**A Festschrift for Herman Rubin**
**Institute of Mathematical Statistics**
**Lecture Notes – Monograph Series**
Vol. 45 (2004) 348–377
© Institute of Mathematical Statistics, 2004

# Scalable mining for classification rules in relational databases

**MinWang[1,*], Bala Iyer [2], and Jeffrey Scott Vitter[3,†]**

*IBM T. J. Watson Research Center, IBM Silicon Valley Lab and Purdue University*

**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 [16]. 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 [20, 19] 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:

[1]Data Management Department, IBM T. J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA. e-mail: `min@us.ibm.com`

[2]IBM Silicon Valley Lab, 555 Bailey Avenue, San Jose, CA 95141, USA. e-mail: `balaiyer@us.ibm.com`

[3]Purdue University, 150 North University Street, West Lafayette, IN 47907, USA. e-mail: `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.

*Keywords and phrases:* data mining, classification, classifier, relation database, SQL.

*AMS 2000 subject classifications:* 68P20.

| *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

- scalability,
- memory hierarchy management [30, 33],
- parallelism [5],
- optimization of the executions [6],
- platform independence, and
- client server API [27].

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 [4, 28, 26, 9]).

Several classification models have been proposed in the literature, including neutral 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 [29]. Therefore, we focus on decision tree model in this paper. A simple illustration of 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 internal memory. In data mining applications, it is common to have training sets with several million examples. It is observed in [24] 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

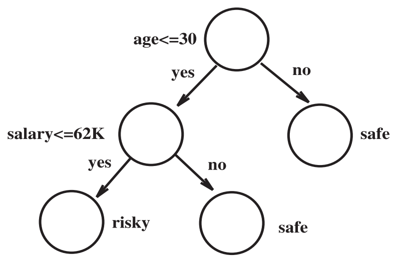

Figure 1: Decision tree for the data in Table 1

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 [13, 21].

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 [24, 29, 18], 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 [3, 22] 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 [36]. 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 [29]. 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.

## 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 $\mathit{value}(A) \leq x$, where $x$ is a value in the domain of $A$. Splits for a categorical attribute $A$ are of the form $\mathit{value}(A) \in S$, where $S$ is a subset of $\mathit{domain}(A)$. We consider only binary splits as in [24, 29] 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 [24] 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 [29], 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 [29], 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 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 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. *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 *leaf_num* value and *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, *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 [4] and used in [24, 29], because it gives acceptable accuracy. The accuracy of our classifier is therefore the same as those in [24, 29].

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 $gini_{\mathrm{split}}(S)$ is given by

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

The attribute containing the split point achieving the smallest *gini* index value is then chosen to split the node [4]. 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 $i$th 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. 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 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 $n$.

## 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
SELECT leaf_num, class, attr_i, COUNT(*)
FROM DETAIL
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 tree) × (#distinct values on $attr_i$) × (#distinct classes), which is very likely to be of the order of several hundreds [25]. 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 [3, 22]. Discretization of variable values into a smaller number of classes is sometimes referred to as "encoding" in data mining practice [3]. Roughly speaking, this is done to obtain a measure of aggregate behavior that may be detectable [25]. 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 [31].

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 function* (udf) [32, 10, 17], which is another reason why MIND is efficient. User-defined functions are used for association in [2]. User-defined function is a new feature provided by DB2 version 2 [10, 17]. In DB2 version 2, the functions available for use in SQL statements extend from the system built-in functions, such as `avg, min, max, sum`, to more general categories, such as user-defined functions (udf). 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*.

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 $d_1.leaf\_num, d_1.attr_i, d_1.class,$ SUM$(d_2.count)$
FROM (FULL OUTER JOIN $DIM_i\ d_1,\ DIM_i\ d_2$
        ON $d_1.leaf\_num = d_2.leaf\_num$ AND
            $d_2.attr_i \leq d_1.attr_i$ AND
            $d_1.class = d_2.class$
        GROUP BY $d_1.leaf\_num, d_1.attr_i, d_1.class)$
```

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 $u_1.leaf\_num, u_1.attr_i,\ f_{gini}$
FROM $C_1\_UP\ u_1, \ldots, C_C\_UP\ u_C,\ C_1\_DOWN\ d_1, \ldots, C_C\_DOWN\ d_C$
WHERE $u_1.attr_i = \cdots = u_C.attr_i = d_1.attr_i = \cdots = d_C.attr_i$ AND
        $u_1.leaf\_num = \cdots = u_C.leaf\_num = d_1.leaf\_num = \cdots = d_C.leaf\_num$
```

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 $leaf\_num, : i,\ attr_i,\ gini$
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 [6, 26, 15].

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 [29], 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:

```
INSERT INTO $S_k\_IN$
SELECT  $p.leaf\_num, p.class, p.v_1, \ldots, p.v_{k-1}, q.v_1, p.count + q.count$
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 attributes 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 $leaf\_num = Partition(attr_1, \ldots, attr_n, class, leaf\_num)$
```

The user-defined function *Partition* defined on a record $r$ of *DETAIL* 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 to the result of the split;
```

| *attr*$_1$ | *attr*$_2$ | *class* | *leaf_num* |
|---|---|---|---|
| 65K | 30 | Safe | 0 |
| 15K | 23 | Risky | 0 |
| 75K | 40 | Safe | 0 |
| 15K | 28 | Risky | 0 |
| 100K | 55 | Safe | 0 |
| 60K | 45 | Safe | 0 |
| 62K | 30 | Risky | 0 |

Table 2: Initial relation *DETAIL* with implicit *leaf_num*

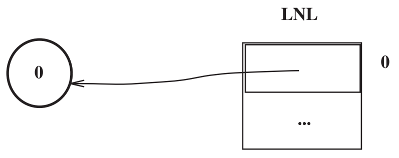

Figure 2: Initial tree

| *leaf_num* | *attr*$_1$ | *class* | *count* |
|---|---|---|---|
| 0 | 15 | 2 | 2 |
| 0 | 60 | 1 | 1 |
| 0 | 62 | 2 | 1 |
| 0 | 65 | 1 | 1 |
| 0 | 75 | 1 | 1 |
| 0 | 100 | 1 | 1 |

Table 3: Relation *DIM*$_1$

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. 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.

**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 *gini* index value for *attr*$_1$. The procedure is depicted in Table 5–13.

| *leaf_num* | $attr_2$ | *class* | *count* |
|---|---|---|---|
| 0 | 23 | 2 | 1 |
| 0 | 28 | 2 | 1 |
| 0 | 30 | 1 | 1 |
| 0 | 30 | 2 | 1 |
| 0 | 40 | 1 | 1 |
| 0 | 45 | 1 | 1 |
| 0 | 55 | 1 | 1 |

Table 4: Relation $DIM_2$

| *leaf_num* | $attr_1$ | *class* | *count* |
|---|---|---|---|
| 0 | 15 | 1 | 0 |
| 0 | 15 | 2 | 2 |
| 0 | 60 | 1 | 1 |
| 0 | 60 | 2 | 2 |
| 0 | 62 | 1 | 1 |
| 0 | 62 | 2 | 3 |
| 0 | 65 | 1 | 2 |
| 0 | 65 | 2 | 3 |
| 0 | 75 | 1 | 3 |
| 0 | 75 | 2 | 3 |
| 0 | 100 | 1 | 4 |
| 0 | 100 | 2 | 3 |

Table 5: Relation *UP*

We can see that the best splits on the two attributes achieve the same *gini* index value, so relation *BEST_SPLIT* is the same as *MIN_GINI* except that it does not contain the column *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 ($attr_2$ in our example) can be chosen.

**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.

**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 [29] forms an *attribute list* (projection of the *DETAIL* table) for each attribute. In order to reduce the cost of computing the

| *leaf_num* | $attr_1$ | *class* | *count* |
|---|---|---|---|
| 0 | 15 | 1 | 4 |
| 0 | 15 | 2 | 1 |
| 0 | 60 | 1 | 3 |
| 0 | 60 | 2 | 1 |
| 0 | 62 | 1 | 3 |
| 0 | 62 | 2 | 0 |
| 0 | 65 | 1 | 2 |
| 0 | 65 | 2 | 0 |
| 0 | 75 | 1 | 1 |
| 0 | 75 | 2 | 0 |

Table 6: Relation *DOWN*

| *leaf_num* | $attr_1$ | *count* |
|---|---|---|
| 0 | 15 | 0.0 |
| 0 | 60 | 1.0 |
| 0 | 62 | 1.0 |
| 0 | 65 | 2.0 |
| 0 | 75 | 3.0 |
| 0 | 100 | 4.0 |

Table 7: Relation $C_1\_UP$

| *leaf_num* | $attr_1$ | *count* |
|---|---|---|
| 0 | 15 | 2.0 |
| 0 | 60 | 2.0 |
| 0 | 62 | 3.0 |
| 0 | 65 | 3.0 |
| 0 | 75 | 3.0 |
| 0 | 100 | 3.0 |

Table 8: Relation $C_2\_UP$

*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

| *leaf_num* | *attr*$_1$ | *count* |
|---|---|---|
| 0 | 15 | 4.0 |
| 0 | 60 | 3.0 |
| 0 | 62 | 3.0 |
| 0 | 65 | 2.0 |
| 0 | 75 | 1.0 |

Table 9: Relation $C_1\_DOWN$

| *leaf_num* | *attr*$_1$ | *count* |
|---|---|---|
| 0 | 15 | 1.0 |
| 0 | 60 | 1.0 |
| 0 | 62 | 0.0 |
| 0 | 65 | 0.0 |
| 0 | 75 | 0.0 |

Table 10: Relation $C_2\_DOWN$

| *leaf_num* | *attr*$_1$ | *gini* |
|---|---|---|
| 0 | 15 | 0.22856 |
| 0 | 60 | 0.40474 |
| 0 | 62 | 0.21428 |
| 0 | 65 | 0.34284 |
| 0 | 75 | 0.42856 |

Table 11: Relation *GINI_VALUE*

*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 [7, 36]. 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 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 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

| *leaf_num* | *attr_name* | *attr_value* | *gini* |
|---|---|---|---|
| 0 | 1 | 62 | 0.21428 |

Table 12: Relation *MIN_GINI* after $attr_1$ is evaluated

| *leaf_num* | *attr_name* | *attr_value* | *gini* |
|---|---|---|---|
| 0 | 1 | 62 | 0.21428 |
| 0 | 2 | 30 | 0.21428 |

Table 13: Relation *MIN_GINI* after $attr_1$ and $attr_2$ are evaluated

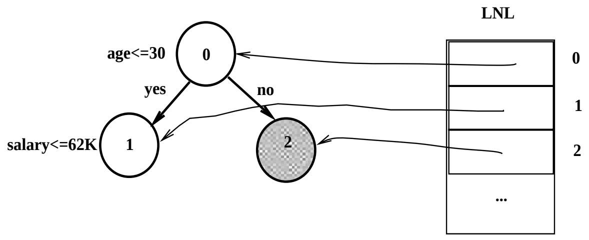

Figure 3: Decision tree at Phase 3

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 $nN$.

**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*

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

*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 $O(LnN/B)$. □

**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*

$$O\left(\frac{LnN}{B}\log_{M/B} n\right). \tag{5}$$

| $attr_1$ | $attr_2$ | *class* | *leaf_num* |
|---|---|---|---|
| 65K | 30 | Safe | 1 |
| 15K | 23 | Risky | 1 |
| 75K | 40 | Safe | 2⇒*STOP* |
| 15K | 28 | Risky | 1 |
| 100K | 55 | Safe | 2⇒*STOP* |
| 60K | 45 | Safe | 2⇒*STOP* |
| 62K | 30 | Risky | 1 |

Table 14: Relation *DETAIL* with implicit *leaf_num* after Phase 3

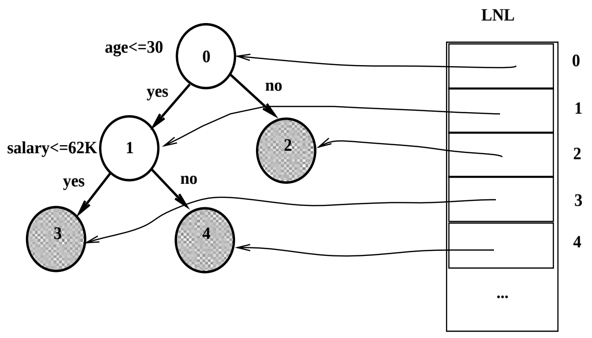

Figure 4: Final decision tree

| $attr_1$ | $attr_2$ | *class* | *leaf_num* |
|---|---|---|---|
| 65K | 30 | Safe | 4⇒*STOP* |
| 15K | 23 | Risky | 3⇒*STOP* |
| 75K | 40 | Safe | *STOP* |
| 15K | 28 | Risky | 3⇒*STOP* |
| 100K | 55 | Safe | *STOP* |
| 60K | 45 | Safe | *STOP* |
| 62K | 30 | Risky | 3⇒*STOP* |

Table 15: Final relation *DETAIL* with implicit *leaf_num*

*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.

| | |
|---|---|
| $M$ | 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 16: Parameters used in analysis

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). $\square$

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 [35]:

$$N = \text{problem size (in units of data items)},$$
$$M = \text{internal memory size (in units of data items)},$$

$$\begin{aligned} B &= \text{block size (in units of data items),} \\ m &= \frac{M}{B}, \text{ number of blocks that fits into internal memory,} \end{aligned}$$

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 $N$ date 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*

$$O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N'}{B}\right) \tag{6}$$

*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 $O(\frac{N}{B})$ 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 $O(\log_m \frac{N/B}{m})$ levels in the merge process, and each level requires $O(\frac{N}{B})$ I/O operations, we obtain the $O(\frac{N}{B} \log_m \frac{N}{B})$ 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 $k = \lceil \log_{M/B} N'/B \rceil$, the length of any run will be bounded by $N'$ and the number of runs is bounded by $\lceil \frac{N/B}{m^k} \rceil$. (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 $\frac{N/B}{m^{k+i-1}}$ runs into $\frac{N/B}{m^{k+i}}$ 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 $p = \log_m \frac{N/B}{n'}$, $n' = N'/B$. So the I/O for the whole special sort procedure is:

$$\begin{aligned}
&2\frac{N}{B}k + \frac{N}{B}(1 + \frac{1}{m}) + \frac{N/B}{m}\left(1 + \frac{1}{m}\right) + \cdots + \frac{N/B}{m^{p-1}}\left(1 + \frac{1}{m}\right) \\
&\leq 2\frac{N}{B}\log_m n' + \frac{N}{B}\left(1 + \frac{1}{m}\right)\frac{1}{1 - 1/m} \\
&\approx 2\frac{N}{B}\log_m n' + \frac{N}{B} \\
&= O\left(\frac{N}{B}\log_m n' + \frac{N}{B}\right) \\
&= O\left(\frac{N}{B}\log_{\frac{M}{B}} \frac{N'}{B}\right).
\end{aligned}$$

□

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*

$$O\left(\frac{nNL}{B} + \frac{nN}{B}\sum_{0 \leq k < L} \log_{M/B} \frac{D_k}{B}\right), \tag{7}$$

*which is*

$$O\left(\frac{LnN}{B}\frac{\log \frac{nN}{B}}{\log \frac{M}{B}}\right). \tag{8}$$

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

$$O\left(\frac{LnN}{B}\right), \tag{9}$$

*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

$$\frac{LnN}{B} + \frac{1}{B}\sum_{0\le k<L} D_k^{'} + \frac{nN}{B}\sum_{0\le k<L} \log_{M/B}\frac{D_k}{B}$$

which is

$$O\left(\frac{LnN}{B} + \frac{nN}{B}\sum_{0\le k<L} \log_{M/B}\frac{D_k}{B}\right).$$

□

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 $r_h N/2^k \le M$, or

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

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

$$IO_{presort} = O\left(\frac{nN}{B}\log_{\frac{M}{B}}\frac{N}{B}\right).$$

From level 0 through level $k-1$, hash tables will not fit in memory. At level $i$ $(0 \le i \le k-1)$, 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

$$O\left(\frac{nNh_i}{B}\right)$$

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

$$O\left(\frac{nN}{B}\log_{\frac{M}{B}}\frac{N}{B} + \sum_{0\le i\le k-1}\frac{nNh_i}{B} + \frac{(L-k)nN}{B}\right) \tag{12}$$

Note that we have

$$h_i = \frac{r_h N}{2^i M} = \frac{N}{2^i M}\left(\frac{1+\log N}{8}\right)$$

So

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

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 schema SQL

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 [23] 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 [23].

In a standard SQL query, the *tuple* variables are declared in the `FROM` clause. A variable declaration has the form $\langle range\rangle\langle var\rangle$. For example, in the query below, the expression *student T* declares $T$ as a variable that ranges over the (tuples of the) relation $student(student\_id, department, GPA)$:

`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 $\langle range\rangle\langle var\rangle$ 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 *range*s 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 FROM $db1 \rightarrow R$, $db1 :: R\ T$. 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 *DETAIL* in a database named *FACT*. We first generate all the dimension tables with the schema (*leaf_num*, *class*, *attr_val*, *count*) in a database named *DIMENSION*, using a simple SchemaSQL statement:

```
CREATE VIEW DIMENSION :: R(leaf_num, class, attr_val, count) AS
SELECT T.leaf_num, T.class, T.R, COUNT(*)
FROM FACT :: DETAIL → R,
     FACT :: DETAIL T
WHERE R <>' class' AND
      R <>' leaf_num' AND
      T.leaf_num <> STOP
GROUP BY T.leaf_num, T.class, T.R
```

The variable $R$ is declared as a column name variable ranging over the column names of relation *DETAIL* in the database *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 *class* and *leaf_num*. If there are $n$ columns in *DETAIL* (excluding columns *class* and *leaf_num*), this query generates $n$ `VIEW`s in database *DIMENSION*, and the name of each `VIEW` is the same as the corresponding column name in *DETAIL*. Note that the attribute name to relation name transformation is done in a very natural way, and the formation of multiple `GROUP BY`s is done by involving *DETAIL* only once.

Those views will be materialized, so that in the later operations we do not need to access *DETAIL* any more.

Relations corresponding to *UP* with the schema (*leaf_num*, *attr_val*, *class*, *count*) can be generated in a database named *UP* by performing a self-outer-join on dimension tables in database *DIMENSION*:

```
CREATE VIEW $UP :: R(leaf\_num, attr\_val, class, count)$ AS
SELECT $d_1.leaf\_num, d_1.attr\_val, d_1.class$, SUM($d_2.count$)
FROM (FULL OUTER JOIN $DIMENSION :: R\ d_1$,
                      $DIMENSION :: R\ d_2$,
                      $DIMENSION \rightarrow R$
      ON $d_1.leaf\_num = d_2.leaf\_num$ AND
         $d_1.attr\_val \leq d_2.attr\_val$ AND
         $d_1.class = d_2.class$
      GROUP BY $d_1.leaf\_num, d_1.attr\_val, d_1.class$)
```

The variable $R$ is declared as a relation name variable ranging over all the relations in database *DIMENSION*. Variables $d_1$ and $d_2$ are both tuple variables over the tuples in each relation $R$ in database *DIMENSION*. For each relation in database *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 *UP*.

Similarly, relations corresponding to *DOWN* can be generated in a database named *DOWN* by just changing the $\leq$ to $>$ in the `ON` clause.

Database *DOWN* and database *UP* contain all the information we need to compute all the *gini* index values. Since standard SQL only allows vertical aggregations, we need to rearrange them before the *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 *gini* index values at each possible split point for each attribute in a database named *GINI_VALUE* directly from relations in *UP* and *DOWN*:

```
CREATE VIEW $GINI\_VALUE :: R(leaf\_num, attr\_val, gini)$ AS
SELECT $u.leaf\_num, u.attr\_val, f_{gini}$
FROM $UP :: R\ u$,
     $DOWN :: R\ d$,
     $UP \rightarrow R$
WHERE $u.leaf\_num = d.leaf\_num$ AND
      $u.attr\_val = d.attr\_val$
GROUP BY $u.leaf\_num, u.attr\_val$
```

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

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

Next, we create a *single* view $MIN\_GINI$ with the schema $MIN\_GINI$(*leaf_num*, *attr_name*, *attr_val*, *gini*) in a database named $SPLIT$ form the *multiple* views in database $GINI\_VALUE$:

```
CREATE VIEW SPLIT :: MIN_GINI(leaf_num, attr_name, attr_val, gini) AS
SELECT T1.leaf_num, R1, T1.attr_val, gini
FROM GINI_VALUE → R1,
     GINI_VALUE :: R1 T1
WHERE T1.gini =(SELECT MIN(T2.gini)
                FROM GINI_VALUE → R2,
                     GINI_VALUE :: D2 T2
                WHERE R1 = R2 AND
                      T1.leaf_num = T2.leaf_num)
```

$R_1$ and $R_2$ are variables ranging over the set of relation names in database $GINI\_VALUE$. $T_1$ and $T_2$ are tuple variables ranging over the tuples in relations specified by $R_1$ and $R_2$, respectively. The clause $R_1 = R_2$ enforces $R_1$ and $R_2$ to be the same relation. Note that relation name $R_1$ in database $GINI\_VALUE$ becomes the column value for the column named *attr_name* in relation $MIN\_GINI$ in database $SPLIT$. Relation $MIN\_GINI$ now contains the best split value and the corresponding *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:

```
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. $T_1$ 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 a 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 [14] 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 [24] and SPRINT [29]. (For brevity, we include only SPRINT in this paper; comparisons showing the improvement of SPRINT over SLIQ are given in [29].) 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 [24, 29] for the same training set, which facilitates meaningful comparisons of run time. The accuracy of SPRINT and SLIQ is discussed in [24, 29], 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 [1]. In this synthetic database, each record consists of nine attributes as shown in Table 17. Ten classifier functions are proposed in [1] 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 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 [29] 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.

| *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 17: Description of the synthetic data

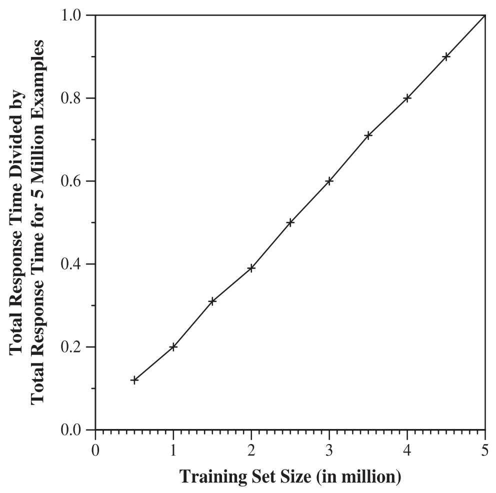

Figure 5: Relative total response time. The $y$-value denotes the total response time for the indicated training set size, divided by the total response time for 5 million examples.

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

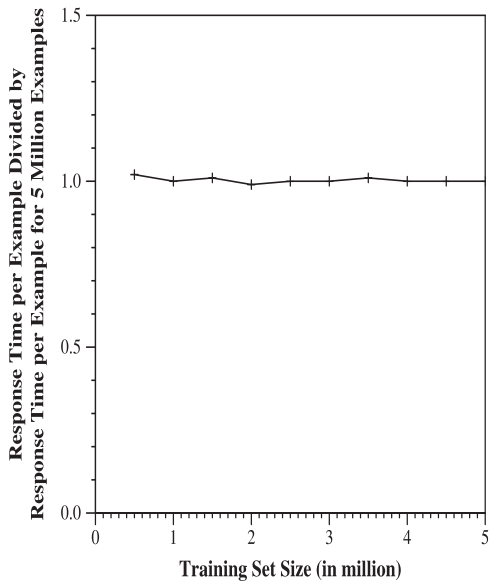

Figure 6: Relative response time per example. The $y$-value denotes the response time per example for the indicated training set size, divided by response time per example when processing 5 million examples.

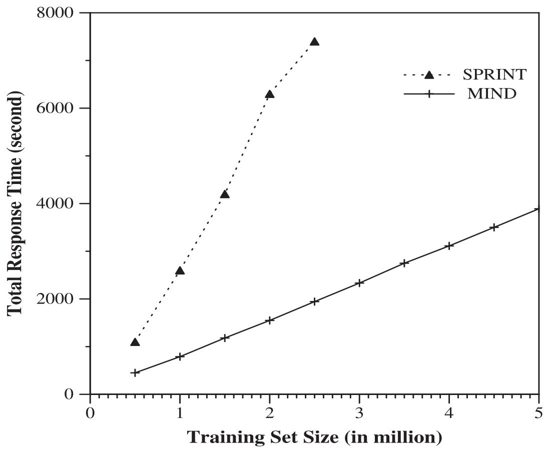

Figure 7: Performance comparison of MIND and SPRINT

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. This is of significant practical interest; for example, imagine *DETAIL* having billions of rows.

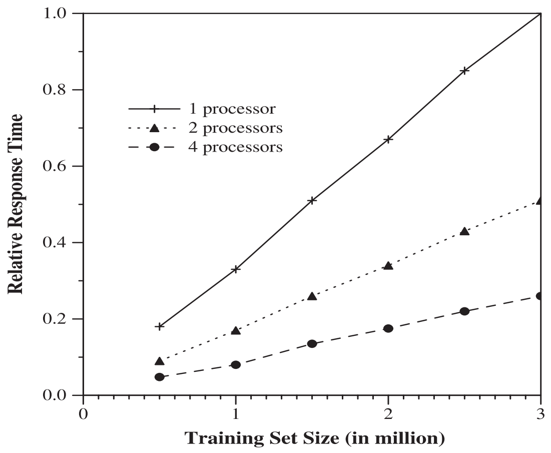

Figure 8: Speedup of MIND for multiprocessors. The $y$-value denotes the total response time for the indicated training set size, divided by the total response time for 3 million examples.

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

[1] 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.

[2] 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.

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

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

[5] C. K. Baru et al. DB2 parallel edition. *IBM Systems Journal*, 34(2), 1995.

[6] 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.

[7] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4), 1997. MR1453082

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

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

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

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

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

[13] 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.

[14] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tabs and subtotals. In *Proceedings of the 12th Annual IEEE Conference on Data Engineering (ICDE '96)*, pages 131–139, 1996.

[15] 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.

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

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

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

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

[20] 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.

[21] 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. MR1427544

[22] 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.

[23] L. Lakshmanan, F. Sadri, and I. Subramanian. SchemaSQL–a language for interoperability in relational multi-database systems. In *Proceedings of the 1996 International Conference on Very Large Databases*, 1996.

[24] 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.

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

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

[27] T. Nguyen and V. Srinivasan. Accessing relational databases from the world web web. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996.

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

[29] 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.

[30] S. Sarawagi. Query processing in tertiary memory databases. In *Proceedings of the 1995 International Conference on Very Large Databases*, 1995.

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

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

[33] S. Sarawagi and M. Stonebraker. Benefits of reordering execution in tertiary memory databases. In *Proceedings of the 1996 International Conference on Very Large Databases*, 1996.

[34] J. Ullman. *Principles of Database Systems*. Computer Science Press, second edition, 1982. MR669881

[35] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001. MR1965532

[36] 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.