Scalable Classification over SQL Databases

Surajit Chaudhuri Usama Fayyad Jeff Bernhardt
Microsoft Research
Redmond, WA 98052, USA
Email: {surajitc,fayyad, jeffbern}@microsoft.com

Abstract

We identify data-intensive operations that are common to classifiers and develop a middleware that decomposes and schedules these operations efficiently using a backend SQL database. Our approach has the added advantage of not requiring any specialized physical data organization. We demonstrate the scalability characteristics of our enhanced client with experiments on Microsoft SQL Server 7.0 by varying data size, number of attributes and characteristics of decision trees.

1 Introduction

We focus on the *classification problem***,** one of the most common operations in data mining [FG*96]. The problem can be simply stated as: given a data set with at least m+1 fields: A_1,..,A_m, and C, build a *model* that predicts the value of the "distinguished" field C given the values of fields A_1,..,A_m. Our goal is to demonstrate a scalable architecture for classification that takes advantage of the functionality of SQL backend. Our approach is based on providing a middleware layer that can interface to a large class of generic classification methods and help scale existing implementations of classification algorithms. The middleware exploits the observations that:

- A wide class of data mining algorithms do not require direct access to data but are driven purely by *sufficient statistics* of the data obtainable via batches group-by/count queries.
- The above queries are numerous, similar in form, but unique (no repetitions). The structure of these waves of query batches can be exploited to achieve significant improvement in performance .

We present our scheme in the context of classification with decision trees, but other classification algorithms such as Naïve Bayes can also "plug-in" to this architecture [CFB97]. We require no changes to the physical design of the SQL database. The key aspects of the proposed middleware *execution module* that contribute to its performance are:

- Batching execution of multiple queries (to compute sufficient statistics) for the classification client in a single scan of data.
- Appropriately staging data from server to client file system and to client main memory.

The middleware's execution module is complemented by a *scheduler* that determines the staging of data as well as batched query execution. The optimization decisions to tradeoff the cost of scanning data at the server versus use of in-memory operations in the middleware is transparent to the client. We report performance results on a variety of experiments run with an implementation of the middleware on Microsoft SQL Server 7.0 using a traditional in-memory classification client. We report experimental results that demonstrate the effect of data size, available memory, and properties of the decision tree being generated on performance of the middleware. For purposes of this discussion we assume all attributes are categorical or have been discretized (see [CFB97] for how numeric-valued attributes are treated).

2 Preliminaries

The classification problem is fundamentally one of statistical estimation: given attributes $A_i \in A$, i=1,..,m, the problem is to estimate the probability that the class variable C takes on some value c, i.e. for each possible value c_j of the class C, determine the probability: $\Pr(C = cj \mid A_1, ..., A_m)$. Not all attributes in A need be relevant to C; classification algorithms attempt to determine key attributes for the problem. In this section, we introduce *decision-tree classifiers* and identify the data-centric operations (gathering sufficient statistics) that critically impact performance of algorithms that construct them.

2.1 Decision Tree Classifiers

We focus on decision trees for various reasons. First, they are widely studied in statistics, pattern recognition and machine learning literature [B*84, Q93, F94]. Next, unlike other classifiers such as the nearest neighbor, neural networks, regression-based statistical techniques, decision trees deal effectively

with high-dimensional data. Finally, decision trees can be examined and understood. The leaves, represented as decision rules, are more easily understood by domain experts [Q93].

Algorithm Grow is an abstract description of a decision tree classifier that generates trees top-down in a greedy fashion. The algorithm starts with all the data set at the root node. A partition is selected, splitting the data into two or more subsets. The procedure is repeated recursively on each new subset until a termination criterion is met:

Algorithm Grow(TreeNode, S)

1. Partition S into subsets Split (S) = $\{S_1,\ldots,S_r\}$
2. For each subset $S_i \in$ Split(S)
3. Create i*th* child of TreeNode: $Child_i$
4. IF Termination Criteria for S_i satisfied THEN: $Child_i$ is a leaf; assign class; return ELSE: **Grow**($Child_i$, S_i)

Several measures have been proposed and used to determine the *partitioning* of data [B*84, Q93, FI92b]. The scheme proposed in this paper can support the above measures (see [CFB97] for examples). We describe the partitioning step in details in Section 2.2. A *class assignment* to a leaf is based on the proportion of counts of class values at the leaf node. Several *termination criteria* exist in the literature. The most common is to terminate if either all cases in S_i are in one class (the node is pure), or it is not possible to split the node further since all attributes have the same values in this subset.

A decision tree algorithm can be structured so that

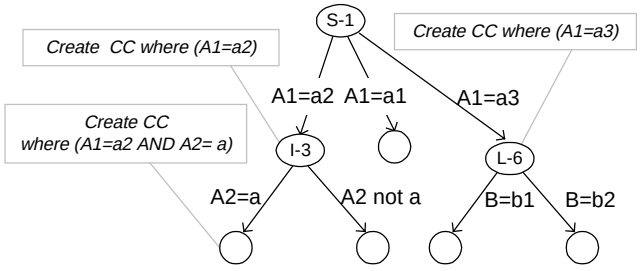


Figure 1: Queries Associated with Tree Nodes

at any stage each node obtains the counts pertinent to the data represented at the node, scores all possible partitions, and selects the best partition. This is illustrated in Figure 1. A node may be in any one of the following three states: *partitioned*, *active* or *leaf*. A node is partitioned if all its children nodes are created. A node that satisfies a stopping criterion becomes a leaf node. All other nodes are *active nodes*. For each active node, its count table needs to be constructed prior to partitioning. We use the term *frontier* to refer to the set of all active nodes.

2.2 Sufficient Statistics

A key insight here is that to score partitions a decision tree requires access to only *sufficient statistics* (a *set of counts*) and not to the data itself. More precisely, for each attribute, for each combination of its attribute value and class value, we need a count of the number of tuples where it co-occurs. Thus, all the splitting criteria used to determine Split(S) can be computed from the *counts table*, which is a simple 4-column table giving the set of counts of co-occurrences of each attribute value with each class value: (attr_name, value, class, count). This table (henceforth called CC table) has many uses. Once obtained, there is no further need to refer to the data again. Hence, the following observation holds:

Observation 1: In a decision tree algorithm, the single operation that needs to reference the data is the construction of CC table.

Furthermore, the following observation guides our tradeoffs in algorithm design:

Observation 2: The Counts table is typically much smaller than the size of the data, and in most applications does not grow with number of records.

2.3 Using the SQL Backend

There are two straightforward ways of using a classification client on a SQL backend. Unfortunately, as we will demonstrate in the experiment section, each of these techniques performs extremely poorly.

Generate a SQL query to extract data needed for all nodes: This corresponds to the case where the entire data set is extracted from the SQL database and loaded in the client secondary storage.

Generate SQL queries for creating CC tables: For each active node, its CC table may be created by executing a SQL statement at the server. Syntactically such a SQL statement may be expressed by using UNION (see below). Unfortunately, optimizers in most database systems are not capable of exploiting the commonality. Observe that the form of the SQL statement using UNION is *different* from the CUBE operation proposed in [GC*97]. Unlike CUBE, the grouping columns only share the class attribute and no grouping is required for combinations of other attributes. The SQL query to create the CC table for a single active node with *m* attributes $A_1,..,A_m$ has the following form:

```
Select "attr1" as attr_name, A1 as
value, class, count(*) From Data_table
Where node_condition
Group By class, A1
UNION … UNION
Select "attrm" as attr_name, Am as
value, class, count(*) From Data_table
Where node_condition
Group By class, Am
```

Since at any given time the decision tree may have multiple active nodes, we will need one UNION query to build the CC table for each of the nodes. (with different WHERE clauses).

3 Architecture

Figure 2 describes the architecture of the data mining system that we implemented on Microsoft SQL Server 7.0. The data-mining client maintains the decision tree structure and implements the scoring functions to be used in selecting the partition at any node in the decision tree. As we explained, the client need not access the individual rows from the data table but needs to see only the sufficient statistics for a node. The basic function of the client is to issue a batch for requests to the middleware for creating *counts tables* for every active node. This interface may be described using the following steps: :

1. Queue batch of requests, one for each active node.
2. Wait for middleware notification that some requests have been fulfilled
3. Process (consume) all counts tables returned by the middleware.
4. Grow the decision tree one level at each node whose request has been fulfilled.
5. If there are any active nodes left, go to 1, else output the decision tree.

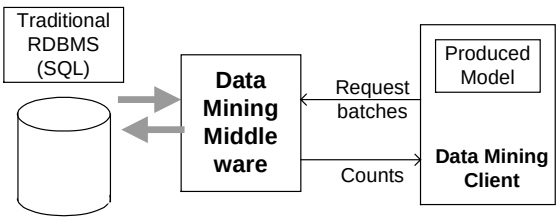


Figure 2: System Architecture, with Middleware

Thus, the middleware is responsible for supporting all data access requests from the clients. It extracts the information from the backend databases as needed (See Section 4).

3.1 Changes to Data Mining Client

Our architecture can support any classification algorithm that is driven by the sufficient statistics with few changes. We discuss how to adapt traditional in-memory classification clients to exploit the middleware that builds the CC tables. The most important change in a client's implementation involves invoking the middleware to obtain information from CC tables instead of having to compute them from data in its own memory. Another change that is needed is that the client no longer decides which nodes in the decision tree should be expanded next. Rather, at each step, the middleware determines (See Section 4.2) which of the active nodes are to be processed next. This behavior is illustrated in Figure 3. Note that the client is free to partition the processed nodes in any order it sees fit. This approach does not affect the decision tree that is finally produced by the classifier.

The specific details of our data mining client are as follows: the selection measure we used in our experiments is the standard entropy measure used in ID3, C4.5, and CART [B*84, Q93]. We did not implement any tree pruning criteria as our experiments are primarily intended to study behavior of the algorithm on generating a full decision tree. This can be easily implemented in our scheme.

4 Scalable Classification Middleware

In processing the set of requests from the data mining client, the scalable classification middleware employs a novel combination of techniques that exploits generic properties of decision-tree classification algorithms. The key techniques supported in the ***execution module*** of the middleware are:

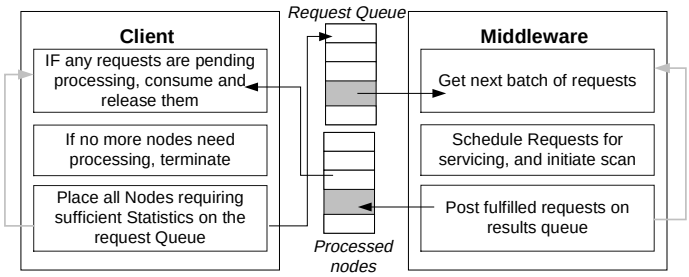


Figure 3: Client and Middleware Interaction

(a) Efficient computation of count tables for multiple active nodes using a single data scan.

(b) Staging data, i.e., copying relevant data from server to middleware file system and eventually to memory.

A ***scheduling module*** in the middleware complements the execution module. The scheduling module is responsible for determining:

(a) The order of processing of the outstanding requests from the client for building CC tables for active nodes.

(b) The data fragments that need to be staged.

4.1 The Execution Module

In this section, we describe how, given the list of the active nodes to be processed (as specified by scheduler), the execution module constructs their counts tables using a single data scan. We also discuss how staging of data from server to middleware is accomplished.

4.1.1 Efficient Computation of Counts Tables

The execution module is able to build counts tables for multiple active nodes efficiently in a single scan of data without the need to use external sorting. This capability allows us to significantly improve performance compared to straightforward SQL implementations where UNION queries (Section 2.3) are used to compute all counts.

The *scan-based-counting* algorithm we use to build the counts tables takes as input the designated set of active nodes (specified by the scheduler) and creates the counts tables. Note that since data may be staged, the data used to build the CC tables may reside either on the database sever, middleware files or in the middleware memory (as determined by middleware policy). However, unless the data for the specified active nodes are already in memory or local file a scan of data at the server is initiated. As each record in the server data set is retrieved via a cursor at the middleware, the relevant counters in the counts table are updated.

As mentioned earlier, the scheduling module ensures that the aggregate memory requirements for the CC tables (scheduled in any single scan) is estimated to be within memory available to the middleware. Any error in estimation is handled at runtime. Specifically, if due to an estimation error, no new count table entries can be accommodated in memory, then the scan-based-counting algorithm dynamically switches to a SQL based implementation. Furthermore, since the full counts table cannot be accommodated in the available memory, the rows of the counts tables are obtained from the server lazily [CFB97].

4.1.2 Staging Data

As the tree is grown, only *a fraction of the entire data set becomes relevant* for processing since over time some of the active nodes turn into leaves. Therefore, as the active nodes are expanded, the relevant data set for all the active nodes that are descendants of the current set of active nodes *decrease monotonically*. Thus, even if at the beginning the data set for any active node cannot be staged to middleware, it may be possible to stage data for a set of active nodes from the server to the middleware as the decision tree is grown. Data may be staged either to the middleware file system or to middleware memory. As the decision tree is grown, the relevant data set associated with a middleware file, where data has been staged, may also shrink significantly. In such cases, selective copying of staged data from the middleware file to middleware memory or creating a smaller middleware file is a useful operation. Thus, as the size of "active" data decreases, data will smoothly migrate from the SQL server, to middleware file system, and to middleware memory. With each node of the tree classifier we track the current location of its data. This is indicated as the prefix of the node number in Figure 1 (S for scan, I for middleware file, and L for in-memory).

When the data set for an active node is staged in the middleware, then the data set for processing all descendant nodes of that node becomes available at the middleware as well. Thus, database server scans to compute CC tables for any descendants of the nodes that have been cached are eliminated.

Finally, note that the application program using the scalable classification tool can customize staging. In particular, staging can be completely disabled or can be restricted to only caching in middleware files from database server or to only memory caching. This allows us to operate our middleware effectively in system environments that do not support a local disk or in which allocated memory is low.

4.2 Scheduler

The scheduler is responsible for driving the execution module. In particular, it provides:

- *Estimators* to determine the size of the data set and the count tables for the given data. This information is used in the following two steps.
- A *priority-based scheduling* scheme to pick a set of active nodes from the request queue.
- *Determining staging of data* for active nodes to enhance performance.

We discuss each of these aspects of the scheduler.

4.2.1 Estimating Data and Count Table Sizes

Let n_i refer to a node in the decision tree and let p_i be its parent. The estimator determines (exactly) the *size of the data set* associated with any node n_i as well as estimates the *sizes of the counts table* for the node n_i. The *data size* of an active node n_i, denoted by $| n_i |$, can be calculated *precisely* from the count table of its parent. This is because the scan of p_i gives the number of records that go to n_i (since partitions are of the form $A = v_i$ or $A \neq$ other). Hence memory load requirements are known.

The *count table size* of an active node, however, can only be estimated. The estimated size of the count

table of an active node is based on the number of distinct values of each attribute A_j appearing in the data at a node. Let us refer to the latter as card(n_i,A_j). To estimate card(n_i, A_j), the inequality card(n_i,A_j) ≤ card(p_i,A_j) provides an upper bound on the count table since we can add the above cardinalities over all attributes present in n_i. Note that the number of attributes present in n_i can be one less than in p_i, e.g., if the splitting condition were $A = v_i$. If size of CC table at parent node p_i is $|CC(p_i)|$, then if n_i is a child of p_i reached via a split on $A_j=v_i$, the quantity $|CC(p_i)|$-1 is a trivial upper bound on $|CC(n_i)|$. Furthermore, if the split at p_i was on every value of A_j, then $|CC(p_i)|$-card(p_i,A_j) is in fact an upper bound on $|CC(n_i)|$. Both these upper bounds are pessimistic. We chose to use the estimate of $|CC(n_i)|$:

$$\text{Est\_cc}(n_i) = \frac{|\,n_i\,|}{|\,p_i\,|}\sum_j \text{card}(p_i, A_j)$$ which is based on independence of the partitioning attribute with respect to the remaining attributes present in the node n_i. Since p_i is the *parent* of n_i, the estimate is (1) fairly conservative in use of available memory, and (2) there is *no propagation of error* since card(p_i, A_j) is known *exactly* without any uncertainty.

4.2.2 Priority-based Scheduling Scheme

Consider the case where for a give node *n*, data is available at middleware file *I*, created for the data set corresponding to an ancestor *p* of *n*. In such a case, we *always* use the file *I* to retrieve data for the node *n* and create CC tables. While not globally optimal, this scheduling policy is pragmatic. Therefore, while scheduling *n*, we *only* schedule other active nodes that are descendants of *p*, i.e., are able to share the same file *I*.

An analogous situation arises for nodes that have their data staged in the middleware memory. This can happen if in an earlier server scan, the data D' for an ancestor node *p* of *n* was loaded into memory. In this case, the goal is to process all active nodes in the subtree rooted at *p* and free up memory as quickly as possible. Hence, while scheduling *n*, we *only* schedule other active nodes that are descendants of *p*, i.e., are able to share the same data set D' in memory. Hence once the data for *p* is loaded, all descendants of *p* get precedence over other nodes in scheduling, leading eventually to flushing D' out of memory and freeing up the resource. The resulting priority-based scheduling scheme is determined by first applying Rule 1, followed by Rule 2 and then Rule 3:

<u>*Relative order among scans*</u>: nodes that require data loaded in middleware memory get preference over nodes that can make use of one or more files (or index structures – see Section 4.3.3). Nodes that require sequential scan get lowest priority:

(**Rule 1**) *In-Memory Scan > Middleware File Scan > Server Scan*.

<u>*Scheduling constraint for in-memory or file scans:*</u> If a set of nodes is to be serviced by local (middleware) file scan, then they must share a common ancestor for which the file was created, ensuring that the same file scan services all nodes. If a set of nodes can be serviced by data loaded in memory, then the set must share an ancestor node p such that data for this node is currently loaded in memory. Such a policy favors depth-first expansion of the subtree rooted at p if its data has been loaded in memory.

(**Rule 2**) *All nodes that are scheduled together must share the same in-memory data set or must share the same file on the middleware*

<u>*Ordering among eligible nodes:*</u> For a selected scan mode and data location (in case of file and in-memory scan), there may be many nodes to be serviced. Servicing each node requires memory to hold its counts table in memory, limiting the number of nodes that may be accommodated. For simplicity, we order eligible nodes by the *increasing* estimated sizes of count tables.

(**Rule 3**) *Node with smallest estimated size of the counts table has the highest precedence.*

4.2.3 Determining Staging of Data

We consider the option of moving data to middleware when there is available memory and/or file space. However, we consider staging in memory *only after* we have scheduled as many nodes as possible using the priority scheduling scheme above. The staging of data is governed by the following rules:

(**Rule 4**) *Only data for one or more of the nodes picked by the priority scheduling scheme qualify to be loaded to middleware file/memory*

(**Rule 5**) *Nodes are ordered by their decreasing size*: pick the node with the largest data size that may be accommodated in the available file/memory.

(**Rule 6**) *Caching server data to local file precedes caching data in memory*, i.e., data is first moved from the database server to local cache. Movement from local file to memory follows the same rules (4 and 5) that prioritize movement of data from database server to local cache.

To summarize, the steps in the scheduler are:

1. Estimate data size and count tables for each active node (unless it has already been evaluated)
2. Pick a set of active nodes using Rules 3-4

3. Exploit available file space in middleware by loading data for a subset of nodes picked in (2)

Exploit remaining memory in middleware by loading data from local files to memory (or, directly from server to memory, if appropriate).

4.3 Discussion

4.3.1 Reducing Data Transmitted from the Server

To ensure that each record fetched from the server to the middleware contributes to one or more of the counts, we generate a *filter expression* to be used in the select query. The filter expression is obtained from the collective set of active tree nodes to be processed, ensuring that only data that is relevant to the nodes are transmitted from the server to the client. With each node *n* of the decision tree, we can associate a predicate which is a conjunction of the predicates on the edges of the path from *n* to the root. Designate this expression by *S*. Given nodes $n_1,\ldots, n_k$ we generate the filter expression $(S_1 \vee \ldots \vee S_k)$. The predicate for each S_i is illustrated in Figure 1 (where clauses). This allows us to avoid the problem of having to tag records in the database explicitly with their memberships (e.g. in [MAR96,SAM96]), thus avoiding writes into a data table and consequently reducing I/O time.

4.3.2 File Splitting in Middleware

If data corresponding to an active node has been loaded in a middleware file, any future references to data for the active node or its descendants must sequentially scan the entire file. However, future descendants of the active node may use only a relatively small fraction of the data in the entire file. To optimize such accesses, smaller files are created to reduce the cost of scanning local files At any point, the fraction of cache data that is being counted for the current set of active nodes acts as the thresholding parameter for creation of new middleware files. If this threshold is set to 100%, then a new cache will be created for each node in the tree. This approach is wasteful early on in the tree growing process because a complete scan of the data is usually needed at each level of the tree, and a price is paid for unnecessarily partitioning the file. However, towards the end of the tree growing process, much of the data will not be scanned at each new level of the tree and the compact size of a node's cache contributes to efficient access.

4.3.3 Using Auxiliary Structures

As the decision tree grows, the fraction of the relevant (active) data set monotonically diminishes. We considered whether this effect could be exploited to reduce the work done by the server on scanning data. In other words, if D' is the relevant subset of data set D, we considered if we can build an auxiliary structure such that the server can scan *only* the records in D'. This leverages the same intuition as in file splitting above, except that we use index structures to simulate it at the server. This can be accomplished in several ways:

a) *Copy data and use new table:* copy the subset of data into a new temporary table. In most cases, such "data copying" results in unacceptably high overhead.

b) *Copy TIDs and make indexed access to D:* We can copy only the tuple identifiers (TID) of the data tuples in D' from D and create a new temp table T'. We can then retrieve the data set D' using a join between T and T' on the TID attribute. Join overhead negatively impacts the improvement in reduced data scans.

c) *Open a Keyset cursor and use Stored Procedure:* In this mode, we define a keyset cursor on D'. This makes it possible to do a sequential scan of D' without paying the overhead of either copying the data set or doing a join at runtime. Use of a keyset cursor will result in the client receiving all the tuples in D' each time the cursor scan is used. However, in future scans, only a subset of D' maybe needed since the active data set decreases monotonically. Therefore, we would like to filter data before they are sent to the middleware. This is achieved by a *stored procedure* that applies the filters on the results obtained by the cursor before the results are returned. Despite the promise, in practice, the gain in efficiency due to this technique was limited. Note that this technique applies only when the relevant data set has shrunk to a small percentage of the given file (around 10%). Unfortunately, at a low threshold such as 10%, in most cases the decision tree is quite close to being complete, limiting the effectiveness of this technique. Thus, despite their potential, our experiments indicate that unlike file splitting in the middleware, optimizing data scan using the above alternatives for reducing the cost of data scans at the server are not beneficial (See Section 5.2.5).

5 Experimental Study

The data scanning and counting algorithms are implemented on Windows NT in C++ as a family of COM Automation objects that a consumer manipulates to extract sufficient statistics from a data source. A queue of pending and completed Counts is maintained. The items are dequeued based on the Scheduler's policy. Counts tables are stored as binary trees. The unique combinations of attribute (column) number and state (value) number specify an entry in the counts table. Because of the way points are sorted in the tree, retrieving a vector of counts for the states of a class correlated with a particular attribute and its state is efficient [CFB97].

5.1 Experimental Setup

Our experiments were conducted using three data sets. The first two are synthetically generated data sets designed to study the behavior of our system in a controlled setting where properties of the tree to be generated are in some sense "predictable". The third data set is a large, publicly available database obtained from the U.S. Census Bureau. The idea here was to ensure that the improvement demonstrated on synthetic data sets still holds when the algorithm is used on a real database that can be publicly accessed for benchmarking purposes. However, for lack of space, these results are omitted and can be found in [CFB97]. We used 4 Pentium-II machines, each with about 128M RAM. All machines were running Microsoft Windows NT 4.0 and had OLE-DB version 1.5 communicating with Microsoft SQL Server 7.0. In this section, we present experimental results that establish:

- Staging directed by the scheduler greatly enhances performance. In Section 5.2.1, we study the effect of staging data to memory. In Section 5.2.2, we study the effect of staging data to local file system and demonstrate its effectiveness.
- Our techniques are scalable with respect to number of attributes and number of rows (Section 5.2.3), and we significantly outperform straightforward SQL Server implementations (Section 5.2.3).
- Our techniques are robust with respect to varying shapes of decision trees (Section 5.2.4).

5.1.1 Data from Random Trees

Given a decision tree, data was generated such that the effect of applying classification on the data will be the given decision tree. Such a generation process allowed us to study the scaling behavior of our scheme as we varied certain properties of the tree.

The tree generation program provides many parameters for controlling the structure of the tree that is used to produce the data. These include number of leaves in the generating tree (measure of tree size), branching factor and tree skewness. The branching parameter allows us to control the bushiness of the generating decision tree. It also makes it possible to explore various distributions of cases and classes in the tree. Furthermore, the number of attributes and the desired average number of values per attribute may be set.

5.1.2 Data from Mixtures of Gaussians

This data set was generated from a mixture of Gaussians in 100 dimensions. The means of the Gaussians are chosen uniformly randomly over the interval [-5, +5] in each dimension. The variances in each dimension are uniformly random over the interval [0.7, 1.5]. We generated 10,000 samples from each Gaussian (class), giving us a starting database of 1,000,000 samples.

These data sets allow us to test performance of our algorithm in a setting where properties of the data are well understood (as opposed to properties of the generating tree in the previous section). Another difference from the data generated in Section 5.1.1 is that the data do not come from a known decision tree. This is to verify that our scheme is not well-tuned for a specific type of data set. Furthermore, with the mixture of Gaussians, we may omit dimensions and still have a mixture of Gaussians, allowing us to vary dimensionality and keep the data properties fixed. Finally, by taking out some of the Gaussians, we can vary the number of classes, again without changing fundamental properties of the data.

5.1.3 Default Settings

Except for the "Real World" data set, the performance and functionality tests were performed on synthetic data produced by the algorithm described in Section 5.1.1. Unless otherwise stated, the following parameters were used to produce data:

(1) Number of attributes = 25 (2) Number of attribute values = 4 (with standard deviation=4) (3) Number of class values = 10 (4) Fanout factor = 0 (5) Complete splits = true, (6) standard deviation on number of cases generated per leaf = 0.0

Although the data was generated to produce even splits of cases amongst an attribute's 4 possible values, only binary trees were grown from the data. In general, this resulted in sufficiently "bushy" trees that rounded out at the bottom.

5.2 Results

5.2.1 Effects of Memory Size

The amount of memory available in the system has two important effects. If not enough memory is available to build count tables for all nodes in the active list, then multiple scans of the database will be needed to build CC tables for active nodes. Second, if any memory is not reserved for count tables, it can be used to stage data in memory. The experiments in this section demonstrate these effects.

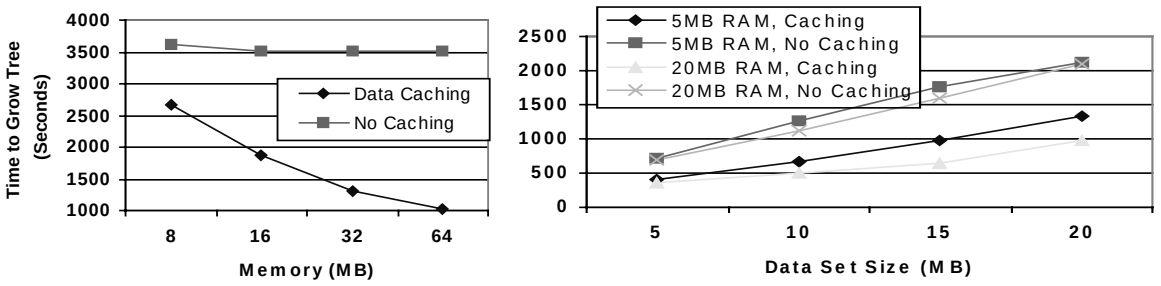
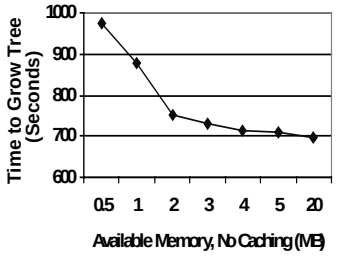


Figure 4: Effect of Memory Buffer size and Database Size



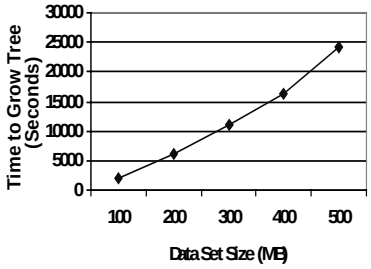


Figure 5: Available memory and database size (no staging)

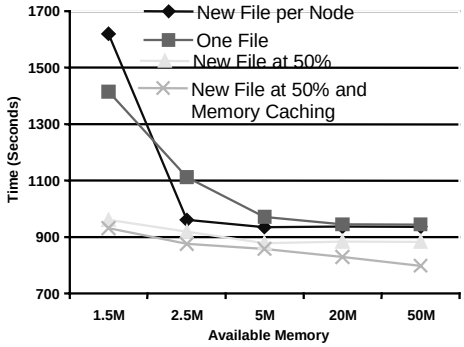


Figure 6: Effect of File Staging

(A) Increasing Memory Usage with a Fixed Data Size and Vice Versa (Fig. 4):

For the data generator, the number of leaves was set to 500 and the cases per leaf were around 950. This produced an approximately 50 MB data set that generates a 7000 node tree. In the 64MB RAM with caching mode, the entire data set is loaded into memory on the first scan. In the other curve (no caching), additional memory has no effect after a full set of count tables can be built in one scan. Both curves flatten off after 64MB. The chart on the right shows the effect of data set size, the number of leaves is set to 500 and the cases per leaf are varied to produce the needed data set size.

(B) Limited Memory for Count Tables (Fig 5a)

The 5MB data set from above is used again here. This graph shows the effect of not having enough memory to hold the count tables for all active nodes, forcing us to make multiple scans to count the frontier. There is no data caching in this experiment. This experiment demonstrates the key effect of staging data in middleware memory.

5.2.2 Effects of Staging Data in Middleware File System

Overall tree building performance is also improved by staging data in the middleware file system. It is particularly effective in situations where available middleware memory is low and adequate local file system space is available. The chart in Figure 6 illustrates the effect on total tree building time for the four configurations of staging as amount of available memory increases. These experiments were run on the Census data set, however the scoring algorithm was adjusted to produce a smaller tree (about 300 nodes)

The four caching configurations are as follows: (1) One staging file for every active node. A new middleware file (cache) is created for each active node of the tree. (2) Singleton file caching, i.e., only one staging file is used for the entire tree and then repeatedly scanned. (3) Hybrid scheme. One staging file is created and repeatedly scanned until the cardinality for the set of nodes being processed is below 50% of the cardinality of the source file cache. (4) The same as (3) but staging data in memory was also enabled. The amount of available memory was varied to demonstrate two effects. First, with lower amounts of memory, only a few of the count tables can fit into memory at one time, thus causing multiple scans of a shared middleware staging file for expanding one level of the tree. This effect goes away as memory increases. Second, in configuration (4) there is a trade off between memory for counting and memory for data staging until the 50M case in which all data and counts fit into memory (no staging file is used)

5.2.3 Effects of Data Size

These experiments demonstrate the ability for the counting algorithm to “scale up” to larger data sets.

(A) Increasing Number of Rows (Fig. 5b): The data generator used 500 leaves and a gradual increase in the number of cases per leaf. The final size of this

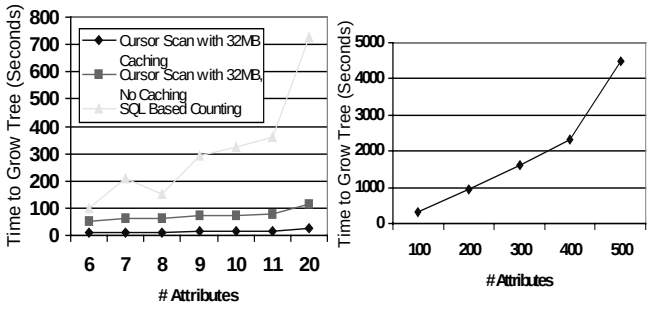


Figure 7: Varying the number of attributes

data set is 5 million records. These runs have 64MB of memory for data staging and counting. One obvious effect as the amount of data being classified increases is that a smaller proportion of the data can be staged into the middleware memory. This in turn leads to more scans of the data.

(B) Increasing Number of Attributes (Fig. 7): One not so obvious effect of a large number of attributes is that the estimated size of a count table goes up. This graph shows an increasing number of attributes with a fixed number of cases. The size of the data set grows from 40MB to 200MB. The data generator uses an increasing number of binary attributes (only 2 attribute values), 200 leaves, and 125 cases per leaf for a total of 100,000 records. These runs have 64MB of memory for data staging and CC tables.

When data is stored in a SQL capable RDBMS, an alternative to counting sufficient statistics with a cursor scan is to harness the power of SQL and have all counting done by the database server via a union of group by queries. An implementation of such an approach was used for comparison. The data set was similar to the one above, but with the number of leaves and cases per leaf scaled down to produce data sets ranging in size from 1MB to 3MB. For larger data sets, the straightforward SQL implementation results in an unacceptably poor performance.

5.2.4 Effects of Tree Shape

It is important to vary the tree shape since tree shape has a direct impact on the relative number of counts (CC) tables that need to be accommodated. Varying tree shape also demonstrates how our algorithms cope with changes in the size of the relevant data set as the execution progresses. For example, a broad (bushy) tree's active nodes may require several cursor scans, and a very thin deep tree requires a scan for every level of the tree.

(A) *Increasing Attribute Values (Fig. 8a):* We used a tree with 200 leaves and 480 cases per leaf to generate 10MB of data from a long lop-sided tree (see

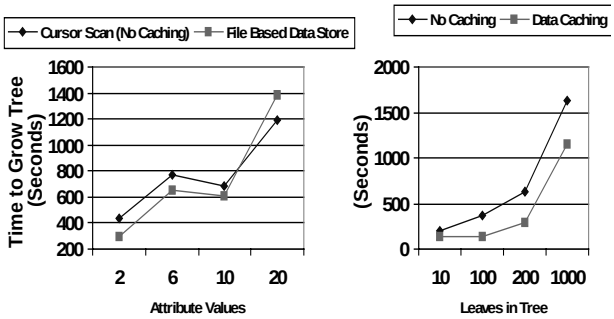


Figure 8: Effect of type of generating tree

[CFB97]) for these tests. The Curve for "File Based Data Store" shows the effect of not using the RDBMS as a continuous source of data. Records are read from the database and saved locally. During early part of the execution, this seems like a good idea because reading from the file is faster than reading from the cursor. However, as the scope of interesting data decreases pulling data from the server becomes faster than reading from the middleware file (server can utilize the WHERE clause to limit records).

(B) Increasing Number of Leaves (Figure 8b): Increasing the number of leaves for a fixed data set size will make the data points less "similar" and therefore harder to classify. This will cause more scans with the cursor, but also greatly increases the size of the request frontier. To show the ill effects of these phenomena, this graph represent runs with and without data caching and only use a small amount of memory (8MB) for count tables for the 10MB data set size.

5.2.5 Use of Index Scans

In Section 4.3.3 we introduced the notion of building server-based index structures on the fly to optimize access to data for active nodes. Our results show that such indexes are not beneficial. Due to space limitations, we omit the details, but we present a brief overview of an experiment. In this experiment, we simulate an *idealized situation* on the server by neglecting the cost of creating index structures. We also pick a tree that maximizes the potential for benefit from indexing. The tree we choose is built from the Census database and has the property that at some point 70% of the data becomes inactive. The tree has one long path (thin subtree) that utilizes 30% of the data and monotonically drops to 1% as subtree is grown. Results showed that even under such favorable circumstances, indexing does not help [CFB97].

6 Related Work

There is fairly extensive literature on decision tree generation in Statistics [B*84] and Machine Learning [Q93, FI92b, FI93, F94]. Since the implementations available for these communities assume the data to be in memory, much of the work has been with very small data sets, typically in the hundreds to thousands of records. More recently, database research has addressed the problem of scaling classification algorithms, e.g., [MAR96, SAM96, GRG98]. These algorithms may need to create new data structures that correspond to vertical partitions of the given data set (one partition for each attribute in the data set), with at least the class attribute replicated. In contrast, we avoid the need to create vertical partitions.

The RainForest work [GRG98] comes closest to the work reported in this paper. Our work was developed independently and contemporaneously. We share with RainForest the observation that constructing CC tables efficiently is at the heart of achieving high scalability for the family of decision tree classifiers. Their AVC-group data structure corresponds to CC-tables in our system. However, despite this similarity, our approach is distinguished by our objective to develop a middleware that works against SQL databases efficiently. We provide the client with complete freedom to consume CC tables in any desired order (not necessarily just depth-first or breadth-first) while allowing the middleware also to serve the active nodes in any order so as to optimize performance. We also support a smooth transition between file staging and staging in middleware memory. This has led to a powerful scheduler that ensures optimized server scans and staging. Also, our experimental results are based on an implementation on a commercial DBMS system (Microsoft SQL Server).

Finally, it is important to point out that while we do need multiple aggregations over the same data set, the nature of the aggregations is quite different from that needed in the CUBE construct [GC*97]. We need to aggregate the data with varying filter expressions and our aggregations require computing count for each combination of attribute value and value from the class variable. We refer the reader back to Section 2.3 for a detailed exposition.

7 References

[B*84] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone, 1984. *Classification and Regression Trees.* Monterey, CA: Wadsworth & Brooks.

[CFB97] S. Chaudhuri, U. Fayyad, J. Bernhardt: 1998. "Scalable Classification over SQL Databases," *Microsoft Research Technical Report* MSR-TR-97-35.

[F94] Fayyad, U.M. 1994. Branching on Attribute Values in Decision Tree Generation. *Proc. 12th National Conf. on Artificial Intelligence*, p. 601--606, MIT Press.

[GC*97] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, 1997. "Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals", *Data Mining and Knowledge Discovery*, vol. 1, no. 1, 1997.

[GRG98] J. Gehrke, R. Ramakrishnan, and V. Ganti, 1998. "RainForest – A Framework for Fast Decision Tree Construction of Large Datasets". *Proc. of VLDB-98.*

[MAR96] M. Mehta, R. Agrawal, and J. Rissanen, 1996. "SLIQ: a fast scalable classifier for data mining", *Proceedings of EDBT-96*, Springer Verlag.

[Q93] J.R. Quinlan, 1993. *C4.5: Programs for Machine Learning*, Morgan Kaufman, 1993.

[SAM96] J.C. Shafer, R. Agrawal, M. Mehta, 1996. "SPRINT: A Scalable Parallel Classifier for Data Mining", *Proc. of VLDB-96.*