# Performance Improvement in the Implementation of DBLEARN

Colin L. Carter
Howard J. Hamilton

Technical Report CS-94-05
January, 1994

Department of Computer Science
University of Regina
Regina, Saskatchewan
S4S 0A2

ISSN 0828-3494
ISBN 0-7731-0264-7

# Performance Improvement in the Implementation of DBLEARN

Colin Carter and Howard J. Hamilton*
Department of Computer Science
University of Regina
Regina, Sask., Canada S4S 0A2
{carter,hamilton}@cs.uregina.ca

January 31, 1994

**Abstract**

We describe efficiency improvements made to the DBLEARN program for performing knowledge discovery in databases. The original DBLEARN prototype implementation suffered from relatively poor performance. Causes of this inefficiency are explored and implementation strategies are suggested to improve performance. They include better memory management, more efficient storage of attribute values, and better searching techniques for matching attribute values with more general concepts. Finally measurements of the performance improvement are provided for one specific computer system.

## 1 Introduction

*Knowledge discovery from databases* is the automated extraction of useful and interesting information from large bodies of diverse data stored in databases. The primary objective of such discovery is to produce new nontrivial information that is understandable, accurate and useful [Frawley et al., 1992]. One of the primary challenges of such discovery is to devise a method which computers can implement efficiently. One such method is attribute oriented generalization based on concept hierarchies [Cai et al., 1991].

*Attribute oriented generalization* seeks to transform diverse data stored in database relations into more general and useful information on an attribute by attribute basis. Generalization is accomplished by replacing the specific attribute data values of each tuple with more and more general concepts. In this way, the number of distinct values of each attribute is reduced as related values are grouped into more general concepts. Many tuples then become

*Supported by Natural Sciences Engineering Research Council of Canada (NSERC) Research Grant OPG0121504,

redundant as the information they contain becomes identical to other tuples. These redundancies are eliminated by removing all but one of the identical tuples. This generalization continues until an acceptable level of information is attained.

The generalization is guided by concept hierarchies associated with each attribute of the relation to be generalized. A *concept hierarchy* is a tree structure defined by a domain expert that provides a hierarchical taxonomy of concepts ranging from a single most general concept at the root of the tree to some representation of all possible attribute values at the leaves. Concepts are formed by grouping related attribute values together and representing all members of this set or range by a single symbol. These symbols in turn may be grouped into more general concepts which are again represented by another unique symbol. This grouping continues until the most general concept, called *ANY*, is reached.

The generalization process is limited by the specification of two tunable threshold values, the attribute threshold and the table threshold. The *attribute threshold* specifies the maximum number of distinct values of any attribute that may exist in the final generalized relation. Generalization proceeds first by reducing the number of distinct values of attributes to less than or equal to the attribute threshold. The *table threshold* specifies the maximum number of tuples that may exist in the final generalized relation. After the first stage of generalization, there still may be more tuples in the partially generalized relation than the table threshold allows. When this occurs, an attribute is chosen by some method to continue generalization. This process iterates until the number of tuples is no more than the table threshold.

As mentioned above, it is desirable that the learning be efficiently implemented. Though attribute oriented generalization using concept hierarchies is an efficient way of extracting information from databases, there are various implementational issues that can drastically affect the performance of the actual program. Three primary factors affect the performance of any learning program. These are space requirements, the efficiency of data representation, and the efficiency of methods of accessing the data. If a program requires inordinately large amounts of memory for data structures, this can cause extreme slowdowns as the operating system is obliged to swap memory to and from secondary disk storage. In addition, if a program uses simple but costly means of representing data, operations on this data will be hampered. Finally, if slow methods of accessing random data values are used, this also will affect final performance.

DBLEARN is a machine learning program which implements attribute oriented generalization using concept hierarchies [Han et al., 1992, Han et al., 1993]. A prototype implementation was made available to us by Jiawei Han of Simon Fraser University. This prototype was an adequate proof of concept for DBLEARN, but it required improved performance for application to large databases.

After our major code reorganization of the DBLEARN prototype, DBLEARN now consists of five major components:

1. The user interface module
2. The command module
3. The database module

4. The concept hierarchy module

5. The learning module

The *user interface module* interacts with the user to specify the type of learning task, the set of data on which to operate, the location of concept hierarchy files and the threshold values to use. The *command module* controls the majority of internal program direction. It receives direction from the user interface and routes the requests to the appropriate module for processing. The *database module* retrieves the data from a database and converts it into DBLEARN data structures. The *concept hierarchy module* reads the concept hierarchy files, extracts the relevant trees and builds the internal representation of these. The *learning module* takes the data and hierarchy trees and reduces the retrieved relation to the level specified by the appropriate thresholds. The concept hierarchy and learning modules are the most relevant components to the performance improvements described in this paper. Further details on the software architecture of DBLEARN are given in [Carter et al., 1994].

The remainder of this paper is organized as follows. In Section 2, we identify several inefficiencies in the prototype implementation. Then in Section 3, we describe design and implementation changes that we incorporated into the program. In Section 4, the preliminary results of comparing the original version of DBLEARN and to our revised version are presented to quantify the actual increase in speed that has been achieved. Finally, in Section 5, we present conclusions and our future research plans for this project.

# 2 Performance Problems in DBLEARN

The primary problems with the performance of the DBLEARN prototype are the result of excessive storage requirements, inefficient data representations, and inefficient data retrieval. For simplicity, we refer to the DBLEARN prototype as the *original version* and the program resulting from our modifications as the *revised version*.

## 2.1 Inefficient Storage

DBLEARN seeks to generalize potentially huge amounts of data. As the program still stands, all initial data are read into internal data structures before any generalization is attempted. In anticipation of this, the original program defined a maximum number of possible tuples in the initial retrieved relation to be in the ten thousand range and defined an array of this many tuple structures.

Each relation also has a number of attributes. The original program anticipated that users would be likely to use no more than twenty attributes and so allocated enough space in each tuple structure for this many attribute values. Each attribute value may be one of several types, the primary ones being either string or numeric (integer or float). Since strings in string format may take much more space than either integers or floats, enough space was allocated in each attribute field to store a string of moderate length.

The summation of these three storage assumptions and a few other less major items resulted in an initial relation structure which required approximately four and a half megabytes

of storage. In a system where many users share the resources of a central server on a network, such excessive initial storage requirements may cause a great deal of disk swapping activity. In actuality, four such permanent relation structures were defined in addition to a temporary one used in a function which removed duplicate tuples from a relation. At any one time therefore, as much as almost twenty three megabytes of data storage is required. It is obvious that such excessive requirements would tax most systems in general use today. In practice, however, many times only several hundred tuples are retrieved from the target database and therefore much space is needlessly wasted.

## 2.2 Inefficient Attribute Value Representations

As mentioned above, the attribute types are primarily string or numeric. The original version of DBLEARN stored all values as strings in fixed length fields. As the relation attributes were defined, their types were examined with respect to the anticipated amount of string storage needed to represent any value of those attributes. The maximum string length was then predicted and each attribute given that much space. In the case of numeric values especially, this is obviously wasteful. Whereas many long integers or double length floats may be stored in eight bytes, often over twice this amount was used for the storage of these attribute types.

In addition to being wasteful of storage, string representation of attributes causes inefficient attribute value comparisons. Whereas two integers in normal numeric representation can be compared in only a few machine instructions, two numbers in string format take many more machine instructions to compare.

This representation of all values as strings, however, was likely motivated by the fact that after the first generalization of each attribute, the original value would be replaced by a string symbol from the concept hierarchy data structure. To represent the value at one point as a number and at a different point as a string would have presented a somewhat more difficult programming task. Regardless, simple representation of even symbols as literal strings is efficient from the standpoint of neither storage nor comparison. This design decision significantly hinders the overall efficiency of the program.

## 2.3 Inefficient Data Retrieval

As a relation is generalized, each attribute value is matched against the values in the appropriate concept hierarchy tree structure. When a match is found, the attribute value is replaced by the value of a more general concept. A key factor in overall program efficiency is the speed of matching a concept with an attribute value. A moderately fast method would be to search an ordered tree for the matching concept. Probably the least efficient method would be a linear search of unordered concepts. This, however, was the way the original version of DBLEARN was implemented. When the concept hierarchy file is read, each concept with an associated more general concept is stored in an unsorted array of concept structures. In addition, the individual concept hierarchy trees for each attribute are not conceptually separated from one another, but all are stored in the same large array. This necessitates an initial extraction of all concepts for a specific attribute before generalization

of that attribute can begin. Since an attribute may be generalized several different times, this repeated extraction is wasteful and inefficient.

# 3 Solutions to Performance Problems in DBLEARN

The three problems of inefficient storage allocation, inefficient data representation, and inefficient searching techniques have been improved respectively by dynamically allocated minimal storage, compact data representation and hashed value searching.

## 3.1 Dynamically Allocated Minimal Storage

Whereas the original version anticipates maximal values for the number of tuples in a relation, the number of attributes in a tuple, and the number of bytes needed for each attribute, the revised version dynamically allocates minimal amounts for each of these elements and expands as needed.

The number of tuples in a relation to be retrieved is not known previous to retrieval unless a specific count operation is issued before the relation retrieval. In lieu of this, however, a nominal number of initial tuples can be allocated. The actual number of tuples retrieved can be tracked and if the initial allocated number is reached, the storage can be expanded and retrieval continued. In fact, the revised version allocates only pointers to tuples and not the full storage for tuples. The storage for the tuples themselves is individually allocated as needed. This keeps the amount of necessary storage at a minimum. In addition, should the number of tuples increase beyond what was previously anticipated as a maximum, the previous version would crash. The revised version, however, will continue to expand as needed until secondary disk storage runs out.

The original version anticipates a maximum of twenty tuples and allocates accordingly. When the learning task is specified, however, the specific attributes to be retrieved are specified. This allows them to be counted and only enough storage allocated for the appropriate amount.

Finally, the original version examined the actual attributes to determine how much storage was needed for each and then allocated the maximum amount for every one. The revised version handles the representation of attributes in a very different way, as explained in the next subsection.

## 3.2 Compact Attribute Value Representation

The definition of concept hierarchies requires that the leaf nodes of the trees represent all possible values that may be encountered in the relation to be generalized. These attribute values may be of two types, range or non-range. Range values are values that can inherently be divided into ranges bounded by lower and upper bounds. Range values include all numbers, dates and times. The notation $x\tilde{}y$ denotes a range value with a lower bound of $x$ and an upper bound of $y$, such that for any $z \in x\tilde{}y, x \leq z < y$. In this way, attribute values with potentially limitless distinct values can be compactly represented in the concept hierarchy tree.

On the other hand, non-range values are all strings and alpha-numeric codes or range types which contain only a finite, limited set of distinct values that need to be individually specified. Since non-range values represent limited sets of distinct values, the concept hierarchy trees can have the actual attribute values as leaves.

The result of this is that every attribute value which can be read will be defined or represented in the concept hierarchy tree before any database access is accomplished. For non-range values, a symbol table can be constructed in which one copy of each symbol is stored as a string and all attribute values can be stored simply as indices into the symbol table. For range type values, the conceptual storage of the actual value can be used, for example, storing integers as integers. The concept hierarchy tree can also have extra fields defined to store the inherent types for the upper and lower bounds of the range values. When the first round of generalization is accomplished, the range values can be matched against the boundary values stored in the concept hierarchy tree. On the first level of generalization, all values, range or non-range are replaced by concept symbols. Range values after generalization, therefore, can simply be replaced by an index into the symbol table, just as for other values. In this way, storage for each attribute value is much more efficient than a string of maximal length.

As for value comparisons, this representation allows much more efficient operations than the former string comparisons. All range values are initially stored as their inherent types. Since these are all numeric in nature, the comparisons for equality when testing against matching concepts are fast and efficient. All non-range values are stored as indices into a symbol table, so all that is needed is a comparison of index values for equality. The efficiency of this over multiple string comparisons is clear.

## 3.3 Efficient Data Retrieval

In the original version, attribute values were matched to concepts by using string comparisons and a linear search of an unordered list of concepts. The revised version has two, more efficient methods of concept matching.

For range values, a numeric value must be compared against all the possible ranges in the concept hierarchy tree until a match is found. Rather than search all symbols in the tree, only leaf concepts need to be searched. The concept hierarchy tree is actually constructed as a tree structure when it is read in. After the trees have been constructed for all attributes, each tree is traversed once to build a list of leaf nodes. A pointer to this list is stored in the structure associated with the root of the tree. On the first generalization of each attribute, the list of terminals is retrieved and a linear search along the linked list is used to locate the matching concept.

For non-range values, we have stated that the values are stored as indices into the symbol table. This, though functionally true, in not entirely accurate. When the symbol table is initially built, the symbols in the concept hierarchy are hashed to get an index into a hash table of a preset size. As with any hashing technique, collisions occur when distinct symbols are transformed (hashed) into the same index. This case is handled by separate chaining, i.e., by associating a list of symbols with each position of the hash table and adding to the list whenever a collision occurs.

A symbol table element contains not only the string value of the symbol but also a

| Task | Tuple Count | Time for Original | Time for Revised | Speedup Factor |
|---|---|---|---|---|
| Task 1 | 486 | 41.906 | 0.330 | 127 |
| Task 2 | 77 | 20.191 | 0.031 | 651 |
| Task 3 | 60 | 19.701 | 0.031 | 635 |

Table 1: Speedup Test Results

pointer into the concept hierarchy tree that points to the matching concept for the symbol. When actual data retrieval begins, the non-range values are hashed into the symbol table and the index of the matching symbol is retrieved. If any collisions have occurred for this index, the list is followed until the exact symbol is found. Now the concept for that symbol can be retrieved by following the symbol's pointer into the concept hierarchy tree. Rather than store the index through which a list can be repeatedly searched to find a pointer to a concept, it is much more efficient to immediately retrieve the pointer to the concept and store that instead. In this way, when the relation is generalized, the matching concept and its more general concept are immediately available through the pointer and generalization is extremely fast. The less general concept can simply be replaced by a pointer to the more general concept, and again by a pointer to a more general concept yet as more levels of generalization occur. Thus, after one initial hashed search of a symbol table, no more searching is necessary. Attribute value comparisons are still efficient as all that is needed is a comparison of pointer values to determine if two attributes have the same value.

# 4 Results

For our experiments, we ran DBLEARN on an IBM compatible 386 microcomputer with OS/2 version 2.1. The relational database used is IBM's DB2/2, a port of their mainframe relational database to the microcomputer platform. The system has eight megabytes of RAM which is the minimum recommended by IBM for running OS/2. However, for testing a program which potentially uses many megabytes of data, this amount of memory is fairly restrictive. This might be compared to runs on a networked computer that is receiving fairly heavy use from many users.

To measure the performance improvement attained, three learning tasks were each run ten times with both the old version and the new version. The three learning tasks were typical of those run with the NSERC Grant Information database, which has been used with previous experiments with DBLEARN. For reference, the tasks are given in Appendix A. We measured the time required for learning (generalization) and did not include the time required for database retrieval. The average of the last nine of each set of ten runs was then computed and used in comparisons.

The results are summarized in Table 1. In Table 1, the *Tuple Count* is the number of tuples in the relation to be generalized, the *Time for Original* is the average time, in seconds, for last 9 runs of the original version, *Time for Revised* is the corresponding amount for the

| Task | Tuple Count | Time for Original | Time for Revised | Speedup Factor |
|---|---|---|---|---|
| Task 1 | 486 | 21.219 | .330 | 64 |
| Task 2 | 77 | 4.421 | .031 | 143 |
| Task 3 | 60 | 3.75 | .031 | 120 |

Table 2: Speedup Test Results, without Disk Swapping

revised version, and the *Speedup Factor* tells how many times faster the revised version is than original. The Speedup Factor is calculated by dividing the Time for Original by the Time for Revised. The value for the Time for Revised for Task 3 in Table 1 is higher than the true time because 3 of the 9 runs reported elapsed times of 0 seconds, while the other 6 runs reported 0.031 or 0.032 seconds. Thus, the true speedup was greater than 635 times for Task 3.

In comparing the average times for the three learning tasks, we notice that the revised version is very significantly faster than the older version on this specific system. This is the actual speed difference in the learning component that would be observed by a user. We identified excess disk swapping due to wasteful use of space as the most likely cause of the majority of the speed difference. The remainder of the difference can be attributed to the more costly string storage and comparisons and to the linear searches for matching values in the overly large concept hierarchy table.

To determine the relative importance of these factors, we ran a separate series of test runs which attempted to eliminate the wasteful use of space and the consequent disk swapping. The results, which are given in Table 2 indicate that the disk swapping was indeed the source of much of the speed difference, but that improved data structures and algorithms alone provided a 65 to 143 times speedup.

As was mentioned, these preliminary test results are accurate for the described system, but they may not be indicative of performance on more powerful computers with more main memory. Currently DBLEARN is being adapted at Simon Fraser University for use with the ORACLE database manager. We will continue testing of the revised version with large data sets and with the ORACLE database manager on SUN workstations.

# 5 Conclusions and Research Directions

Attribute oriented generalization using concept hierarchies is efficient and useful for the discovery of interesting information from databases. The DBLEARN prototype was hampered, by inefficient implementation, in its ability to deal with large amounts of data. In this paper, we identified several efficiency problems of the learning component of DBLEARN. These included overallocation of storage, inefficient data representation, and poor search and replace techniques for generalizing concepts. The solutions described and implemented were dynamic allocation of minimal storage, efficient storage of attribute values through the use of hashing and a symbol table, the representation of the concept hierarchy as a tree structure,

and the representation of attributes as pointers into this tree. These changes resulted in extremely fast generalization of relations compared to previous performance.

Currently, the generalization process reads in a complete relation before attempting a reduction of tuples. It would be much more efficient if the reduction was done on the fly. That is, as the tuples are read in from the database, keep track of the number of distinct values of each attribute. As soon as the number of distinct values of a given attribute exceeds the attribute threshold, the current amount of data can be generalized, and the concept hierarchy tree adjusted so that level one generalization occurs immediately on reading that attribute. Again, the number of distinct values for the second lowest level of the concept hierarchy can be tracked and level two generalization begun as soon as the attribute threshold is again reached. If the portion of the relation already read in is also generalized each step, the storage requirements would never be great at all, but would hover relatively close to the table threshold value. In addition for any subsequent data, multiple levels could be generalized in one step rather than several. Both storage gains and efficiency gains could be realized.

# References


[Cai et al., 1991] Cai, Y., Cercone, N., and Han, J. (1991). Attribute-oriented induction in relational databases. In *Knowledge Discovery in Databases*, pages 213–228. AAAI/MIT Press, Cambridge, MA.

[Carter et al., 1994] Carter, C., Hamilton, H., and Cercone, N. (1994). The software architecture of DBLEARN. Technical Report 94-4, Department of Computer Science, University of Regina, Regina, Sask., Canada. 27 pages.

[Frawley et al., 1992] Frawley, W., Piatetsky-Shapiro, G., and Metheus, C. (1992). Knowledge discovery in databases: An overview. *AI Magazine*, 13(3):57–70.

[Han et al., 1992] Han, J., Cai, Y., and Cercone, N. (1992). Knowledge discovery in databases: An attribute-oriented approach. In *Proc. of 18th Int'l Conf. on Very Large Data Bases*, Vancouver.

[Han et al., 1993] Han, J., Cai, Y., and Cercone, N. (1993). Data-driven discovery of quantitative rules in relational databases. *IEEE Trans. on Knowledge and Data Engineering*, 5(1):29–40.

# Appendix A: Learning Tasks

This appendix gives the three learning tasks applied to the NSERC Grant Information database in our experiments.

**Task 1:**

```
learn characteristic rule for "CS_Operating_Grants"
from award a, organization o, grant_type g
where   o.org_code = a.org_code and
        g.grant_order = "Operating_Grants" and
        a.grant_code = g.grant_code and
        a.disc_code = "Computer"
in relevance to amount, prop(amount), prop(votes)
using table threshold 18
```

**Task 2:**

```
learn characteristic rule for "CS_HW_Op_Grants"
from award A, organization O
where  A.disc_code = "Hardware" and A.org_code = O.org_code
in relevance to amount, province, prop(votes), prop(amount)
using attribute threshold 7
using table threshold 12
```

**Task 3:**

```
learn characteristic rule for "CS_SW_Op_Grants"
from award A, organization O
where  A.disc_code = "Software" and A.org_code = O.org_code
in relevance to amount, province, prop(votes), prop(amount)
using table threshold 12
using attribute threshold 7
```