a small fraction of the number of generated tuples (Mine relation size). Note that the very high scanning rate of the WX9010 - 2.4 million tuples/sec - is very useful to *select* the tuples satisfying the example-set descriptor of the Count by Group primitive. However, the *selected* tuples still have to be *grouped* and *counted*, which (unlike the *selection* operation) requires communication among the processors. On the other hand, this communication overhead among processors is minimized on the IFS/2, due to the SIMD nature of its Search Engines.

Note that the IFS/2 is particularly fast in the case of 4 Where attributes (corresponding to queries which count tens or a few hundred tuples), as shown in Table 3. This was expected, since the more the number of *Where* attributes, the smaller the number of unknown hash bit values, and so the shorter the associative-search time. Actually, in Table 3 the IFS/2's time is

widely dominated by a nearly-constant (independent of the number of tuples) communication overhead - i.e. the time to translate the SQL query into lower-level commands and to send the translated query down to the IFS/2 - of 0.2 secs. For instance, in the row with 40000 tuples, 56 tuples are counted in 0.224 secs, i.e. 0.2 secs of communication time and 0.024 secs of associative-search time.

5 Improving the computational efficiency of Count by Group.

In this Section we suggest some ways of improving the computational efficiency of the Count by Group primitive.

5.1 Increasing the granularity of the primitive.

The Count by Group primitive is a very repetitive operation in KDD algorithms. Hence, the query shown in Figure 2 would have to be sent to the DB server many times, which incurs a repeated communication overhead between host and DB server, as well as a repeated query-initialization overhead. These overheads can be reduced by exploiting the facts that the goal attribute is always the same throughout the KDD algorithm and a number of successive queries can refer to the same set of examples. Thus, candidate attribute is the only input parameter that varies among queries. In this case, the set of queries can be transformed into a single, coarser-grained query, which we will call Multiple Count by Group, as shown in Figure 7. The term Attribute_Id in the SELECT and GROUP BY clauses ranges over the attribute names specified in the Candidate_attribute_list of the FOREACH clause. Note that the clause FOREACH represents an extension to SQL which could be useful in DB applications other than KDD.

```
SELECT Attribute_Id, Goal_attribute, COUNT(*)
FROM  Mine_Relation
WHERE  Example_Set_Descriptor
GROUP BY  Attribute_Id, Goal_attribute
FOREACH  Attribute_Id  IN  Candidate_attribute_list
```

Figure 7 - Extended SQL query underlying Multiple Count by Group primitive

At first glance, a reduction in communication/query-initialization time may seem to be irrelevant in very large data sets, where this time could be a small fraction of the total processing time. However, in general, a KDD algorithm will execute database queries where the number of *selected* (*or counted*) tuples (and so the processing time) can vary widely[1], as discussed in Section 4.3. Communication/query-initialization time is significant in queries selecting few tuples - particularly for the IFS/2, as shown in Table 3. Now we examine the benefit to be gained by using Multiple Count by Group.

We define the total Count by Group processing time as the sum of the Start up time (S) plus the Execution time (E). S has essentially three components, viz. the time to send a query from the host to the server, to compile the query, and to generate a workload schedule. E has essentially three components, viz. the time to execute a *selection* operation - *Where* clause - (E1), to execute a *group by* with *count* operation (E2) and to return results to the host (E3). Let T be the time to execute a given iteration of a KDD algorithm where there are k candidate attributes to be evaluated. To execute that iteration, Count by Group would be called k times. S and E1 times are approximately constant for all calls (the same tuples are *selected* in every call). E2 and E3 times vary according to the candidate attribute. To simplify, let E2* and E3* denote the average of E2 and E3 over all k calls. Then T is approximately (S + E1 + E2* + E3*)k, or (S + E1)k + (E2* + E3*)k.

Using Multiple Count by Group, T could be reduced to T' = S' + E1 + (E2* + E3*)k, where S' is the Start up time associated with Multiple Count by Group. S' is expected to be somewhat longer than S (due e.g. to the additional query-scheduling complexity), but for large values of k (which is commonplace in large real-world data sets[2]) we expect S' << kS, so that (S' + E1) << (S + E1)k. More precisely, the percentual time saving obtained with Multiple Count by Group can be estimated as 1 - T'/T.

---

[1] E.g., in decision-tree learners, and in top-down learning algorithms in general, tree nodes close to the root node correspond to queries accessing many tuples (in particular, the root node corresponds to all the tuples), while leaf nodes correspond to queries accessing relatively few tuples.

[2] For example, the Quarterly Labour Force Survey produced by the UK's Department of Employment has about 350 attributes, so that the Mine relation would probably have tens of attributes.

S' is difficult to estimate, since it involves modifications to the low-level, internal software of the server. To simplify, we will neglect the difference between S' and S, so that S' = S. Assume that k = 30 (see footnote 2 in the previous page). As an example, let us take the second row of Table 2, i.e. 40000 tuples in the Mine relation and 3337 tuples counted (or selected). For the WX9010, (S + E1) has been estimated as 0.9 sec, so that (E2 + E3) = 4.59 - 0.9 = 3.69. Hence, the percentual time saving obtained with Multiple Count by Group would be 19%. For the IFS/2, (S + E1) has been estimated as 0.2 sec, so that (E2 + E3) = 0.99 - 0.2 = 0.79. Hence, the time saving would be 20%. A much greater time saving for the IFS/2 would be obtained with Multiple Count by Group in the case of Table 3. Taking as an example the second row of this table (and with the assumptions above), the time saving would be 86%.

5.2 Tailoring access methods for KDD tasks.

Recall, from Section 2, that we store the task-relevant data set in the Mine relation, created as a pre-processing step for the KDD algorithm. This allows us to create the Mine relation with access methods tailored for KDD tasks, rather than simply using access methods of current relations in the DB. This basic idea can be useful in any relatively complex KDD algorithm, where one expects most tuples to be accessed several times by the KDD algorithm. This helps to justify the overhead of creating the Mine relation. A quantification of the benefit to be gained from this idea will be subject of future research. For now, we briefly discuss below how this idea can be exploited.

We could create indexes (or hash) on attributes which exhibit a greater estimated frequency of occurrence in KDD queries. Unfortunately, that frequency is difficult to estimate, since it usually depends on both the KDD algorithm and the data set. However, in the case of the Count by Group primitive, we do have some clues to pave the way for this estimation. For example, the goal attribute is always the same in every instance of a Count by Group query. Moreover, we could sometimes use an estimation criterion based on the same attribute correlation measure computed by the KDD algorithm. Taking again the well-known example of decision-tree learners [15], the attribute in the root node of the tree (determined

in the very first iteration of the algorithm) will occur in the *Where* clause (example-set descriptor) of every query corresponding to lower-level tree nodes. Clearly, we could create an index (or hash) on that attribute, to improve performance.

5.3 Efficient aggregations.

One of the basic operations within the Count by Group primitive is the *count* aggregation function. Counting the number of tuples satisfying some conditions is a ubiquitous operation in KDD, and counting can be implemented in hardware at relatively low cost. Hence, a way of reducing the time taken by Count by Group is to use a hardware counter unit to count tuples "on the fly", as they are selected as having satisfied the example-set descriptor. This idea seems especially attractive in the case of the IFS/2, due to the limited degree of parallelism offered by that machine in count operations - recall that, in the IFS/2, count operations are done on the node-controller transputers, and the number of node-controllers is far less than the number of Search Engines.

6 Conclusions and future research.

We have proposed a KDD primitive which underlies many candidate rule evaluation mechanisms. We specified this primitive in terms of SQL relational operations, so that it can be encapsulated in a back-end DB server. Then we compared the time taken by the primitive on three different (but contemporary) computational architectures, viz. a Sun IPC workstation, the White Cross WX9010 (a distributed-memory MIMD DB computer) and the IFS/2 (a pseudo-associative-memory DB computer prototype). We showed that, in general, both the WX9010 and the IFS/2 obtain speed ups of one order of magnitude over a Sun workstation. We also showed that the WX9010 is faster than the IFS/2 when the KDD primitive selects a large (at least tens of thousands) number of tuples, whereas the IFS/2 is faster than the WX9010 when the primitive selects a relatively small (a few thousands or less) number of tuples.

We also proposed some ways to improve the computational efficiency of the KDD primitive. The most straightforward improvement is to increase the granularity of the

primitive, to reduce the client/server communication overhead and the query-initialization overhead. We showed that this idea is particularly attractive for queries selecting a relatively small number of tuples, which can be the case in KDD tasks even if the whole task-relevant data set is very large. The other proposed improvements are, in essence, to create a Mine relation with access methods tailored for KDD tasks, rather than simply using access methods of current relations in the DB; and to use some hardware counter unit(s) to count tuples "on the fly", as they are scanned (whose cost/benefit ratio remains to be investigated).

Moreover, additional KDD primitives may be appropriate for other phases of the total process. For instance, an interesting research direction is to design primitives for the Evaluation component of the KDD model discussed in Section 2. Note that a primitive for the Evaluation component would evaluate the final discovered rules, rather than candidate rules considered during the search, which requires new rule-evaluation criteria.

There are two main advantages in encapsulating KDD primitives in a parallel DB server. First, it simplifies the host software, saving host CPU time and releasing the application programmer from the complexity associated with parallel programming. This is a form of extended DB functionality and automatic parallelization, which increases application-programmer productivity. Second, the primitive can be implemented within the DB server in a much more efficient way than by host software. 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, which tends to increase the accuracy of the discovered rules and may even allow the discovery of new rules which cannot be discovered from relatively small data samples. We believe that automatic parallelization and greater query-processing efficiency are essential to make the use of parallel KDD commonplace among DB users.

Acknowledgements.

This research was funded by EPSRC (a British government agency) grant number GR/J/43301 and CNPq (a Brazilian government agency) grant number 200384/93-7. We thank other members of the IFS group for valuable discussions, particularly D.R. Thoen, N.E.J. Dewhurst, R.A. Gamble and A.J. Marsh. We also thank Steve Hassan for his help in using the WX9010.

References.

[1] R. Agrawal, T. Imielinski and A. Swami. Database mining: a performance perspective. IEEE Trans. Knowledge and Data Engineering, 5(6), Dec. 1993. pp. 914-925.

[2] P. Clark, T. Niblett. The CN2 induction algorithm. Machine Learning 3, 1989. 261-283.

[3] C.J. Date. An Introduction to Database Systems. 6th Ed. Reading, Massachusetts: Addison-Wesley, 1995.

[4] D. DeWitt et. al. The Gamma database machine project. IEEE Trans. Knowledge Engineering, 2(1), Mar. 1990, pp. 44-62.

[5] W.J. Frawley, G. Piatetsky-Shapiro and C.J. Matheus. Knowledge discovery in databases: an overview. In: G. Piatetsky-Shapiro and W.J. Frawley (Eds.) Knowledge Discovery in Databases. Menlo Park, CA: AAAI, 1991.

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

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

[8] IBC Ltd. Proc. of the Conf. on Commercial Parallel Processing, London, Nov. 1994. (Available from IBC Technical Services Ltd., 57-61 Mortimer Street, London, W1N 7TD).

[9] S.H. Lavington, M.E. Waite, J. Robinson and N.E.J. Dewhurst. Exploiting parallelism in primitive operations on bulk data types: some results. Computers and Artificial Intelligence, 12(4), 1993. pp. 313-336.

[10] C.J. Matheus, P.K. Chan and G. Piatetsky-Shapiro. Systems for knowledge discovery in databases. IEEE Trans. Knowledge and Data Engineering, 5(6), Dec. 1993, 903-913.

[11] R.S. Michalski, L. Kerschberg and K.A. Kaufman. Mining for knowledge in databases: the INLEN architecture, initial implementation and first results. Journal of Intelligent Information Systems, 1(1), 1992. pp. 85-113.

[12] J. Mingers. An empirical comparison of selection measures for decision-tree induction. Machine Learning, 3, 1989. pp. 319-342.

[13] D. Norris. White Cross Systems' Database Servers come to market. Large Computer Strategies, P-071-1544. April 5th 1994, Published by Gartner Group Inc. 56 Top Gallant Road, PO Box 10212, Stamford, CT-06904-2212, USA.

[14] G. Piatetsky-Shapiro and C.J. Matheus. Knowledge discovery workbench for exploring business databases. International Journal of Intelligent Systems, 7(7), 1992. p. 675-686.

[15] J.R. Quinlan. Induction of decision trees. Machine Learning, 1, 1986. pp. 81-106.

[16] J.C. Schlimmer. Self-modeling databases: learning and applying partial integrity constraints to detect database errors. IEEE Expert, April 1993. pp. 35-43.

[17] S. Shekhar, B. Hamidzadeh, A. Kohli and M. Coyle. Learning transformation rules for semantic query optimization: a data-driven approach. IEEE Trans. Knowledge and Data Engineering, 5(6), Dec. 1993. pp. 950-964.

[18] A. Silberchatz, M. Stonebraker and J. Ullman. Database systems: achievements and opportunities. Communications of the ACM, 34(10), 1991. pp. 110-120.

[19] S.M. Weiss and C.A. Kulikowski. Computer systems that learn. San Mateo, CA: Morgan Kaufmann, 1991.

[20] J. Zytkow and R. Zembowicz. Database exploration in search of regularities. Journal of Intelligent Information Systems, Vol. 2, No. 1, Mar. 1993, pp. 39-81.