Scalable Mining for Classification Rules in Relational Databases

Min Wang∗
Data Management Department
IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532, USA
min@us.ibm.com
Phone: (914) 784-6268
FAX: (914) 784-7455

Bala Iyer
IBM Silicon Valley Lab
555 Bailey Avenue
San Jose, CA 95141, USA
balaiyer@us.ibm.com

Jeffrey Scott Vitter†
Purdue University
150 North University Street
West Lafayette, IN 47907, USA
jsv@purdue.edu

∗Contact author. Support was provided in part by an IBM Graduate Fellowship.
†Support was provided in part by the Army Research Office through research grant DAAD19-03-1-0321, by the National Science Foundation through research grant CCR-9877133, and by an IBM research award.
Abstract

Data mining is a process of discovering useful patterns (knowledge) hidden in extremely large datasets. Classification is a fundamental data mining function, and some other functions can be reduced to it. 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 have built a prototype of MIND in the relational database management system DB2 and have 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 datasets but also with the number of processors on an IBM SP2 computer system. Even on uniprocessors, MIND scales well beyond 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 [Has96]. 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 of IM as well as previous investigations [Imi96, IM96] have pointed to the need for the relational database management systems to have these functions built in. Besides reducing I/O costs, this approach leverages over 20 years of research and development in DBMS technology, among them are:

- scalability,
- memory hierarchy management [Sar95, SS96],
- parallelism [B+95],
- optimization of the executions [BGI95],
- platform independence, and
- client server API [NS96].
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 [B+84, Qui93, Mur95, Cat91]).

Several classification models have been proposed in the literature, including neural network, decision trees, statistical models, and genetic models. Among these models, decision tree model is particularly suited for data mining applications due to the following reasons: (1) ease of construction, (2) simple and easy to understand, and (3) acceptable accuracy [SAM96]. Therefore, we focus on decision tree model in this paper. A simple illustration of training data is shown in Table 1. The examples reflect the past experience of an organization extending credit. From these 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 internal memory. In data mining applications, it is common to have training sets with several million examples. It is observed in [MAR96] 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 [DKM96, KM96].

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 [MAR96, SAM96, IBM96], 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 data 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 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 [AZ96, L+96] 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 [VV96]. 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 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 [SAM96]. 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; an illustrative example is given in Section 4. A theoretical performance analysis is given in Section 5. We revisit MIND algorithm in Section 6 using a general extension of current SQL standards. In Section 7, we present our experimental results. We make concluding remarks in Section 8.

\(^1\)SQL is simply an implementation \([C^+74]\) of the relational calculus proposed in \([Cod70]\). A few extensions have been done since then \([Ull82, C^+74]\).
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 [MAR96, SAM96] 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 of 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 [MAR96] 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 [SAM96], 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 [SAM96], 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 method, the information we need to evaluate the split and perform the partition is stored in relations in a database. Thus we can take advantage of DBMS functionalities and memory management. The only thing we need to do is to incorporate a data structure that relates the database relations to the growing classification tree. We assign a unique number to

---

1A splitting index is used to choose from alternative splits. We use the gini index (defined in Section 2.3) in our algorithm.
each node in the tree. When loading the training data into the database, imagine the addition of a hypothetical column \( \text{leaf}_\text{num} \) to each row. For each training example, \( \text{leaf}_\text{num} \) will always indicate which leaf node in the current tree it belongs to. When the tree grows, the \( \text{leaf}_\text{num} \) value changes to indicate that the record is moved to a new node by applying a split. A static array called \( \text{LNL} \) (leaf node list) is used to relate the \( \text{leaf}_\text{num} \) value in the relation to the corresponding node in the tree. By using a labeling technique, we insure that at each tree growing stage, the nodes always have the identification numbers 0 through \( N - 1 \), where \( N \) is the number of nodes in the tree. \( \text{LNL}[i] \) is a pointer to the node with identification number \( i \). For any record in the relation, we can get the leaf node it belongs to from its \( \text{leaf}_\text{num} \) value and \( \text{LNL} \) and hence we can get the information in the node (e.g. split attribute and value, number of examples belonging to this node and their class distribution).

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

### 2.3 Computing the \( \text{gini} \) index

A splitting index is used to choose from alternative splits for each node. Several splitting indices have recently been proposed. We use the \( \text{gini} \) index, originally proposed in [B+84] and used in [MAR96, SAM96], because it gives acceptable accuracy. The accuracy of our classifier is therefore the same as those in [MAR96, SAM96].

For a data set \( S \) containing \( N \) examples from \( C \) classes, \( \text{gini}(S) \) is defined as

\[
\text{gini}(S) = 1 - \sum_{i=1}^{C} p_i^2
\]

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 \( \text{gini} \) index of the divided data \( \text{gini}_{\text{split}}(S) \) is given by

\[
\text{gini}_{\text{split}}(S) = \frac{N_1}{N} \text{gini}(S_1) + \frac{N_2}{N} \text{gini}(S_2)
\]

The attribute containing the split point achieving the smallest \( \text{gini} \) index value is then chosen to split the node [B+84]. Computing the \( \text{gini} \) index is the most expensive part of the algorithm since finding the best split for a node requires evaluating the \( \text{gini} \) 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: \( \text{DETAIL}(\text{attr}_1, \text{attr}_2, \ldots, \text{attr}_n, \text{class}, \text{leaf}_\text{num}) \), where \( \text{attr}_i \) is the \( i \)th attribute, for \( 1 \leq i \leq n \), \( \text{class} \) is the classifying attribute, and \( \text{leaf}_\text{num} \) denotes which leaf in the classification tree the record belongs to.\(^2\) In actuality \( \text{leaf}_\text{num} \) 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 \( \text{leaf}_\text{num} \) value of each record in the training set keeps changing. Because \( \text{leaf}_\text{num} \) is a computed attribute, the \( \text{DETAIL} \) table is never updated, a key reason why MIND is efficient for the DB2 relational database. 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 \( n \).

\(^{2}\) \( \text{DETAIL} \) itself may be a summary of atomic transactional data, or the atomic data.
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 $DIM_i(\text{leaf\_num, class, attr}_i, \text{count})$ using a simple SQL SELECT statement on $DETAIL$:

\begin{verbatim}
INSERT INTO $DIM_i$  
SELECT leaf\_num, class, $attr_i$, COUNT(*) 
FROM $DETAIL$ 
WHERE leaf\_num <> STOP 
GROUP BY leaf\_num, class, $attr_i$
\end{verbatim}

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 tree) \times ($\#$distinct values on $attr_i$) \times ($\#$distinct classes), which is very likely to be of the order of several hundreds [Mes97]. 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 [AZ96, L+96]. Discretization of variable values into a smaller number of classes is sometimes referred to as “encoding” in data mining practice [AZ96]. Roughly speaking, this is done to obtain a measure of aggregate behavior that may be detectable [Mes97]. 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 5.

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 $DIM_i$.

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 [Sin97].

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:

\begin{verbatim}
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) [Cha97]).
SELECT FROM DETAIL
INSERT INTO \textit{DIM}_1\{leaf\_num, class, \textit{attr}_1, \text{COUNT}(*)
  WHERE \textit{predicate}
GROUP BY \textit{leaf\_num, class, \textit{attr}_1}
INSERT INTO \textit{DIM}_2\{leaf\_num, class, \textit{attr}_2, \text{COUNT}(*)
  WHERE \textit{predicate}
GROUP BY \textit{leaf\_num, class, \textit{attr}_2}
  
\vdots
INSERT INTO \textit{DIM}_n\{leaf\_num, class, \textit{attr}_n, \text{COUNT}(*)
  WHERE \textit{predicate}
GROUP BY \textit{leaf\_num, class, \textit{attr}_n}
\end{verbatim}

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 \textit{user-defined function} (udf) [SR86, Cha96, IBM], which is another reason why MIND is efficient. User-defined functions are used for association in [AS96]. User-defined function is a new feature provided by DB2 version 2 [Cha96, IBM]. In DB2 version 2, the functions available for use in SQL statements extend from the system built-in functions, such as \textit{avg}, \textit{min}, \textit{max}, \textit{sum}, to more general categories, such as \textit{user-defined functions} (udf). An external udf is a function that is written by a user in a host programming language. The \texttt{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 \textit{DETAIL}.

For each leaf in the tree, possible split values for attribute \textit{i} are all distinct values of \textit{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 \textit{gini} index. We collect such distribution information in two relations, \textit{UP} and \textit{DOWN}.

Relation \textit{UP} with the schema \textit{UP}(leaf\_num, \textit{attr}_i, class, count) can be generated by performing a self-outer-join on \textit{DIM}_i:

\begin{verbatim}
INSERT INTO \textit{UP}
SELECT \textit{d}_1.leaf\_num, \textit{d}_1.\textit{attr}_i, \textit{d}_1.class, \text{SUM}(\textit{d}_2.count)
FROM (FULL OUTER JOIN \textit{DIM}_i \textit{d}_1, \textit{DIM}_i \textit{d}_2
  ON \textit{d}_1.leaf\_num = \textit{d}_2.leaf\_num AND
    \textit{d}_2.\textit{attr}_i \leq \textit{d}_1.\textit{attr}_i AND
    \textit{d}_1.class = \textit{d}_2.class
GROUP BY \textit{d}_1.leaf\_num, \textit{d}_1.\textit{attr}_i, \textit{d}_1.class)
\end{verbatim}

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

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

\begin{verbatim}
CREATE VIEW \textit{C}_k\_UP(leaf\_num, \textit{attr}_i, count) AS
SELECT leaf\_num, \textit{attr}_i, count
FROM \textit{UP}
WHERE class = \textit{k}
\end{verbatim}
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 u$1$.leaf$\_num$, u$1$.attr$\_i$, $f_{gini}$
FROM $C_1$ UP u$1$, ..., $C_n$ UP u$C$, $C_1$ $DOWN$ d$1$, ..., $C_n$ $DOWN$ d$C$
WHERE u$1$.attr$\_i$ = ... = u$C$.attr$\_i$ = d$1$.attr$\_i$ = ... = d$C$.attr$\_i$ AND
u$1$.leaf$\_num$ = ... = u$C$.leaf$\_num$ = d$1$.leaf$\_num$ = ... = d$C$.leaf$\_num$

where $f_{gini}$ is a function of $u_1.count$, ..., $u_n.count$, $d_1.count$, ..., $d_n.count$ 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$\_num$, i$^1$, attr$\_i$$^2$, 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 [BGI95, Mur95, GW87].

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 [SAM96], but formed with set-oriented operators.

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 a recursive 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:

---

$^1$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 $attr_i$. 

9
INSERT INTO $S_k \downarrow \text{IN}$
SELECT $p.leaf\_num, p.class, p.v_1, \ldots, p.v_{k-1}, q.v_1, p.count + q.count$
FROM (FULL OUTTER JOIN $S_{k-1} \downarrow \text{IN} \ p$, $S_j \downarrow \text{IN} \ q$
  ON $p.leaf\_num = q.leaf\_num$ AND
  $p.class = q.class$ AND
  $q.v_1 > p.v_{k-1}$)

We generate relation $S_k \downarrow \text{OUT}$ from $S_k \downarrow \text{IN}$ in a manner similar to how we generate $\text{DOWN}$ from $\text{UP}$. Then we treat $S_k \downarrow \text{IN}$ and $S_k \downarrow \text{OUT}$ exactly as $\text{DOWN}$ and $\text{UP}$ for numerical attributes in order to compute the $\text{gini}$ 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 $\text{UPDATE}$ performs the split for each leaf:

UPDATE $DETAIL$
SET $leaf\_num = \text{Partition}(\text{attr}_1, \ldots, \text{attr}_n, \text{class}, \text{leaf}\_num)$

The user-defined function $\text{Partition}$ defined on a record $r$ \footnote{$r = (\text{attr}_1, \ldots, \text{attr}_n, \text{class}, \text{leaf}\_num)$} of $DETAIL$ as follows:

$\text{Partition}(\text{record } r)$
  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 to 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 $\text{Partition}$ is merely applying the current tree to the original training set. We avoid the update by replacing $leaf\_num$ by function $\text{Partition}$ in the statement forming $\text{DIM}_i$. 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 $\text{gini}$ index computation for all leaves involves only dimension tables.

4 An Example

We illustrate our algorithm by an example. The example training set is the same as the data in Table 1.

Phase 0: Load the training set and initialize the tree and $LNL$. At this stage, relation $DETAIL$, the tree, and $LNL$ are shown in Table 2 and Figure 2.

Phase 1: Form the dimension tables for all attributes in one pass over $DETAIL$ using user-defined function. The result dimension tables are show in Table 3–4.
<table>
<thead>
<tr>
<th>\textit{attr}_1</th>
<th>\textit{attr}_2</th>
<th>\textit{class}</th>
<th>\textit{leaf_num}</th>
</tr>
</thead>
<tbody>
<tr>
<td>65K</td>
<td>30</td>
<td>Safe</td>
<td>0</td>
</tr>
<tr>
<td>15K</td>
<td>23</td>
<td>Risky</td>
<td>0</td>
</tr>
<tr>
<td>75K</td>
<td>40</td>
<td>Safe</td>
<td>0</td>
</tr>
<tr>
<td>15K</td>
<td>28</td>
<td>Risky</td>
<td>0</td>
</tr>
<tr>
<td>100K</td>
<td>55</td>
<td>Safe</td>
<td>0</td>
</tr>
<tr>
<td>60K</td>
<td>45</td>
<td>Safe</td>
<td>0</td>
</tr>
<tr>
<td>62K</td>
<td>30</td>
<td>Risky</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2: Initial relation \textit{DETAIL} with implicit \textit{leaf\_num}

![Initial tree diagram]

Figure 2: Initial tree

<table>
<thead>
<tr>
<th>\textit{leaf_num}</th>
<th>\textit{attr}_1</th>
<th>\textit{class}</th>
<th>\textit{count}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>60</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>62</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>65</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3: Relation \textit{DIM}_1

<table>
<thead>
<tr>
<th>\textit{leaf_num}</th>
<th>\textit{attr}_2</th>
<th>\textit{class}</th>
<th>\textit{count}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>23</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>28</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>30</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>30</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>40</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>45</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>55</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4: Relation \textit{DIM}_2

**Phase 2:** Find the best splits for current leaf nodes. A best split is found through a set of operations on relations as described in Section 2.

First we evaluate the \textit{gini} index value for \textit{attr}_1. The procedure is depicted in Table 5–13.

We can see that the best splits on the two attributes achieve the same \textit{gini} index value, so relation \textit{BEST\_SPLIT} is the same as \textit{MIN\_GINI} except that it does not contain the column \textit{gini}. We store the best split in each leaf node of the tree (the root node in this phase). In case of a tie for best split at a node, any one of them (\textit{attr}_2 in our example) can be chosen.
<table>
<thead>
<tr>
<th>leaf_num</th>
<th>attr1</th>
<th>class</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>15</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>60</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>60</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>62</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>62</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>65</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>65</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>75</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>75</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5: Relation UP

<table>
<thead>
<tr>
<th>leaf_num</th>
<th>attr1</th>
<th>class</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>15</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>60</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>60</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>62</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>62</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>65</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>65</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>75</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6: Relation DOWN

<table>
<thead>
<tr>
<th>leaf_num</th>
<th>attr1</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>60</td>
<td>1.0</td>
</tr>
<tr>
<td>0</td>
<td>62</td>
<td>1.0</td>
</tr>
<tr>
<td>0</td>
<td>65</td>
<td>2.0</td>
</tr>
<tr>
<td>0</td>
<td>75</td>
<td>3.0</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>4.0</td>
</tr>
</tbody>
</table>

Table 7: Relation C1_UP

Phase 3: Partitioning. According to the best split found in Phase 2, we grow the tree and partition the training set. The partition is reflected as leaf_num updates in relation DETAIL. Any new grown node that is pure or “small enough” is marked and reassigned a special leaf_num value STOP so that it is not processed further. The tree is shown in Figure 3 and the new DETAIL is shown in Table 14. Again, note leaf_num is never stored in DETAIL, so no update to DETAIL is necessary.
<table>
<thead>
<tr>
<th>leaf_num</th>
<th>attr1</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>2.0</td>
</tr>
<tr>
<td>0</td>
<td>60</td>
<td>2.0</td>
</tr>
<tr>
<td>0</td>
<td>62</td>
<td>3.0</td>
</tr>
<tr>
<td>0</td>
<td>65</td>
<td>3.0</td>
</tr>
<tr>
<td>0</td>
<td>75</td>
<td>3.0</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Table 8: Relation $C_2\_UP$

<table>
<thead>
<tr>
<th>leaf_num</th>
<th>attr1</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>4.0</td>
</tr>
<tr>
<td>0</td>
<td>60</td>
<td>3.0</td>
</tr>
<tr>
<td>0</td>
<td>62</td>
<td>3.0</td>
</tr>
<tr>
<td>0</td>
<td>65</td>
<td>2.0</td>
</tr>
<tr>
<td>0</td>
<td>75</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 9: Relation $C_1\_DOWN$

<table>
<thead>
<tr>
<th>leaf_num</th>
<th>attr1</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>1.0</td>
</tr>
<tr>
<td>0</td>
<td>60</td>
<td>1.0</td>
</tr>
<tr>
<td>0</td>
<td>62</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>65</td>
<td>0.0</td>
</tr>
<tr>
<td>0</td>
<td>75</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 10: Relation $C_2\_DOWN$

<table>
<thead>
<tr>
<th>leaf_num</th>
<th>attr1</th>
<th>gini</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>0.22856</td>
</tr>
<tr>
<td>0</td>
<td>60</td>
<td>0.40474</td>
</tr>
<tr>
<td>0</td>
<td>62</td>
<td>0.21428</td>
</tr>
<tr>
<td>0</td>
<td>65</td>
<td>0.34284</td>
</tr>
<tr>
<td>0</td>
<td>75</td>
<td>0.42856</td>
</tr>
</tbody>
</table>

Table 11: Relation $GINI\_VALUE$

<table>
<thead>
<tr>
<th>leaf_num</th>
<th>attr_name</th>
<th>attr_value</th>
<th>gini</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>62</td>
<td>0.21428</td>
</tr>
</tbody>
</table>

Table 12: Relation $MIN\_GINI$ after $attr_1$ is evaluated

<table>
<thead>
<tr>
<th>leaf_num</th>
<th>attr_name</th>
<th>attr_value</th>
<th>gini</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>62</td>
<td>0.21428</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>leaf_num</th>
<th>attr_name</th>
<th>attr_value</th>
<th>gini</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>30</td>
<td>0.21428</td>
</tr>
</tbody>
</table>

Table 13: Relation $MIN\_GINI$ after $attr_1$ and $attr_2$ are evaluated
Figure 3: Decision tree at Phase 3

<table>
<thead>
<tr>
<th>attr1</th>
<th>attr2</th>
<th>class</th>
<th>leaf_num</th>
</tr>
</thead>
<tbody>
<tr>
<td>65K</td>
<td>30</td>
<td>Safe</td>
<td>1</td>
</tr>
<tr>
<td>15K</td>
<td>23</td>
<td>Risky</td>
<td>1</td>
</tr>
<tr>
<td>75K</td>
<td>40</td>
<td>Safe</td>
<td>2⇒STOP</td>
</tr>
<tr>
<td>15K</td>
<td>28</td>
<td>Risky</td>
<td>1</td>
</tr>
<tr>
<td>100K</td>
<td>55</td>
<td>Safe</td>
<td>2⇒STOP</td>
</tr>
<tr>
<td>60K</td>
<td>45</td>
<td>Safe</td>
<td>2⇒STOP</td>
</tr>
<tr>
<td>62K</td>
<td>30</td>
<td>Risky</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 14: Relation DETAIL with implicit leaf_num after Phase 3

Figure 4: Final decision tree

<table>
<thead>
<tr>
<th>attr1</th>
<th>attr2</th>
<th>class</th>
<th>leaf_num</th>
</tr>
</thead>
<tbody>
<tr>
<td>65K</td>
<td>30</td>
<td>Safe</td>
<td>4⇒STOP</td>
</tr>
<tr>
<td>15K</td>
<td>23</td>
<td>Risky</td>
<td>3⇒STOP</td>
</tr>
<tr>
<td>75K</td>
<td>40</td>
<td>Safe</td>
<td>STOP</td>
</tr>
<tr>
<td>15K</td>
<td>28</td>
<td>Risky</td>
<td>3⇒STOP</td>
</tr>
<tr>
<td>100K</td>
<td>55</td>
<td>Safe</td>
<td>STOP</td>
</tr>
<tr>
<td>60K</td>
<td>45</td>
<td>Safe</td>
<td>STOP</td>
</tr>
<tr>
<td>62K</td>
<td>30</td>
<td>Risky</td>
<td>3⇒STOP</td>
</tr>
</tbody>
</table>

Table 15: Final relation DETAIL with implicit leaf_num

**Phase 4:** Repeat Phase 1 through Phase 3 until all the leaves in the tree become STOP leaves. The final tree and DETAIL are shown in Figure 4 and Table 15.
5 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 [SAM96] forms an attribute list (projection of the $DETAIL$ table) for each attribute. In
order to reduce the cost of computing the $gini$ index, SPRINT presents 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 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 [BGV97, VV96]. This optimization can be obtained automatically if SQL has the multiple
grouping operator proposed in Section 3.1 and with appropriate query optimization, or by ap-
propriate 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 16 (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

$$D_k \leq \min\{nN, VC2^kr_d\}. \quad (3)$$

In practice, the second expression in the min term is typically the smaller one, but in our worst-
<table>
<thead>
<tr>
<th>$M$</th>
<th>size of internal memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>size of disk block</td>
</tr>
<tr>
<td>$N$</td>
<td># of rows in DETAIL</td>
</tr>
<tr>
<td>$n$</td>
<td># of attributes in DETAIL (not including class label)</td>
</tr>
<tr>
<td>$C$</td>
<td># of distinct class labels</td>
</tr>
<tr>
<td>$L$</td>
<td>depth of the final classifier</td>
</tr>
<tr>
<td>$D_k$</td>
<td>total size of all dimension tables at depth $k$</td>
</tr>
<tr>
<td>$V$</td>
<td># of distinct values for all attributes</td>
</tr>
<tr>
<td>$r$</td>
<td>size of each record in DETAIL</td>
</tr>
<tr>
<td>$r_a$</td>
<td>size of each attribute value in DETAIL (for simplicity, we assume that all attribute values are of similar size.)</td>
</tr>
<tr>
<td>$r_d$</td>
<td>size of each record in a dimension table</td>
</tr>
<tr>
<td>$r_h$</td>
<td>size of each record in a hash table (used in SPRINT)</td>
</tr>
</tbody>
</table>

Table 16: Parameters used in analysis

case expressions below we will often bound $D_k$ by $nN$.

Claim 1 If all dimension tables fit in memory, that is, $D_k \leq M$ for all $k$, the I/O complexity of MIND is

$$O\left(\frac{\ln N}{B}\right),$$

which is essentially best possible.

Proof: If all dimension tables fit in memory, then we only need to read DETAIL once at each level. Dimension tables for all attributes are accumulated in memory when each DETAIL record is read in. When the end of DETAIL table is reached, we’ll have all the unsorted dimension tables in memory. Then sorting and gini index computation are performed for each dimension table, best split will be found for each current leaf node.

The I/O cost to read in DETAIL once is $rN/B = O(nN/B)$, and there are $L$ levels in the final classifier, so the total I/O cost is $O(L/nN/B)$.

Claim 2 In the case when not all dimension tables fit in memory at the same time, but each individual dimension table does, the I/O complexity of MIND is

$$O\left(\frac{\ln N}{B} \log_{M/B} n\right).$$

Proof: In the case when not all dimension tables fit in memory at the same time, but each individual dimension table does, we can form, use and discard each dimension table on the fly. 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. In scanning DETAIL, for each dimension, its buffer is used to store the accumulated information. Whenever a buffer is full, it is written to disk. When the scanning of DETAIL is finished, many blocks have been obtained for each dimension based on which the final dimension table can be formed easily. For example, there might be two entries $(1, 1, 1, count_1)$, $(1, 1, 1, count_2)$ in two blocks for $attr_1$. They are corresponding to an entry with $leaf\_num = 1$, $class = 1$, $attr_1 = 1$ in the final dimension table.
for \( attr_1 \) and will become a entry \((1, 1, 1, count_1 + count_2)\) in the final dimension table. All those blocks that corresponds to one dimension are collectively called an intermediate dimension table for that dimension.

Now the intermediate dimension table for the first attribute is read into memory, summarized, and sorted into a final dimension table. Then MIND calculates the gini index values with respect to this dimension for each leaf node, and keeps the current minimum gini index value and the corresponding \((attribute\_name, attribute\_value)\) pair in each leaf node. When the calculation for the first attribute is done, the in-memory dimension table is discarded. MIND repeats the same procedure for the second attribute, and so on. Finally, we get the best splits for all leaf nodes and we are ready to grow the tree one more level. The I/O cost at level \( k \) is scanning \( DETAIL \) once, plus writing out and reading in all the intermediate dimension tables once. We denote the total size of all intermediate dimension tables at level \( k \) by \( D'_k \). Note that the intermediate dimension tables are a compressed version of the original \( DETAIL \) table, and they take much less space than the original records they represent. So we have

\[
D'_k \leq nN.
\]

The I/O cost at each level is

\[
O \left( \frac{1}{B} \sum_{0 \leq k < L} D'_k + \frac{LnN}{B} \right) = O \left( \frac{LnN}{B} \right).
\]

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 in (5).

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 tables, whose size, although too large to fit in memory, is typically small in comparison with \( N \).

We formally define the special sort problem. We adopt the notations used in [Vit01]:

\[
\begin{align*}
N &= \text{problem size (in units of data items),} \\
M &= \text{internal memory size (in units of data items),} \\
B &= \text{block size (in units of data items),} \\
m &= \frac{M}{B}, \text{ number of blocks that fits into internal memory,}
\end{align*}
\]

where \( 1 \leq B \leq M < N \).

The special sort problem can be defined as follows:

**Definition 1** There are \( N'(N' \ll N) \) distinct keys, \( \{k_1, k_2, \ldots, k_{N'}\} \), and we assume \( k_1 < k_2 < \ldots < k_{N'} \) for simplicity. We have \( N \) data items \( (k_{x(i)}, count_i) \), for \( 1 \leq i \leq N \), \( 1 \leq x(i) \leq N' \).

The goal is to obtain \( N' \) data items with the key in sorted increasing order and the corresponding count summarized; that is, \((k_i, COUNT_i)\), where

\[
COUNT_i = \sum_{1 \leq k \leq N, x(k) = i} count_k
\]
for $1 \leq i \leq N'$.

**Lemma 1** The I/O complexity of the special sort problem is

$$O\left(\frac{N}{B} \log m \frac{N'}{B}\right)$$

**(Proof)** We perform a modified merge sort procedure for the special sort problem. First $N/M$ sorted “runs” are formed by repeatedly filling up the internal memory, sorting the records according to their key values, combining the records with the same key and summarizing their counts, and writing the results to disk. This requires $O\left(\frac{N}{B}\right)$ I/Os. Next $m$ runs are continually merged and combined together into a longer sorted run, until we end up with one sorted run containing all the $N'$ records.

In a traditional merge sort procedure, the crucial property is that we can merge $m$ runs together in a linear number of I/Os. To do so we simply load a block from each of the runs and collect and output the $B$ smallest elements. We continue this process until we have processed all elements in all runs, loading a new block from a run every time a block becomes empty. Since there are $O\left(\log_m \frac{N'}{m}\right)$ levels in the merge process, and each level requires $O\left(\frac{N}{B}\right)$ I/O operations, we obtain the $O\left(\frac{N'}{B} \log_m \frac{N'}{B}\right)$ complexity for the normal sort problem.

An important difference between the special sort procedure and the traditional one is that in the former, the length of each sorted run will not go beyond $N'$ while in the latter, the length of sorted runs at each level keeps increasing (doubling) until reaching $N$.

In the special sort procedure, at and after level $k = \lceil \log_m \frac{N'}{B} \rceil$, the length of any run will be bounded by $N'$ and the number of runs is bounded by $\lceil \frac{N'}{m}\rceil$. (For simplicity, we will ignore all the floors and ceilings in the following discussion.) From level $k + 1$ on, the operation we perform at each level is basically combining each $m$ runs (each with a length less than or equal to $N'$) into one run whose length is still bounded by $N'$. We repeat this operation at each level until we get a single run. At level $k + i$, we combine $\frac{N/B}{m^{i+1}}$ runs into $\frac{N/B}{m^{i+2}}$ runs and the I/O at this level is

$$\frac{N/B}{m^{i+1}} \left(1 + \frac{1}{m}\right).$$

We will finish the combining procedure at level $k + p$ where $p = \log_m \frac{N'/B}{m'}$, $n' = N'/B$. So the I/O for the whole special sort procedure is:

$$2 \frac{N}{B} k + \frac{N}{B}(1 + \frac{1}{m}) + \frac{N/B}{m}(1 + \frac{1}{m}) + \ldots + \frac{N/B}{m^{i-1}}(1 + \frac{1}{m})$$

$$\approx 2 \frac{N}{B} \log_m n' + \frac{N/B}{B}(1 + \frac{1}{m}) - \frac{1}{1/m}$$

$$= O\left(\frac{N}{B} \log_m \frac{N'}{B}\right).$$

$\square$

Now we are ready to give the I/O complexity of MIND in the worst case.

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

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

(7)
which is

\[
O \left( \frac{L_n N \log \frac{n N}{B}}{B \log \frac{n N}{B}} \right). 
\]

(8)

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

\[
O \left( \frac{L_n N}{B} \right), 
\]

(9)

which matches the optimal time of (4).

Proof: This is similar to the proof in Claim 2. At level \( k \) of the tree growth phase, MIND first forms all the intermediate dimension tables with total size \( D'_k \) in external memory. This can be done by a single pass through the DETAIL table, as follows. MIND keeps a buffer of size \( O(M/n) \) for each dimension. In scanning DETAIL, MIND accumulates information for each dimension in its corresponding buffer; whenever a buffer is full, it is written to disk. When the scanning of DETAIL is finished, MIND performs the special merge sort procedure for the disk blocks corresponding to all (not individual) dimension tables. At the last level of the special sort, the final dimension table for each attribute will be formed one by one. MIND calculates the \( gini \) index values with respect to each dimension for each leaf node, and keeps the current minimum \( gini \) index value and the corresponding \( (attribute\_name, attribute\_value) \) pair in each leaf node. When the calculation for the last attribute is done, we get the best splits for all leaf nodes and we are ready to grow the tree one more level.

The I/O cost at level \( k \) is scanning DETAIL once, which is \( O(nN/B) \), plus the cost of writing out all the intermediate dimension tables once, which is bounded by \( O(nN/B) \), plus the cost for the special sort, which is \( O(\frac{N}{B} \log_{M/B} D_k/B) \).

So the I/O for all levels is

\[
\frac{L_n N}{B} + \frac{1}{B} \sum_{0 < k < L} D'_k + \frac{n N}{B} \sum_{0 < k < L} \log_{M/B} \frac{D_k}{B},
\]

which is

\[
O \left( \frac{L_n N}{B} + \frac{n N}{B} \sum_{0 < k < L} \log_{M/B} \frac{D_k}{B} \right).
\]

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

Each entry in a hash table contains a \( tid \) (transaction identifier) which is an integer in the range of 1 to \( N \), and one bit that indicates which child this record should be partitioned to in the next level of the classifier. So we have

\[
r_h = \frac{1 + \log N}{8}.
\]
We can estimate when the hash tables will fit in memory, given the optimistic assumptions that all memory is allocated to hash tables and all hash tables at each node have equal size; that is, a hash table at level \( k \) contains \( N/2^k \) entries. Thus, a hash table at level \( k \) fits in memory if \( r_k N/2^k \leq M \), or

\[ 2^k \geq \frac{N}{M} \left( \frac{1 + \log N}{8} \right). \tag{10} \]

For sufficiently large \( k \), (10) will be satisfied, that is, hash tables become smaller at deeper nodes and thus fit in memory. But it is clear that even for moderately large detail tables, hash tables at upper levels will not fit in memory.

During the partition phase, each non-splitting attribute list at each node needs to be partitioned into two parts based on the corresponding hash table. One way to do this is to do a random hash table search for each entry in the list, but this is very expensive. Fortunately, there is a better way: First, we bring a large portion of the hash table into memory. The size of this portion is limited only by the availability of the internal memory. Then we scan the non-splitting list once, block by block, and for each entry in the list, we search the in-memory portion of the hash table. In this way, the hash table is swapped into memory only once, and each non-splitting attribute list is scanned \( N/M \) times. For even larger \( N \), it is better to do the lookup by batch sorting, but that approach is completely counter to the founding philosophy of the SPRINT algorithm.

A careful analysis gives us the following estimation:

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

\[ O \left( \frac{nN^2 \log N}{BM} \right) \tag{11} \]

**Proof:** To perform the pre-sort of the SPRINT algorithm, we need to read \( DETAIL \) once, write out the unsorted attribute lists, and sort all the attribute lists. So we have

\[ IO_{pre-sort} = O \left( \frac{nN \log M}{B} \frac{N}{B} \right). \]

From level 0 through level \( k - 1 \), hash tables will not fit in memory. At level \( i \) (\( 0 \leq i \leq k - 1 \)), SPRINT will perform the following operations:

1. Scan the attribute lists one by one to find the best split for each leaf node.
2. According to the best split found for each leaf node, form the hash tables and write them to disk.
3. Partition the attribute list of the splitting attribute for each leaf node.
4. Partition the attribute lists for the \( n - 1 \) non-splitting attributes for each leaf node.

Among these operations, the last one incurs the most I/O cost and we perform it by bringing a portion of a hash table into memory first. The size of this portion is limited only by the availability of the main memory. Then we scan each non-splitting list once, block by block, and for each entry in the list, we search the in-memory portion of the hash table and decide which child this entry should go in the next level. In this way, the hash table is swapped into memory only once, and the non-splitting list is scanned multiple times. The I/O cost of this operation is

\[ O \left( \frac{nNnhi}{B} \right) \]
where \( h_i \) is the number of portions we need to partition a hash table into due to the limitation of the memory size.

From level \( k \) to level \( L \) the hash table will fit in memory, and the I/O costs for those levels is \( O((L - k)nN/B) \), which is significantly smaller than those for the previous levels.

So the I/O cost of SPRINT becomes

\[
O \left( \frac{nN}{B} \log_{\frac{M}{2^M}} \frac{N}{M} + \sum_{0 \leq i \leq k-1} \frac{nNh_i}{B} + \frac{(L-k)nN}{B} \right) \tag{12}
\]

Note that we have

\[
h_i = \frac{r_hN}{2^i M} = \frac{N}{2^i M} \left( \frac{1 + \log N}{8} \right)
\]

So

\[
\frac{N}{M} \left( \frac{1 + \log N}{8} \right) \leq \sum_{0 \leq i \leq k-1} h_i \leq \frac{2N}{M} \left( \frac{1 + \log N}{8} \right) \tag{13}
\]

Applying (13) to (12), we get the I/O complexity of SPRINT in (11).

Examination of (8) and (11) 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.

6 Algorithm Revisited Using SchemaSQL

In Section 3.1, we described the MIND algorithm using SQL-like statements. Due to the limitation of current SQL standards, most of those SQL-like statements are not supported directly in today’s DBMS products. Therefore, we need to convert them to currently supported SQL statements, augmented with new facilities like user defined functions. Putting logic within a user-defined function hides the operator from query optimization. If classification was a subquery or part of a large query, it would not be possible to obtain all join reorderings, thereby risking suboptimal execution.

Current SQL standards are mainly designed for efficient OLTP (On-Line Transactional Processing) queries. For non-OLTP applications, it is true that we can usually reformulate the problem and express the solution using standard SQL. However, this approach often results in inefficiency. Extending current SQL with ad-hoc constructs and new optimization considerations might solve this problem in some particular domain, but it is not a satisfactory solution. Since supporting OLAP (On-Line Analytical Processing) applications efficiently is such an important goal for today’s RDBMSs, the problem deserves a more general solution.

In [LSS96] an extension of SQL, called SchemaSQL, is proposed. SchemaSQL offers the capability of uniform manipulation of data and meta-data in relational multi-database systems. By examining the SQL-like queries in Section 3.1, we can see that this capability is what we need in the MIND algorithm. To show the power of extended SQL and the flexibility and general flavor of MIND, in this section, we rewrite all the queries in Section 3.1 using SchemaSQL.

First we give an overview of the syntax of SchemaSQL. For more details see [LSS96].

In a standard SQL query, the \( \text{tuple} \) variables are declared in the \texttt{FROM} clause. A variable declaration has the form \( \texttt{range} \langle \text{var} \rangle \). For example, in the query below, the expression \( \texttt{student} \ T \) declares \( T \) as a variable that ranges over the (tuples of the) relation \( \texttt{student} \langle \texttt{sid, department, GPA} \rangle \):
SELECT student_id  
FROM student T  
WHERE T.department = CS AND T.GPA = A  

The SchemaSQL syntax extends SQL syntax in several directions:

1. The federation consists of databases, with each database consisting of relations.
2. To permit meta-data queries and reconstruction views, SchemaSQL permits the declaration of other types of variables in addition to the tuple variables permitted in SQL.
3. Aggregate operations are generalized in SchemaSQL to make horizontal and block aggregations possible, in addition to the usual vertical aggregation in SQL.

SchemaSQL permits the declaration of variables that can range over any of the following five sets:

1. names of databases in a federation,
2. names of the relations in a database,
3. names of the columns in the scheme of a relation,
4. tuples in a given relation in database, and
5. values appearing in a column corresponding to a given column in a relation.

Variable declarations follow the same syntax as \( (range)(var) \) as in SQL, where \( var \) is any identifier. However, there are two major differences:

1. The only kind of range permitted in SQL is a set of tuples in some relation in the database, where in SchemaSQL any of the five kinds of range can be used to declare variables.
2. The range specification in SQL is made using constant, i.e., an identifier referring to a specific relation in a database. By contrast, the diversity of ranges possible in SchemaSQL permits range specifications to be nested, in the sense that it is possible to say, for example, that \( R \) is a variable ranging over the relation names in database \( D \), and that \( T \) is a tuple in the relation denoted by \( R \).

Range specifications are one of the following five types of expressions, where \( db, rel, col \) are any constant or variable identifiers.

1. The expression \( \rightarrow \) denotes a range corresponding to the set of database names in the federation.
2. The expression \( db \rightarrow \) denotes the set of relation names in the database \( db \).
3. The expression \( db :: rel \rightarrow \) denotes the set of names of column in the schema of the relation \( rel \) in the database \( db \).
4. \( db :: rel \) denotes the set of tuples in the relation \( rel \) in the database \( db \).
5. \( db :: rel.col \) denotes the set of values appearing in the column named \( col \) in the relation \( rel \) in the database \( db \).
For example, consider the clause $\text{FROM } db1 \rightarrow R, db1 \rightarrow R T$. It declares $R$ as a variable ranging over the set of relation names in the database $db1$ and $T$ as a variable ranging over the tuples in each relation $R$ in the database $db1$.

Now we are ready to rewrite all the SQL-like queries in Section 3.1 using SchemaSQL. Assume that our training set is stored in relation $\text{DETAIL}$ in a database named $\text{FACT}$. We first generate all the dimension tables with the schema $(\text{leaf\_num, class, attr\_val, count})$ in a database named $\text{DIMENSION}$, using a simple SchemaSQL statement:

\[
\text{CREATE VIEW } \text{DIMENSION} :: R(\text{leaf\_num, class, attr\_val, count}) \text{ AS}
\]

\[
\text{SELECT } T.\text{leaf\_num}, T.\text{class}, T.R, \text{COUNT}(*)
\]

\[
\text{FROM } \text{FACT} :: \text{DETAIL} \rightarrow R,
\]

\[
\text{FACT} :: \text{DETAIL} T
\]

\[
\text{WHERE } R <='\text{class}' \text{ AND}
\]

\[
R <='\text{leaf\_num}' \text{ AND}
\]

\[
T.\text{leaf\_num} <= \text{STOP}
\]

\[
\text{GROUP BY } T.\text{leaf\_num}, T.\text{class}, T.R
\]

The variable $R$ is declared as a column name variable ranging over the column names of relation $\text{DETAIL}$ in the database $\text{FACT}$, and the variable $T$ is declared as a tuple variable on the same relation. The conditions on $R$ in the WHERE clause make the variable $R$ range over all columns except the columns named $\text{class}$ and $\text{leaf\_num}$. If there are $n$ columns in $\text{DETAIL}$ (excluding columns $\text{class}$ and $\text{leaf\_num}$), this query generates $n$ views in database $\text{DIMENSION}$, and the name of each view is the same as the corresponding column name in $\text{DETAIL}$. Note that the attribute name to relation name transformation is done in a very natural way, and the formation of multiple GROUP BYs is done by involving $\text{DETAIL}$ only once.

Those views will be materialized, so that in the later operations we do not need to access $\text{DETAIL}$ any more.

Relations corresponding to $\text{UP}$ with the schema $(\text{leaf\_num, attr\_val, class, count})$ can be generated in a database named $\text{UP}$ by performing a self-outer-join on dimension tables in database $\text{DIMENSION}$:

\[
\text{CREATE VIEW } \text{UP} :: R(\text{leaf\_num, attr\_val, class, count}) \text{ AS}
\]

\[
\text{SELECT } d1.\text{leaf\_num}, d1.\text{attr\_val}, d1.\text{class}, \text{SUM}(d2.\text{count})
\]

\[
\text{FROM (FULL OUTER JOIN } \text{DIMENSION} :: R d1,
\]

\[
\text{DIMENSION} \rightarrow R
\]

\[
\text{ON } d1.\text{leaf\_num} = d2.\text{leaf\_num} \text{ AND}
\]

\[
d1.\text{attr\_val} <= d2.\text{attr\_val} \text{ AND}
\]

\[
d1.\text{class} = d2.\text{class}
\]

\[
\text{GROUP BY } d1.\text{leaf\_num}, d1.\text{attr\_val}, d1.\text{class}
\]

The variable $R$ is declared as a relation name variable ranging over all the relations in database $\text{DIMENSION}$. Variables $d1$ and $d2$ are both tuple variables over the tuples in each relation $R$ in database $\text{DIMENSION}$. For each relation in database $\text{DIMENSION}$, a self-outer-join is performed according to the conditions specified in the query, and the result is put into a VIEW with the same name in database $\text{UP}$.

Similarly, relations corresponding to $\text{DOWN}$ can be generated in a database named $\text{DOWN}$ by just changing the $\leq$ to $>$ in the ON clause.
Database \textit{DOWN} and database \textit{UP} contain all the information we need to compute all the \textit{gini} index values. Since standard SQL only allows vertical aggregations, we need to rearrange them before the \textit{gini} index is actually calculated as in Section 3.1. In SchemaSQL, aggregation operations are generalized to make horizontal and block aggregations possible. Thus, we can generate views that contain all \textit{gini} index values at each possible split point for each attribute in a database named \textit{GINI\_VALUE} directly from relations in \textit{UP} and \textit{DOWN}:

\begin{verbatim}
CREATE VIEW GINI\_VALUE :: R(leaf\_num, attr\_val, gini) AS
SE\textit{L}\textit{..CT} u.leaf\_num, u.attr\_val, f_{gini}
FROM \textit{UP} :: R u,
    \textit{DOWN} :: R d,
    \textit{UP} \rightarrow R
WHERE u.leaf\_num = d.leaf\_num AND
    u.attr\_val = d.attr\_val
GROUP BY u.leaf\_num, u.attr\_val
\end{verbatim}

where \textit{f_{gini}} is a function of \textit{u.class}, \textit{d.class}, \textit{u.count}, \textit{d.count} according to (1) and (2).

\textit{R} is declared as a variable ranging over the set of relation names in database \textit{UP}, \textit{u} is a variable ranging over the tuples in each relation in database \textit{UP}, and \textit{d} is a variable ranging over the tuples in the relation with the same name as \textit{R} in database \textit{DOWN}. Note that the set of relation names in databases \textit{UP} and \textit{DOWN} are the same. For each of the relation pairs with the same name in \textit{UP} and \textit{DOWN}, this statement will create a view with the same name in database \textit{GINI\_VALUE} according to the conditions specified. It is interesting to note that \textit{f_{gini}} is a block aggregation function instead of the usual vertical aggregation function in SQL. Each view named \textit{R} in database \textit{GINI\_VALUE} contains the \textit{gini} index value at each possible split point with respect to attribute named \textit{R}.

Next, we create a single view \textit{MIN\_GINI} with the schema \textit{MIN\_GINI}(leaf\_num, attr\_name, attr\_val, gini) in a database named \textit{SPLIT} form the multiple views in database \textit{GINI\_VALUE}:

\begin{verbatim}
CREATE VIEW SPLIT :: MIN\_GINI(leaf\_num, attr\_name, attr\_val, gini) AS
SE\textit{L}\textit{..CT} T_1.leaf\_num, R_1, T_1.attr\_val, gini
FROM GINI\_VALUE \rightarrow R_1,
    GINI\_VALUE :: R_1 T_1
WHERE T_1.gini = (SELECT MIN(T_2.gini)
    FROM GINI\_VALUE \rightarrow R_2,
    GINI\_VALUE :: D_2 T_2
    WHERE R_1 = R_2 AND
    T_1.leaf\_num = T_2.leaf\_num)
\end{verbatim}

\textit{R_1} and \textit{R_2} are variables ranging over the set of relation names in database \textit{GINI\_VALUE}. \textit{T_1} and \textit{T_2} are tuple variables ranging over the tuples in relations specified by \textit{R_1} and \textit{R_2}, respectively. The clause \textit{R_1 = R_2} enforces \textit{R_1} and \textit{R_2} to be the same relation. Note that relation name \textit{R_1} in database \textit{GINI\_VALUE} becomes the column value for the column named \textit{attr\_name} in relation \textit{MIN\_GINI} in database \textit{SPLIT}. Relation \textit{MIN\_GINI} now contains the best split value and the corresponding \textit{gini} index value for each leaf node of the tree with respect to all attributes.

The overall best split for each leaf is obtained from executing the following:

\begin{verbatim}

\end{verbatim}
CREATE VIEW SPLIT :: BEST_SPLIT(leaf_num, attr_name, attr_val) AS
SELECT T1.leaf_num, T1.attr_name, T1.attr_val
FROM SPLIT :: MIN_GINI T1
WHERE T1.gini = (SELECT MIN(gini)
                 FROM SPLIT :: MIN_GINI T2
                 WHERE T1.leaf_num = T2.leaf_num)

This statement is similar to the statement generating relation BEST_SPLIT in Section 3.1. T1 is declared as a tuple variable ranging over the tuples of relation MIN_GINI in database SPLIT. For each leaf_num, (attr_name, attr_val) pair that achieving the minimum gini index value is inserted into relation BEST_SPLIT.

We have shown how to rewrite all the SQL-like queries in MIND algorithm using SchemaSQL. In our current prototype of MIND, the first step, generating all the dimension tables from DETAIL, is most costly and all the later steps only need to access small dimension tables. We use udf to reduce the cost of the first step. All the SQL-like queries in Section 3.1 in the later steps are translated into equivalent SQL queries. Those translations usually lead to poor performance. But since those queries only access small relations in MIND, the performance loss is negligible. While udf provides a solution to our classification algorithm, we believe general extension of SQL is needed for efficient support of OLAP applications.

An alternative way to generate all the dimension tables from DETAIL would be using the newly proposed data cube operator [GBLP96] since dimension tables are different subcubes. But it usually takes a long time to generate the data cube without precomputation and the fact that the leaf_num column in DETAIL keeps changing from level to level when we grow the tree makes precomputation infeasible.

7 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 [MAR96] and SPRINT [SAM96]. (For brevity, we include only SPRINT in this paper; comparisons showing the improvement of SPRINT over SLIQ are given in [SAM96].) 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 [MAR96, SAM96] for the same training set, which facilitates meaningful comparisons of run time. The accuracy of SPRINT and SLIQ is discussed in [MAR96, SAM96], 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 [AGI+92]. In this synthetic database, each record consists of nine attributes as shown in Table 17. Ten classifier functions are proposed in [AGI+92] to produce databases with different
<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>salary</td>
<td>uniformly distributed from 20K to 150K</td>
</tr>
<tr>
<td>commission</td>
<td>( salary \geq 75K \Rightarrow commission = 0 ) else</td>
</tr>
<tr>
<td>age</td>
<td>uniformly distributed from 20 to 80</td>
</tr>
<tr>
<td>loan</td>
<td>uniformly distributed from 0 to 500K</td>
</tr>
<tr>
<td>elevel</td>
<td>uniformly chosen from 0 to 4</td>
</tr>
<tr>
<td>car</td>
<td>uniformly chosen from 1 to 20</td>
</tr>
<tr>
<td>zipcode</td>
<td>uniformly chosen from 10 available zipcodes</td>
</tr>
<tr>
<td>hvalue</td>
<td>uniformly distributed from 0.5k(100000 ) to 1.5k(100000, )</td>
</tr>
<tr>
<td>hyear</td>
<td>where ( k \in {0, \ldots, 9} ) is zipcode</td>
</tr>
</tbody>
</table>

Table 17: Description of the synthetic data

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
\[
((\text{age} < 40) \land (50K \leq \text{salary} \leq 100K)) \lor \\
((40 \leq \text{age} < 60) \land (75K \leq \text{salary} \leq 125K)) \lor \\
((\text{age} \geq 60) \land (25K \leq \text{salary} \leq 75K))
\]

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 5 and Figure 6 respectively. Figure 5 hints that our algorithm achieves linear scalability with respect to the training set size. Figure 6 shows that the time per example curve stays flat when the training set size increases. The corresponding curve for [SAM96] appears to be growing slightly on the largest cases. Figure 7 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 8 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 speedup on multiprocessors, it is critical that this step is parallelized. In the current working prototype of MIND, it is done by user-defined function with a scratch-pad accessible from multiple processors.

8 Conclusions

The MIND algorithm solves the problem of classification within the relational database management systems. 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
1. MIND represents the data mining function classification as a classic DBMS problem of sum-

data sets larger than those tested before.

There are four reasons why MIND is best, exhibits excellent scalability, and is able to handle
functions of $N$. If also runs faster than previous algorithms on the systems,

on a multiprocessor and yet demonstrates excellent non-increasing response time per example as a

believe MIND is the first classifier to successfully run on datasets of $N = 5$ million examples

for the indicated training set size, divided by response time per example when processing 5 million

Figure 6: Relative response time per example. The y-axis denotes the response time per example

Figure 5: Relative total response time for the

indicated training set size, divided by the total response time for 5 million examples.
Figure 7: Performance comparison of MIND and SPRINT

Figure 8: 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.

2. MIND avoids any update to the DETAIL table of examples. 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.

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


