RainForest - A Framework for Fast Decision Tree Construction of Large Datasets

Johannes Gehrke[*] *Raghu Ramakrishnan* *Venkatesh Ganti*[†]

Department of Computer Sciences, University of Wisconsin-Madison
{johannes,raghu,vganti}@cs.wisc.edu

Abstract

Classification of large datasets is an important data mining problem. Many classification algorithms have been proposed in the literature, but studies have shown that so far no algorithm uniformly outperforms all other algorithms in terms of quality. In this paper, we present a unifying framework for decision tree classifiers that separates the *scalability* aspects of algorithms for constructing a decision tree from the central features that determine the *quality* of the tree. This generic algorithm is easy to instantiate with specific algorithms from the literature (including C4.5, CART, CHAID, FACT, ID3 and extensions, SLIQ, Sprint and QUEST).

In addition to its generality, in that it yields scalable versions of a wide range of classification algorithms, our approach also offers performance improvements of over a factor of five over the Sprint algorithm, the fastest scalable classification algorithm proposed previously. In contrast to Sprint, however, our generic algorithm requires a certain minimum amount of main memory, proportional to the *set* of distinct values in a column of the input relation. Given current main memory costs, this requirement is readily met in most if not all workloads.

[*]Supported by an IBM Corporate Fellowship

[†]This research was supported by Grant 2053 from the IBM Corporation.

Proceedings of the 24th VLDB Conference
New York, USA, 1998

1 Introduction

Classification is an important data mining problem[AIS93]. The input is a database of *training records*; each record has several attributes. Attributes whose underlying domain is totally ordered are called *ordered* attributes, whereas attributes whose underlying domain is not ordered are called *categorical* attributes. There is one distinguished attribute, called *class label*, which is a categorical attribute with a very small domain. (We will denote the elements of the domain of the class label attribute as *class labels*; the semantics of the term class label will be clear from the context). The remaining attributes are called *predictor attributes*; they are either ordered or categorical in nature. The goal of classification is to build a concise model of the distribution of the class label in terms of the predictor attributes. The resulting model is used to assign class labels to a database of *testing records* where the values of the predictor attributes are known but the value of the class label is unknown. Classification has a wide range of applications, including scientific experiments, medical diagnosis, fraud detection, credit approval and target marketing.

Many classification models have been proposed in the literature. (For overviews of classification methods see [WK91, MST94].) Decision trees are especially attractive for a data mining environment for three reasons. First, due to their intuitive representation, they are easy to assimilate by humans [BFOS84]. Second, they can be constructed relatively fast compared to other methods [MAR96, SAM96]. Last, the accuracy of decision tree classifiers is comparable or superior to other models [LLS97, Han97]. In this paper, we restrict our attention to decision tree classifiers.

Within the area of decision tree classification, there exist a large number of algorithms to construct decision trees (also called classification trees; we will use both terms interchangeably). Most algorithms in the machine learning and statistics community are main memory algorithms, even though today's databases are in general much larger than main memory [AIS93].

There have been several approaches to dealing with large databases. One approach is to discretize each ordered attribute and run the algorithm on the discretized data.

But all discretization methods for classification that take the class label into account when discretizing assume that the database fits into main memory [Qui93, FI93, Maa94, DKS95]. Catlett [Cat91] proposed sampling at each node of the classification tree, but considers in his studies only datasets that could fit in main memory. Methods for partitioning the dataset such that each subset fits in main memory are considered by Chan and Stolfo [CS93a, CS93b]; although this method enables classification of large datasets their studies show that the quality of the resulting decision tree is worse than that of a classifier that was constructed taking the complete database into account at once.

In this paper, we present a framework for scaling up existing decision tree construction algorithms. This general framework, which we call RainForest for rather whimsical reasons[1], closes the gap between the limitations to main memory datasets of algorithms in the machine learning and statistics literature and the scalability requirements of a data mining environment. The main insight, based on a careful analysis of the algorithms in the literature, is that most (to our knowledge, all) algorithms (including C4.5 [Qui93], CART [BFOS84], CHAID [Mag93], FACT [LV88], ID3 and extensions [Qui79, Qui83, Qui86, CFIQ88, Fay91], SLIQ and Sprint [MAR96, MRA95, SAM96] and QUEST [LS97]) access the data using a common pattern, as described in Figure 1. We present data access algorithms that scale with the size of the database, adapt gracefully to the amount of main memory available, and are not restricted to a specific classification algorithm. (This aspect of decision tree classification is addressed extensively in statistics and machine learning.) Our framework applied to algorithms in the literature results in a scalable version of the algorithm *without modifying the result of the algorithm*. Thus, we do not evaluate the quality of the resulting decision tree, which is not affected by our framework; instead we concentrate on scalability issues.

The rest of the paper is organized as follows. In Section 2, we formally introduce the problem of decision tree classification and describe previous work in the database literature. In Section 3, we introduce our framework and discuss how it encompasses previous work. In Section 4, we present scalable algorithms to construct decision trees, and in Section 5 we present results from a detailed performance evaluation. We conclude in Section 6.

2 Decision tree classifiers

2.1 Problem definition

A *decision tree* T is a model of the data that encodes the distribution of the class label in terms of the predictor attributes. It is a directed, acyclic graph in form of a tree. The root of the tree does not have any incoming edges. Every other node has exactly one incoming edge and zero or more outgoing edges. If a node n has no outgoing edges we call n a *leaf node*, otherwise we call n an *internal node*. Each leaf node is labeled with one class label; each internal node is labeled with one predictor attribute called the *splitting attribute*. Each edge e originating from an internal node n has a predicate q associated with it where q involves only the splitting attribute of n. The set of predicates P on the outgoing edges of an internal node must be *non-overlapping* and *exhaustive*. A set of predicates P is *non-overlapping* if the conjunction of any two predicates in P evaluates to `false`. A set of predicates P is *exhaustive* if the disjunction of all predicates in P evaluates to `true`. We will call the set of predicates on the outgoing edges of an internal node n the *splitting predicates of* n; the combined information of splitting attribute and splitting predicates is called the *splitting criteria* of n and is denoted by $\texttt{crit}(n)$.

For an internal node n, let $E = \{e_1, e_2, \ldots, e_k\}$ be the set of outgoing edges and let $Q = \{q_1, q_2, \ldots, q_k\}$ be the set of predicates such that edge e_i is associated with predicate q_i. Let us define the notion of the *family* of tuples of a node with respect to database D. The family $F(r)$ of the root node r of decision tree T is the set of all tuples in D. For a non-root node $n \in T$, $n \neq r$, let p be the parent of n in T and let $q_{p\to n}$ be the predicate on the edge $e_{p\to n}$ from p to n. The family of the node n is the set of tuples $F(n)$ such that for each tuple $t \in F(n)$, $t \in F(p)$ and $q_{p\to n}(t)$ evaluates to `true`. Informally, the family of a node n is the set of tuples of the database that follows the path from the root to n when being classified by the tree. Each path W from the root r to a leaf node n corresponds to a classification rule $R = P \to c$, where P is the conjunction of the predicates along the edges in W and c is the class label of node n.

There are two ways to control the size of a classification tree. A *bottom-up pruning* algorithm [MRA95] has two phases: In phase one, the growth phase, a very deep tree is constructed. In phase two, the *pruning phase*, this tree is cut back to avoid overfitting the training data. In a *top-down pruning* algorithm [RS98] the two phases are interleaved: Stopping criteria are calculated during tree growth to inhibit further construction of parts of the tree when appropriate. In this paper, we will concentrate on the tree growth phase, since it is due to its data-intensive nature the most time-consuming part of decision tree construction [MAR96, SAM96]. Whether the tree is pruned top-down or bottom-up is an orthogonal issue.

2.2 Previous work in the database literature

Agrawal et al. introduce in [AGI$^+$92] an interval classifier that could use database indices to efficiently retrieve portions of the classified dataset using SQL queries. However, the method does not scale to large training sets [SAM96]. Fukuda et al. [FMM96] construct decision trees with two-dimensional splitting criteria. Although their algorithm can produce rules with very high classification accuracy, scalability was not one of the design goals. In addition, the decision tree no longer has the intuitive representation of a tree with one-dimensional splits at each node. The decision tree classifier in [MAR96], called SLIQ, was designed

[1] There are lots of trees to choose from, and they all grow fast in RainForest! We also happen to like rainforests.

for large databases but uses an in-memory data structure that grows linearly with the number of tuples in the training database. This limiting data structure was eliminated in [SAM96], which introduced Sprint, a scalable classifier.

Sprint works for very large datasets and removes all relationships between main memory and size of the dataset. Sprint builds classification trees with binary splits using the `gini` index [BFOS84] to decide the splitting criterion; it controls the final quality of the decision tree through an application of the MDL principle [Ris89, MRA95]. To decide on the splitting attribute at a node n, the algorithm requires access to $F(n)$ for each ordered attribute in sorted order. So conceptually, for each node n of the decision tree, a sort of $F(n)$ for each ordered attribute is required. Sprint avoids sorting at each node through the creation of *attribute lists*. The attribute list L_a for attribute a is a vertical partition of the training database D: For each tuple $t \in D$ the entry of t into L_a consists of the projection of t onto a, the class label and t's record identifier. The attribute lists are created at the beginning of the algorithm and sorted once as a preprocessing step.

During the tree growth phase, whenever an internal node n splits, $F(n)$ is distributed among n's children according to $\texttt{crit}(n)$. Since every tuple is vertically partitioned over all attribute lists, each attribute list needs to be distributed separately. The distribution of an attribute list is performed through a hash-join with the attribute list of the splitting attribute; the record identifier, which is duplicated into each attribute list, establishes the connection between the parts of the tuple. Since during the hash-join each attribute list is read and distributed sequentially, the initial sort order of the attribute list is preserved.

In recent work, Morimoto et al. developed algorithms for decision tree construction for categorical predictor variables with large domains [YFM$^+$98]; the emphasis of this work is to improve the quality of the resulting tree. Rastogi and Shim developed PUBLIC, a scalable decision tree classifier using top-down pruning [RS98]. Since pruning is an orthogonal dimension to tree growth, their techniques can be easily incorporated into our schema.

2.3 Discussion

One can think of Sprint as a *prix fixe* all-you-can-eat meal in a world-class restaurant. Sprint runs with a minimal amount of main memory and scales to large training databases. But it also comes with some drawbacks. First, it materializes the attribute lists at each node, possibly tripling the size of the database (it is possible to create only one attribute list for all categorical attributes together as an optimization). Second, there is a large cost to keep the attribute lists sorted at each node n in the tree: Since the connection between the vertically separated parts of a tuple can only be made through the record identifier, a costly hash-join needs to be performed. The size of the hash table is proportional to the number of tuples in $F(n)$ and thus can be very large. Overall, Sprint pays a significant price for its scalability. As we will show in Section 3, some observations about the nature of decision tree algorithms enable us to speed up Sprint significantly in most cases. To return to our restaurant analogy, the techniques in Section 3 allow you to sample some RainForest Crunch (TM) ice-cream in the restaurant, paying for just what you ordered.

The emphasis of the research in the machine learning and statistics community has been on improving the accuracy of classifiers. Many studies have been performed to determine which algorithm has the highest prediction accuracy [SMT91, BU92, DBP93, CM94, MST94]. These studies indicate that no algorithm is uniformly most accurate over all the datasets studied. (Mehta et al. also show quality studies [MRA95, MAR96] which indicate that the accuracy of the decision tree built by Sprint is not uniformly superior.) We have therefore concentrated on developing a unifying framework that can be applied to most decision tree algorithms, and results in a scalable version of the algorithm without modifying the result. That is, *the scalable versions of the algorithms produce exactly the same decision tree as if sufficient main memory were available to run the original algorithm on the complete database in main memory*. To carry our restaurant analogy one (last!) step further, the techniques in Section 4 allow you to pick a different restaurant every day, eat there as little or much as you want, and pay only for what you order.

3 The RainForest Framework

We first introduce the well-known greedy top-down decision tree induction schema. Then we show how this schema can be refined to the generic RainForest Tree Induction Schema and detail how the separation of scalability issues from quality concerns is achieved. Concluding this section, we overview the resulting design space for the algorithms presented in Section 4.

Decision tree algorithms build the tree top-down in the following way: At the root node r, the database is examined and the best splitting criterion $\texttt{crit}(r)$ is computed. Recursively, at a non-root node n, $F(n)$ is examined and from it $\texttt{crit}(n)$ is computed. (This is the well-known schema for top-down decision tree induction; for example, a specific instance of this schema for binary splits is shown in [MAR96]). This schema is shown in Figure 1.

A thorough examination of the algorithms in the literature shows that the greedy schema can be refined to the generic *RainForest Tree Induction Schema* shown in Figure 1. Most decision tree algorithms (including C4.5, CART, CHAID, FACT, ID3 and extensions, SLIQ, Sprint and QUEST) proceed according to this generic schema and we do not know of any algorithm in the literature that does not adhere to it. In the remainder of the paper, we denote by $\mathcal{CL}$ a representative classification algorithm.

Note that at a node n, the utility of a predictor attribute a as a possible splitting attribute is examined independent of the other predictor attributes: The information about the class label distribution for each distinct attribute value of a is sufficient. We define the *AVC-set* of a predictor attribute a at node n to be the projection of $F(n)$ onto a and the

```
Input: node $n$, partition $D$, classification algorithm $\mathcal{CL}$
Output: decision tree for $D$ rooted at $n$

Top-Down Decision Tree Induction Schema:
BuildTree(Node n, datapartition $D$, algorithm $\mathcal{CL}$)
(1)   Apply $\mathcal{CL}$ to $D$ to find crit($n$)
(2)   let $k$ be the number of children of $n$
(3)   if (k > 0)
(4)       Create $k$ children $c_1, \ldots, c_k$ of $n$
(5)       Use best split to partition $D$ into $D_1, \ldots, D_k$
(6)       for ($i = 1$; $i \leq k$; i++)
(7)           BuildTree($c_i$, $D_i$)
(8)       endfor
(9)   endif
RainForest Refinement:
(1a)  for each predictor attribute $p$
(1b)      Call $\mathcal{CL}$.find_best_partitioning(AVC-set of $p$)
(1c)  endfor
(2a)  $k$ = $\mathcal{CL}$.decide_splitting_criterion();
```
Figure 1: Tree Induction Schema and Refinement

class label whereby counts of the individual class labels are aggregated. We define the *AVC-group* of a node n to be the set of all AVC-sets at node n. (The acronym AVC stands for for **A**ttribute-**V**alue, **C**lasslabel.) Note that the size of the AVC-set of a predictor attribute a at node n depends only on the number of distinct attribute values of a and the number of class labels in $F(n)$.

The main difference between the greedy top-down schema and the subtly refined RainForest Schema is, that the latter isolates an important component, the AVC-set. The AVC-set allows the separation of scalability issues of the classification tree construction from the algorithms to decide on the splitting criterion: Consider the main memory requirements at each step of the RainForest Schema shown in Figure 1. In lines (1a)–(1c), the AVC-sets of each predictor attribute are needed in main memory, one at a time, to be given as argument to procedure $\mathcal{CL}$.find_best_partitioning. Thus, the total main memory required in lines (1a)–(1c) is the maximum size of any single AVC-set. In addition, Algorithm $\mathcal{CL}$ stores for each predictor attribute the result of procedure $\mathcal{CL}$.find_best_partitioning as input to the procedure $\mathcal{CL}$.decide_splitting_criterion; the size of these statistics is negligible. In line (2a), all the statistics collected in lines (1a)–(1c) are evaluated together in procedure $\mathcal{CL}$.decide_splitting_criterion; the main memory requirements for this step are minimal. Lines (3)–(9) distribute tuples from one partition to several others; one page per open file is needed.

Following the preceding analysis based on insights from the RainForest Schema, we can make the (now rather trivial) observation that as long as we can find an efficient way to construct the AVC-group of node n, we can scale up any classification algorithm $\mathcal{CL}$ that adheres to the generic RainForest Schema.

Consider the size S_a ofthe AVC-set of predictor attribute a at a node n. Note that S_a is proportional to the number of distinct attribute values of attribute a in $F(n)$, and not to the size of the family $F(n)$ of n. Thus, for most real-life datasets, we expect that the whole AVC-group of the root node will fit entirely in main memory, given current memory sizes; if not it is highly likely that at least the AVC-set of each individual predictor attribute fits in main memory.

The assumption that the AVC-group of the root node r fits in-memory does not imply that the input database fits in-memory! The AVC-group of r is *not* a compressed representation of $F(r)$; $F(r)$ can not be reconstructed from the AVC-group of r. Rather the AVC-group of r contains aggregated information that is sufficient for decision tree construction. In Section 5, we calculate example numbers for the AVC-group of the root node generated by a synthetic data generator introduced by Agrawal et al. in [AIS93] (which was designed to model real-life data). The maximum memory size for the AVC-group of the generated datasets is about 16 megabytes. With current memory sizes of 64 megabytes for home computers, we believe that in a corporate data mining environment the AVC-group of the root node will almost always fit in main memory; otherwise at least each single AVC-set of the root node will fit in-memory. Depending on the amount of main memory available, three cases can be distinguished:

1. The AVC-group of the root node fits in main memory. We describe algorithms for this case in Sections 4.1, 4.2, and 4.3.

2. Each individual AVC-set of the root node fits in main memory, but the AVC-group of the root node does not fit in main memory. We describe algorithms for this case in Section 4.4.

3. None of the individual AVC-sets of the root fit in main memory. This case is discussed in the full paper.

In understanding the RainForest family of algorithms, it is useful to keep in mind that the following steps are carried out for each tree node n, according to the generic schema in Figure 1:

1. **AVC-group Construction:** If an AVC-group does not already exist when the node n is considered, we must read $F(n)$ in order to construct the AVC-group. This involves a scan of the input database D or a materialized partition of D that is a superset of $F(n)$. Sometimes, we need to construct the AVC-group one AVC-set at a time.

2. **Choose Splitting Attribute and Predicate:** This step uses the decision tree algorithm $\mathcal{CL}$ that is being scaled using the RainForest framework; to our knowledge all decision tree algorithms make these choices by examining the AVC-sets of the node one by one.

3. **Partition D Across the Children Nodes:** We must read the entire dataset and write out all records, partitioning them into child “buckets” according to the splitting criterion chosen in the previous step. If there is sufficient memory, we can build the AVC-groups for one or more children at this time, as an optimization.

State of n	Precondition	Processing Behavior of tuple t
Send	crit(n) has been computed; n's children nodes are allocated; n is root, or parent of n is in state Send	t is sent to a child according to crit(n)
Fill	n is root node, or parent of n is in state Send	The AVC-group at n is updated
Write	n is root, or n's parent is in state Send	t is appended to a n's partition
FillWrite	n is root node, or parent of n is in state Send	The AVC-group at n is updated by t; t is appended to n's partition
Undecided		No processing takes place
Dead	crit(n) has been computed; either n does not split or all children of n are in state Dead	No processing takes place

Figure 2: States and Processing Behavior

The algorithms that we present in Section 4 differ primarily in how they utilize additional memory in the third step, and how they deal with insufficient memory to hold an AVC-group in the first step.

Comparing the size of the AVC-group of a node n to the attribute lists created in Sprint [SAM96] for n, the AVC-group is typically much smaller than even a single attribute list, because the AVC-set size is proportional to the number of distinct values in the columns of D, rather than to the number of records in D. Although we outperform Sprint by about a factor of five, the primary design goal of the RainForest framework was not to outperform Sprint, but rather to provide a general framework to scale up a broad range of decision-tree classification algorithms from the literature. The reason why the techniques used in Sprint do not straightforwardly extend to a broader range of algorithms is that the data management of Sprint is designed to enable efficient *sequential* access to ordered attributes in sorted order. Thus, decision tree algorithms that exhibit this access pattern (e.g., CART [BFOS84]) can be implemented with the data management of Sprint. But other decision tree algorithms (e.g., ID3 [Qui86] or GID3 [CFIQ88]) that do not exhibit a sequential access pattern can not be scaled using this approach.

4 Algorithms

In this section, we present algorithms for two of the three cases listed above. The first three algorithms, RF-Write, RF-Read and RF-Hybrid, require that the AVC-group of the root node r (and thus the AVC-group of each individual node in the tree) fits into main memory; we assume that this is the most common case, as discussed in Section 3. The remaining algorithm, Algorithm RF-Vertical, works in the case that each single AVC-set of r fits in-memory, but the complete AVC-group of r does not fit. Since scalability and splitting criterion selection are orthogonal in the RainForest Schema, we do not dwell on any issues dealing with the quality of the resulting decision tree.

In order to describe the following algorithms precisely, we introduce the notion of the *state* of a node; possible states are Send, Fill, FillWrite, Write, Undecided, and Dead. The state S of a node n determines how a tuple is processed at n. A list of the states and the preconditions and processing behavior is shown in Figure 2. Whenever a node is created, its state is set to Undecided (unless mentioned otherwise), and we will call such a node a *new* node. A node whose state is Dead will be called a *dead* node.

4.1 Algorithm RF-Write

For Algorithm RF-Write, we assume that the AVC-group of the root node r fits into main memory. Algorithm RF-Write works as follows: We make one scan over the database and construct the AVC-group of r. Algorithm $\mathcal{CL}$ is applied and k children of r are created. An additional scan over the database is made, where each tuple t is written into one of the k partitions. The algorithm then recurses in turn on each partition. In the remainder of this paragraph, we describe Algorithm RF-Write in more detail.

At the beginning, the state of r is set to Fill and one scan over the database D is made. Since r is in state Fill, its AVC-group is constructed during the scan. Algorithm $\mathcal{CL}$ is called with the AVC-group of r as argument and computes crit(r). Assume that $\mathcal{CL}$ splits on attribute a into k partitions. Algorithm RF-Write allocates k children nodes of r, sets the state of r to Send, the state of each child to Write, and makes one additional pass over D. Each tuple t that is read from D is processed by the tree: Since r is in state Send, crit(r) is applied to t and t is sent to a child c_t. Since node c_t is in state Write, t is appended to c_t's partition. After the scan, the partition of each child node c_t consists of $F(c_t)$. The algorithm is then applied on each partition recursively.

For each level of the tree, Algorithm RF-Write reads the entire database twice and writes the entire database once.[2]

4.2 Algorithm RF-Read

The basic idea behind Algorithm RF-Read is to always read the original database instead of writing paritions for the children nodes. Since at some point all AVC-groups of the new nodes will not fit into main memory, we will read the original database many times, each time constructing

[2] This simple analysis assumes that the tree is balanced. More precisely, at a level l, only those tuples that belong to families of nodes at level l are read twice and written once. Since there might be dead nodes in the tree, the set of tuples processed at level l does not necessarily constitute the whole input database.

AVC-groups for an unexamined subset of the new nodes in the tree.

More precisely, in the first step of Algorithm RF-Read, the state of the root node r is set to to `Fill`, one scan over the database D is made, and `crit`(r) is computed. The children nodes $\{c_1, c_2, \ldots, c_k\}$ of r are created. Suppose that at this point there is enough main memory to hold the AVC-groups of all children nodes $\{c_1, c_2, \ldots, c_k\}$ of r in-memory. (We will address the problem of size estimation of the AVC-groups in Section 4.5.) In this case, there is no need to write out partitions for the c_i's as in Algorithm RF-Write. Instead, we can in another scan over D construct the AVC-groups of all children simultaneously: We set the state of r to `Send`, change the state of each newly allocated child c_i from `Undecided` to `Fill`, and build in a second scan over D the AVC-groups of the nodes $c_1, \ldots, c_k$ simultaneously in main memory. After the scan of D, Algorithm $\mathcal{CL}$ is applied to the in-memory AVC-group of each child node c_i to decide `crit`(c_i). If c_i splits, its children nodes are allocated and its state is set to `Send`; otherwise c_i's state is set to `Dead`. Note that so far we have made only two scans over the original database to construct the first two levels of the tree.

We can proceed in the same way for each level of the tree, as long as there is sufficient main memory available to hold the AVC-groups of all new nodes N at the level. Suppose that we arrive at a level L where there is not sufficient memory to hold all AVC-groups of the new nodes in-memory. In this case, we can divide the set of new nodes N into groups $G_1, \ldots, G_{g_L}$, $\cup G_i = N$, $G_i \cap G_j = \emptyset$ for $i \neq j$, such that the all AVC-groups of the nodes in a given group G_i fit in-memory. Each group is then processed individually: the states of the nodes in G_i are changed from `Undecided` to `Fill` and one scan over the database is made to construct their AVC-groups; after the scan, their splitting criteria are computed. Once all g_L groups for level L have been processed, we proceed to the next level of the tree. Note that for level L, g_L scans over the database D were necessary.

With increasing L, usually the number of nodes at a level L of the tree and thus usually the overall main memory requirements of the collective AVC-groups of the nodes at that level grow. Thus, Algorithm RF-Read makes an increasing number of scans over the database per level of the tree. Therefore it is not efficient for splitting algorithms that apply bottom-up pruning (except for the case that the families at the pure leaf nodes are very large — and this is usually not known in advance). But for splitting algorithms that prune the tree top-down [Fay91, RS98], this approach might be a viable solution.

We included Algorithm RF-Read for completeness: it marks one end of the design spectrum in the RainForest framework and it is one of the two parents of the Algorithm RF-Hybrid described in the next section. We do not think that it is very important in practice due to its restrictions in usability.

4.3 Algorithm RF-Hybrid

Combining Algorithm RF-Write and Algorithm RF-Read gives rise to Algorithm RF-Hybrid. We first describe a simple form of RF-Hybrid; in the next paragraph we will refine this version further. RF-Hybrid proceeds exactly like RF-Read until a tree level L is reached where all AVC-groups of the new nodes N together do not fit any more in main memory. At this point, RF-Hybrid switches to RF-Write: Algorithm RF-Hybrid creates m partitions and makes a scan over the database D to partition D over the m partitions. The algorithm then recurses on each node $n \in N$ and to complete the subtree rooted at n. This first version of RF-Hybrid uses the available memory more efficiently than RF-Write and does not require an increasing number of scans over the database for lower levels of the tree as does RF-Read.

We can improve upon this simple version of Algorithm RF-Hybrid using the following observation: Assume that we arrive at tree level L where all AVC-groups of the new nodes N together do not fit any more in main memory. Algorithm RF-Hybrid switches from RF-Read to RF-Write, but during this partitioning pass, we do not make use of the available main memory. (Each tuple is read, processed by the tree and written to a partition — no new information concerning the structure of the tree is gained during this pass.) We exploit this observation as follows: We select a set $M \subset N$ for which we construct AVC-groups in main memory while writing the partitions for the nodes in N. After the partitioning pass, Algorithm $\mathcal{CL}$ is applied to the in-memory AVC-groups of the nodes in M and their splitting criteria are computed.

The concurrent construction of AVC-groups for the nodes in M has the following advantage. Let $n \in M$ be a node whose AVC-group has been constructed, and consider the recursion of Algorithm RF-Hybrid on n. Since `crit`(n) is already known, we saved the first scan over n's partition: we can immediately proceed to the second scan during which we construct AVC-groups for the children of n. Thus, due to the concurrent construction of the AVC-groups of the nodes in M, we save for each node $n \in M$ one scan over n's partition.

How do we choose $M \subset N$? Since we save for each node $n \in M$ one scan over n's partition, we would like to maximize the sum of the sizes of the families of the nodes in M. The restricting factor is the size of main memory: For each node $n \in M$ we have to maintain its AVC-group in main memory. We can formulate the problem as follows: Each node $n \in M$ has an associated *benefit* (the size of $F(n)$) and an associated *cost* (the size of its AVC-group which has to be maintained in main memory).

Assume for now that we have estimates of the sizes of the AVC-groups of all nodes in N. (We will address the problem of size estimation of AVC-groups in Section 4.5.) According to the formulation in the preceding paragraph, the choice of M is an instance of the *knapsack problem* [GJ79]. An instance of the knapsack problem consists of a knapsack capacity and a set of items where each item

State of n	Precondition	Processing Behavior of a tuple t
`Fill`	n is root node, or parent of n is in state `Send`	The AVC-group at n is updated and the projection of t onto P_{large} and the class label is written to a file
`FillWrite`	n is root node, or parent of n is in state `Send`	The AVC-group at n is updated and t is appended to n's partition; the projection of t onto $Plarge$ and the class label is written to a temporary file F

Figure 3: Description of States with Modified Processing Behavior in Algorithm RF-Vertical

has an associated cost and benefit. The goal is to find a subset of the items such that the total cost of the subset does not exceed the capacity of the knapsack while maximizing the sum of the benefits of the items in the knapsack. The knapsack problem is well-known to be NP-complete [GJ79]. We decided to use a modified greedy approximation which finds a packing that has at least half the benefit of the optimal packing and works well in practice. (We call the greedy algorithm modified, because it considers the item with the largest benefit separately; this special case is necessary to get the stated bound with respect to the optimal solution.) The output of the Greedy Algorithm is the subset M of the new nodes N such that: (i) We can afford to construct the AVC-groups of M in-memory, and (ii) The benefit (the number of saved I/O's) is maximized.

Note that the Greedy Algorithm only addresses the problem of selecting an optimal set M of new nodes N for which we can construct AVC sets in-memory. As an extension of Algorithm RF-Hybrid, we could consider writing partitions for only the nodes in $N \setminus M$; we will consider this extension in further research.

4.4 Algorithm RF-Vertical

Algorithm RF-Vertical is designed for the case that the AVC-group of the root node r does not fit in main memory, but each individual AVC-set of r fits in-memory. For the presentation of RF-Vertical, we assume without loss of generality that there are predictor attributes $P_{large} = \{a_1, \ldots, a_v\}$ with very large AVC-sets such that each individual AVC-set fits in main memory, but no two AVC-sets of attributes in P_{large} fit in-memory. We denote the remaining predictor attributes by $P_{small} = \{a_{v+1}, \ldots, a_m\}$; the class label is attribute c. We limited the presentation to this special scenario for the ease of explanation; our discussion can easily be extended to the general case.

RF-Vertical proceeds similar to RF-Hybrid, but we process predictor attributes $a \in P_{large}$ in a special way: For each node, we write a temporary file Z from which we can reconstruct the AVC-sets of the attributes in P_{large}. After "normal" processing of the attributes $a \notin P_{small}$ has completed, Z is read v times and for each $a \in P_{large}$ its AVC-set is constructed in turn.

Let n be a node in the tree and let t be a tuple from the database D. In Algorithm RF-Vertical, the processing of a tuple t at a node n has slightly changed for some states of n. Assume that n is in state `Fill`. Since we can not afford to construct n's complete AVC-group in main memory, we only construct the attribute lists for predictor attributes P_{small} in-memory. For predictor attributes in P_{large}, we write a temporary file Z_n, into which we insert t's projection onto P_{large} and the class label. Thus, Z_n has the schema $\langle a_1, a_2, \ldots, a_v, c\rangle$. After the scan over D is completed, Algorithm $\mathcal{CL}$ is applied to the in-memory AVC-groups of the attribute in P_{small}. Algorithm $\mathcal{CL}$ can not yet compute the final splitting criterion (i.e., procedure $\mathcal{CL}$.`decide_splitting_criterion` can not be called yet), since the AVC-sets of the attributes $a \in P_{large}$ have not yet been examined. Therefore, for each predictor attribute $a \in P_{large}$, we make one scan over Z_n, construct the AVC-set for a and call procedure $\mathcal{CL}$.`find_best_partitioning` on the AVC-set. After all v attributes have been examined, we call procedure $\mathcal{CL}$.`decide_splitting_criterion` to compute the final splitting criterion for node n. This slightly modified processing behavior of a node for states `Fill` and `FillWrite` has been summarized in Figure 3.

In the description above, we concentrated on one possibility to construct the AVC-set of the predictor attributes P_{large}. In general, there are several possibilities for preparing the construction of the AVC-sets of the predictor attributes P_{large} at a node n. The complete set of options is given in the full paper.

4.5 AVC-Group Size Estimation

To estimate the size of the AVC-group of new node n, note that we can not assume that n's AVC-group is much smaller than the AVC-group of its parent p even though $F(p)$ might be considerably larger than $F(n)$. We estimate the size of the AVC-group of a new node n in a very conservative way: We estimate it to be the same size as its parent p — except for the AVC-set of the splitting attribute a. (If parent p of node n splits on a we know the size of a's AVC-set at node n exactly). Even though this approach usually overestimates the sizes of AVC-groups, it worked very well in practice. There are algorithms for the estimation of the number of distinct values of an attribute ([ASW87, HNSS95]); we intend to explore their use in future research.

5 Experimental results

In the machine learning and statistics literature, the two main performance measures for classification tree algorithms are: (i) The quality of the rules of the resulting tree, and (ii) The decision tree construction time [LLS97]. The generic schema described in Section 3 allows the instantiation of most (to our knowledge, all) classification tree algorithms from the literature *without modifying the result of the algorithm*. Thus, quality is an orthogonal issue in our framework, and we can concentrate solely on decision tree

Predictor Attribute	Distribution	Maximum number of entries
Salary	U(20000,150000)	130001
Commission	If salary $>$ 75k, then commission = 0 else U(10000,75000)	65001
Age	U(20,80)	61
Education Level	U(0,4)	5
Car	U(1,20)	20
ZipCode	Uniformly chosen from nine zipcodes	9
House Value	U($0.5 \cdot k \cdot 100000$, $1.5 \cdot k \cdot 100000$) where k depends on ZipCode	1350001
Home Years	U(1,30)	30
Loan	U(0,500000)	500001
Overall size of the AVC-group of the root		2045129

Figure 4: The sizes of the AVC-sets of `Generator` in [AIS93]

construction time. In the remainder of this section we study the performance of the techniques that enable classification algorithms to be made scalable.

5.1 Datasets and Methodology

The gap between the scalability requirements of real-life data mining applications and the sizes of datasets considered in the literature is especially visible when looking for possible benchmark datasets to evaluate scalability results. The largest dataset in the often used Statlog collection of training databases [MST94] contains only 57000 records, and the largest training dataset considered in [LLS97] has 4435 tuples. We therefore use the synthetic data generator introduced by Agrawal et al. in [AIS93], henceforth referred to as `Generator`. The synthetic data has nine predictor attributes as shown in Table 4. Included in the generator are classification functions that assign labels to the records produced. We selected two of the functions (Function 1 and Function 7) from [AIS93] for our performance study. Function 1 generates relatively small decision tree whereas the trees generated by Function 7 are large. (Note that this adheres to the methodology used in the Sprint performance study [SAM96]).

Since the feasibility of our framework relies on the size of the initial AVC-group, we examined the sizes of the AVC-group of the training data sets generated by `Generator`. The overall maximum number of entries in the AVC-group of the root node is about 2.1 million, requiring a maximum memory size of about 17 MB. If we partition the predictor attribute *house value* vertically, the main memory requirements to hold the AVC-groups of the root node in main memory are reduced to about 11 MB (1.35 million entries). The maximal AVC-set sizes of each predictor attribute are displayed in Table 4. The function $U(x, y)$ denotes the integer uniform distribution with values $v : x \leq v \leq y$. Since we will change the memory available to the RainForest algorithms during our experiments, let us call the number of AVC-set entries that fit in-memory the *buffer size*. So in order to run RF-Write on the datasets generated by `Generator`, we need a buffer size of at least 2.1 million entries, whereas RF-Vertical can be run with a buffer size of 1.35 million entries. All our experiments were performed on a Pentium Pro with a 200 Mhz processor running Solaris X86 version 2.5.1 with 128 MB of main memory. All algorithms are written in C++ and were compiled using `gcc` version 2.7.2.1 with the `-O3` compilation option.

We are interested in the behavior of the RainForest algorithms for datasets that are larger than main memory, therefore we uniformly stopped tree construction for leaf nodes whose family was smaller than 10000 tuples; any clever implementation would switch to a main memory algorithm at a node n whenever $F(n)$ fits into main memory.

5.2 Scalability results

First, we examined the performance of Algorithms RF-Write, RF-Hybrid and RF-Vertical as the size of the input database increases. For Algorithms RF-Write and RF-Hybrid, we fixed the size of the AVC-group buffer to 2.5 million entries; for Algorithm RF-Vertical we fixed the size of the AVC-group buffer to 1.8 million entries. Figures 5 and 6 show the overall running time of the algorithms as the number of tuples in the input database increases from 1000000 to 5000000. (Function 7 constructs very large decision trees and thus tree growth takes much longer than for Function 1). The running time of all algorithms grows nearly linearly with the number of tuples. Algorithm RF-Hybrid outperforms both Algorithms RF-Write and RF-Vertical in terms of running time; the difference is much more pronounced for Function 7. Figures 7 and 8 show the number of page accesses during tree construction (assuming a pagesize of 32 KB).

In the next four experiments, we investigated how internal properties of the AVC-groups of the training database influence performance. (We expected that only the size of the input database and the buffer size matter which the experiments confirm.) We fixed the size of the input database to 2000000 tuples and the classification function to Function 1. Figure 9 shows the effect of an increase in the absolute size of the AVC-group in the input database while holding the available buffer sizes constant at 2.5 million entries for RF-Write and RF-Hybrid and at 1.8 million entries for RF-Vertical: We varied the size of the AVC-group through manipulation of the data generator from 200000 entries (20% of the original size) to 2000000 entries (original size). For small AVC-group sizes (40% and below), the times for RF-Vertical and RF-Hybrid are identical. The larger buffer size only shows its effect for larger AVC-group-sizes: RF-Hybrid writes partitions less frequently than RF-Vertical. The running time of RF-Write is not affected through a change in AVC-group size, since RF-Write writes partitions regardless of the amount of memory available. Figure 10 shows the effect of an increase in the absolute size of the AVC-group in the input database while varying the buffer sizes. The buffer size for RF-Write and RF-Hybrid is set such that exactly the AVC-group of the root node fits in-memory; the buffer size of RF-Vertical is set such that exactly the largest AVC-set of the root node fits in-memory. Since both AVC-group size and buffer size are increased simultaneously (keeping their ratio constant), the running times stay constant.

Figure 11 shows how the effect of skew between two attributes within an AVC-group affects performance. The number of tuples remained constant at 2000000; we set the buffer sizes for RF-Write and RF-Hybrid to 250000, and the buffer size for RF-Vertical to 1800000. We duplicated the `loan` attribute (thus increasing the number of attributes to ten), but skewed the distribution of distinct attributes values between the two loan attributes. We reduced the number of attribute values of the remaining attributes to make the loan attributes the dominant contributors to the overall AVC-group size. During the skew, we held the overall number of distinct attribute values for the two loan attributes at a combined size of 1200000 entries. For example, a skew value of 0.1 indicates that the first loan attribute had 10% (120000) distinct attribute values and the second loan attribute had 90% (1080000) distinct values. As we expected, the overall running time is not influenced by the skew, since the overall AVC-group size remained constant.

In our last experiment shown in Figure 12, we added extra attributes with random values to the records in the input database, while holding the overall number of entries constant at 4200000 for RF-Hybrid and RF-Write and at 2100000 entries for RF-Vertical. Adding attributes increases tree construction time since the additional attributes need to be processed, but does not change the final decision tree. (The splitting algorithm will never choose such a "noisy" attribute in its splitting criterion.) As can be seen in Figure 12, the RainForest family of algorithms exhibits a roughly linear scaleup with the number of attributes.

5.3 Performance comparison with Sprint

In this section, we present a performance comparison with Sprint [SAM96]. We tried to make our implementation of Sprint as efficient as possible, resulting in the following two implementation improvements over the algorithm described in [SAM96]. First, we create only one attribute list for all categorical attributes together. Second, when a node n splits into children nodes n_1 and n_2, we create the histograms for the categorical attributes of n_1 and n_2 during the distribution of the categorical attribute list, thus saving an additional scan. We made the in-memory hash-table large enough to perform each hash-join in one pass over an attribute list.

Figures 13 and 14 show the comparison of Sprint and the RainForest algorithms for Functions 1 and 7. For algorithms RF-Hybrid and RF-Write, we set the AVC buffer size to 2500000 entries (the AVC-group of the root fits in-memory); for RF-Vertical we set the buffer size such that the largest AVC-set of a single attribute of the root node fits in-memory. The figures show that for Function 1, RF-Hybrid and RF-Vertical outperform Sprint by a factor of about 5. Function 7 generates larger trees than function 1; thus the speedup factor is about 8.

Where does this speed-up come from? First, we compared the cost of the repeated in-memory sorting of AVC-groups in the RainForest algorithms with the cost of creation of attribute lists in Sprint through which repeated sorts can be avoided. The numbers in Figure 15 show that repeated in-memory sorting of the AVC-groups is about ten times faster then the initial attribute list creation time. Second, we compared the cost to arrive at a splitting criterion for a node n plus distribution of $F(n)$ among n's children. In Sprint, the splitting criterion is computed through a scan over all attribute lists; the distribution of $F(n)$ is performed through a hash-join of all attribute lists with the attribute list of the splitting attribute. In the RainForest family of algorithms, $F(n)$ is read twice and written once; RF-Vertical needs to write vertical partitions if necessary. We set the buffer size of RF-Write such that the AVC-group of the root fits in-memory and the buffer size of RF-Vertical such that the largest AVC-set fits in-memory. Figure 16 shows that the cost of determining the splitting criterion plus partitioning in the original database is about a factor of 5 faster than scanning and hash-joining the attribute lists. This cost is the overall dominant cost during tree construction and thus explains why the RainForest family of algorithms outperforms Sprint by a factor of 5.

6 Conclusions

In this paper, we have developed a comprehensive approach to scaling decision tree algorithms that is applicable to all decision tree algorithms that we are aware of. The key insight is the observation that decision trees in the literature base their splitting criteria at a tree node on the AVC-group for that node, which is relatively compact.

The best splitting criteria developed in statistics and machine learning can now be exploited for classification in a scalable manner. In addition, depending upon the available memory, our algorithms offer significant performance improvements over the Sprint classification algorithm, which is the fastest scalable classifier in the literature. If there is enough memory to hold individual AVC-sets, as is very likely, we obtain very good speed-up over Sprint; if there is enough memory to hold all AVC-sets for a node, the speed-up is even better.

Acknowledgements. We wish to thank Wei-Yin Loh for insightful discussions.

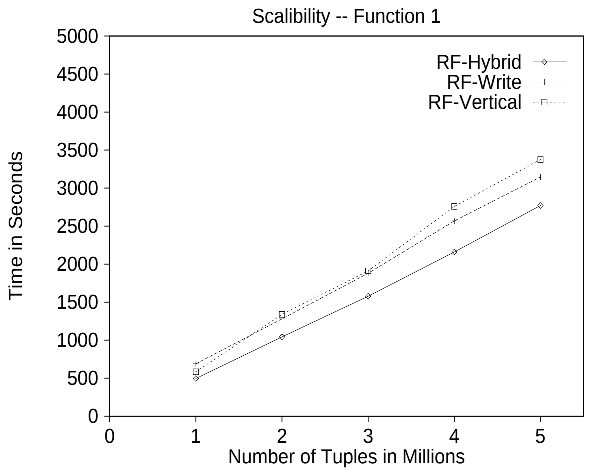

Figure 5: Scalability — F1: Overall Time

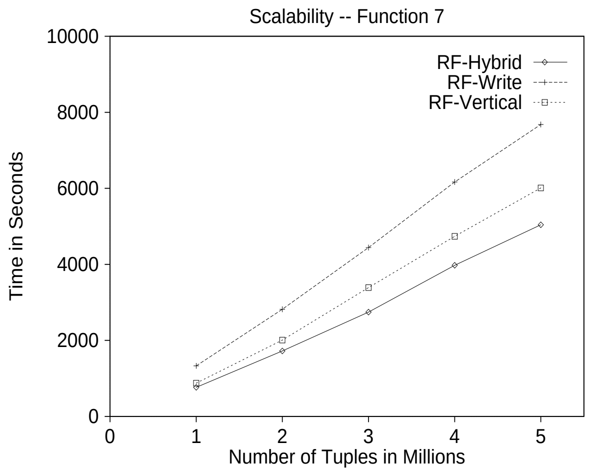

Figure 6: Scalability — F7: Overall Time

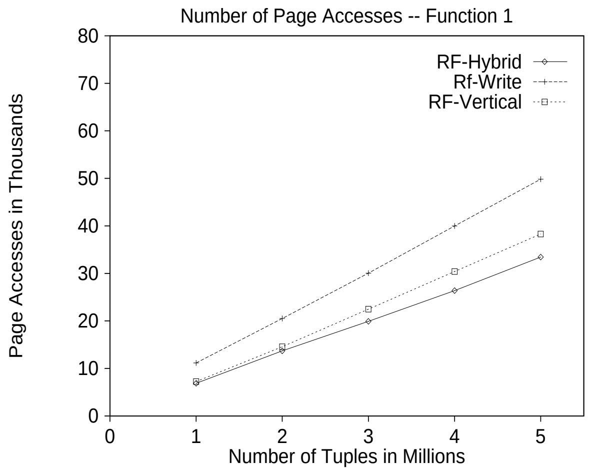

Figure 7: Scalability — F1: Page Accesses

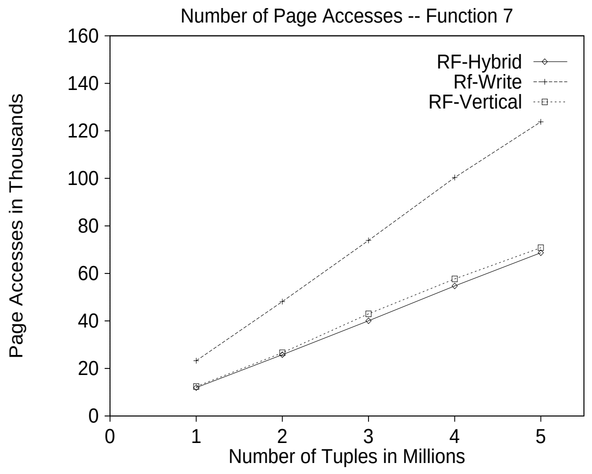

Figure 8: Scalability — F7: Page Accesses

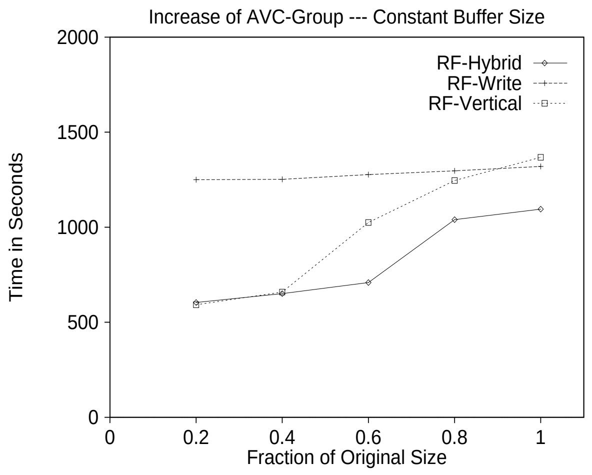

Figure 9: Changing AVC-group Size

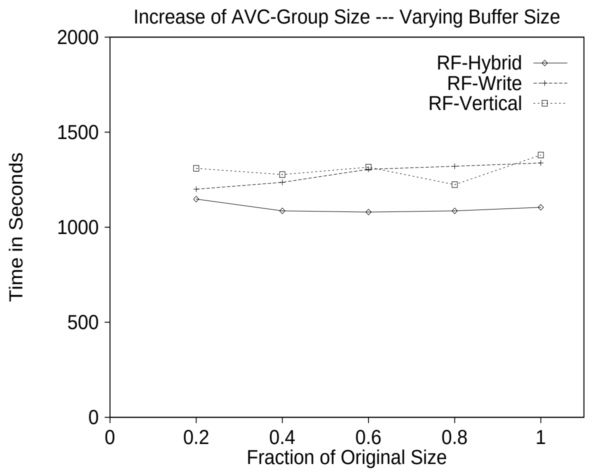

Figure 10: Changing AVC-group Size

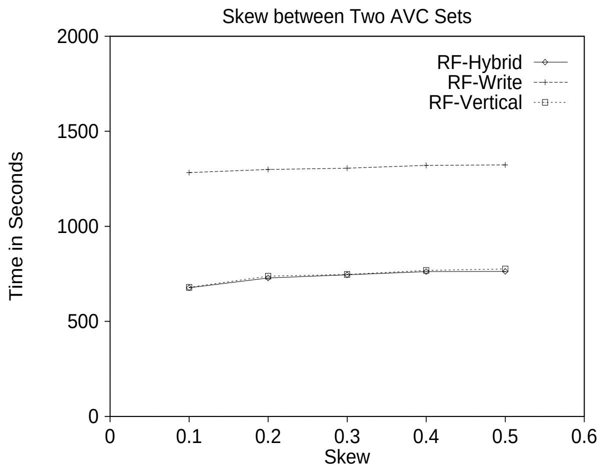

Figure 11: Changing AVC-group Skew

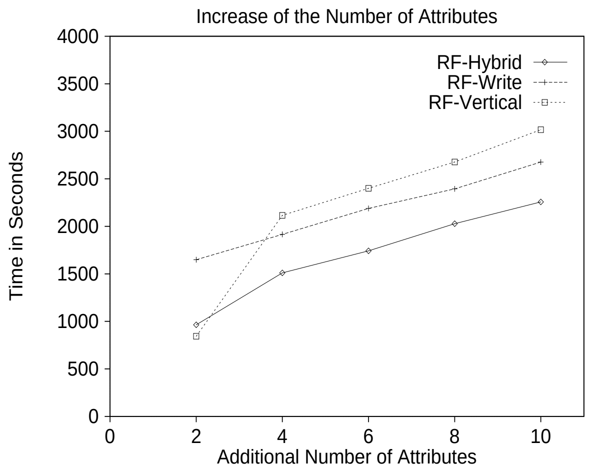

Figure 12: Changing Number of Attributes

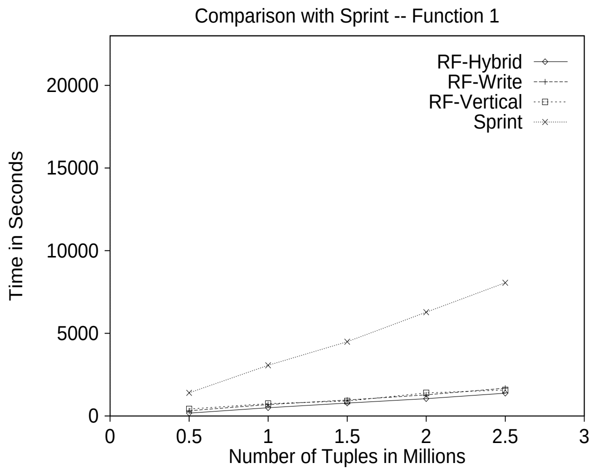

Figure 13: Comparison with Sprint — F1

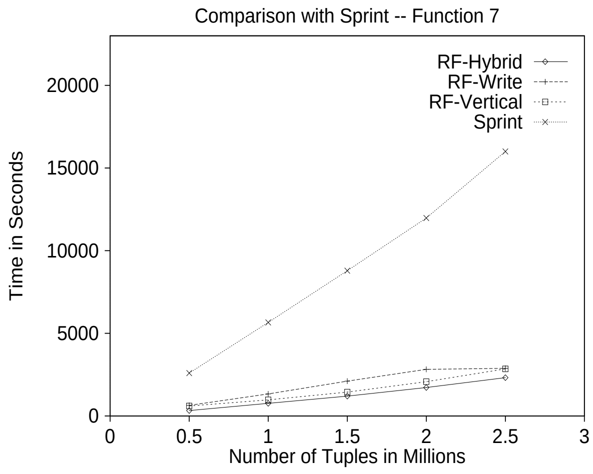

Figure 14: Comparison with Sprint — F7

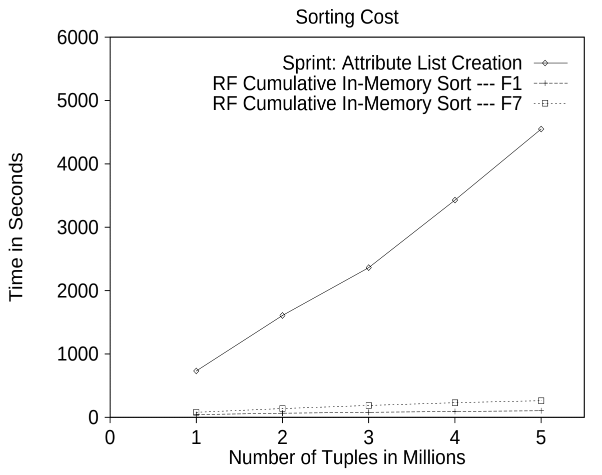

Figure 15: Sorting Cost Comparison

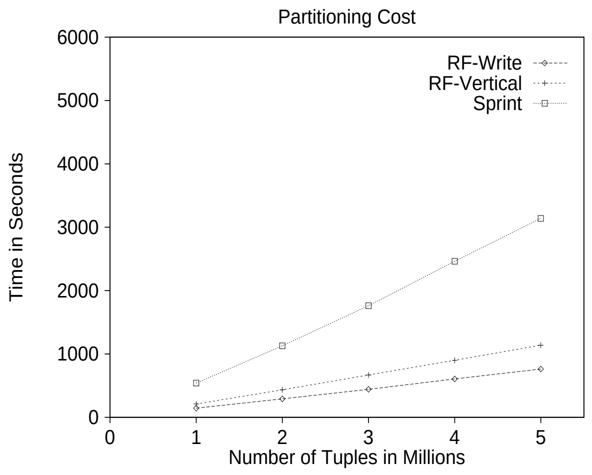

Figure 16: Partitioning Cost Comparison

References

[AGI$^+$92] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An interval classifier for database mining applications. In *Proc. of VLDB*, 1992.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE TKDE*, December 1993.

[ASW87] M.M. Astrahan, M. Schkolnick, and K.-Y. Whang. Approximating the number of unique values of an attribute without sorting. *Information Systems*, 12(1):11–15, 1987.

[BFOS84] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*.

[BU92] C.E. Brodley and P.E. Utgoff. Multivariate versus univariate decision trees. TR 8, Department of Computer Science, University of Massachussetts, 1992.

[Cat91] J. Catlett. *Megainduction: Machine Learning on Very Large Databases*. PhD thesis, University of Sydney, 1991.

[CFIQ88] J.. Cheng, U.M. Fayyad, K.B. Irani, and Z. Qian. Improved decision trees: A generalized version of ID3. In *Proc. of Machine Learning*, 1988.

[CM94] S.P. Curram and J. Mingers. Neural networks, decision tree induction and discriminant analysis: an empirical comparison. *Journal of the Operational Research Society*, 45:440–450, 1994.

[CS93a] P. K. Chan and S. J. Stolfo. Experiments on multistrategy learning by meta-learning. In *Proc. of CIKM*, 1993.

[CS93b] P. K. Chan and S. J. Stolfo. Meta-learning for multistrategy and parallel learning. In *Proc. Second Intl. Workshop on Multistrategy Learning*, 1993.

[DBP93] V. Corruble D.E. Brown and C.L. Pittard. A comparison of decision classifiers with backpropagation neural networks for multimodal classification problems. *Pattern Recognition*, 26:953–961, 1993.

[DKS95] J. Dougherty, R. Kahove, and M. Sahami. Supervised and unsupervised discretization of continous features. In *Proc. of Machine Learning*, 1995.

[Fay91] U.M. Fayyad. *On the induction of decision trees for multiple concept learning*. PhD thesis, EECS Department, The University of Michigan, 1991.

[FI93] U.M. Fayyad and K. Irani. Multi-interval discretization of continous-valued attributes for classification learning. In *Proc. of the International Joint Conference on Artificial Intelligence*, 1993.

[FMM96] T. Fukuda, Y. Morimoto, and S. Morishita. Constructing efficient decision trees by using optimized numeric association rules. In *Proc. of VLDB*, 1996.

[GJ79] M.R. Garey and D.S. Johnson. *Computer and Intractability*, 1979.

[Han97] D.J. Hand. *Construction and Assessment of Classification Rules*, 1997.

[HNSS95] P.J. Haas, J.F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proc. of VLDB*, 1995.

[LLS97] T.-S. Lim, W.-Y. Loh, and Y.-S. Shih. An empirical comparison of decision trees and other classification methods. TR 979, Department of Statistics, UW Madison, June 1997.

[LS97] W.-Y. Loh and Y.-S. Shih. Split selection methods for classification trees. *Statistica Sinica*, 7(4), October 1997.

[LV88] W.-Y. Loh and N. Vanichsetakul. Tree-structured classification via generalized disriminant analysis (with discussion). *Journal of the American Statistical Association*, 83:715–728, 1988.

[Maa94] W. Maass. Efficient agnostic pac-learning with simple hypothesis. In *Proc. of Conference on Computational Learning Theory*, 1994.

[Mag93] J. Magidson. The CHAID approach to segmentation modeling. In *Handbook of Marketing Research*, 1993.

[MAR96] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. of EDBT*, 1996.

[MRA95] M. Mehta, J. Rissanen, and R. Agrawal. MDL-based decision tree pruning. In *Proc. of KDD*, 1995.

[MST94] D. Michie, D.J. Spiegelhalter, and C.C. Taylor, editors. *Machine Learning, Neural and Statistical Classification*, 1994.

[Qui79] J.R. Quinlan. Discovering rules by induction from large collections of examples. In *Expert Systems in the Micro Electronic Age*, 1979.

[Qui83] J.R Quinlan. Learning efficient classification procedures. In *Machine Learning: An Artificial Intelligence Approach*, 1983.

[Qui86] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Qui93] J.R. Quinlan. *C4.5: Programs for Machine Learning*, 1993.

[RS98] R.Rastogi and K.Shim. PUBLIC: A Decision Tree Classifier that Integrates Pruning and Building. In *Proc. of VLDB*, 1998.

[Ris89] J. Rissanen. *Stochastic Complexity in Statistical Inquiry*, 1989.

[SAM96] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. of VLDB*, 1996.

[SMT91] J.W. Shavlik, R.J. Mooney, and G.G. Towell. Symbolic and neural learning algorithms: an empirical comparison. *Machine Learning*, 6:111–144, 1991.

[WK91] S.M. Weiss and C.A. Kulikowski. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems*, 1991.

[YFM$^+$98] Y.Morimoto, T.Fukuda, H.Matsuzawa, T.Tokuyama, and K.Yoda. Algorithms for Mining Association Rules for Binary Segmentations of Huge Categorical Databases. In *Proc. of VLDB*, 1998.