Scalable Mining for Classification Rules in Relational Databases

Min Wang
Dept. of Computer Science
Duke University
Durham, NC 27708

Bala Iyer
Database Technology Institute
IBM Santa Teresa Lab
San Jose, CA 95161

Jeffrey Scott Vitter
Dept. of Computer Science
Duke University
Durham, NC 27708

Abstract

Classification is a key function of many "business intelligence" toolkits and a fundamental building block in data mining. Immense data may be needed to train a classifier for good accuracy. The state-of-art classifiers [21, 25] need an in-memory data structure of size $O(N)$, where N is the size of the training data, to achieve efficiency. For large data sets, such a data structure will not fit in the internal memory. The best previously known classifier does a quadratic number of I/Os for large N.

In this paper, we propose a novel classification algorithm (classifier) called MIND (MINing in Databases). MIND can be phrased in such a way that its implementation is very easy using the extended relational calculus SQL, and this in turn allows the classifier to be built into a relational database system directly. MIND is truly scalable with respect to I/O efficiency, which is important since scalability is a key requirement for any data mining algorithm.

We built a prototype of MIND in the relational database manager DB2 and benchmarked its performance. We describe the working prototype and report the measured performance with respect to the previous method of choice. MIND scales not only with the size of the datasets but also with the number of processors on an IBM SP2 computer system. Even on uniprocessors, MIND scales well beyond the dataset sizes previously published for classifiers. We also give some insights that may have an impact on the evolution of the extended relational calculus SQL.

1 Introduction

Information technology has developed rapidly over the last three decades. To make decisions faster, many companies have combined data from various sources in relational databases [14]. The data contain patterns previously undeciphered that are valuable for business purposes. Data mining is the process of extracting valid, previously unknown, and ultimately comprehensible information from large databases and using it to make crucial business decisions. The extracted information can be used to form a prediction or classification model, or to identify relations between database records.

Since extracting data to files before running data mining functions would require extra I/O costs, users as well as previous investigators [18, 17] have pointed to the need for the relational database managers to have these functions built in.

The classification problem can be described informally as follows: We are given a *training set* (or *DETAIL* table) consisting of many training examples. Each training example is a row with multiple attributes, one of which is a *class label*. The objective of classification is to process the *DETAIL* table and produce a *classifier*, which contains a description (model) for each class. The models will be used to classify future data for which the class labels are unknown (see [6, 24, 23, 7]). A simple illustration of training data is shown in Table 1. The examples reflect the past experience of an organization extending credit. From those examples, we can generate the classifier shown in Figure 1.

Although memory and CPU prices are plunging, the volume of data available for analysis is immense and getting larger. We may not assume that the data are memory-resident. Hence, an important research problem is to develop accurate classification algorithms that are scalable with respect to I/O and parallelism. Accuracy is known to be domain-specific (e.g., insurance fraud, target marketing). However, the problem of scalability for large amounts of data is more amenable to a general solution. A classification algorithm should scale well; that is, the classification algorithm should work well even if the training set is huge and vastly overflows the internal memory. In

Authorized licensed use limited to: KnowledgeGate from IBM Market Insights. Downloaded on December 15, 2008 at 10:52 from IEEE Xplore. Restrictions apply.

salary	*age*	*credit rating*
65K	30	Safe
15K	23	Risky
75K	40	Safe
15K	28	Risky
100K	55	Safe
60K	45	Safe
62K	30	Risky

Table 1: Training set

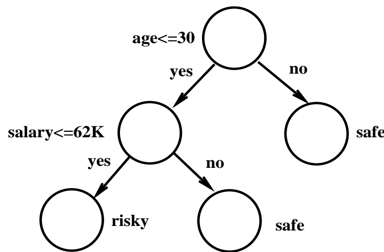

Figure 1: Decision tree for the data in Table 1

data mining applications, it is common to have training sets with several million examples. It is observed in [21] that previously known classification algorithms do not scale.

Random sampling is often an effective technique in dealing with large data sets. For simple applications whose inherent structures are not very complex, this approach is efficient and gives good results. However, in our case, we do not favor random sampling for two main reasons:

1. In general, choosing the proper sample size is still an open question. The following factors must be taken into account:
   - The training set size.
   - The convergence of the algorithm. Usually, many iterations are needed to process the sampling data and refine the solution. It's very difficult to estimate how fast the algorithm will give a satisfactory solution.
   - The complexity of the model.

   The best known theoretical upper bounds on sample size suggest that the training set size may need to be immense to assure good accuracy [12, 19].
2. In many real applications, customers insist that *all* data, not just a sample of the data, must be processed. Since the data are usually obtained from valuable resources at considerable expense, they should be used as a whole throughout the analysis.

Therefore, designing a scalable classifier may be necessary or preferable, although we can always use random sampling in places where it is appropriate.

In [21, 25, 16], data access for classification follows "a record at a time" access paradigm. Scalability is addressed individually for each operating system, hardware platform, and architecture. In this paper, we introduce the MIND (MINing in Databases) classifier. MIND rephrases classification as a classic database problem of summarization and analysis thereof. MIND leverages the extended relational calculus SQL, an industry standard, by reducing the solution to novel manipulations of SQL [1] statements embedded in a small program written in C.

MIND scales, as long as the database primitives it uses scale. We can follow the recommendations in [1, 20] that numerical data be discretized so that each attribute has a reasonable number of distinct values. If so, operations like histogram formation, which have a significant impact on performance, can be done in a linear number of I/Os, usually requiring one, but never more than two passes over the *DETAIL* table [29]. Without the discretization, the I/O performance bound has an extra factor that is logarithmic but fortunately with a very large base M/B, which is the number of disk blocks that can fit in the internal memory.

One advantage of our approach is that its implementation is easy. We have implemented MIND as a stored procedure, a common feature in modern DBMSs. In addition, since most modern database servers have very strong parallel query processing capabilities, MIND runs in parallel at no extra cost. A salient feature of MIND and one reason for its efficiency is its ability to do classification without any update to the *DETAIL* table.

We analyze and compare the I/O complexities of MIND and the previous method of choice, the interesting method called SPRINT [25]. Our theoretical analysis and experimental results show that MIND scales well whereas SPRINT can exhibit quadratic I/O times.

We describe our MIND algorithm in the next section. The database implementation of our algorithm is described in Section 3. A theoretical performance analysis is given in Section 4. In Section 5, we present our experimental results. We make concluding remarks in Section 6.

[1] SQL is simply an implementation of the relational calculus proposed in [11]. A few extensions have been done since then [28, 10].

Authorized licensed use limited to: KnowledgeGate from IBM Market Insights. Downloaded on December 15, 2008 at 10:52 from IEEE Xplore. Restrictions apply.

2 The Algorithm

2.1 Overview

A decision tree classifier is built in two phases: a growth phase and a pruning phase. In the growth phase, the tree is built by recursively partitioning the data until each partition is either "pure" (all members belong to the same class) or sufficiently small (according to a parameter set by the user). The form of the split used to partition the data depends upon the type of the attribute used in the split. Splits for a numerical attribute A are of the form $value(A) \leq x$, where x is a value in the domain of A. Splits for a categorical attribute A are of the form $value(A) \in S$, where S is a subset of $domain(A)$. We consider only binary splits as in [21, 25] for purpose of comparisons. After the tree has been fully grown, it is pruned to remove noise in order to obtain the final tree classifier.

The tree growth phase is computationally much more expensive than the subsequent pruning phase. The tree growth phase accesses the training set (or *DETAIL* table) multiple times, whereas the pruning phase only needs to access the fully grown decision tree. We therefore focus on the tree growth phase. The following pseudo-code gives an overview of our algorithm:

```
GrowTree(TrainingSet DETAIL)
  Initialize tree T and put all records of DETAIL
    in the root;
  while (some leaf in T is not a STOP node)
    for each attribute i do
      form the dimension table (or histogram) DIM_i;
      evaluate gini index for each non-STOP leaf at
        each split value with respect to attribute i;
    for each non-STOP leaf do
      get the overall best split for it;
    partition the records and grow the tree for one
      more level according to the best splits;
    mark all small or pure leaves as STOP nodes;
  return T;
```

2.2 Leaf node list data structure

A powerful method called SLIQ was proposed in [21] as a semi-scalable classification algorithm. The key data structure used in SLIQ is a *class list* whose size is linear in the number of examples in the training set. The fact that the *class list* must be memory-resident puts a hard limitation on the size of the training set that SLIQ can handle.

In the improved SPRINT classification algorithm [25], new data structures *attribute list* and *histogram* are proposed. Although it is not necessary for the attribute list data structure to be memory-resident, the histogram data structure must be in memory to insure good performance. To perform the split in [25], a *hash table* whose size is linear in the number of examples of the training set is used. When the hash table is too large to fit in memory, splitting is done in multiple steps, and SPRINT does not scale well.

In our MIND classifier, the information we need to evaluate the split and perform the partition is stored as a many to one function, registered with the object-relational database manager. When the function is evaluated on a record, it returns the partition number. This is the key reason for I/O reduction, efficient classification and update elimination.

We define a data structure that is a representation of the growing classification tree. We assign a unique number to each node in the tree. When loading the training data into the database, imagine the addition of a hypothetical column *leaf_num* to each row. For each training example, *leaf_num* will always indicate which leaf node in the current tree it belongs to. When the tree grows, the *leaf_num* value changes to indicate that the record is moved to a new node by applying a split. A static array called *LNL (*leaf node list) is used to relate the *leaf_num* value in the relation to the number assigned to the corresponding node in the tree.

To insure the performance of our algorithm, *LNL* is the only data structure that needs to be memory-resident. The size of *LNL* is equal to the number of nodes in the tree, so *LNL* can always be stored in memory.

2.3 Computing the *gini* index

A splitting index is used to choose from alternative splits for each node. Several splitting indices have recently been proposed. We use the *gini* index, originally proposed in [6] and used in [21, 25], because it gives acceptable accuracy. The accuracy of our classifier is therefore the same as those in [21, 25].

For a data set S containing N examples from C classes, $gini(S)$ is defined as

$$gini(S) = 1 - \sum_{i=1}^{C} p_i^2 \qquad (1)$$

where p_i is the relative frequency of class i in S. If a split divides S into two subset S_1 and S_2, with sizes N_1 and N_2 respectively, the *gini* index of the divided data $gini_{\text{split}}(S)$ is given by

$$gini_{\text{split}}(S) = \frac{N_1}{N} gini(S_1) + \frac{N_2}{N} gini(S_2) \qquad (2)$$

Authorized licensed use limited to: KnowledgeGate from IBM Market Insights. Downloaded on December 15, 2008 at 10:52 from IEEE Xplore. Restrictions apply.

The attribute containing the split point achieving the smallest *gini* index value is then chosen to split the node [6]. Computing the *gini* index is the most expensive part of the algorithm since finding the best split for a node requires evaluating the *gini* index value for each attribute at each possible split point.

The training examples are stored in a relational database system using a table with the following schema: $DETAIL(attr_1, attr_2, \ldots, attr_n, class, leaf_num)$, where $attr_i$ is the ith attribute, for $1 \leq i \leq n$, *class* is the classifying attribute, and *leaf_num* denotes which leaf in the classification tree the record belongs to.[1] In actuality *leaf_num* can be computed from the rest of the attributes in the record and does not need to be stored explicitly. As the tree grows, the *leaf_num* value of each record in the training set keeps changing. Because *leaf_num* is a computed attribute, the *DETAIL* table is never updated, a key reason why MIND is efficient for relational databases. We denote the cardinality of the class label set by C, the number of the examples in the training set by N, and the number of attributes (not including class label) by n.

3 Database Implementation of MIND

To emphasize how easily MIND is embeddable in a conventional database system using SQL and its accompanying optimizations, we describe our MIND components using SQL.

3.1 Numerical attributes

For every level of the tree and for each attribute $attr_i$, we recreate the dimension table (or histogram) called DIM_i with the schema $DIM_i(leaf_num, class, attr_i, count)$ using a simple SQL `SELECT` statement on *DETAIL*:

```
INSERT INTO DIM_i ²
SELECT leaf_num, class, attr_i, COUNT(*)
FROM DETAIL ³
WHERE leaf_num <> STOP
GROUP BY leaf_num, class, attr_i
```

Although the number of distinct records in *DETAIL* can be huge, the maximum number of rows in DIM_i is typically much less and is no greater than (#leaves in the tree) $\times$ (#distinct values on $attr_i$) $\times$ (#distinct classes), which is very likely to be the order of several hundreds [22]. By including *leaf_num* in the attribute list for grouping, MIND collects summaries for every leaf in one query. In the case that the number of distinct values of $attr_i$ is very large, preprocessing is often done in practice to further discretize it [1, 20]. Discretization of variable values into a smaller number of classes is sometimes referred to as "encoding" in data mining practice [1]. Roughly speaking, this is done to obtain a measure of aggregate behavior that may be detectable [22]. Alternatively, efficient external memory techniques can be used to form the dimension tables in a small number (typically one or two) linear passes, at the possible cost of some added complexity in the application program to give the proper hints to the DBMS, as suggested in Section 4.

It is apparent for each attribute i that its DIM_i table may be created in one pass over the *DETAIL* table. It is straightforward to schedule one query per dimension (attribute). Completion time is still linear in the number of dimensions. Commercial DBMSs store data in row-major sequence. I/O efficiencies may be obtained if it is possible to create dimension tables for all attributes in one pass over the *DETAIL* table. Concurrent scheduling of the queries populating the DIM_i tables is the simple approach. Existing buffer management schemes that rely on I/O latency appear to synchronize access to *DETAIL* for the different attributes. The idea is that one query piggy-backs onto another query's I/O data stream. Results from early experiments are encouraging [26].

It is also possible for SQL to be extended to insure that, in addition to optimizing I/O, CPU processing is also optimized. Taking liberty with SQL standards, we write the following query as a proposed SQL operator:

```
SELECT FROM DETAIL
INSERT INTO DIM_1{leaf_num, class, attr_1, COUNT(*)
       WHERE predicate
       GROUP BY leaf_num, class, attr_1}
INSERT INTO DIM_2{leaf_num, class, attr_2, COUNT(*)
       WHERE predicate
       GROUP BY leaf_num, class, attr_2}
    ⋮
INSERT INTO DIM_n{leaf_num, class, attr_n, COUNT(*)
       WHERE predicate
       GROUP BY leaf_num, class, attr_n}
```

The new operator forms multiple groupings concurrently and may allow further RDBMS query optimization.

Since such an operator is not supported, we make use of the object extensions in DB2, the *user-defined*

[1] *DETAIL* itself may be a summary of atomic transactional data, or the atomic data.

[2] Note the structural transformation that takes an attribute name in a schema and turns it into the table name of an aggregate.

[3] *DETAIL* may refer to data either in a relation or a file (e.g. on tape). In the case of a file, *DETAIL* resolves to an execution of a user-defined function (e.g. fread in UNIX) [9]).

Authorized licensed use limited to: KnowledgeGate from IBM Market Insights. Downloaded on December 15, 2008 at 10:52 from IEEE Xplore. Restrictions apply.

function (udf) [27, 8, 15], which is another reason why MIND is efficient. User-defined functions are used for association in [3]. An external udf is a function that is written by a user in a host programming language. The CREATE FUNCTION statement for an external function tells the system where to find the code that implements the function. In MIND we use a udf to accumulate the dimension tables for all attributes in one pass over $DETAIL$.

After populating DIM_i, we evaluate the *gini* index value for each leaf node at each possible split value of the attribute i by performing a series of SQL operations that only involve accessing DIM_i.

For each leaf in the tree, possible split values for attribute i are all distinct values of $attr_i$ among the records that belong to this leaf. For each possible split value, we need to get the class distribution for the two parts partitioned by this value in order to compute the corresponding *gini* index. We collect such distribution information in two relations, UP and $DOWN$.

Relation UP with the schema $UP(leaf_num, attr_i, class, count)$ can be generated by performing a self-outer-join on DIM_i:

```
INSERT INTO UP
SELECT d1.leaf_num, d1.attr_i, d1.class, SUM(d2.count)
FROM (FULL OUTER JOIN DIM_i d1, DIM_i d2
ON d1.leaf_num = d2.leaf_num AND
   d2.attr_i <= d1.attr_i AND
   d1.class = d2.class
GROUP BY d1.leaf_num, d1.attr_i, d1.class)
```

Similarly, relation $DOWN$ can be generated by just changing the $\leq$ to $>$ in the ON clause. We can also obtain $DOWN$ by using the information in the leaf node and the *count* column in UP without doing a join on DIM_i again.

$DOWN$ and UP contain all the information we need to compute the *gini* index at each possible split value for each current leaf, but we need to rearrange them in some way before the *gini* index is calculated. The following intermediate view can be formed for all possible classes k:

```
CREATE VIEW C_k_UP(leaf_num, attr_i, count) AS
SELECT leaf_num, attr_i, count
FROM UP
WHERE class = k
```

Similarly, we define view C_k_DOWN from $DOWN$.

A view $GINI_VALUE$ that contains all *gini* index values at each possible split point can now be generated. Taking liberty with SQL syntax, we write

```
CREATE VIEW GINI_VALUE(leaf_num,
                       attr_i, gini) AS
SELECT u1.leaf_num, u1.attr_i, f_gini
FROM   C1_UP u1, ...,C_C_UP u_C,
       C1_DOWN d1, ...,C_C_DOWN d_C
WHERE u1.attr_i = ... = u_C.attr_i =
      d1.attr_i = ... = d_C.attr_i
AND   u1.leaf_num = ... = u_C.leaf_num =
      d1.leaf_num = ... = d_C.leaf_num
```

where f_{gini} is a function of $u_1.count, \ldots, u_n.count, d_1.count, \ldots, d_n.count$ according to (1) and (2).

We then create a table MIN_GINI with the schema $MIN_GINI(leaf_num, attr_name, attr_value, gini)$:

```
INSERT INTO MIN_GINI
SELECT leaf_num, :i¹, attr_i², gini
FROM GINI_VALUE a
WHERE a.gini=(SELECT MIN(gini)
              FROM GINI_VALUE b
              WHERE a.leaf_num = b.leaf_num)
```

Table MIN_GINI now contains the best split value and the corresponding *gini* index value for each leaf node of the tree with respect to $attr_i$. The table formation query has a nested subquery in it. The performance and optimization of such queries are studied in [5, 23, 13].

We repeat the above procedure for all other attributes. At the end, the best split value for each leaf node with respect to all attributes will be collected in table MIN_GINI, and the overall best split for each leaf is obtained from executing the following:

```
CREATE VIEW BEST_SPLIT(leaf_num, attr_name,
                       attr_value) AS
SELECT leaf_num, attr_name, attr_value
FROM MIN_GINI a
WHERE a.gini=(SELECT MIN(gini)
              FROM MIN_GINI b
              WHERE a.leaf_num = b.leaf_num)
```

3.2 Categorical attributes

For categorical attribute i, we form DIM_i in the same way as for numerical attributes. DIM_i contains all the information we need to compute the *gini* index for any subset splitting. In fact, it is an analog of the *count matrix* in [25], but formed with SQL operators.

[1] i is a host variable, the value applies on invocation of the statement.

[2] Again, note the transformation for the table name DIM_i to column value i and column name $attr_i$.

Authorized licensed use limited to: KnowledgeGate from IBM Market Insights. Downloaded on December 15, 2008 at 10:52 from IEEE Xplore. Restrictions apply.

A possible split is any subset of the set that contains all the distinct attribute values. If the cardinality of attribute i is m, we need to evaluate the splits for all the 2^m subsets. Those subsets and their related counts can be generated in an inductive way. The schema of the relation that contains all the k-sets is $S_k_IN(leaf_num,\ class,\ v_1, v_2, ..., v_k,\ count)$. Obviously we have $DIM_i = S_1_IN$. S_k_IN is then generated from S_1_IN and S_{k-1}_IN as follows:

```
INSERT INTO S_k_IN
SELECT p.leaf_num, p.class, p.v_1, ..., p.v_{k-1},
       q.v_1, p.count + q.count
FROM (FULL OUTER JOIN S_{k-1}_IN p, S_1_IN q
ON p.leaf_num = q.leaf_num AND
   p.class = q.class AND
   q.v_1 > p.v_{k-1})
```

We generate relation S_k_OUT from S_k_IN in a manner similar to how we generate $DOWN$ from UP. Then we treat S_k_IN and S_k_OUT exactly as $DOWN$ and UP for numerical attribute in order to compute the *gini* index for each k-set split.

A simple observation is that we don't need to evaluate all the subsets. We only need to compute the k-sets for $k = 1, 2, \ldots, \lfloor m/2 \rfloor$ and thus save time. For large m, greedy heuristics are often used to restrict search.

3.3 Partitioning

Once the best split attribute and value have been found for a leaf, the leaf is split into two children. If *leaf_num* is stored explicitly as an attribute in *DETAIL*, then the following UPDATE performs the split for each leaf:

```
UPDATE DETAIL
SET leaf_num
    = Partition(attr_1, ..., attr_n, class, leaf_num)
```

The user-defined function *Partition* is as follows:

Partition(record r)
  Use the *leaf_num* value of r to locate the tree node n that r belongs to;
  Get the best split from node n;
  Apply the split to r, grow a new child of n if necessary;
  Return a new *leaf_num* according the result of the split;

However, *leaf_num* is not a stored attribute in *DETAIL* because updating the whole relation *DETAIL* is expensive. We observe that *Partition* is merely applying the current tree to the original training set. We avoid the update by replacing *leaf_num* by function *Partition* in the statement forming DIM_i. If *DETAIL* is stored on non-updatable tapes, this solution is required. It is important to note that once the dimension tables are created, the *gini* index computation for all leaves involves only dimension tables.

4 Performance Analysis

Building classifiers for large training sets is an I/O bound application. In this section we analyze the I/O complexity of both MIND and SPRINT and compare their performances.

As we described in Section 2.1, the classification algorithm iteratively does two main operations: computing the splitting index (in our case, the *gini* index) and performing the partition. SPRINT [25] forms an *attribute list* (projection of the *DETAIL* table) for each attribute. In order to reduce the cost of computing the *gini* index, SPRINT presorts each attribute list and maintains the sorted order throughout the course of the algorithm. However, the use of attribute lists complicates the partitioning operation. When updating the leaf information for the entries in an attribute list corresponding to some attribute that is *not* the splitting attribute, there is no local information available to determine how the entries should be partitioned. A *hash table* (whose size is linear in the number of training examples that reach the node) is repeatedly queried by random access to determine how the entries should be partitioned. In large data mining applications, the hash table is therefore not memory-resident, and several extra I/O passes may be needed, resulting in highly nonlinear performance.

MIND avoids the external memory thrashing during the partitioning phase by the use of dimension tables DIM_i that are formed while the *DETAIL* table, consisting of all the training examples, is streamed through memory. In practice, the dimension tables will likely fit in memory, as they are much smaller than the *DETAIL* table, and often preprocessing is done by discretizing the examples to make the number of distinct attribute values small. While vertical partitioning of *DETAIL* may also be used to compute the dimension tables in linear time, we show that it is not a must. Data in and data archived from commercial databases are mostly in row major order. The layout does not appear to hinder the performance.

If the dimension tables cannot fit in memory, they can be formed by sorting in linear time, if we make the weak assumption that $(M/B)^c \geq D/B$ for some small positive constant c, where D, M, and B are respectively the dimension table size, the internal memory size, and the block size [4, 29]. This optimization can be obtained automatically if SQL has the multiple

Authorized licensed use limited to: KnowledgeGate from IBM Market Insights. Downloaded on December 15, 2008 at 10:52 from IEEE Xplore. Restrictions apply.

M	size of internal memory
B	size of disk block
N	# of rows in *DETAIL*
n	# of attributes in *DETAIL* (not including class label)
C	# of distinct class labels
L	depth of the final classifier
D_k	total size of all dimension tables at depth k
V	# of distinct values for all attributes
r	size of each record in *DETAIL*
r_a	size of each attribute value in *DETAIL* (for simplicity, we assume that all attribute values are of similar size.)
r_d	size of each record in a dimension table
r_h	size of each record in a hash table (used in SPRINT)

Table 2: Parameters used in analysis

grouping operator proposed in Section 3.1 and with appropriate query optimization, or by appropriate restructuring of the SQL operations. The dimension tables themselves are used in a stream fashion when forming the *UP* and *DOWN* relations. The running time of the algorithm thus scales linearly in practice with the training set size.

Now let's turn to the detailed analysis of the I/O complexity of both algorithms. We will use the parameters in Table 2 (all sizes are measured in bytes) in our analysis.

Each record in *DETAIL* has n attribute values of size r_a, plus a class label that we assume takes one (byte). Thus we have $r = nr_a + 1$. For simplicity we regard r_a as some unit size and thus $r = O(n)$. Each entry in a dimension table consists of one node number, one attribute value, one class label and one count. The largest node number is 2^L, and it can therefore be stored in L bits, which for simplicity we assume can fit in one word of memory. (Typically L is on the order of 10–20. If desired, we can rid ourselves of this assumption on L by rearranging *DETAIL* or a copy of *DETAIL* so that no *leaf_num* field is needed in the dimension tables, but in practice this is not needed.) The largest count is N, so $r_d = O(\log N)$. Counts are used to record multiple instances of a common value in a compressed way, so they always take less space than the original records they represent. We thus have

$$D_k \leq \min\{nN, VC2^k r_d\}. \tag{3}$$

In practice, the second expression in the min term is typically the smaller one, but in our worst-case expressions below we will often bound D_k by nN.

If all dimension tables fit in memory, that is, $D_k \leq M$ for all k, then we only need to read *DETAIL* once at each level, and the I/O complexity of MIND is

$$O\left(\frac{LnN}{B}\right), \tag{4}$$

which is essentially best possible.

In the case when not all dimension tables fit in memory at the same time, but each individual dimension table does, we can form the dimension tables in external memory. This can be done by a single pass through the *DETAIL* table when $M/n > B$ (which is always true in practice). MIND keeps a buffer of size $O(M/n)$ for each dimension table. In a scanning pass of *DETAIL*, MIND accumulates each dimension table in its corresponding buffer; whenever a buffer is full, it is written to disk. When the scanning of *DETAIL* is finished, disk blocks corresponding to each individual dimension table are read into memory, summarized, and sorted into a final dimension table. In the very unlikely scenario where $M/n < B$, a total of $\log_{M/B} n$ passes over *DETAIL* are needed, resulting in a total I/O complexity of $O(\frac{1}{B}\sum_{0\leq k<L} D_k + \frac{LnN}{B}\log_{M/B} n)$.

Now let's consider the worst case in which some individual dimension tables do not fit in memory. We employ a merge-sort process. An interesting point is that the merge sort process here is different from the traditional one: After several passes in the merge sort, the lengths of the runs will not increase anymore; they are upper bounded by the number of rows in the final dimension table, whose size, although too large to fit in memory, is typically small in comparison with N.

Theorem 1 *In the worst case the I/O complexity of MIND is*

$$O\left(\frac{1}{B}\sum_{0\leq k<L} D_k + \frac{nN}{B}\sum_{0\leq k<L}\log_{M/B}\frac{D_k}{B}\right), \tag{5}$$

which is

$$O\left(\frac{LnN}{B}\frac{\log\frac{nN}{B}}{\log\frac{M}{B}}\right). \tag{6}$$

In most applications, the log term is negligible, and the I/O complexity of MIND becomes

$$O\left(\frac{LnN}{B}\right), \tag{7}$$

which matches the optimal time of (4).

Now we analyze the I/O complexity of the SPRINT algorithm. There are two major parts in SPRINT:

Authorized licensed use limited to: KnowledgeGate from IBM Market Insights. Downloaded on December 15, 2008 at 10:52 from IEEE Xplore. Restrictions apply.

the pre-sorting of all attribute lists and the constructing/searching of the corresponding hash tables during partition. Since we are dealing with a very large *DETAIL* table, it is unrealistic to assume that N is small enough to allow hash tables to be stored in memory. Actually those hash tables need to be stored on disk and brought into memory during the partition phase. It is true that hash tables will become smaller at deeper levels and thus fit in memory, but at the early levels they are very large; for example, the hash table at level 0 has N entries.

A careful analysis gives us the following estimation:

Theorem 2 *The I/O complexity of SPRINT is*

$$O\left(\frac{nN^2 \log N}{BM}\right) \tag{8}$$

Examination of (6) and (8) reveals that MIND is clearly better in terms of I/O performance. For large N, SPRINT does a quadratic number of I/Os, whereas MIND scales well.

5 Experimental Results

There are two important metrics to evaluate the quality of a classifier: *classification accuracy* and *classification time*. We compare our results with those of SLIQ [21] and SPRINT [25]. (For brevity, we include only SPRINT in this paper; comparisons showing the improvement of SPRINT over SLIQ are given in [25].) Unlike SLIQ and SPRINT, we use the classical database methodology of summarization. Like SLIQ and SPRINT, we use the same metric (*gini* index) to choose the best split for each node, we grow our tree in a breadth-first fashion, and we prune it using the same pruning algorithm. Our classifier therefore generates a decision tree identical to the one produced by [21, 25] for the same training set, which facilitates meaningful comparisons of run time. The accuracy of SPRINT and SLIQ is discussed in [21, 25], where it is argued that the accuracy is sufficient.

For our scaling experiments, we ran our prototype on large data sets. The main cost of our algorithm is that we need to access *DETAIL* n times (n is the number of attributes) for each level of the tree growth due to the absence of the multiple `GROUP BY` operator in the current SQL standard. We recommend that future DBMSs support the multiple `GROUP BY` operator so that *DETAIL* will be accessed only once regardless of the number of attributes. In our current working prototype, this is done by using user-defined function as we described in Section 3.1.

Owing to the lack of a classification benchmark, we used the synthetic database proposed in [2]. In

attribute	*value*
salary	uniformly distributed from 20K to 150K
commission	$salary \geq 75K \Rightarrow commission = 0$ else uniformly distributed from 10K to 75K
age	uniformly distributed from 20 to 80
loan	uniformly distributed from 0 to 500K
elevel	uniformly chosen from 0 to 4
car	uniformly chosen form 1 to 20
zipcode	uniformly chosen from 10 available zipcodes
hvalue	uniformly distributed from $0.5k100000$ to $1.5k100000$, where $k \in \{0, \ldots, 9\}$ is zipcode
hyear	uniformly distributed from 1 to 30

Table 3: Description of the synthetic data

this synthetic database, each record consists of nine attributes as shown in Table 3. Ten classifier functions are proposed in [2] to produce databases with different complexities. We run our prototype using function 2. It generates a database with two classes: Group A and Group B. The description of the class predicate for Group A is shown below.

Function 2, Group A
$((\text{age} < 40) \wedge (50\text{K} \leq \text{salary} \leq 100\text{K})) \vee$
$((40 \leq \text{age} < 60) \wedge (75\text{K} \leq \text{salary} \leq 125\text{K})) \vee$
$((\text{age} \geq 60) \wedge (25\text{K} \leq \text{salary} \leq 75\text{K}))$

Our experiments were conducted on an IBM RS/6000 workstation running AIX level 4.1.3. and DB2 version 2.1.1. We used training sets with sizes ranging from 0.5 million to 5 million records. The relative response time and response time per example are shown in Figure 2 and Figure 3 respectively. Figure 2 hints that our algorithm achieves linear scalability with respect to the training set size. Figure 3 shows that the time per example curve stays flat when the training set size increases. The corresponding curve for [25] appears to be growing slightly on the largest cases. Figure 4 is the performance comparison between MIND and SPRINT. MIND ran on a processor with a slightly slower clock rate. We can see that MIND performs better than SPRINT does even in the range where SPRINT scales well, and MIND continues to scale well as the data sets get larger.

We also ran MIND on an IBM multiprocessor SP2 computer system. Figure 5 shows the parallel speedup of MIND.

Another interesting measurement we obtained from uniprocessor execution is that accessing *DETAIL* to form the dimension tables for all attributes takes 93%–96% of the total execution time. To achieve linear

Authorized licensed use limited to: KnowledgeGate from IBM Market Insights. Downloaded on December 15, 2008 at 10:52 from IEEE Xplore. Restrictions apply.

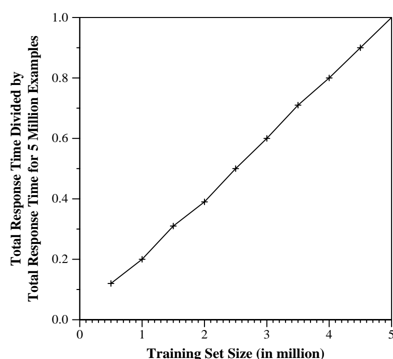

Figure 2: Relative total response time. The y-value denotes the total response time for the indicated training set size, divided by the total response time for 5 million examples.

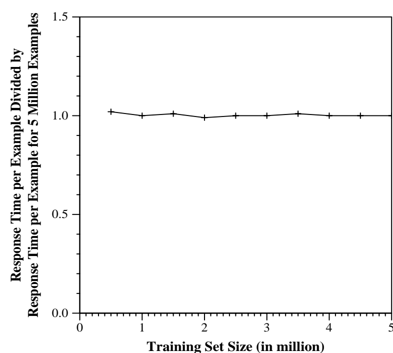

Figure 3: Relative response time per example. The y-value denotes the response time per example for the indicated training set size, divided by response time per example when processing 5 million examples.

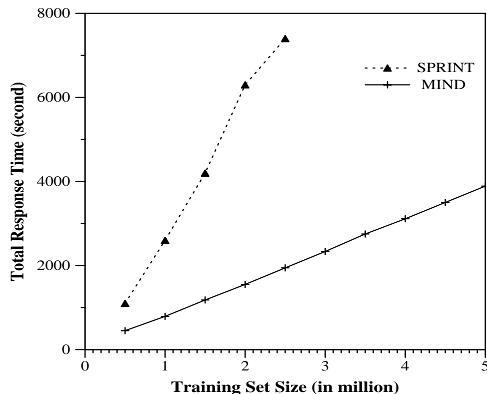

Figure 4: Performance comparison of MIND and SPRINT

speedup on multiprocessors, it is critical that this step is parallelized. In the current working prototype

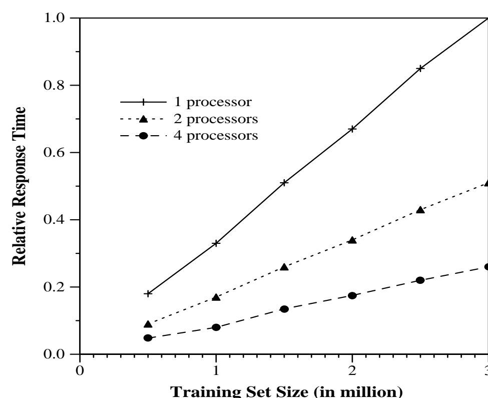

Figure 5: Speedup of MIND for multiprocessors. The y-value denotes the total response time for the indicated training set size, divided by the total response time for 3 million examples.

of MIND, it is done by user-defined function with a scratch-pad accessible from multiple processors.

6 Conclusions

The MIND algorithm solves the problem of classification within the relational database manager and reduces it I/O complexity significantly. Our performance measurements show that MIND demonstrates scalability with respect to the number of examples in training sets and the number of parallel processors. We believe MIND is the first classifier to successfully run on datasets of $N = 5$ million examples on a uniprocessor and yet demonstrate effectively non-increasing response time per example as a function of N. It also runs faster than previous algorithms on file systems.

There are four reasons why MIND is fast, exhibits excellent scalability, and is able to handle data sets larger than those tackled before:

1. MIND rephrases the data mining function classification as a classic DBMS problem of summarization and analysis thereof.
2. MIND avoids any update to the *DETAIL* table of examples using the object extension of DB2. This is of significant practical interest; for example, imagine *DETAIL* having billions of rows.
3. In the absence of a multiple concurrent grouping SQL operator, MIND takes advantage of the user-defined function capability of DB2 to achieve the equivalent functionality and the resultant performance gain.
4. Parallelism of MIND is obtained at little or no extra cost because the RDBMS parallelizes SQL queries.

Authorized licensed use limited to: KnowledgeGate from IBM Market Insights. Downloaded on December 15, 2008 at 10:52 from IEEE Xplore. Restrictions apply.

We recommend that extensions be made to SQL to do multiple groupings and the streaming of each group to different relations. Most DBMS operators currently take two streams of data (tables) and combine them into one. We believe that we have shown the value of an operator that takes a single stream input and produces multiple streams of outputs.

References

[1] P. Adrians and D. Zantinge. *Data Mining.* Addison-Wesley, 1996.

[2] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An interval classifier for database mining applications. In *Proceedings of the 1992 International Conference on Very Large Databases*, pages 560–573, Vancouver, Canada, August 1992.

[3] R. Agrawal and K. Shim. Developing tightly-coupled data mining applications on a relational database system. In *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining*, August 1996.

[4] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4), 1997. Special issue on parallel I/O. An earlier version appears in Proc. of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '96), Padua, Italy, June 1996, 109–118.

[5] G. Bhargava, P. Goel, and B. Iyer. Hypergraph based reordering of outer join queries with complex predicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1995.

[6] L. Breiman et al. *Classification and Regression Trees.* Wadsworth, Belmont, 1984.

[7] J. Catlett. *Megainduction: Machine Learning on Very Large Databases.* PhD thesis, University of Sydney, 1991.

[8] D. Chamberlin. *Using the New DB2: IBM's Object-Relational Database System.* Morgan Kaufmann, 1996.

[9] D. Chamberlin. Personal communication, 1997.

[10] D. Chamberlin et al. Seqel: A structured english query language. In *Proc. of ACM SIGMOD Workshop on Data Description, Access, and Control*, May 1974.

[11] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6), June 1970.

[12] T. Dietterich, M. Kearns, and Y. Mansour. Applying the weak learning framework to understand and improve C4.5. In *Proceedings of the 13th International Conference on Machine Learning*, pages 96–104, 1996.

[13] R. A. Ganski and H. K. T. Wong. Optimization of nested sql queried revisited. In *Proceeding of the 1987 ACM SIGMOD International Conference on Management of Data*, 1987.

[14] S. Hasty. Mining databases. *Apparel Industry Magazine*, 57(5), 1996.

[15] IBM. *IBM DATABASE 2 Application Programming Guide-for common servers*, version 2 edition.

[16] IBM Germany. *IBM Intelligence Miner User's Guide*, version 1 edition, July 1996.

[17] T. Imielinsk and H. Mannila. A database perspective on knowledge discovery. *Communication of the ACM*, 39(11), November 1996.

[18] T. Imielinski. From file mining to database mining. In *Proceedings of the 1996 SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, May 1996.

[19] M. Kearns and Y. Mansour. On the boosting ability of top-down decision tree learning algorithms. In *Proceedings of the 28th ACM Symposium on the Theory of Computing*, pages 459–468, 1996.

[20] H. Lu et al. On preprocessing data for efficient classification. In *Proceedings of SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, May 1996.

[21] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proceedings of the 5th International Conference on Extending Database Technology*, Avignon, France, March 1996.

[22] H. Messatfa. Personal communications, 1997.

[23] S. K. Murthy. *On Growing Better Decision Trees from Data.* PhD thesis, Johns Hopkins University, 1995.

[24] J. Ross Quilan. *C4.5: Programs for Machine Learning.* Morgan Kaufman, 1993.

[25] J. C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the 1996 International Conference on Very Large Databases*, Mumbai (Bombay), India, September 1996.

[26] J. B. Sinclair. Personal communication, 1997.

[27] M. Stonebraker and L. A. Rowe. The design of postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, 1986.

[28] J. Ullman. *Principles of Database Systems.* Computer Science Press, second edition, 1982.

[29] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of the Goddard Conference on Mass Storage Systems and Technologies*, NASA Conference Publication 3340, Volume II, pages 553–570, College Park, MD, September 1996.

Authorized licensed use limited to: KnowledgeGate from IBM Market Insights. Downloaded on December 15, 2008 at 10:52 from IEEE Xplore. Restrictions apply.