DATA MANAGEMENT TECHNIQUES FOR FAST FUNCTION APPROXIMATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Biswanath Panda

May 2009

DATA MANAGEMENT TECHNIQUES FOR FAST FUNCTION APPROXIMATION

Biswanath Panda, Ph.D.

Cornell University 2009

Computing and information technology is fundamentally changing the face of modern science. Traditional methods of performing scientific studies are now making way for the next generation of methods that use computing technology. However, the kinds of calculations that scientists wish to perform, and the ways in which they want to collect, archive and analyze information has posed several new challenges in data management and algorithm design. Achieving the goal of using computing technology effectively in scientific applications has become an important area of research in computer science, called eScience or data-driven science.

Most data-driven scientific applications are aimed at studying and understanding some real world physical phenomenon. The general methodology followed by a scientist is to first model the physical phenomenon either directly from the mathematical equations governing the phenomenon, or from a large dataset of observations about the phenomenon. Recent advances in data management, data mining and machine learning have addressed numerous challenges that arise in this first stage of model building. However, the state of the art methods are inadequate in addressing challenges that arise in the second stage of a data driven scientific study, where the scientist uses the model she has built to help her understand the physical phenomenon, using tools such as computer simulation and visualization.

This thesis identifies and addresses data management challenges that arise when a complex model built for a real world phenomenon is analyzed by a scientist to gain insights about the phenomenon. The first part of the thesis concentrates on high-dimensional function approximation (HFA), a problem relevant to virtually all applications that use computer simulation as the methodology for understanding complex models. We explore various aspects of HFA in depth, identify key data management problems, and propose solutions that significantly speedup long running scientific simulations. Besides computer simulation, visualizing low dimensional summaries of a complex model is another method commonly used by scientists to understand models. Most real world models are complex and involve thousands of attributes. In order to get a very good understanding of a model, a scientist generates a very large number of low dimensional summaries for the model. Generating large sets of summaries for a complex model presents a challenging data management task and the second part of the thesis develops scalable algorithms for solving this data management problem.

BIOGRAPHICAL SKETCH

Biswanath Panda hails from the steel city of India called Jamshedpur. He completed his under-graduate education in Computer Science and Engineering at the Indian Institute of Technology, Kharagpur in 2003. Soon after he moved to the United States of America to pursue his doctorate in Computer Science at Cornell University. Cornell stimulated his research interests in tackling challenging problems related to large scale data management. In collaboration with the Sibley School of Mechanical Engineering and the Cornell Lab of Ornithology, he worked on several data management problems that arose when scientists used complex models in their scientific studies. He received a special Masters in 2007 for his research on minimizing simulation time in long running scientific simulations. After graduation, Biswanath is taking up a full time engineering position with Google.

To my parents Anjan and Kasturi.

ACKNOWLEDGEMENTS

Five years ago when I started graduate school, I embarked on a long journey of learning and discovery. Today, as I near the end of this journey it gives me immense pleasure to write this note of appreciation thanking all the people who have helped and supported me along the way.

First and foremost, I would like to thank my adviser Johannes Gehrke, for his continuous support and mentoring throughout my Ph.D. Over the years I have learned a lot from Johannes. His emphasis on the overall development of a graduate student has not only helped me learn to conduct research, but has also honed my skills in writing, oral communication and teaching. In addition to Johannes, my committee member Mirek Riedewald was also instrumental in the development of the ideas that have made their way into this thesis. I am extremely grateful to him for being available at all times for lengthy technical discussions, and providing me with invaluable advice whenever I got stuck or seemed to have reached a dead end.

Other than Johannes and Mirek, I would like to take this opportunity to thank some other faculty members at Cornell whose guidance has been invaluable to me. I would like to thank Stephen Pope, from the Department of Mechanical and Aerospace Engineering for helping me understand the ISAT algorithm and other computational challenges in combustion simulations. Steve was very patient in helping me understand the nuances in scientific simulations and was a great sounding board for the different ideas that we had for improving ISAT. I would also like to thank my minor adviser David Weinbaum for helping me achieve my goal of learning finance and receiving a minor degree in business. I would also like to thank Ken Birman for being part of my committee and providing me with good feedback on my research. Ken also provided me

with the opportunity to instruct the operating systems practicum course which was a great learning experience for me.

I cannot understate the role played by my family and friends in making this dissertation possible. I would like to thank my parents to whom I have dedicated this thesis. In spite of being miles away in India, their love, support and encouragement has always been with me and has helped me tide over several difficult times in my life. Finally, I would like to thank all my friends at Cornell, who made my stay at Ithaca the most enriching and entertaining part of my life. While I refrain from listing their names for the sole fear of missing out a few, I am deeply indebted to all of them for helping me make Ithaca my new home.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INDEXING FOR FUNCTION APPROXIMATION

Simulation is one of the most powerful tools that scientists have at their disposal for studying and understanding real-world physical phenomena. In order to be realistic, the mathematical models which drive simulations are often very complex and run for a very large number of simulation steps. The required computational resources often make it infeasible to evaluate simulation models exactly at each step, and thus scientists trade accuracy for reduced simulation cost.

In this paper, we explore function approximation for a combustion simulation. In particular, we model high-dimensional function approximation (HFA) as a storage and retrieval problem, and we show that HFA defines a novel class of applications for high dimensional index structures. The interesting property of HFA is that it imposes a mixed query/update workload on the index which leads to novel tradeoffs between the efficiency of search versus updates. We investigate in detail one specific approach to HFA based on Taylor Series expansions and we analyze tradeoffs in index structure design through a thorough experimental study.

1.1 Introduction

Studying physical phenomena through computer simulation is an important method of scientific research. Application areas include studies of heat and

Reprinted, with permission, from B.Panda, M. Riedewald, S. B. Pope, J. Gehrke, L. P. Chew. "Indexing For Function Approximation". *Published in the proceedings of the* 32^{nd} *International Conference On Very Large Databases*.

mass transfer, fluid dynamics, combustion, evaporation and many more [69, 70]. The general methodology in these application areas is similar. Scientists first understand the physical laws that govern the observed phenomenon. These laws then drive a mathematical model that is used in simulations as an approximation of reality.

In practice scientists often face serious computational challenges. The more realistic the model, the more complex the corresponding mathematical equations. As an example, consider the simulation of a combustion process [67], the application that brought our group together. Simulation of combustion requires tracking the composition of gases in a combustion chamber and the change in their compositions over time. The transition from one composition to the next is defined by a complex high-dimensional *transition function*. Depending on the gases studied, a composition can be described using nine dimensions for a simple Hydrogen simulation and fifty or more dimensions for a Methane simulation. A single transition step (which is an evaluation of this function) can require millions of floating point operations. On a modern processor with the optimized code of the domain scientists, a simple Hydrogen simulation of low dimensionality would run for a few hours, while a Methane simulation with higher dimensionality would require several weeks. Simulations need to be run for many different gases and different configurations for each gas. This problem is not specific to combustion simulation, but representative of a large class of scientific simulations that require repeated evaluations of computationally expensive functions that govern a physical process [34, 49, 55].

To trade accuracy for simulation time, the domain scientists do not expend the resources to evaluate the function for each transition step, but instead use

cheaper *approximations*. The main approach to function approximation is to build a model $\hat{f}$ of the function. Given a query point $\mathbf{x}$ at which the function f must be evaluated, function approximation techniques return $\hat{f}(\mathbf{x})$ which approximates the true value $f(\mathbf{x})$ within a specified error tolerance. There are two main types of function approximations. Global approximation techniques use a single model to represent f. Local approximation techniques break the domain into regions, representing each region with a different model. It has been shown that local approximations work better for the class of simulations that are the focus of this paper [34, 45].

A local function approximation scheme poses two main challenges. First, we need to decide how to select appropriate regions in the domain of the function, such that f can be approximated well within each region by a model $\hat{f}$. Approaches based on the Taylor Series are generally accepted for approximating high-dimensional functions in this domain [48, 55, 67]. The second challenge is to efficiently store the regions such that given a query point $\mathbf{x}$, we can efficiently find the region responsible for $\mathbf{x}$ in order to calculate $\hat{f}(\mathbf{x})$. It is this storage and retrieval part that we think the database community can make significant contributions to. The domain scientists already took the first steps by developing the ISAT method [55] for indexing of regions. In collaboration with them, we are now studying the general problem of high-dimensional function approximation (HFA) and the design of efficient index structures for it. The workload imposed on indexing structures by HFA is very different from the workloads studied so far in the literature [7, 29].

In summary, this paper makes the following contributions:

- We introduce the novel problem of high-dimensional function approxi-

mation as an application of high-dimensional indexing, and we describe an algorithmic framework that abstracts the salient elements of an HFA application (Sections 1.2 and 1.3).

- We give an analysis of the index design tradeoffs that this problem poses, and we identify their effects on the overall performance of HFA (Section 1.4).
- We perform a thorough evaluation of a set of candidate index structures from the database community for this application (Section 1.5).

Section 1.6 discusses related work and Section 1.7 concludes the paper.

We would like to emphasize that this paper presents an exciting new direction of research into high-dimensional indexing, and it is this connection between an active application area and the database community that we believe is one of the major contributions of this paper.

1.2 Problem Formulation

We first introduce the basic problem of high-dimensional function approximation and then show in Section 1.3 how it leads to a challenging indexing problem. We start with our case study, an example of a typical scientific application that uses HFA to improve the running time of simulations. We then formally define the resulting HFA problem.

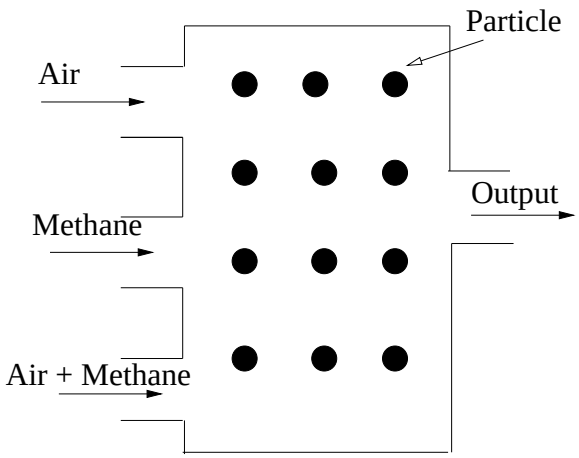

Figure 1.1: Combustion Chamber

1.2.1 Simulating Combustion

The application simulates the combustion of a hydrocarbon in a reaction chamber. The chamber has three inflows (air, the hydrocarbon being studied — Methane in the diagram, and a mixture of air and hydrocarbon) and a single outflow (see Figure 1.1). The gases flow into the chamber at different rates which are input parameters to the simulation. The simulation starts with a user-specified number of particles in the chamber. Each particle p has a user-specified chemical composition, which is described by its thermochemical composition vector $\phi^p(t) = \langle Y_1^p(t), Y_2^p(t), \ldots, Y_{\mathrm{s}}^p(t), h \rangle$, where s is the number of chemical species in p, $Y_i^p(t)$ is the mass fraction of chemical species i in particle p at time t of the simulation, and h (which is a constant) is the enthalpy of the particle [67].

Each simulation step consists of the following three phases:

1. **Inflow-Outflow:** Some of the particles in the reactor leave through the outflow and the same number of new particles enter from the inflows in ratios proportional to the rates of the inflow.

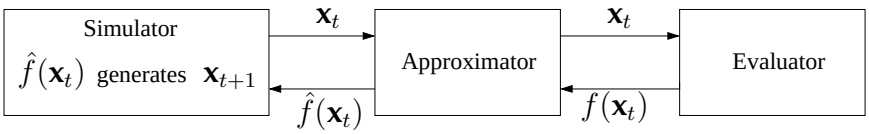

Figure 1.2: Application Model

2. **Mixing:** Particles in the reactor mix with each other, generating new particle compositions.

3. **Reaction:** The particle compositions evolve due to reaction and the new compositions must be calculated for all particles.

The computationally expensive part is the reaction step, which for a particle p is described by a reaction function f that maps one thermochemical composition to the next composition. Moreover, typical simulations require 10^8 to 10^{10} reaction function evaluations. These factors can cause simulations to run for years if the function value is calculated at each step. Thus in practice, the domain scientists accept approximations to f in order to be able to run large, complex simulations.

1.2.2 Application Model

Combustion simulation is representative of the class of applications that our methods apply to in general. Figure 1.2 shows the general framework. There is an application simulating some mathematical model, which we call the simulator. The simulator generates query points at which the value of a function f is required. These function values are used by the simulator to generate future query points. The application queries a function approximator for the func-

tion values. The approximator can calculate the exact value[1] using the function evaluator, which is an expensive operation, or it can use some algorithm to return approximate values within a user specified error tolerance. Typically, the approximator has limited knowledge about f, e.g., only previously calculated function values.

1.2.3 Problem Definition

Let us now define the function approximation problem for the above application model. We start by defining an ε-approximation of a function value.

Definition 1 *Let $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$ be a function, let $\mathbf{x} \in \mathbb{R}^m$ and let $\varepsilon \in \mathbb{R}$. We say that $\hat{f}(\mathbf{x}) \in \mathbb{R}^n$ is an* ε-approximation *of $f(\mathbf{x})$ at $\mathbf{x}$ if $\| \hat{f}(\mathbf{x}) - f(\mathbf{x}) \| < \varepsilon$.*

We can now formalize the function approximation problem as a game between two players, the simulator (application) and the function approximator. In the first round of the game, the simulator produces query point $\mathbf{x}_1$, and the function approximator computes $\hat{f}(\mathbf{x}_1)$ at computational cost c_1, where $\hat{f}$ is an ε-approximation to f. In the next round, the simulator takes $\hat{f}(\mathbf{x}_1)$ and computes $\mathbf{x}_2$, the function approximator generates $\hat{f}(\mathbf{x}_2)$ at cost c_2 and so on. Note that in general $\mathbf{x}_{i+1}$, among other things depends on $\hat{f}(\mathbf{x}_i)$, $i = 1, \ldots, n-1$. The game stops after n rounds. The goal of function approximation is, for a given ε, to minimize the total cost, $\min \sum_{i=1}^{n} c_i$.

In this paper we study approximators that attempt to minimize the total cost by partitioning the input domain into local regions and modeling each region

[1] Up to the accuracy of the evaluator. In practice the evaluator is a differential equation solver that introduces some error as well.

with some $\hat{f}$. The motivation is that the local regions enable cheaper approximate computation. We make this notion of local region more formal in the next definition.

Definition 2 *An* ε-Local Region $R_{f,\hat{f}}(\mathbf{x}, \varepsilon) \subseteq \mathbb{R}^m$ *for function* f *based on approximation* $\hat{f}$ *at point* $\mathbf{x}$ *is a maximal connected region containing* $\mathbf{x} \in \mathbb{R}^m$ *such that* $\forall \mathbf{x}' \in R_{f,\hat{f}}(\mathbf{x}, \varepsilon) : \hat{f}(\mathbf{x}')$ *is an* ε*-approximation of* $f(\mathbf{x}')$.

As a shortcut, we will often refer to an ε-Local Region $R_{f,\hat{f}}(\mathbf{x}, \varepsilon)$ for function f based on approximation $\hat{f}$ at point $\mathbf{x}$ as *Local Region* when the parameters are clear from the context.

There is a cost associated with finding the Local Region around a point. At the very least a function evaluation is required for the "center" of the region, together with additional computation for determining the extent of the region. Assuming that this cost is approximately the same for all regions, we can minimize total function approximation cost by finding the smallest set of Local Regions that covers all query points. The hardness of this problem is analyzed in the next section.

1.2.4 Analysis

In this section we show that the function approximation problem is hard. This applies to both its offline and its online formulation. We show hardness for an easier version of the problem, where the Local Region of $\mathbf{x}$ is obtained for free when $f(\mathbf{x})$ is computed. The more general case, i.e., where determining

the extent of the Local Region around $\mathbf{x}$ has some cost assigned to it as well, is therefore at least as hard.

Offline problem: Given a set $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ of query points, find the smallest set $L = \{l_1, \ldots, l_k\}$ of Local Regions in the data space (not limited to Local Regions around the query points), such that for each $\mathbf{x}_i \in X$ there is an $l_j \in L$, which contains $\mathbf{x}_i$.

If the Local Regions are constrained to be hyper-spheres, we can show by reduction from Geometric Covering By Discs [39], that the offline problem is *NP-complete*. This implies that the more general formulation above is at least as hard.

Using a similar reduction, we can show the same hardness results even for a restricted version of the offline problem. In this restricted version, we constrain L to be a subset of $\{R_{f,\hat{f}}(x_1, \varepsilon), \ldots, R_{f,\hat{f}}(x_n, \varepsilon)\}$, i.e., we can only choose from the Local Regions around query points.

In practice the algorithm for function approximation does not know the query points in advance. It has to solve an *online* problem, where query points are presented one-by-one.

Online problem: For $i > 0$ let $X(i) = \{\mathbf{x}_1, \ldots, \mathbf{x}_i\}$ and $L(i) = \{l_1, \ldots, l_{k(i)}\}$, where $k(i)$ is some integer with $k(i-1) \leq k(i)$ for all $i > 1$. Find the smallest set $L(n)$, such that the following holds for each set $X(i)$: Each $\mathbf{x} \in X(i)$ is contained in some Local Region $l \in L(i)$.

Intuitively the set $X(i)$ contains the query points seen until time i, and $L(i)$ contains the Local Regions that have been materialized until time i. To be able

to compute the function for query point $\mathbf{x}_i$, $\mathbf{x}_i$ has to be contained in one of the Local Regions that are available at time i. If no such Local Region exists, it has to be inserted into $L(i)$.

A standard performance measure for online algorithms is the *competitive ratio* [8]. It measures the cost of the online algorithm for an input sequence $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ relative to an optimal offline algorithm, which knows the whole input in advance. The competitive ratio is defined as the worst (i.e., highest) ratio over all possible inputs of length n. If this ratio is independent of n, i.e., bounded by a constant c, then the online algorithm is c-competitive. E.g., if the online algorithm never materializes more than twice as many Local Regions as the optimal offline algorithm, then it is 2-competitive. Notice that it is often possible to analyze competitiveness even without knowing the optimal algorithm [8].

We can show that *there exists no deterministic online algorithm that is competitive*. Due to space constraints we only sketch the proof. The adversary can always construct a function (even if we are restricted to smooth functions), such that the online algorithm has to materialize a new Local Region for every query $\mathbf{x}_i$, while the offline algorithm can pick a single Local Region that contains all query points.

Figure 1.3 illustrates the construction. The figure shows a one-dimensional function where linear interpolations are used as $\hat{f}$, i.e., $\hat{f}$ for a Local Region is $\hat{f}(\mathbf{a}) = f(\mathbf{x}) + s_{\mathbf{x}} \times (\mathbf{x} - \mathbf{a})$, where $s_{\mathbf{x}}$ is the derivative of f at $\mathbf{x}$. Note that the Local Regions in this case are intervals. For $q_1, \ldots, q_4$, the Local Regions around them do not contain any other q_i, but the Local Region of q_5 covers the whole domain. Since the online algorithm is deterministic, the adversary can always select a function such that all Local Regions computed by the online algorithm

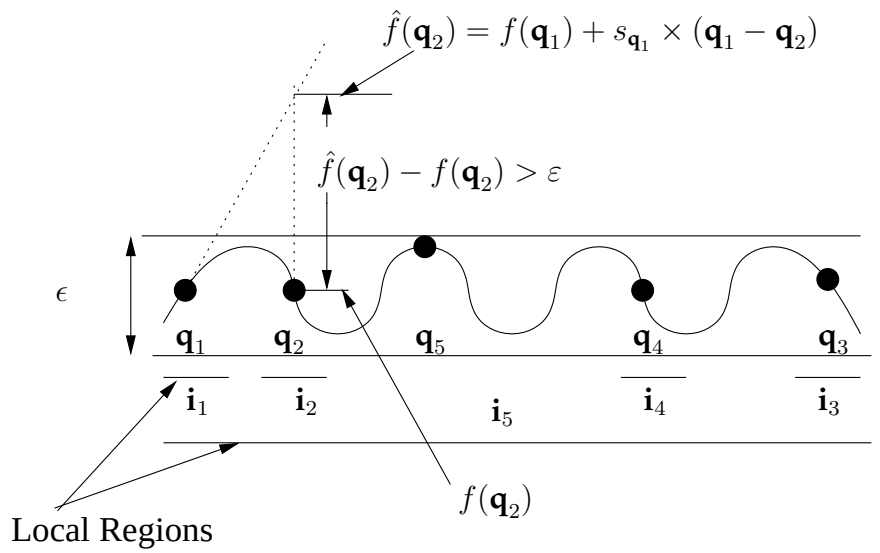

Figure 1.3: Example For Non-competitiveness

are around points like $q_1, \ldots, q_4$, while the offline algorithm can choose an optimal point like q_5. This construction even works for the more restricted online problem, where materialized Local Regions have to be selected from the Local Regions defined by the query points. The adversary constructs the same function and input; it simply selects a point like q_5 as the *last* query point.

The proof of non-competitiveness might appear contrived. However, without knowing the nature of the function it is impossible to rule out the case that there are "large" Local Regions that contain many "small" Local Regions. We are currently exploring what properties of a function could be taken advantage of to obtain better competitiveness.

Algorithm 1: Framework For Function Approximation

Require: Query Point $\mathbf{x}$, Index Structure S

1: **if** $\exists \langle R_{f,\hat{f}}(\mathbf{x'}, \varepsilon), \hat{f} \rangle \in S$ such that $\mathbf{x} \in R_{f,\hat{f}}(\mathbf{x'}, \varepsilon)$ **then**
2: Compute $\mathbf{y} = \hat{f}(\mathbf{x})$
3: **else**
4: Compute $\mathbf{y} = f(\mathbf{x})$
5: Update(S, $\mathbf{x}$, $f(\mathbf{x})$)
6: **end if**
7: **return** $\mathbf{y}$

Algorithm 2: Simple Update

Require: S, $\mathbf{x}$, $f(\mathbf{x})$

1: Add new $\langle R_{f,\hat{f}}(\mathbf{x}, \varepsilon), \hat{f} \rangle$ to S

1.3 An Algorithmic Framework

In this section we introduce an algorithmic framework for the problem, which highlights the indexing problem in function approximation. Since the results from our analysis in 1.2.4 are discouraging, we start with algorithms that we abstracted from the approach of the domain scientists: A greedy heuristic (Section 1.3.1) and its refinements (Section 1.3.2). We then summarize the instantiation of choice of this framework by the domain scientists (Section 1.3.3).

Algorithm 3: Update With Grow

Require: S, $\mathbf{x}$, $f(\mathbf{x})$

1: **if** $\exists \langle \hat{R}_{f,\hat{f}}(\mathbf{x'}, \varepsilon), \hat{f} \rangle \in S$: $\mathbf{x}$ can be included in $\hat{R}_{f,\hat{f}}(\mathbf{x'}, \varepsilon)$ **then**
2: **for all** $\langle \hat{R}_{f,\hat{f}}(\mathbf{x'}, \varepsilon), \hat{f} \rangle \in S$ **do**
3: **if** $\mathbf{x}$ can be included in $\hat{R}_{f,\hat{f}}(\mathbf{x'}, \varepsilon)$ **then**
4: Update $\langle \hat{R}_{f,\hat{f}}(\mathbf{x'}, \varepsilon), \hat{f} \rangle$ to include $\mathbf{x}$
5: **end if**
6: **end for**
7: **else**
8: Add new $\langle \hat{R}_{f,\hat{f}}(\mathbf{x}, \varepsilon), \hat{f} \rangle$ to S
9: **end if**

1.3.1 A Greedy Heuristic

Since the problem is hard and there is no hope for a competitive online algorithm, we use a simple greedy strategy as shown in Algorithm 1. The algorithm maintains an index structure S, which contains the Local Regions around previously evaluated query points. Given a new query point $\mathbf{x}$, it first tries to find a Local Region that contains $\mathbf{x}$ (Lines 1-2). If the point lies in some Local Region, an approximate value of $f(\mathbf{x})$ is calculated and returned. If the point does not lie in any indexed region, then the algorithm has to compute $f(\mathbf{x})$ (Line 4). It then updates the index based on the knowledge of $f(\mathbf{x})$ as it belongs to a part of the domain that is yet to be indexed (Line 5). In the simple version of the algorithm, the update routine creates a new Local Region containing $\mathbf{x}$ and inserts it into the index (Algorithm 2).

1.3.2 Practical Constraints

In practice, it is often impossible to accurately compute the Local Region ($R_{f,\hat{f}}(\mathbf{x}, \varepsilon)$) around a point. We will see why this is the case in Section 1.3.3. Usually an initial (conservative) guess of the Local Region is first obtained and inserted into the index. We denote these approximate Local Regions as $\hat{R}_{f,\hat{f}}(\mathbf{x}, \varepsilon)$. As the simulation proceeds and larger portions of the domain are seen, better approximations of the existing Local Regions in S are obtained. This calls for a modified update operation in Algorithm 1. Algorithm 3 is an update stub which replaces Algorithm 2.

In our initial algorithm, if the function was evaluated because no existing region contained the query point, then a new Local Region was created and inserted into the index. The new update stub on the other hand first checks to see if the current query point can be part of any existing Local Region (Line 1). If such regions exist, then it finds the regions that can include **x** and updates them (Lines 2-6). Finally, only if no existing region can include **x**, then a new Local Region is initialized and inserted into S in Line 8. Updating existing Local Regions is usually more beneficial than adding new ones because it reduces the total number of Local Regions. We will also see later that updating an existing region is cheaper than creating a new one.

1.3.3 An Instantiation

In practice finding a representation of the Local Regions of a function is not easy. In this section we review a method based on the Taylor Series [55]. This method, commonly used by scientists finds the Local Regions in two steps. It first creates

an initial approximation, which is then refined over time.

Initializing Local Regions

Under fairly general conditions a function $f(\mathbf{x+a})$ can be expanded using the Taylor Series as

$$f(\mathbf{x+a}) = \sum_{j=0}^{k} [\frac{1}{j!}(\mathbf{a}.\nabla_{\mathbf{x}})^j f(\mathbf{x})] + \phi_k(\mathbf{x}, \mathbf{a}), \tag{1.1}$$

where $\nabla_{\mathbf{x}}$ is the gradient $[\frac{\partial}{\partial x_1} \frac{\partial}{\partial x_2} \ldots \frac{\partial}{\partial x_m}]$ and the error ϕ_k is $O(|\mathbf{a}|^k)$, i.e., $lim_{h \downarrow 0} \frac{\phi_k(\mathbf{x}, h\mathbf{a})}{h|\mathbf{a}|^k} = 0$.

The Taylor Series provides us with a simple mechanism for function approximation. Given the value of f at a point **x**, the value at any point **x+a** "near" **x** is usually approximated using the first few terms of the summation in Equation 1.1. For example, using the first term only, we get a constant approximation $\hat{f}_{0,x}$ of f as follows:

$$f(\mathbf{x+a}) \approx \hat{f}_{0,x}(\mathbf{x+a}) \equiv f(\mathbf{x}) \tag{1.2}$$

Similarly, the first two terms give the following linear approximation $\hat{f}_{1,x}$

$$f(\mathbf{x+a}) \approx \hat{f}_{1,x}(\mathbf{x+a}) \equiv f(\mathbf{x}) + (\mathbf{a}.\nabla_{\mathbf{x}}) f(\mathbf{x}) \tag{1.3}$$

The errors of the above approximations can be obtained from the remaining terms in the summation of Eq. 1.1, i.e., those terms not used in the approximation. For small values of |a|, high-order terms in Eq. 1.1 are dominated by low-order terms and are therefore commonly ignored. The domain scientists only use the single lowest-order term to estimate the approximation error. More precisely, the approximation quality requirement is defined for the constant approximation as

$$\| \ (\mathbf{a}.\nabla_{\mathbf{x}}) f(\mathbf{x}) \ \| < \varepsilon, \tag{1.4}$$

and similarly for the linear approximation

$$\| \frac{1}{2!}(\mathbf{a}.\nabla_{\mathbf{x}})^2 f(\mathbf{x}) \| < \varepsilon. \tag{1.5}$$

Equation 1.5 for a high dimensional function is the equation of a tensor and hence, except under special conditions, it is computationally infeasible to compute the Local Region defined by it. However, it can be shown that Eq. 1.4 defines a hyper-ellipsoid around **x** [55]. Therefore, using a constant approximation, the local region around **x** is a hyper-ellipsoid.

Growing

The linear approximation is preferred over the constant approximation, because it tends to generate much larger Local Regions. Unfortunately, since the region defined by Eq. 1.5 is difficult to compute, we have to start out with the more conservative (and hence smaller) region defined by Eq. 1.4. Since we know that the true Local Region is much larger, we use the grow operation to extend the initial region over time as follows. Consider a query point **x** and an ellipsoid e around it. Suppose, there exists another query point **x′** such that **x′** lies just outside e but $\hat{f}_{1,x}(\mathbf{x'})$ is an ε-approximation. Then **x′** is assumed to be part of the Local Region of **x**; therefore e is grown to a larger ellipsoid that contains **x′**. This simple heuristic of growing has been found to work well in practice [48]. For applications with stricter error guarantees, growing can be further controlled by using domain specific information like the maximum allowable size of ellipsoids or it may even be turned off and other function specific methods may be used to find the true Local Regions.

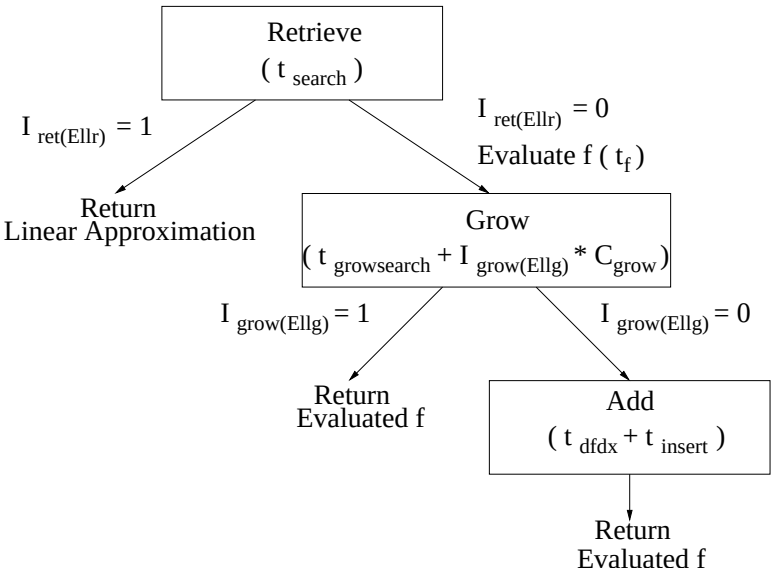

Figure 1.4: Costs In Function Approximation

Final Algorithm

Instantiating the framework with ellipsoids as the Local Regions is straightforward. The algorithm performs the following high level operations on a query point **x**.

Retrieve: The algorithm first tries to find an ellipsoid that contains **x** (Line 1 of Algorithm 1).

Grow: If the retrieve fails then the algorithm attempts to grow existing ellipsoids in the index to include **x** (Lines 1-6 of Algorithm 3).

Add: If both the retrieve and the grow fail a new ellipsoid derived from the constant approximation is initialized and inserted into the index (Line 8 of Algorithm 3).

1.4 Indexing Problem

We now turn to the indexing problem in function approximation, which produces a challenging workload for the operations on index S in Algorithms 1, 2 and 3. The retrieve requires the index to support fast lookups. The grow requires both a fast lookup to find growable ellipsoids and then an efficient update process once an ellipsoid is grown. Finally, an efficient insert operation is required for the add step. Also, past decisions about growing and adding affect future performance of the index, therefore the algorithm produces a query/update workload that is not common in traditional indexing applications.

A straightforward implementation of the algorithm introduced in the previous section would search in the index for an ellipsoid containing the query point until it finds an ellipsoid or has established that no such ellipsoid exists. The grow similarly would try to find all ellipsoids that can be grown and finally the add is performed if these operations fail. Our initial experiments showed that this implementation can result in poor performance.

This observation brings us to the most interesting aspect of the indexing problem in function approximation: It presents a very different framework in which indexes must be evaluated. Traditionally, the performance of index structures has been measured simply in terms of the cost of a search and in some cases update. There are two distinct cost factors in the function approximation problem. First, there are the costs associated with the search and update operations on the index. Second, there are costs of the function approximation application which include function evaluations and ellipsoid operations. Since,

the goal of function approximation is to minimize the total cost of the simulation, all these costs must be accounted for when evaluating the performance of an index. We will see that a principled analysis leads to the discovery of novel tradeoffs. These tradeoffs produce significant and different effects on different index structures. This makes indexing for function approximation a challenging problem.

1.4.1 Costs

This section introduces a cost model for the function approximation algorithm. We use the cost model to qualitatively explore the tradeoffs in the indexing problem. The formulation of a quantitative cost model for optimization purposes has limited use. This is because the benefits from an operation can be determined only in the future after the operation has been done. As a simple example, consider the grow operation. The benefit from the grows cannot be estimated accurately until the actual grown ellipsoids are known, which requires the grow operation to be performed. Therefore, we do not explore a true quantitative cost model further in the paper. Table 1.1 is a summary of the commonly referenced variables in this section and the next.

The total cost (C_{tot}) of processing a query $\mathbf{q}$ can be expressed as

$$C_{tot} = t_{search} + I_{ret} \times t_{la} + (1 - I_{ret}) \times C_{miss} \tag{1.6}$$

Note that the the costs in Eq. 1.6 vary from one query to the next. However, the dependence on $\mathbf{q}$ has been dropped for ease of notation. t_{search} is the cost of searching the index for an ellipsoid containing $\mathbf{q}$. This is essentially the cost of the retrieve step. I_{ret} is an indicator which is 1 if there exists an ellipsoid

Table 1.1: Notation For Cost Model

Name	Description
C_{tot}	Total cost of a query
C_{miss}	Total cost of a miss
C_{grow}	Total cost of a grow
C_{add}	Total cost of an add
I_{ret}	Indicator variable for successful retrieve
I_{grow}	Indicator variable for successful grow
t_{search}	Index search cost during Retrieve
t_f	Cost of a function evaluation
t_{la}	Cost of a linear approximation
$t_{growsearch}$	Index search cost during grow
$t_{inellipsoid}$	Cost of checking if ellipsoid contains point
t_{grow}	Cost to grow ellipsoid
t_{update}	Index update cost
t_{dfdx}	Cost of a derivative evaluation
t_{insert}	Index insertion cost
Ellr	Max. number ellipsoids examined for Retrieve
Ellg	Max. number ellipsoids examined for Grow
Nfpos	Number of false positives during Retrieve
N_{found}	Number of growable ellipsoids found
$N_{growmax}$	Max. number of grows allowed
N_{grown}	Number of ellipsoids grown

containing $\mathbf{q}$ in the index and 0 otherwise. t_{la} is the cost of calculating the linear approximation on a successful retrieve. C_{miss} is the cost incurred if $\mathbf{q}$ does not result in a retrieve operation. A miss results in either a grow or an add. C_{miss}

Table 1.2: Relative Costs Of The Ellipsoid Operations

Ellipsoid Operation	Cost
t_f	2000
t_{dfdx}	1200
t_{grow}	10
t_{la}	1
$t_{inellipsoid}$	1
Usual search cost	1

can therefore be written as

$$C_{miss} = t_f + t_{growsearch} + I_{grow} \times C_{grow} + (1 - I_{grow}) \times C_{add} \tag{1.7}$$

The cost of a grow comprises two parts. $t_{growsearch}$ is the cost associated with searching for growable ellipsoids. C_{grow} is the cost of actually growing the ellipsoids and updating the index. I_{grow} is an indicator which is 1 if there is some ellipsoid in the index that can be grown and 0 otherwise. Finally, if no ellipsoids were grown, an ellipsoid is added at a cost of C_{add}.

Table 1.2 describes typical ratios between costs of the application in the combustion problem, which is a typical example of an application that our methods apply to. The table also lists the commonly observed cost of searching for an ellipsoid in an index. It is important to note that for most indexes the costs of the application are more expensive than the index operations.

1.4.2 Effects And Tradeoffs

In the previous section we outlined the cost of processing a query. Here we will analyze the different components of C_{tot}. Figure 1.4 displays the cost associated with each high-level operation.

Retrieve: The first component of C_{tot} is t_{search}. In most high dimensional index structures the ellipsoid containing a query point is usually not the first ellipsoid found. The index ends up looking at a number of ellipsoids before finding "the right one" (Section 1.5.1). The additional ellipsoids that are examined by the index are called *false positives*, the number of which is denoted by Nfpos. Taking Nfpos into account, we can rewrite t_{search} as

$$t_{search} = (\text{Nfpos} + 1) \times (t_r + t_{inellipsoid}).$$

For each false positive the algorithm pays to search and retrieve the ellipsoid from the index (t_r) and to check if the ellipsoid contains the query point ($t_{inellipsoid}$). In practice using an iterator for the search could lead to different values of t_r for each false positive. However, this is not important for this qualitative study and hence we ignore such effects to keep the analysis simple.

In traditional indexing problems, if an object that satisfies the query condition exists in the index, then finding this object during search is mandatory. Therefore, Nfpos is a fixed property of the index. However, the function approximation problem provides the flexibility to tune Nfpos, because we can evaluate the function if the index search was not successful. The number of false positives can be tuned by limiting the number of ellipsoids examined during the retrieve step. We denote this parameter by Ellr. Ellr places an upper bound on the number of false positives for a query. Taking this parameter into account,

the total cost of processing a query can be rewritten as

$$C_{tot} = t_{search} + I_{ret(Ellr)} \times t_{la} + (1 - I_{ret(Ellr)}) \times C_{miss} \tag{1.8}$$

$I_{ret(Ellr)}$ is 1 if an ellipsoid containing the query point is found by the index *before Ellr ellipsoids are examined*, and 0 otherwise. Notice that for a given query, $I_{ret} \geq I_{ret(Ellr)}$. Tuning Ellr introduces the following tradeoffs:

Effect 1: Decreasing Ellr restricts the number of ellipsoids examined during the retrieve for a given query. This effectively reduces the number of false positives, therefore decreasing t_{search}.

Effect 2: Decreasing Ellr decreases the probability that $I_{ret(Ellr)} = 1$ for a given query $\mathbf{q}$, thereby lowering the probability of a query resulting in a retrieve. This is because there might be an ellipsoid containing $\mathbf{q}$ in the index but it is not found before Ellr ellipsoids are examined. Reducing the probability of retrieve increases the probability of an expensive miss operation.

Effect 3: The previous tradeoffs demonstrated the effects of decreasing Ellr on the probability of a retrieve for a given query. There is another tradeoff unique to the problem. Misses that result from decreasing Ellr can grow and add ellipsoids. These grows and adds index new parts of the domain and also change the overall structure of the index. Both of these affect the probability of retrieves for future queries.

Grow: The next major cost component is the cost of the grow operation: $t_{growsearch} + I_{grow} \times C_{grow}$. In the first part of the grow process the index is traversed to find ellipsoids that can be grown. Every ellipsoid in the index is a candidate for growing. Checking an ellipsoid for growing involves a cost of retrieving it from the index (t_{rg}) and then checking to see if the ellipsoid can be

grown (t_{la}). Therefore, most indexes prune the search space for finding growable ellipsoids using some domain information or heuristic. Just like the retrieve operation, the cost incurred in searching for growable ellipsoids can be tuned by restricting the number of ellipsoids examined for growing (Ellg). Taking this parameter into account, C_{miss} can be rewritten as $C_{miss} =$

$$t_f + t_{growsearch} + I_{grow(Ellg)} \times C_{grow} + (1 - I_{grow(Ellg)}) \times C_{add}.$$

$I_{grow(Ellg)}$ is 1 if at least one growable ellipsoid is found before Ellg ellipsoids are examined, and 0 otherwise.

The cost of searching for growable ellipsoids can be written as

$$t_{growsearch} = Ellg \times (t_{rg} + t_{la}).$$

Tuning Ellg introduces the following tradeoffs:

Effect 4: Decreasing Ellg decreases $t_{growsearch}$ because fewer ellipsoids are examined for growing.

Effect 5: Decreasing Ellg decreases the number of ellipsoids examined for growing and hence the number of growable ellipsoids that are found (N_{found}). Therefore, restricting Ellg also limits N_{grown}(the number of ellipsoids that are finally grown). The effects associated with N_{grown} are described in Effects 6 and 7.

After finding the ellipsoids that can be grown, the next step of the grow operation is to actually grow the ellipsoids, which costs C_{grow}. There is another tuning parameter, $N_{growmax}$, which represents the maximum number of ellipsoids that are allowed to be grown during a grow operation. Hence the number of ellipsoids that are actually grown, N_{grown}, is only $\min\{N_{found}, N_{growmax}\}$. For

each ellipsoid that is grown during the grow step, the algorithm incurs a cost to grow the ellipsoid (t_{grow}) and update the index (t_{update}).

$$C_{grow} = N_{grown} \times (t_{grow} + t_{update})$$

N_{grown} has the following effects on the cost of the algorithm:

Effect 6: Lower values of N_{grown} decreases C_{grow}. This can affect the simulation time significantly because t_{grow} and t_{update} can be expensive.

Effect 7: Larger values of N_{grown} increase the fraction of the domain that is covered by ellipsoids. Therefore, I_{ret} changes from 0 to 1 for future query points that lie in the newly covered part of the domain.

Effect 8: For $N_{grown} > 1$ the false positive rate of the index can increase, because the grown ellipsoids overlap (they all cover the new query point). This in turn might negatively affect $I_{ret(Ellr)}$. The reason for this is that the higher false positive rate can result in failed retrieves due to the search limit imposed by Ellr.

Add: The last cost component is the cost of adding an ellipsoid. It includes the cost of finding the derivative of the function (t_{dfdx}), which is costly (see Table 1.2), and inserting the new ellipsoid into the index (t_{insert}). The derivative is needed to estimate the initial ellipsoid and for computing the linear approximation [55]. A new ellipsoid is added only if the retrieve and grow both fail. Therefore there is no direct way of controlling the number of adds.

Effect 9: Lowering the effort spent on the retrieve and grow can cause the number of add operations to increase. This can be undesirable because the add operation is expensive and a newly added ellipsoid is a conservative approxi-

mation of a Local Region. Adds also increase the index size.

In summary, the algorithm provides us with a set of tunable parameters, namely the number of ellipsoids examined during retrieve (Ellr), the number of ellipsoids examined during grow (Ellg), and the maximum number of ellipsoids allowed to be grown ($N_{growmax}$). Each parameter can have different effects on C_{tot}. What makes the problem interesting is that these effects often move in opposite directions. Moreover, tuning affects indexes differently and to varying degrees, which makes it necessary to analyze each index individually. In the next section we will demonstrate the effects of these parameters on the performance of different index structures when used in the function approximation algorithm.

1.5 Experiments

In the previous section we introduced the tuning parameters of the function approximation problem and we identified nine qualitative effects these parameters could have on the runtime. In this section we study the corresponding tradeoffs for a concrete instance of the problem and different index structures.

All experimental results are for a Methane combustion simulation. In the simulation, the number of species was set to $s = 31$, i.e., the thermochemical composition vector has 32 dimensions. There are 100 particles in the combustion chamber; at each time step a single particle enters the reaction step. The simulation was run for 6×10^6 time steps, thereby generating 6×10^6 query points for function evaluation. The error tolerance, unless otherwise noted, was set to $\varepsilon = 5 \times 10^{-5}$. All reported measurements are wall-clock time for an execu-

tion on a Windows XP machine with a 2.4Ghz processor and 2GB of memory.

1.5.1 Candidate Index Structures

Our goal in this paper is *not* to find the best index for function approximation. In fact because of the diversity of function approximation applications in terms of their cost structure, dimensionality, and locality of access, the existence of a single best index is unlikely. For this reason we selected a very diverse set of indexes, without attempting a comparison of all existing ones. Another criterion for selection for this initial study was to pick only well understood index structures with predictable behavior, rather than highly optimized and complex indexes.

We chose a candidate from each of the different classes of commonly used indexes, namely linear scan, spatial partitioning, balanced index for points and balanced index for extended objects. In the simulations we studied, the indexes were small enough to fit in main memory, therefore we limited our attention to in-memory performance. Extending our experiments to other indexes and I/O performance is part of our future work.

Bounding Box Rtree (Bbox Rtree): An obvious choice for function approximation is an index that can manage the Local Regions, i.e., the ellipsoids in our case. The most well-known data structure with this functionality is the Rtree, a balanced multidimensional generalization of the B-tree which can handle both point and hyper-rectangular objects. There exists a large number of Rtree-variants (see Section 1.6); as a representative we selected the robust R*-tree [3]. Bbox Rtree indexes the axis-parallel minimum bounding boxes of the el-

lipsoids, using the standard R*-tree algorithms. The retrieve operation finds leaf objects (minimum bounding boxes of ellipsoids) that contain the query point. Then it needs to verify that the corresponding ellipsoid also contains the query point. Growing of ellipsoids is implemented by a deletion, followed by an insertion. Growable ellipsoids are found by performing a nearest-neighbor (NN) search on the bounding boxes.

Point Rtree: Managing objects with extent is far more challenging than handling points [29]. We can map our problem to a point-indexing problem by only indexing the center points of the ellipsoids in an Rtree. The Point Rtree does not have to deal with overlapping leaf objects (bounding boxes of *inner* nodes can still overlap) and growing does not require an index update, because the center of an ellipsoid is not modified by the grow operation. Unfortunately, without any information about the dimensions of the ellipsoid, the index has no way of pruning search—as long as the ellipsoid is large enough, even a center point far away from the query could be relevant. Intuitively based on the Taylor Series, the Euclidean distance between query point and ellipsoid center should be correlated with the probability that the query point is within the ellipsoid. We therefore implemented both the retrieve and the search for growable ellipsoids as a NN-query.

Binary Tree: This is a binary space partitioning tree [29], which was introduced for the ISAT function approximation problem [55]. The Binary Tree indexes the centers of the ellipsoids by recursively partitioning the space with cutting planes. Leaf nodes of the tree correspond to ellipsoid centers and non-leaf nodes represent cutting planes. During the retrieve step, the index is traversed from the root by following the subtree corresponding to the side of the cutting

plane that the query point lies on. Like the Point Rtree, the Binary Tree requires no update when an ellipsoid is grown, because it only indexes the ellipsoid center points. A more detailed discussion of the index can be found in Section 1.5.2.

MRU List + Rtree: For high dimensional data, it has been shown that a simple linear scan often outperforms any sophisticated indexing technique [73]. We therefore include a list-based data structure. This simple structure has the advantage that if there is locality of access, we can directly apply existing cache-replacement policies. The MRU List stores the ellipsoids ordered by their most recent access. The retrieve operation simply scans the list, starting with the most recently used object. To improve the search for growable ellipsoids, we index the ellipsoids with a "secondary" point Rtree. This tree is identical to the Point Rtree described above, but it is not used for the retrieve operation. Notice that the leaf objects also contain a pointer to the corresponding ellipsoid in the MRU list.

1.5.2 Tradeoffs: Detailed Analysis

We examined the tradeoffs for all candidate index structures. Due to space constraints, we only discuss the two best-performing indexes in detail and report the overall performance for the others. Somewhat to our surprise, the Binary Tree and the simple MRU List + Rtree have the best performance after tuning. If we do not tune any index, i.e., set Ellr $=$ Ellg $= \mathrm{N}_{\mathrm{growmax}} = \infty$, the Bbox Rtree clearly outperforms the Binary Tree. However, the Binary Tree benefits much more from tuning. This will be discussed in more detail in this section.

For our experiments we do not examine the effect of the $\mathrm{N}_{\mathrm{growmax}}$ parame-

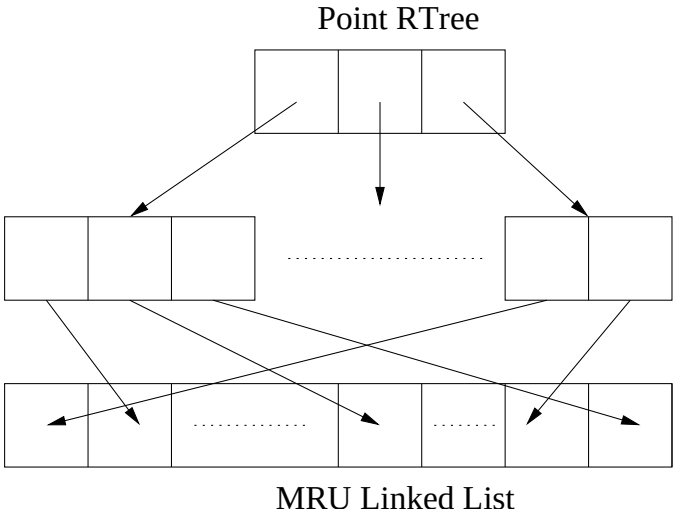

Figure 1.5: MRU List + Rtree

ter. Recall that we grow $N_{grown} = \min\{N_{found}, N_{growmax}\}$ ellipsoids, i.e., $N_{growmax}$ limits the number of finally grown ellipsoids if there are too many growable candidates. However, we currently do not have any meaningful way of preferring one grow candidate over another, therefore we set $N_{growmax} = \infty$. Notice also that Ellg directly limits N_{found}, and therefore we can control N_{grown} through Ellg. Hence in our experiments we only study the effect of two tuning parameters: Ellr and Ellg.

MRU List + Rtree

The MRU List + Rtree index (Figure 1.5) uses the list for the retrieve operations. The list contains pointers to the ellipsoids. During the retrieve step, it is scanned, starting with the most recently used ellipsoid. This continues until either an ellipsoid is found that contains the query point, or Ellr ellipsoids have been examined unsuccessfully. If a containing ellipsoid is found, it is moved to the front of the list.

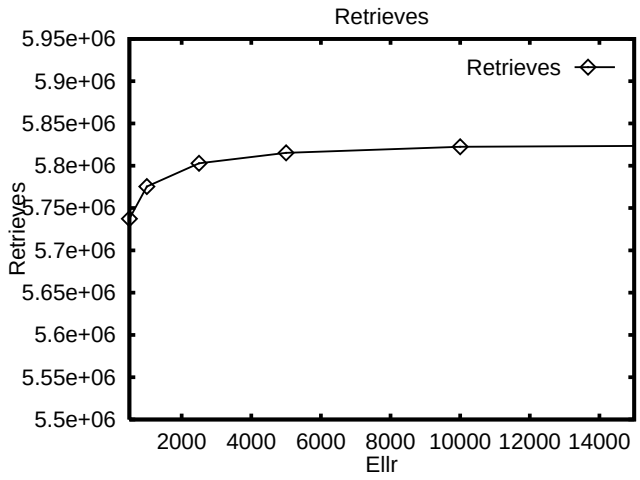

Figure 1.6: Number Of Retrieves vs. Ellr (MRU List + Rtree)

As mentioned earlier, the Rtree improves the performance of the search for growable ellipsoids during the grow step. It indexes the ellipsoid centers (see Figure 1.5 for an illustration). Instead of scanning the list for growable ellipsoids, we perform a NN-search for the query point to find grow candidates sorted by the Euclidean distance to the ellipsoid centers. The search terminates once the Ellg nearest neighbors have been examined. All growable ellipsoids are grown. Since their center points do not change, we do not need to update the Rtree. The linked list is updated by moving all grown ellipsoids to the front.

An add operation adds the new ellipsoid to the front of the list. Its center point, together with a pointer to the ellipsoid object, is inserted into the Rtree.

We first examine the effect of tuning Ellr, the number of ellipsoids in the list that are examined during the retrieve step. For this experiment, we held the grow search limit constant at Ellg $= 3000$. Limiting the grow search to 3000 nearest neighbors was found to be enough to find all growable ellipsoids. We report the total number of retrieves (Figure 1.6) and the total cost of the simulation, also broken down into retrieve and miss (grow and add) cost (Figure 1.7).

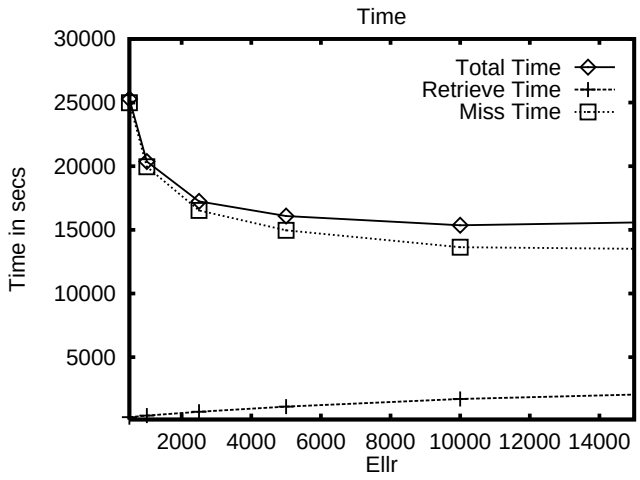

Figure 1.7: Time vs. Ellr (MRU List + Rtree)

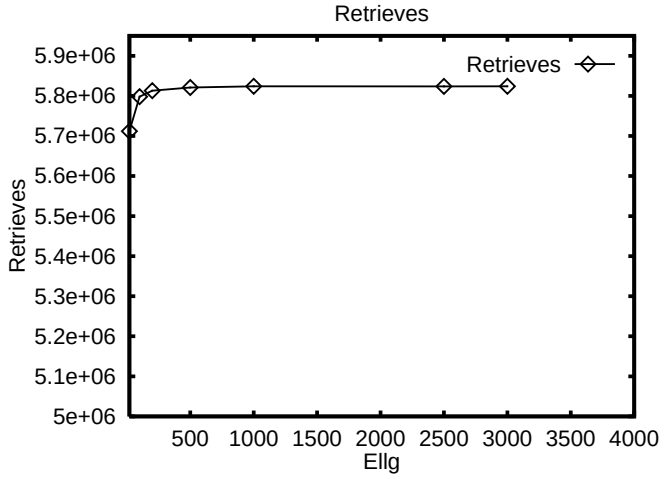

Figure 1.8: Number Of Retrieves vs. Ellg (MRU List + Rtree

As we increase Ellr, t_{search} increases in accordance with Effect 1 (Figure 1.7). At the same time the number of retrieves increases (Figure 1.6), because we are searching further, thereby reducing C_{miss} (Effect 2). As we increase Ellr, Effect 2 initially causes the simulation time to decrease. However, at some point further increasing Ellr will only add very few additional retrieves, hence the total number of retrieves asymptotes and the increased effort in searching does not pay off any more. At this point Effect 1 causes the overall simulation time to

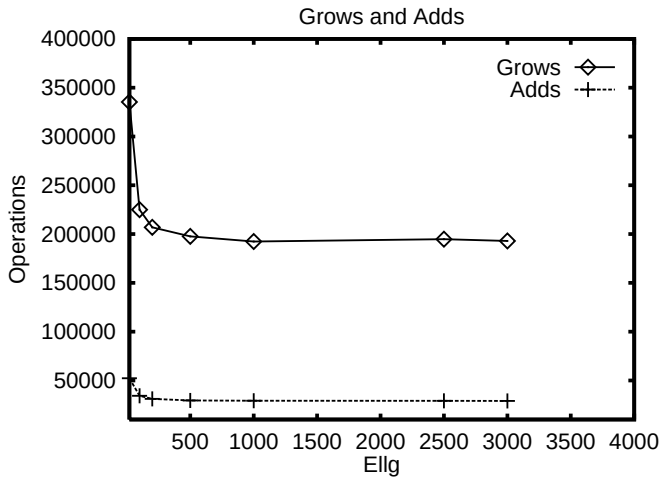

Figure 1.9: Number Of Grows, Add vs. Ellg (MRU List + Rtree)

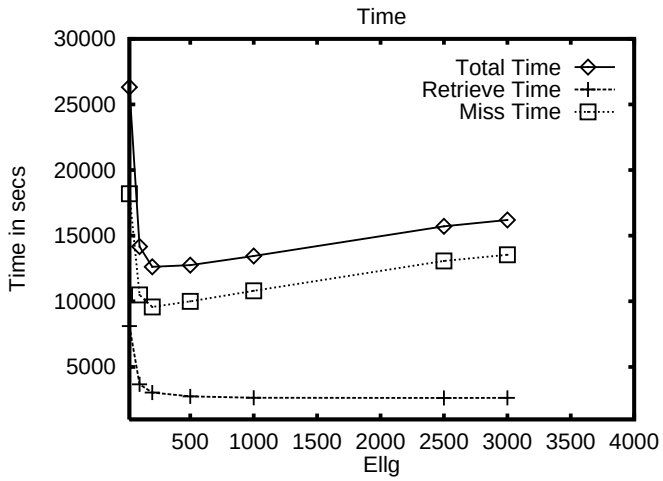

Figure 1.10: Time vs. Ellg (MRU List + Rtree)

increase slightly.

Next we examine the effect of tuning Ellg, the number of nearest neighbors examined for growing. In this experiment there were no restrictions placed on Ellr. The Rtree examines ellipsoids for growing in nearest neighbor order. Therefore, as we start to increase Ellg, larger ellipsoids are grown and the domain is indexed more aggressively. The number of retrieves increases (Figure 1.8) and the number of misses decreases (Figure 1.9). The total simulation

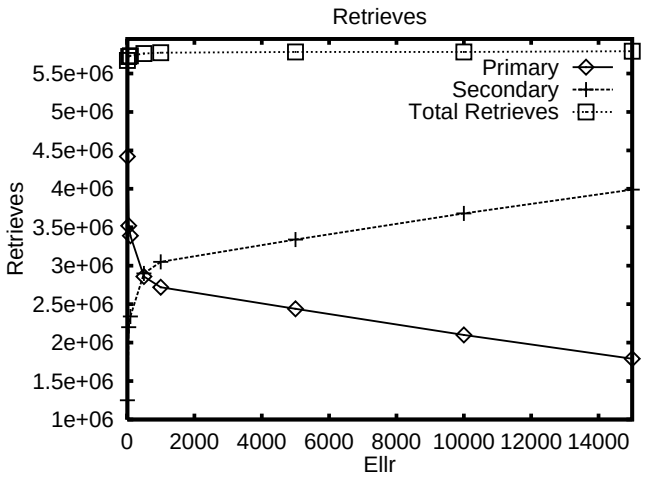

Figure 1.11: Number of retrieves vs. Ellr (Binary Tree)

time (Figure 1.10) therefore decreases (Effect 7). Increasing Ellg increases the chances of finding at least one growable ellipsoid, hence reduces the number of adds (Effect 9). Reduction in the number of adds causes the list size to decrease for larger values of Ellg. This accounts for the decrease in t_{search} as we start increasing Ellg. As we go on increasing Ellg, the number of retrieves asymptotes (Figure 1.8) because no additional ellipsoids are found for growing. There is no increase in retrieve time. However, the total simulation time then increases because $t_{growsearch}$ increases with Ellg caused by Effect 4 (Figure 1.10).

Binary Tree

The Binary Tree partitions the data space recursively, using cutting planes [55]. It might be unbalanced, i.e., leaves can be at different depths. Figure 1.12 shows an example tree with three ellipsoids A, B, C and two cutting planes X and Y. For now we focus on the tree in the top half of the figure, together with the corresponding view of the data space showing the cutting planes and ellipsoids.

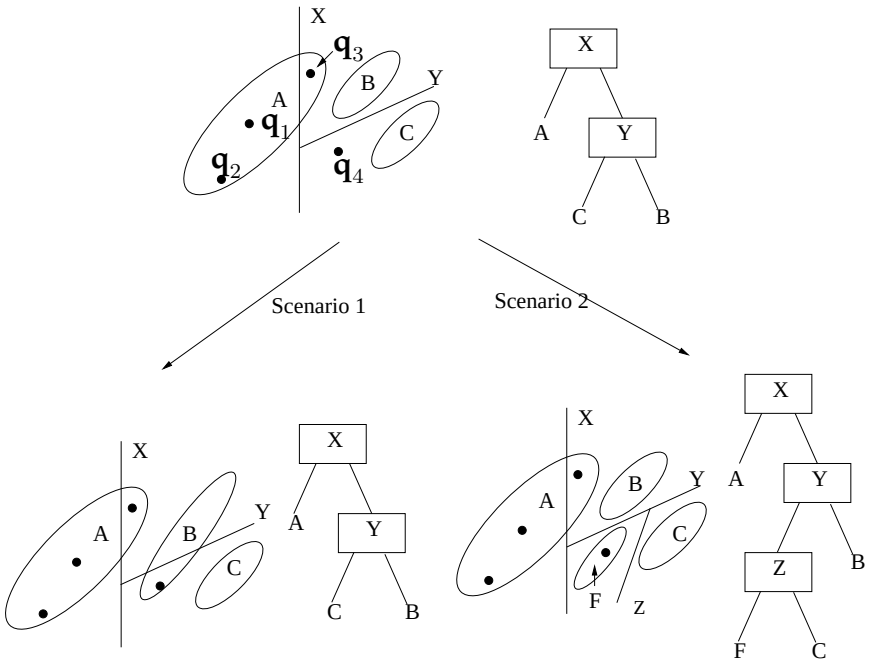

Figure 1.12: Binary Tree

We illustrate the retrieve step with query point $\mathbf{q}_2$. The retrieve starts at the root, checking on which side of hyper-plane X the query point lies. The search continues recursively with the corresponding subtree, the left one in our example. When we reach a leaf node, we test if the ellipsoid in the leaf contains the query point. In the example, A contains $\mathbf{q}_2$, therefore we have found a containing ellipsoid. This process, i.e., when the traversal from root to leaf is successful, will be denoted as a *Primary Retrieve* [55].

Notice that ellipsoids can straddle cutting planes, e.g., A covers volume on both sides of cutting plane X. If ellipsoids are straddling planes, then the Primary Retrieve can result in a false negative. For example, $\mathbf{q}_3$ lies to the right of X and so the Primary Retrieve fails even though there exists an ellipsoid A containing it. To overcome this problem the Binary Tree performs a *Secondary Retrieve* if the Primary fails. The main idea of the Secondary Retrieve is to explore

the "neighborhood" around the query point by examining "nearby" subtrees. In the case of $\mathbf{q}_3$, the failed Primary Retrieve ended in leaf B. Nearby subtrees are explored by moving up a level in the tree and exploring the other side of the cutting plane. Specifically, we first examine C (after moving up to Y, C is in the unexplored subtree). Then the search would continue with A (now moving up another level to X and accessing the whole left subtree). This process continues until a containing ellipsoid is found, or Ellr ellipsoids have been examined unsuccessfully.

The search for growable ellipsoids proceeds in exactly the same way as a Secondary Retrieve, starting where the failed Primary Retrieve ended. Assume that in the example in Figure 1.12, ellipsoid B can be grown to include $\mathbf{q}_4$, but C and A cannot. After the retrieve failed, the grow operation first attempts to grow C. Then it continues to examine B, then A (unless Ellg < 3). B is grown to include $\mathbf{q}_4$, as shown on the bottom left (Scenario 1). Growing of B made it straddle hyper-plane Y. Hence, for any future query point near $\mathbf{q}_4$ and "below" Y, a Secondary Retrieve is necessary to find containing ellipsoid B, which is "above" Y.

The alternative to growing B is illustrated on the bottom right part of Figure 1.12 (Scenario 2). Assume Ellg $= 1$, i.e., after examining C, the grow search ends unsuccessfully. Now we add a new ellipsoid F with center $\mathbf{q}_4$ to the index. This is done by replacing leaf C with an inner node , which stores the hyper-plane that best separates C and F. The add step requires the expensive computation of F, but it will enable future query points near $\mathbf{q}_4$ to be found by a Primary Retrieve.

As we can see from this example, tuning parameter Ellg affects the Binary

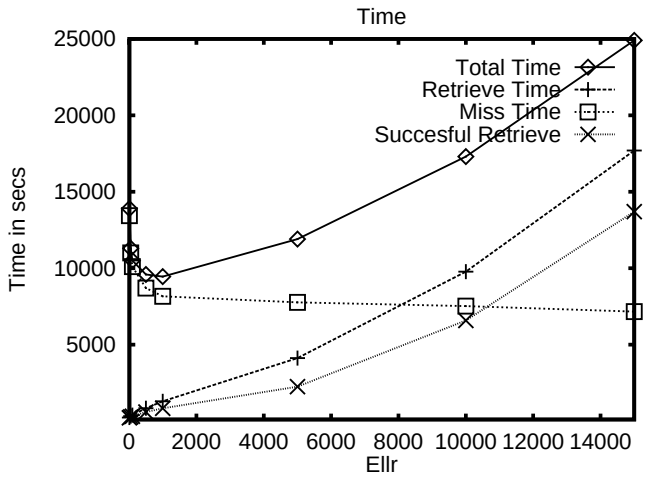

Figure 1.13: Time vs. Ellr (Binary Tree)

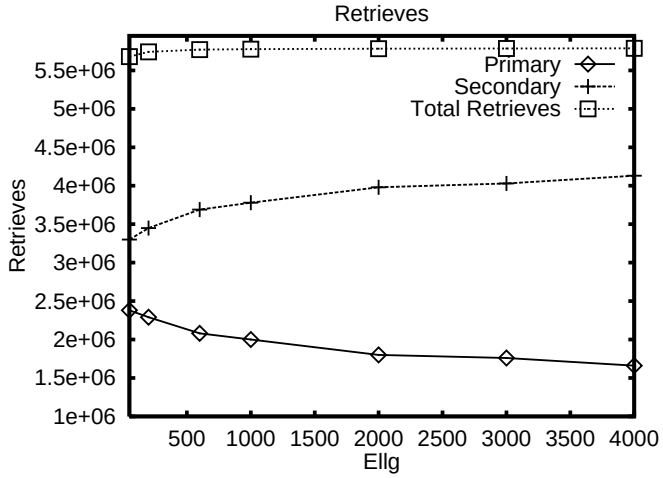

Figure 1.14: Number Of Retrieves vs. Ellg (Binary Tree)

Tree in its choice of scenario 2 over 1. Furthermore, this choice, i.e., performing an add instead of a grow operation, reduces Nfpos for future queries, but adds extra-cost for the current query. The experiments show that this tradeoff has a profound influence on the overall simulation cost. We will also see that the effect of the tuning parameters is very different for the Binary Tree as compared to the MRU List + Rtree.

We first study the effect of varying Ellr, which limits the number of ellip-

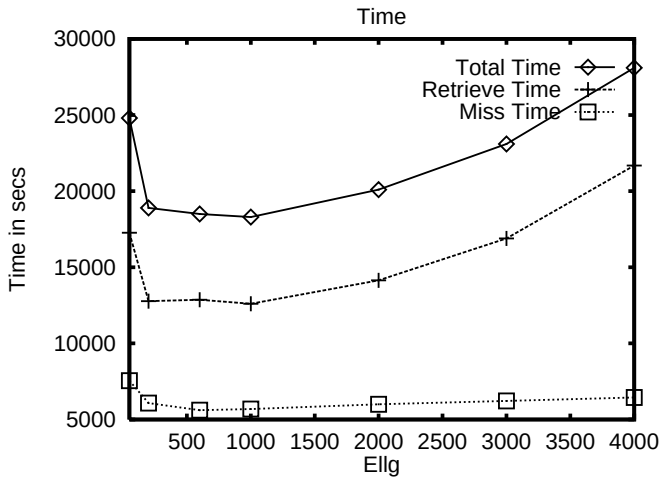

Figure 1.15: Time vs. Ellg (Binary Tree)

soids examined during the Secondary Retrieve phase. For this experiment we set Ellg $= \infty$. It can be seen from Figure 1.13 that as we increase Ellr, t_{search} goes up (Effect 1). This increase in the retrieve time is accompanied by a reduction in miss time, which is caused by the improved total number of retrieves (hence fewer misses) due to the more aggressive Secondary Retrieves (Effect 2, see Figure 1.11).

One of the most interesting observations from Figure 1.13 is the super-linear increase in the time for successful retrieves, which starts dominating the total simulation time. Figure 1.11 reveals the explanation: As we increase Ellr, Secondary Retrieves (and hence also Nfpos) are increasing, because we are searching the index more extensively. Therefore we are reducing the number of add operations, ultimately causing the Primary Retrieve rate to decrease (Effect 3). At the same time, the average cost of a Secondary Retrieve also increases, because the search proceeds further in the tree. These two effects together—increase in number of Secondary Retrieves and in average cost per Secondary Retrieve—create the superlinear trend of the retrieve time with increasing Ellr.

Lastly, we examine the effect of varying Ellg, the number of ellipsoids examined for growing, while setting Ellr $= \infty$. As we start increasing Ellg, because of Effect 7 the total number of retrieves increases slightly (Figure 1.14). Therefore there are fewer misses, which results in lower miss cost and better total simulation time (see Figure 1.15). Note the initial drop in retrieve time in Figure 1.15. The reason is that t_{search} includes the cost of *all* searches, including unsuccessful ones. A better retrieve rate therefore also reduces the total retrieve cost.

Figure 1.14 shows that as we increase Ellg, Primary Retrieves are being replaced by Secondary Retrieves, while the overall number of successful retrieves stays fairly constant for larger values of Ellg. This is because increasing Ellg is replacing adds with grow operations, which as we discussed earlier increases Nfpos and is a manifestation of Effect 8. The explanation of the super-linear increase in retrieve time is similar to that described for tuning Ellr. As we increase Ellg the miss cost also increases slightly (see Figure 1.15) because of Effects 4 and 6. Overall the total simulation time first decreases because of the dominance of Effect 7, but later starts to increase because of Effects 4 and 8.

1.5.3 Tradeoffs: The Big Picture

It is evident from the detailed analysis in Section 1.5.2 that the tuning parameters can have a significant effect on the performance of the indexes and the overall function approximation algorithm. We performed a similar analysis for the other index structures and selected the best parameter setting for each of them accordingly. Table 1.3 lists the overall running time of the Methane combustion simulation; the times are for the indexes *after tuning*, unless explicitly

Table 1.3: Total Simulation Time (sec)

Index Type	Value of ε		
	0.0005	0.00005	0.00004
Binary Tree (tuned)	1073	10181	13100
MRU List + Rtree	1125	14000	19920
Bbox Rtree	1201	14700	20850
Random Projection Rtree	1378	15800	22051
Binary Tree (default)	1344	29186	31200
FIFO List + Rtree	2164	33770	42900
Point RTree	10431	> 44000	-
Ellipsoidal Rtree	14328	> 44000	-

stated otherwise. We report times for different values of ε, because the index size increases with lower error tolerance and hence smaller ellipsoids.

The tuned Binary Tree performs significantly better than the Binary Tree with default parameter settings (Ellr $= \infty$). In fact, it outperforms all competitors. The "natural" index for this problem, the Bbox Rtree, performs well, but is 20-40% slower than the tuned Binary Tree. We established that the cause for this difference was the ability of the Binary Tree to achieve a large number of Primary Retrieves because it partitions the space, rather than searching through levels of overlapping bounding boxes. Careful tuning can bias the Binary Tree toward a high rate of Primary Retrieves, with little reduction in overall retrieval rate. On the other hand, tuning had comparatively little effect on the Bbox Rtree. The overlap of bounding boxes at all levels of the tree resulted in large numbers of false positives during search. We note here that the difference in performance of the Bbox Rtree and the Binary Tree is not due to the update costs of the Rtree. We have found in our experiments that Effect 6 does not significantly affect per-

formance because very few ellipsoids actually grow during a single grow step.

The dramatic difference between the FIFO List and MRU List indexes is caused by locality in the combustion simulation. Both index structures are identical; they only differ in the order of the ellipsoids in the list. MRU sorts by most recent access, while FIFO maintains the ellipsoids in the order in which they were added.

A surprising result was the poor performance of the Point Rtree. Since it does not know the spatial extent of the ellipsoids, we implemented the retrieve operation with an NN-query to find the "best" ellipsoids early on. Unfortunately because of the limited pruning power and the high cost of the NN-search, the Point Rtree was not more successful than scanning the FIFO List. The MRU List + Rtree essentially uses the same Rtree during the grow step to find grow candidates. Hence the performance difference between it and the Point Rtree approach is mostly due to the poor retrieve performance of the Point Rtree.

We also experimented with two extensions of the Bbox Rtree to explore ways to improve its performance. Both are motivated by the problem that in high dimensions, hyper-rectangular bounding boxes are only poor approximations of ellipsoids. The bounding boxes contain a large fraction of "dead space", i.e., volume that is outside the ellipsoid, which creates many false positives during search.

The **Random Projection Rtree** addresses the problem by projecting all ellipsoids onto a fixed set of k randomly selected lines. This transforms a d-dimensional ellipsoid into a k-dimensional hyper-rectangle in the transformed space defined by the projection lines. We can now use a standard Rtree to index

the objects. By using larger numbers of projections, we can achieve a tighter bounding polyhedron around an ellipsoid, at the cost of more expensive index operations in the higher-dimensional space. The results for $k = 60$ showed the expected lower false positive rates (compared to Bbox Rtree), but slightly worse overall performance (see Table 1.3) because of higher dimensionality. A detailed study of Random Projection Rtrees is part of our future work.

We can also reduce dead space by using ellipsoids as the bounding shape at all tree levels. The corresponding **Ellipsoid Rtree** performed very poorly, because of the high cost of basic index operations like testing if a point is within a bounding ellipsoid or splitting nodes and computing the new bounding ellipsoids, which is done approximately.

1.6 Related Work

The ISAT function approximation approach was first introduced by Pope [55]. It is one of the most widely used techniques for function approximation in the scientific community and is now a part of Fluent's CFD package [69]. Machine learning and data mining research have extensively studied the problem of automatically learning unknown functions [33]. By treating the known function values as a training sample, machine learning techniques can be used for function approximation. For the combustion simulation, neural networks have been proposed and used [34]. However, there has been very little work on studying function approximation as an indexing problem. Pope [55] and later Veljkovic et al. [68] propose new index structures for combustion simulation. Our work is the first principled analysis of the indexing problem.

A large variety of index structures have been proposed by the database and computational geometry communities, and their suitability for the function approximation problem needs to be studied. Work prior to 2001 is surveyed in [7] and [29]. In the following we discuss a few selected indexes, which are most related to the ones studied in this paper.

The Rtree [32] is a commonly used multidimensional index in the database community. It is a balanced data structure based on axis-parallel bounding boxes and can manage both point and extended objects. It is thus a natural choice for indexing Local Regions for function approximation. Several variants of the Rtree have been proposed, e.g., R*-tree [3], $R+$-tree [62], and Xtree [6]. The goal of most improvements is to reduce the overlap of bounding boxes in tree nodes, which is a major factor in degrading performance for high-dimensional data. The SS-Tree [74] takes this a step further by using spheres as bounding regions. It is therefore a good candidate for managing spherical or ellipsoidal regions of accuracy.

To avoid overlap of bounding regions, some index structures partition the space, e.g., the Binary Space Partitioning (BSP) tree [29]. The Binary Tree used in our experiments is an adaptation of this index structure.

It has been shown that in high dimensions linear scans are sometimes faster than complex index structures, especially when data is accessed on disk. The VA-file [73] improves the performance of linear scans by quantizing the space. A simple approach based on scanning files at different resolutions has been shown to outperform sophisticated bulk-loaded Rtrees [56].

The Random Projection Rtree in this paper was motivated by work on us-

ing projections for containment queries [14] and approximate nearest-neighbor queries [43]. Multidimensional problems can be mapped to lower dimensions by hashing [31, 38]. Other common approaches to combat the curse of dimensionality are dimensionality reduction and principal component analysis, e.g., used by the TV-tree [46], the Δ-tree [20] and the VA+ file [26].

1.7 Conclusions and Future Work

In this paper we introduced the function approximation problem. We showed its hardness and how it motivates an interesting indexing problem. The workload generated for the indexing problem posed two challenges. First, the workload introduces interesting tradeoffs that affect different index structures to varying degrees and therefore each index must be analyzed individually. Second, the performance of indexes can no longer be measured only in terms of traditional metrics like search and update cost. We addressed these challenges by providing a systematic framework to reason about the performance of indexes when used for function approximation. Our study shows that even simple indexes can be highly efcient, if they can be tuned predictably and can take advantage of domain knowledge.

CHAPTER 2

HIGH-SPEED FUNCTION APPROXIMATION

Learning methods for predictive models have traditionally focused on prediction quality and model building time, while prediction time (the time taken to make a prediction) is often ignored. However there is an increasing need for models that are not only accurate, but also make fast predictions. Some of the most accurate models like ensemble models are often too slow to be used in practice. We believe that exploring the tradeoff between prediction time and model accuracy is an exciting new direction for data mining research.

In this paper, we make a first step toward exploring this tradeoff. We introduce a new learning problem where we minimize model prediction time subject to a constraint on model accuracy. Our solution is a generic framework that leverages existing data mining algorithms while taking prediction time into account. We show a first application of our framework to a combustion simulation, and our results show significant improvements over existing methods.

2.1 Introduction

Predictive models, both for classification and for regression problems, play a major role in machine learning and data mining. After a predictive model is learned from a given set of training cases, it can be used to make predictions for

Portions, reprinted, with permission, from B.Panda, M. Riedewald, S. B. Pope, J. Gehrke. "High-Speed Function Approximation". *Published in the proceedings of the* 7^{th} *IEEE International Conference On Data Mining*.

new inputs. Traditionally, learning algorithms for such models have focused on improving prediction quality, e.g., measured by accuracy, root mean squared error (RMSE), area under the ROC curve and other metrics [15]. Research in data mining also considered model building time, i.e., to improve the time it takes to learn predictive models for large or high-dimensional data sets. However, there is another aspect of a predictive model, which is usually ignored by learning algorithms—*prediction time* — the time taken by the model to process an input and make a prediction. Let us describe a concrete application where prediction time is important.

High-dimensional function approximation (HFA) for combustion simulations was recently introduced by [54]. Scientists study how the composition of gases in a combustion chamber changes over time due to chemical reactions. The composition of a gas particle is described by a high-dimensional vector. The simulation consists of a series of time steps. During each time step some particles in the chamber react, causing their compositions to change. This reaction is described by a complex high-dimensional function, which, given a particle's current composition vector and other simulation properties, produces a new composition vector. Combustion simulations usually require up to 10^8 to 10^{10} reaction function evaluations. For most experiments, a single evaluation of the reaction function costs tens of milliseconds of CPU time on a modern PC. This makes running large scale simulations computationally infeasible. Scientists address this problem by building computationally less expensive models that approximate the reaction function within a user defined error tolerance of ϵ [55]. Our work is motivated by these specialized solutions for building models with low prediction time.

Combustion represents one of many physical phenomena studied by scientists using simulation methods. In most cases the mathematical model describing the phenomenon is complex, making it necessary to build approximate models that improve simulation runtime. Recently Bucila et al. [12] observed that ensemble models, while being the most accurate in many scenarios, are often too slow to be used in practice. In addition to scientific simulations, predictive models with low prediction time are also important for online transactions, financial forecasting, fraud detection and numerous other applications where it is important to be both fast and accurate. Building models for applications where prediction time is crucial is the focus of this paper.

One approach to reducing prediction time would be to concentrate on a given data mining model and its construction algorithm and modify them to take prediction time into account. This modification would have to be made for each model/algorithm combination, an arduous task. We instead propose a meta-learning framework that leverages existing data mining models and model building algorithms. The main idea is a local model approach, where we divide the domain of the learning problem into regions with associated data mining models. The search algorithms in our framework select appropriate regions and models across a large space of possible region/model configurations. Our work shows that this novel local model approach that uses different model types in different parts of the space can significantly reduce prediction time while maintaining high prediction accuracy. We make the following contributions.

- We introduce a new learning problem, *Low Prediction Time Learning*, with the goal to minimize model prediction time while maintaining a user-

defined model accuracy. (Section 2.2)

- We propose a generic framework for Low Prediction Time Learning. Our framework is application-independent and it is not limited to any particular model type or learning algorithm. (Section 2.3)
- We show how our ideas lead to significant speed-up for real simulation workloads. (Sections 2.4 and 2.5)

Section 2.6 discusses related work and Section 2.7 concludes the paper.

2.2 Problem Formulation

We formally define the Low Prediction Time Learning problem and then describe a detailed example, which illustrates several aspects that make the problem challenging.

Assume we are given a probability distribution $\mathcal{D}$ on R^m and two functions $f : R^m \rightarrow R^n$ and M : $R^m \rightarrow R^n$. Let X be a random variable that takes on values from R^m according to distribution $\mathcal{D}$. We say that M is an (ϵ, δ)-approximation of f with respect to $\mathcal{D}$, if on expectation at least $1-\delta$ fraction of points are within ϵ of the true function value, i.e.,

$$\mathrm{E}[\mathrm{I}(X)] \geq 1 - \delta, \tag{2.1}$$

where $||$ is some metric and, I() is an indicator function such that for all $\mathbf{x} \in R^m$, $\mathrm{I}(\mathbf{x}) = 1$ if $||f(\mathbf{x}) - \mathrm{M}(\mathbf{x})|| \leq \epsilon$ and 0 otherwise. Also, let $c_{\mathrm{M}}(\mathbf{x})$ be the time taken by M to compute M($\mathbf{x}$).

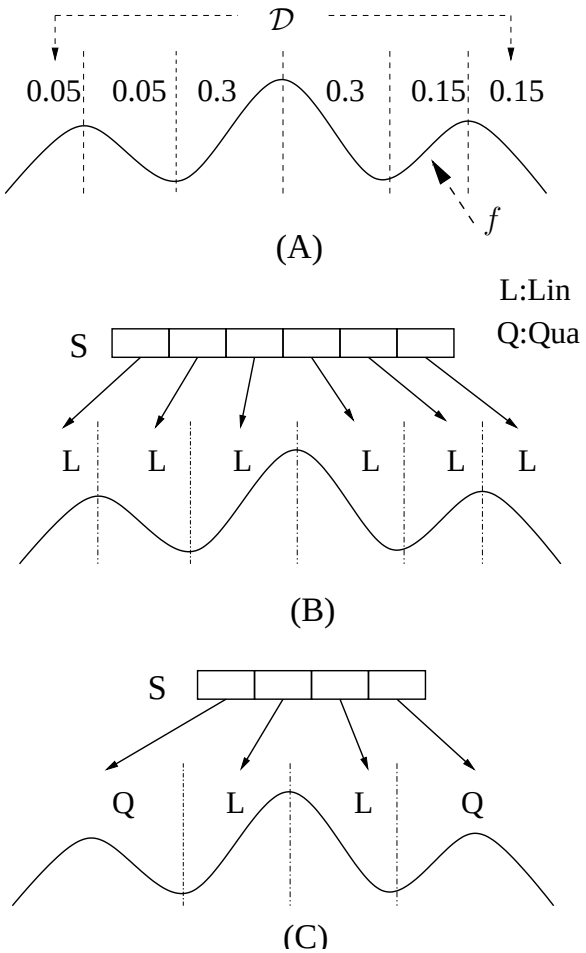

Figure 2.1: Example Of Tradeoffs

We can now define the *Low Prediction Time Learning Problem* as follows. Given a set $\mathcal{I} = \{(\mathbf{x}_1, f(\mathbf{x}_1)), (\mathbf{x}_2, f(\mathbf{x}_2)), \ldots, (\mathbf{x}_N, f(\mathbf{x}_N))\}$ find a function M (the *model*) such that M is an (ϵ, δ) approximation of f while minimizing

$$\text{ModelCost} = E_{\mathcal{D}}[c_{\text{M}}(\mathbf{x})]$$

We now describe a simple example to illustrate why Low Prediction Time Learning is an interesting problem. The example will also provide insights into the overall solution described in the next section. Suppose we want to approximate the one dimensional function f shown in Figure 2.1(A) within a specified

(ϵ, δ) error constraint for the distribution $\mathcal{D}$ shown in the figure. Further assume that we have a set of model types denoted by $\mathcal{M}$ that can be used to approximate the function. Let this set consist of polynomials up to degree 10, that is

$$\mathcal{M} = \{\sum_{i=0,n} a_i \cdot x^i | n = 0, 1, \ldots, 10\}$$

For simplicity, assume the cost of evaluating a polynomial of degree n is equal to the number of multiplication operations, i.e., it is $2n - 1$. (Note that one can compute x^i as $x^{i-1} \cdot x$, hence all powers of x up to the n-th can be obtained with $n - 1$ multiplications.)

Suppose the true function f is a polynomial of degree 10. Then it is clearly possible to approximate f with a polynomial of degree 10 with (ϵ, δ) error. If we approximate f using a polynomial of degree 10, then the model will take 19 time units per prediction.

Observation 1: Assume f can also be approximated within (ϵ, δ) by a 6th-degree polynomial. This reduces model cost to 11 time units per prediction.

Observation 2: Assume further that polynomials of degree less than 6 do not approximate the function well in all parts of the domain. However, lower degree polynomials may work well in some parts of the space. For example, in part (B) of Figure 2.1 the function domain has been divided into 6 parts and a polynomial of degree 1 is fit in each part. Assume that for all points in a particular partition the linear model in that partition approximates the function within (ϵ, δ). Therefore, this set of linear models defines another model that overall satisfies the (ϵ, δ) constraint. However, now the prediction time is not just an evaluation of a polynomial, but actually involves two steps. Given a query point, we first have to find the partition that contains the point (*search time*) and then evaluate

the polynomial in the partition (*approximation time*).

In order to find a partition containing the query point, we need a search structure S on the partitions. In this example we use a simple linear list S as shown in part B of the figure. For a given query point, the list is scanned until the corresponding partition is found. For simplicity we assume that the search cost is equal to the number of list elements accessed. Hence for the overall prediction time we obtain on expectation $0.05 \cdot 1 + 0.05 \cdot 2 + 0.3 \cdot 3 + 0.3 \cdot 4 + 0.15 \cdot 5 + 0.15 \cdot 6 = 3.9$ units for search and 1 unit for evaluating the corresponding degree-1 polynomial, for a total cost of 4.9 units per query.

Observation 3: Part (C) of Figure 2.1 shows another partitioning of the function. In this case the first and the last partitions have polynomials of degree 2, while the second and third partitions have polynomials of degree 1. Using an argument similar to Observation 2, assume that this model also satisfies the (ϵ, δ) constraint and again we use a list S to search for partitions. In this case the average approximation time per query is $0.1 \cdot 3 + 0.3 \cdot 1 + 0.3 \cdot 1 + 0.3 \cdot 3 = 1.8$ time units and the average search time per query similarly is $0.1 \cdot 1 + 0.3 \cdot 2 + 0.3 \cdot 3 + 0.3 \cdot 4 = 2.8$ time units, resulting in a total prediction time of 4.6 time units per query.

The example illustrates several interesting tradeoffs for Low Prediction Time Learning.

- Observation 1 showed that at a particular error tolerance there may exist several models of different complexity that can approximate f. As the error tolerance is increased, simpler models can be used, reducing prediction time. We call this the *Accuracy-Prediction Time Tradeoff*.
- Observation 2 showed that there exists a tradeoff between search time and

approximation time. Fitting a polynomial of degree 6 resulted in a model with no search time but high approximation time. Partitioning the domain and using a linear model in each partition resulted in model with high search cost and low approximation cost. We call this the *Search-Approximation Time Tradeoff*.

- Observation 3 indicated that exploiting the Search-Approximation Time Tradeoff is challenging because there are many different ways to partition the function domain and build models for each part. In this simple example the difference in prediction times did not vary significantly between the two partitioning schemes, but for more complex functions it can be significant.

In the following sections we will develop a cost-model based optimization framework in order to find models that exploit both the tradeoffs described in this example.

2.3 Algorithmic Framework

Recall that in the example in the previous section, different partitionings of the input domain and using different model types in the partitions resulted in varying prediction times. In this section we formalize the approach and discuss how to explore the design space of possible regions and models.

2.3.1 Model Definition

A region-model M for a function $f : R^m \rightarrow R^n$ consists of a set of convex regions $\mathrm{R} = \{r_i | r_i \subseteq R^m\}$, stored in some search structure S, and a mapping Q of regions to standard data mining models such that $\forall r_i \in \mathrm{R} : \mathrm{Q}[r_i] = m_i$. Here m_i is an instantiation of a model type in $\mathcal{M}$, where $\mathcal{M}$ is a set of types of data mining models.

The search structure S supports a Lookup(S,**x**) operation that returns a region $r \in \mathrm{R}$ containing **x**. Given a query point **x** the prediction process consists of the following steps: (1) find $r = \mathrm{Lookup}(\mathrm{S}, \mathbf{x})$, (2) then select $m = \mathrm{Q}[r]$, and (3) compute prediction $m(\mathbf{x})$. We can now revisit the notion of an (ϵ, δ)-approximation of a function f with respect to a region-model. We say that a region-model M is a (ϵ, δ)-approximation of a function f if the following holds:

$$E_{\mathcal{D}}[||f(\mathbf{x}) - \mathrm{Q}[\mathrm{Lookup}(\mathrm{S}, \mathbf{x})](\mathbf{x})|| \leq \epsilon] \geq 1 - \delta.$$

Notice that there might be (R,Q) configurations where some query points are not covered by any of the regions in R, i.e., Lookup(S,**x**) returns no result. To handle this, we assume the existence of a ground truth model of function f, which would be evaluated for such query points. The ground truth model returns $f(\mathbf{x})$ for any $\mathbf{x} \in R^m$ at some (high) cost C. For scientific simulations, this ground truth model is usually a differential equation solver. For traditional machine learning prediction problems this could be a highly accurate, but expensive ensemble model. If such a ground truth model does not exist, we can still apply our approach by simply setting $C = \infty$.

As described earlier, the prediction time per query consists of two costs: search time and approximation time. Let $\mathrm{s_S}(\mathbf{x})$ be the time taken by Lookup

to find a region r containing $\mathbf{x}$ using search structure S. Similarly, let $\mathrm{a}_m(\mathbf{x})$ be the time taken to compute an approximation using model $m = \mathrm{Q}[r]$. Then the expected total prediction time per query can be written as ModelCost $=$ $E_{\mathcal{D}}[\mathrm{s}_{\mathrm{S}}(\mathbf{x}) + \mathrm{a}_{\mathrm{Q}[r]}(\mathbf{x})]$.

Important properties: We would like to point out some important observations about the model definition above. First, we do not impose any restrictions on what model types can be included in set $\mathcal{M}$ and what search structure S to use. Any predictive model (e.g. neural nets, decision trees, SVMs) that can represent parts of the target function could be used. Similarly, the search structure could be a spatial index, a point index with post-processing to take region extent into account, a simple list, or any other structure that supports lookup functionality. Second, the models in $\mathcal{M}$ need not be modified to be included in our framework. This way we can leverage existing techniques, without having to modify each technique individually. Third, "global models", i.e., those where a single model is learned for the entire function domain, are a special case of our model definition. For a global model search time is zero. Finally, it has been observed that models similar to ours may exhibit variance because of discontinuities at region boundaries, that is addressed using a more general mixture model framework [40]. We discuss ways to address this in Section 1.7.

2.3.2 Algorithms

Let $\mathcal{I}$ denote a set of input points with known function values. We partition this set into a training set ($\mathcal{T}$) and a validation set ($\mathcal{V}$) for model building. Generalization error and model cost (ModelCost) will be measured on an independent

test set not used for model building.

An exhaustive exploration of all possible combinations of region partitioning, models used for each region, and index for managing regions, is practically infeasible. To reduce the complexity, we divide the problem into smaller subproblems. In particular, our algorithm has two major steps:

1. Generate a set of regions and find the best model for each region.
2. For each index structure under consideration, select the set of region-model pairs that minimizes expected prediction time for this index. Return the best solution.

These two steps that we call *Region-Model Candidate Set Selection* and *Region-Model Selection* are discussed in more detail below.

Region-Model Candidate Set Selection: Any subset of points in $\mathcal{T}$ could be connected as a candidate region, resulting in a number of regions exponential in the training set size. We therefore have to resort to heuristics for generating "the most promising" candidate regions. To reduce the search space, without being overly restrictive, we propose the following general approach. Assume we are given a set of relatively small regions, which we refer to as *base regions*. These base regions could be obtained from a regular grid partitioning of the space, from the leaves in a regression tree [11], or based on ISAT's regions of accuracy [55]. Notice that base regions do not need to be disjoint. We restrict region candidates to be either base regions or larger *derived* regions, which are the union of some base regions that are *near* each other. We will present a concrete algorithm in Section 2.4.

For each region under consideration, base region or derived, the next step is

to find a local model for that region. This is described in Algorithm 4. Using the points from $\mathcal{T}$ and $\mathcal{V}$ that lie in a given region r (called $\mathcal{T}_r$ and $\mathcal{V}_r$), the algorithm finds the lowest prediction time (t_m) model instantiation (m) from $\mathcal{M}$ that can be learned in the region and produces ϵ-approximations for at least $1-\delta$ fraction of the points in $\mathcal{V}_r$.

Two observations make the implementation of Algorithm 4 efficient. First, $\mathcal{T}_r$ and $\mathcal{V}_r$ for a derived region can be approximated by merging the corresponding lists from base regions. Second, it is common for more complex models to have higher prediction time. Rather, than trying all models in a region we sort $\mathcal{M}$ in increasing order of model complexity and iterate the list till a model satisfying the error constraint is found.

Region-Model Selection: The region-model generation algorithm produces a set with elements of the form $(r_i, m_i, \mathrm{t}_{m_i})$. We call this set of region model pairs RM. Notice that each of the models in RM satisfies the (ϵ, δ) error constraint for its region. Region-model selection involves selecting a subset of RM and initializing a model M (as defined in Section 2.3.1) that has lowest prediction time. Therefore, selection finds a model that minimizes $\sum_{\mathbf{x}\in\mathcal{V}}(\mathrm{s}_S(\mathbf{x}) + \mathrm{a}_{Q[r]}(\mathbf{x}))$. There are two important observations about this problem formulation.

- A selected subset of regions need not cover all points in $\mathcal{V}$. The ground truth model (Section 2.3.1) will be used to make predictions for such non-covered points. A ground truth model with approximation time of ∞ forces the selection algorithm to search for subsets of RM that completely cover the function domain.
- Algorithm 4 guarantees that every region-model pair in RM satisfies the (ϵ, δ) error constraint. However, if regions are allowed to overlap this does

not guarantee that the (ϵ, δ) error constraint will hold for a model M consisting of a subset of RM. In our experience having all regions satisfy the error constraint leads to tighter error for M. This is not surprising, because M will only have worse error for some corner cases. As the experiments show, in practice enforcing (ϵ, δ) for each region usually leads to better global error.

Several factors make the region-model selection problem difficult. First, lookup cost in a search structure depends on the properties of the regions it stores like their degree of overlap, extent, and orientation. If multiple regions in the search structure S contain a given query point, then approximation cost depends on the region-model pair that will be finally used in the prediction. These issues aside, we can show that even if we make very restrictive assumptions about the search time and approximation time of a query point, the region model selection problem is NP-hard.

Given the complexity of the selection problem, we use a greedy heuristic, shown in Algorithm 5. The algorithm starts out with an initial solution of base regions. This initial solution is biased toward high search cost and low approximation cost. In each step the algorithm replaces a set of regions in the current solution with a larger region from the set of candidate regions, such that the larger region covers all the removed regions. This is done greedily by selecting the region that brings about the largest reduction in prediction time. The algorithm stops when no more improvement is possible.

Notice that Algorithm 5 assumes the existence of a cost function ($\mathcal{C}$), which, given a set of region-model pairs and a validation set $\mathcal{V}$, returns the prediction time of the best model that can be created using the given region-model pairs.

Algorithm 4: Model Generation

Require: Training set $\mathcal{T}$, Validation Set $\mathcal{V}$, Region r, Model Set $\mathcal{M}$, Error ϵ, Error Rate δ

1: $\mathcal{T}_r = \{(\mathbf{x}, f(\mathbf{x}))|\mathbf{x} \in r \wedge (\mathbf{x}, f(\mathbf{x})) \in \mathcal{T}\}$
2: $\mathcal{V}_r = \{(\mathbf{x}, f(\mathbf{x}))|\mathbf{x} \in r \wedge (\mathbf{x}, f(\mathbf{x})) \in \mathcal{V}\}$
3: **for all** model types $\in \mathcal{M}$ in ascending order of complexity **do**
4: **if** model instantiation m using $\mathcal{T}_r$ exists **then**
5: Y=$\{(\mathbf{x}, f(\mathbf{x}))|(\mathbf{x}, f(\mathbf{x})) \in \mathcal{V}_r \wedge ||m(\mathbf{x}) - f(\mathbf{x})|| \leq \epsilon\}$
6: **if** $\frac{|Y|}{|\mathcal{V}_r|} > 1 - \delta$ **then**
7: **return** (m,t_m)
8: **end if**
9: **end if**
10: **end for**
11: **return** "No model found"

Finding such a cost function is challenging, because of reasons pointed out earlier. We will discuss this in more detail in the next section.

2.4 Instantiations

There are many ways to instantiate the above framework, differing in how base regions are generated and merged and the search structure used to store the regions. One can define a grid-based partitioning of the function domain [4], attempt to merge adjacent grid cells and use a search structure that performs a binary search along each dimension to find the cell the query point lies in. Another possible instantiation is a regression tree style partitioning of the function domain with a binary tree search structure. In this case the base regions

Algorithm 5: Greedy Region Selection

Require: RM, Validation Set $\mathcal{V}$, Cost function $\mathcal{C}$

1: Sol ($\subseteq$ RM) = $\{(r_i, m_i, \mathrm{t}_{m_i}) | r_i$ is a base region$\}$
2: Cost = $\mathcal{C}$(Sol)
3: **while** Cost improves **do**
4: TempSol={}
5: **for all** $(r, m, \mathrm{t}_m) \in S \wedge (r, m, \mathrm{t}_m) \notin$ Sol **do**
6: Rem = $\{(r_i, m_i, \mathrm{t}_{m_i}) | (r_i, m_i, \mathrm{t}_{m_i}) \in \text{Sol} \wedge r_i \subseteq r\}$
7: tSol_r = Sol + (r, m, t_m) - Rem
8: $\mathrm{tCost}_r = \mathcal{C}(\mathrm{tSol}_r)$
9: TempSol = TempSol $\cup(\mathrm{tSol}_r, \mathrm{tCost}_r)$
10: **end for**
11: **if** $\exists(\mathrm{tSol}_r, \mathrm{tCost}_r) \in$ TempSol s.t. $\mathrm{tCost}_r <$ Cost **then**
12: (Sol,Cost)=$(\mathrm{tSol}_r, \mathrm{tCost}_r)$
13: **end if**
14: **end while**
15: S= Regions in Sol, Q= Region-Model map for Sol
16: **return** S,Q

correspond to the leaf nodes of a regression tree (T) like CART [11]. The region merge process could then attempt to merge a subtree of T into a single region with a more complex model. Intuitively the selection algorithm would prune away subtrees of T whenever it is cheaper to use the complex model in the merged region to make a prediction compared to traversing the subtree and using the simpler models in the leaves. For both the grid-based and the regression tree approach defining cost function $\mathcal{C}$ is fairly straightforward and we omit the details.

A third and more general instantiation is to treat each individual point in $\mathcal{I}$ as a base region and define a merge that creates regions enclosing the 1, 2,..., n nearest neighbors of a point. In this case the set of regions can have arbitrary shape, size, overlap; and the search structure (S) can be any high dimensional index. We discuss a variation of this idea for the combustion simulation where scientists build models with flexible region definitions.

2.4.1 Simulation Instantiation

The ISAT algorithm used by the domain scientists [54] approximates the combustion reaction function by a set of (possibly overlapping) high-dimensional ellipsoids with linear models inside these ellipsoids. These regions are obtained based on selective evaluations of the reaction function, which is the ground truth model for this application.

To ensure that the ellipsoids satisfy the model definition in Section 2.3.1, we use a slightly modified version of the algorithm described in [54]. The main modification is a stricter error control mechanism that periodically checks existing regions in the model and updates region boundaries to not include parts of the space where the model is producing poor approximations. Studies also indicated that hyper-rectangular regions work at least as well as ellipsoids, we will therefore use hyper-rectangular base regions. In the remainder of the paper, this modified algorithm is referred to as the ISAT algorithm.

Domain scientists also observed that their long-running simulations ($> 10^9$ queries) almost always have the following two properties. First, the future query distribution of the simulation can be fairly accurately estimated after a

few million queries. Second, simulation time is dominated by model prediction time, i.e., model construction and maintenance time are negligible. We describe the instantiation of our framework for such simulations.

Without loss of generality we model the simulation as a 2-phase process. During the first phase (a few million queries) the ISAT algorithm is run. This algorithm produces a set of base regions in the function domain with a similar model in each region. In order to create this set of region-model pairs, the ISAT algorithm has to evaluate the reaction function for some query points. These points will be used as the training and validation data for our technique ($\mathcal{I}$). At the end of the first phase we apply our framework using $\mathcal{I}$ as the input data set and build a new model optimized for prediction time. This model is used for the rest of the simulation. Long-running simulations need not have exactly two phases; in that case the above procedure can be repeated periodically. Note that the framework instantiation for the combustion simulation can also be applied to improve prediction time in a *traditional supervised learning model*, using the training data explicitly provided.

Our instantiation for the combustion problem starts with the set of regions created by the ISAT algorithm during phase one as the base regions. Larger regions are created by merging a base region with its nearest neighbors. Specifically, for each base region r, we add the following derived regions: r merged with its first nearest base region, r merged with its two nearest base regions, and so on until some upper limit n of neighbors. Duplicate derived regions are eliminated. Since the base regions are hyper-rectangles, we define a derived region as the smallest bounding hyper-rectangle of the merged base regions. Conceptually, we do not need to use ISAT's regions as base regions, and could

use individual points in $\mathcal{I}$ as base regions instead. However, if cardinality of $\mathcal{I}$ is large this would make nearest neighbor search costly.

Having defined the region creation process, the next step is to find models for each region (Algorithm 4). We now turn our attention to the major challenge for the next step—defining cost function $\mathcal{C}$.

Cost Function ($\mathcal{C}$): For high dimensional indexes, it is difficult to accurately estimate the search cost of a query just based on the set of regions to be stored, without actually building the index. Unfortunately, building the index for each iteration of the greedy region selection algorithm (step 8 in Algorithm 5) is very expensive.

We discuss cheaper alternatives for selected index structures. Due to space constraints we omit implementation details. The main idea is to take advantage of two properties of the problem. (1) The selection process picks region-models from a fixed set and optimizes the solution for a fixed set of points ($\mathcal{V}$). Hence we can precompute information like the subset of $\mathcal{V}$ in each region. (2) At each step the algorithm leaves most of the solution unchanged and only replaces a small set of regions with a single larger region. We can leverage this property for incremental computation.

Random List stores regions in a simple list. The lookup operation scans the list from the beginning until a region containing the query point is found. While lists are not sophisticated index structures, linear scans are known to perform well for disk-based accesses in high dimensions [73] and also as in-memory data structures for combustion simulations [54].

Different orders of regions in the list will result in different prediction costs.

Given a set of regions it is infeasible to try all possible orders to find the best one. The idea behind the random list approach is to compute and minimize the expected cost assuming all region orders are equally likely, and then to pick the best order for the set of regions with the lowest expected cost.

Given a selected set of region model pairs of size |S|, the cost function computes $\sum_{\mathbf{x}_i \in \mathcal{V}} (\frac{|S|}{f_i+1} + Avg(\mathrm{t}_{m_1} \ldots \mathrm{t}_{m_{f_i}}))$. The intuition for the formula is as follows. For a set of regions, if a query point lies in multiple regions, then in any random order of the list it is very likely that a region containing the query point is found early. Therefore, the search time for a query point is approximated as $\frac{|S|}{f_i+1}$ where f_i is the number of regions that query point $\mathbf{x}_i$ lies in. The approximation time for a query point is simply the average of the cost of the models in the regions that the query point lies in (each one is equally likely to be found first in the list). After the selection algorithm finds a set of regions with the lowest expected cost, we try a few different sort orders of these regions and pick one with the lowest cost.

MFU List: In practice it is often a good heuristic to store the most frequently queried regions in front of the list. This strategy is called Most Frequently Used (MFU). Notice that this need not be an optimal order, because the model in a frequently accessed region might be expensive and the query point might also be covered by a region with a cheaper model later in the list. We use the validation set $\mathcal{V}$ to estimate the fraction of future queries that will fall into a given region.

In a MFU list the order in which a set of regions will be stored is known and therefore search and approximation cost for all query points can be accurately computed. In this case an efficient implementation exists by first sorting

all candidate regions in RM according to the number of points in $\mathcal{V}$ that they contain.

RTree: For hierarchical indexes like the RTree, it is known that finding accurate cost models for high-dimensional data is very difficult [44]. Fortunately, for our technique we do not need absolute costs, but rather an estimate of the net benefit of merging a set of regions into a single region. In this section we propose a fairly simple and robust heuristic that can be used for optimizing any index structure which prunes search space by building a hierarchical structure on the set of regions being indexed. We describe the heuristic for the RTree [32], a popular index for spatial data. One can develop more accurate cost models for different index structures but our aim is to show that even a simple heuristic works well for improving model prediction time. More sophisticated cost models can be easily plugged into our algorithm (Line 8 in Algorithm 5).

The RTree is a balanced tree structure. Nodes in the tree correspond to hyper-rectangles in the data space. If the tree indexes hyper-rectangles, a leaf node stores actual data objects (up to a specified maximum), while a non-leaf node stores the minimum bounding box of hyper-rectangles in its subtree. During a search, all subtrees whose bounding boxes contain the query point are examined, hence the search cost is determined by number of hyper-rectangles examined till a data object containing the point is found, often called the *false positive rate* of an index. A tree can have a non-zero false positive rate because in high dimensions it is difficult to partition objects well, causing the bounding boxes of non-leaf nodes to overlap. This results in multiple search paths in the tree for a given query point and some paths may not have a data object containing the query point (hence false positives). Our goal is to estimate the reduction in false

positives for queries if a region merge is done in Lines 7 and 8 of Algorithm 5. This cost reduction has to be compared with the cost increase associated with a more complex model in the larger merged region.

We estimate the benefit of merging as follows. Assume the RTree on average has k false positives for a query. Since RTrees (and any hierarchical index) tend to cluster nearby objects, all false positives of a query tend to be in the neighborhood of the query. Hence, if we merge some neighboring regions, then nearby query points will see a reduction in their false positive rate because some of their false positives have been merged. We estimate this reduction in false positives by defining a neighborhood around the merged regions, such that it contains all queries that are affected by the merge.

The order in which these affected queries will access regions in the tree depends on the actual tree layout. Lacking further knowledge, we assume that all regions in the neighborhood are accessed in some random order. Hence we use the random list cost model (see above) to estimate the benefit of a region merge in the affected neighborhood. The main challenge is to select the correct neighborhood. We define the neighborhood by selecting a small number of nearest neighbors, parameterized by γ, of each region that participates in the merge. Details on γ and the performance of the heuristic are described in the experiments.

2.5 Experiments

As a proof of concept, we implemented and tested our approach for the combustion simulation application. We use libraries and data from a Hydrogen+Air

Table 2.1: Notation For Experiments

Name	Description
ISAT	ISAT algorithm
Opt	Proposed optimization algorithm
C	Constant model
L	Linear model
Q	Second order model
|S|	Index size grouped by model type
k	Average number of false positives
Obs δ	Observed δ on test set
Search Time	Total cost of index lookup
Approx Time	Total cost of model evaluation
Total Time	Total prediction time
StdDev	Standard deviation of total time

simulation provided by the authors of [54]. The dataset comprises 5 million simulation query points. Each query point is a 10 dimensional composition vector. The reaction function that describes the simulation in this case is a high dimensional function $f : R^{10} \rightarrow R^{11}$.

The overall setup is as follows. We run the ISAT algorithm on the first 3 million query points to generate the base regions and training/validation data set $\mathcal{I}$, which are used by our algorithm as discussed in Section 2.4.1. A random sample of $\approx 2 \times 10^5$ query points from the last 2 million queries is used as an independent test set. We compare total simulation time on the test set against the original ISAT model as it is currently used by the domain scientists.

Table 2.2: Results For Hydrogen + Air Simulation

ExptNo:(S, ϵ)	Method	$\mathcal{M}$	$\lvert\mathrm{S}\rvert$	k	Obs δ	Search Time(ms)	Approx Time(ms)	Total Time(ms)	StdDev (ms)
1: $(\mathrm{RL}, 5 \times 10^{-3})$	ISAT	L	L:63	26	0.01	623	337	960	68
	Opt	L,Q	L:28,Q:9	6	0.005	114	434	548	-
	Only S	L,Q	L:26,Q:41	1	0.0002	84	1750	1834	-
2: $(\mathrm{RL}, 5 \times 10^{-5})$	ISAT	L	L:2263	977	0.05	20477	383	20860	2983
	Opt	L,Q	L:1430,Q:332	122	0.01	2071	1620	3691	-
3: $(\mathrm{MFU}, 3 \times 10^{-3})$	ISAT	C	C:2226	113	0.08	2367	93	2460	-
	Opt	C,L	C:1362,L:115	19	0.003	414	342	756	-
4: $(\mathrm{RTree}, 3 \times 10^{-3})$	ISAT	C	C:2226	212	0.11	15530	78	15608	819
	Opt	C,L	C:687,L:229	92	0.07	6751	266	7017	-
5: $(\mathrm{RTree}, 5 \times 10^{-5})$	ISAT	L	L:2263	166	0.06	12238	380	12618	1327
	Opt	L,Q	L:1986,Q:36	124	0.05	8927	385	9312	-

Table 2.3: Neighborhood Effect

γ	|S|	Avg Total Time(ms)	StdDev(ms)
ISAT	2226	15608	819
0.004	1210	10584	1303
0.008	916	8350	736
0.012	**802**	**8050**	**950**
0.02	653	6151	667
0.03	555	5362	773

All experiments used a $70 - 30$ split of $\mathcal{I}$ into training ($\mathcal{T}$) and validation ($\mathcal{V}$) set and $\delta = 0.1$. For each base region, 8 derived regions are created by merging the base region with its 1,2,...,8 nearest neighbor base regions. For a fair comparison we use exactly the same data that ISAT uses for model building. Notice that $\mathcal{I}$ usually is not exactly a uniform sample of the query points due to peculiarities of the ISAT algorithm. This puts our algorithm at a slight disadvantage, but overall we did not find significant differences between the distribution of $\mathcal{I}$ and the test set. All experiments were run on a Windows XP PC with a 2.79GHz processor and 8GB RAM.

2.5.1 Results

We ran simulations using different values of ϵ, index structures and model types ($\mathcal{M}$). Table 2.2 summarizes the results; variables are explained in Table 2.1. All measurements are on the test set and times reported are in milliseconds, rounded to the nearest integer.

Experiment 1 is for $\epsilon = 5 \times 10^{-3}$ and the Random List (RL) index. ISAT built regions with linear models (L)[1] and our framework used both linear and quadratic (Q) models. ISAT created 63 regions. Since index size and search cost are small in this case, our method (Opt) does not merge many of the linear regions into quadratic ones (only 9). Nevertheless a significant reduction in prediction cost by $\approx 30\%$ is achieved. The increase in approximation cost (some query points are approximated using quadratic models) is offset by the decrease in search cost. Recall that our algorithm for a random list tries a few random orders and returns the best as the solution. For ISAT there is no optimization algorithm for selecting the best list order, therefore we report average cost across 30 different random sort orders and standard deviation.

To show that both approximation and search cost must be considered for prediction time optimization, we repeated Experiment 1 using a simpler optimization goal—only minimize search cost (”Only S”). In this case the selection algorithm aggressively merges regions to cover validation points with the smallest number of derived regions containing quadratic models. As the results show, the additional decrease in search cost is not significant enough to offset the higher approximation cost. A surprising observation in this experiment is that the number of regions created by ”Only S” is greater than for ISAT, even though the selection algorithm usually *replaces* a set of regions with a larger region. This happens because it is possible to select a candidate region that does not completely contain any regions in the current solution, but significantly overlaps with a lot of them (i.e., Rem={} in Line 6 and 7 of Algorithm 4). Adding such a region increases list size but may still reduce expected search cost per query as some query points now are covered by multiple regions (recall that

[1]ISAT always uses the same model in every region; specified when the simulation starts.

search cost=$\frac{|S|}{f_i} + 1$).

Experiment 2 uses the same setup as Experiment 1 but with $\epsilon = 5 \times 10^{-5}$. As ϵ is stricter, it is not surprising that ISAT creates a larger number of regions and hence search cost dominates prediction time. Opt in this case more aggressively selects regions with quadratic models, causing the approximation time to increase significantly. An even larger decrease in search time results in $\approx 70\%$ improvement in total time.

When we repeated Experiments 1 and 2 for the MFU List, Opt did not merge any regions and simply continued to use the base regions created by ISAT. The reason is the skewed distribution of query points over base regions. The first few regions in the list account for the vast majority of accesses, resulting in very low search cost. Hence for the MFU List the benefit of merging regions would be too low to offset the higher approximation cost of a quadratic model in a merged region. Stated differently, if the search cost is low, then it is preferable to stay with the simplest models in each region.

To show more clearly that Opt makes the right decisions even for the MFU List, we performed **Experiment 3** using a MFU List and $\epsilon = 3 \times 10^{-3}$, but this time setting ISAT to produce base regions with *constant models* (C). Now Opt chooses between linear and constant models. Because constant models lead to smaller base regions (to guarantee the error), the list has now more elements and hence higher search cost. Again Opt automatically makes the right choice to merge regions into larger ones with linear models, significantly improving cost.

Experiments 4 and 5 report results for the RTree index. In Section 2.4.1 we

introduced parameter γ to control the affected neighborhood size of a region merge. We use a simple heuristic to set γ. First, an RTree is built from the base regions. Points in $\mathcal{V}$ are queried using the tree and the average number of base regions probed per query (k_l) is recorded. Based on the assumption that if on average k_l leaves are scanned per query, this corresponds to a random list of size $2 \cdot k_l$ being examined, we set γ such that at most[2] $2 \cdot k_l$ regions are affected by a merge, when the current solution has only base regions. Once initialized, we do not change γ. Hence affected neighborhood size decreases with index size.

Using this heuristic and otherwise the same setup as Experiment 3, in **Experiment 4** Opt shows $\approx 50\%$ improvement over ISAT. Our RTree implementation [3] uses a one-by-one insertion scheme and different insertion orders can lead to slightly different RTrees. Therefore, for ISAT we report average measurements and standard deviation in total runtime across 10 different insertion orders. Opt uses its cost model to select the best among a few different RTree insertion orders.

Experiment 5 uses the same setup as Experiment 1, but with an RTree. The improvement in runtime is comparably small, suggesting either that linear models are good or poor choice of γ. Notice that even though approximation time remains almost unchanged, search cost actually decreases. This can be explained by regions that contain very few query points next to heavily accessed regions. Merging the lightly accessed regions does not change approximation cost. But it does help reduce search cost for the heavily accessed region, because less false positives are encountered.

As we mentioned earlier, the goal of this paper is not to develop the most ac-

[2]We say "at most" because we scale γ according to the size of the merge. Larger merges affect larger neighborhoods.

curate cost models for high dimensional indexes. Rather, we wanted to provide a proof of concept that pursuing optimization of prediction time is worthwhile. More accurate cost models can easily be leveraged in our framework. However, we end the discussion here with a micro benchmark that shows that the proposed simple Rtree cost model is robust (i.e., not sensitive to γ). Table 2.3 shows index size, average runtime and standard deviation (across 10 different insertion orders) for Rtrees optimized using different values of γ. For instance, $\gamma = 0.012$ for an index of size |S| implies $0.012 \cdot$ |S| nearest neighbors of each replaced region are assumed to be affected be a region merge. These results are for the setup of Experiment 4, with the line in bold face representing the default γ value used in that experiment. Results were similar for Experiment 5, hence are not reported here explicitly.

The first conclusion from the results is that index size decreases with increasing γ. This is expected since a larger neighborhood size implies that the tree is expected to have a larger false positive rate and hence our algorithm predicts more cost savings by merging regions and using complex models. While it is clear from the table that total time is not very sensitive to γ, in this case it tends to improve as γ increases. This is an artifact of the setup and happens because here RTree search cost far exceeds approximation cost. As a result, Opt uses linear models in most regions. As we increase γ, Opt merges more regions. But this only insignificantly increases approximation cost, because most regions are already linear for smaller γ. However, search cost may still decrease significantly.

Discussion. Our experiments show that different indexes and model types work well in different simulation settings. Our proposed method (Opt) cor-

rectly and automatically captures the tradeoffs in the problem and effectively adapts the model to the index and simulation parameters. Our method does not improve runtime at the cost of degrading prediction quality (see δ values in Table 2.2). In fact, in most cases Opt produces a δ value better than the original ISAT model. This is because our algorithm performs robust error control by checking each region-model pair before admitting it as a candidate for region selection. ISAT on the other hand only randomly checks regions for error. Finally, in order for the method to be useful in practice, it should not generate a significant computational overhead for the simulation. In all our experiments the cost of the optimization algorithm was negligible compared to the total cost of a long-running simulation as used by the domain scientists.

2.6 Related Work

Closest to our work is recent work on model compression [12] where an ensemble model is approximated with a neural network to improve prediction time. Robot motion planning algorithms developed techniques to optimize prediction time in local regression models [58]. However, none of the prior work formalizes the learning problem and examines the various tradeoffs we discuss.

There is lot of work on local models. Instance based learning [50] is a special class of local models where rather than explicitly defining regions, function values at unknown points are interpolated from neighboring training samples. A regression tree [11] creates regions in the function domain. Regression trees are often pruned for accuracy [11], and our framework when applied to a regression tree uses the same idea for improving prediction time. No work has

focused on optimizing regression trees for prediction time. [16, 42] propose new split criteria for accuracy, [42, 66] use complex models in the leaves (again for accuracy), [16] assumes that shorter trees are easy to interpret and always creates the most complex model for a subtree, [27, 40] propose methods to reduce variance in regression tree models, and [24] optimizes tree construction costs. We optimize and build a more general class of models and the regression tree is only an instance of a model in the class.

Existing techniques in the combustion community use region-models that differ in the types of regions and models used. ISAT [55] uses ellipsoids with linear models, PRISM [4] uses a grid partitioning with polynomials and [18] uses self organizing maps to define regions and neural networks as models. [54, 68] focus on finding good indexes for local models built by ISAT. However, no work in this community addresses search and approximation costs together.

Numerous methods have been proposed for finding cost models for high dimensional index structures. Most of the analysis is for disk-based indexes and point data only. [1, 5, 41, 72] predict the query cost in RTree style index structures by assuming a uniform data distribution to estimate size of index pages on disk and the approximate number of pages a query accesses. Since the uniform distribution assumption usually does not hold in high dimensions, [25, 52] model the data using global parameters like fractal dimensionality, while [19, 51, 64] partition the data and model data in a partition independently. [44] shows that the parametric models perform badly with increasing dimensionality and proposes predicting index costs from a smaller index built using a subset of the data.

2.7 Conclusions

We introduced and formalized the low prediction time learning problem. We proposed a general framework that leverages existing data mining models to minimize model prediction time and used it to significantly speed up a scientific application. Understanding how existing data mining models can be optimized for prediction time is an interesting direction for future research.

Future directions include reduction of the model variance using the overlap among regions (recall our remark from 2.3.1) and periodic application of our framework for long running combustion simulations.

CHAPTER 3

FAST SUMMARIES OF COMPLEX MODELS

Modern science is collecting massive amounts of data from sensors, instruments, and through computer simulation. It is widely believed that analysis of this data will hold the key for future scientific breakthroughs. Unfortunately, deriving knowledge from large high-dimensional scientific datasets is difficult. One emerging answer is exploratory analysis using data mining; but data mining models that accurately capture natural processes tend to be very complex and are usually not intelligible. Scientists therefore generate model summaries to find the most important patterns learned by the model. Generating model summaries creates serious data management challenges: Scientists usually want to analyze patterns in different "slices" and "dices" of the data space, comparing the effects of various input variables on the output. We propose novel techniques for efficiently generating such model summaries. Our techniques leverage commonalities on two levels—per-summary and between multiple summaries. We also propose a scalable implementation of our techniques in a MapReduce framework. For both sequential and parallel implementation, we typically achieve speedups of one or more orders of magnitude, while guaranteeing the exact same results as the naive approach.

3.1 Introduction

Powerful learning methods like tree-based ensembles, SVM's, and artificial neural nets can be used for training highly accurate prediction models even when there is only a limited understanding of the underlying process that generated

a given data sample. The models are flexible enough to model complex interactions between many variables and they can handle large datasets. This makes them ideal candidates for *exploratory analysis* in scientific and business applications [35]. During exploratory analysis users want to understand "what the model has learned" about functional dependencies between different input variables and the output. This is important for two reasons. First, it provides insights about the data being modeled. Second, it helps serve as a sanity check to see if the model has the expected behavior. Unfortunately, for non-trivial datasets models tend to be complex and often difficult to understand. Even single decision trees become unintelligible once they have thousands of nodes.

One of the most popular approaches used by scientists to understand high-dimensional functions, including prediction models, is through *low dimensional summaries* [28, 37, 47, 35]. Consider the following example from bird ecology. A scientist wants to discover which variables strongly affect occurrence of endangered wild bird species. She has access to millions of records about bird sightings, each with hundreds of attributes describing time and location of the sighting and location properties like climate, vegetation, land cover, elevation, and so on. Such data is available through the Avian Knowledge Network (AKN) at `www.avianknowledge.net`. Using the available data the scientist trains a good model for predicting occurrence of bird species. Now she wants to see a low dimensional summary for how elevation alone, affects occurrence. To do this, the scientist computes a set of summaries that average out from the model the effects of all attributes other than elevation, and visualizes the results as shown in Figure 3.1. Each graph in the figure shows how elevation alone affects the model response and the different graphs correspond to different ways of averaging out the effects of other attributes. We discuss the exact method used for

generating these summaries in Section 3.2.

In general, scientists like to explore large collections of model summaries by either searching for specific patterns or using an OLAP-style [17] analysis, i.e., examining different attributes in many "slices" and "dices" of the data space. In practice, though, there is a high computational cost associated with generating the model summaries. This forces them to limit their attention to a small number of coarse summaries. Even producing a single model summary using the existing approaches can be very expensive, because they require the model to be evaluated at a large number of combinations of attribute values. Exploring this challenging data management problem that arises when generating large sets of low dimensional summaries for complex data mining models is the focus of this paper. We propose novel techniques for making the generation of these summaries computationally feasible. The main idea is to leverage inherent structure in summary computation workloads, both within a summary and between different summaries. We show that by careful materialization and precomputation, one can avoid duplicate computation and dramatically reduce computation cost. This paper focuses on tree-based data mining models. We typically achieve a speedup of one or more orders of magnitude over existing approaches, without sacrificing the quality of the estimates produced. Specifically, we make the following contributions:

- We introduce the *summary computation* problem for complex data mining models and identify common structure in the workloads for generating model summaries. (Sections 3.2 and 3.3)
- We propose algorithms that take advantage of the structure for speeding up summary computations in tree-based models, including state of the art

bagged and boosted tree ensembles. (Sections 3.4, 3.5 and 3.6)

- We evaluate our algorithms using models built from real world data and show that our proposed algorithms provide impressive speedups in computing large sets of summaries. (Section 3.7)

Section 3.8 discusses briefly some extensions to our algorithms. Section 3.9 discusses related work and Section 3.10 concludes the paper.

3.2 Generating Model Summaries

We introduce the model summaries that scientists would like to generate for complex data mining models. Our goal is to support efficient computation for all of them. In this section we will slightly abuse notation by using the same symbol for both an attribute name and the set of all values of this attribute. In particular, we will write $y \in \mathrm{Year}$ to state that y is a value of attribute Year.

3.2.1 Example

Assume a scientist has trained a model $F(\mathrm{Elev}, \mathrm{Year}, \mathrm{Hpop})$, which for a given combination of elevation ($e \in \mathrm{Elev}$), year ($y \in \mathrm{Year}$), and human population density ($h \in \mathrm{Hpop}$) can accurately predict the probability of observing some bird species of interest. Now she wants to visualize the effect of elevation, i.e., how does the probability of observing the bird species of interest depend on elevation. In this example, and in the remainder of this paper, we will refer to the attributes that the scientist wants to visualize with a model summary

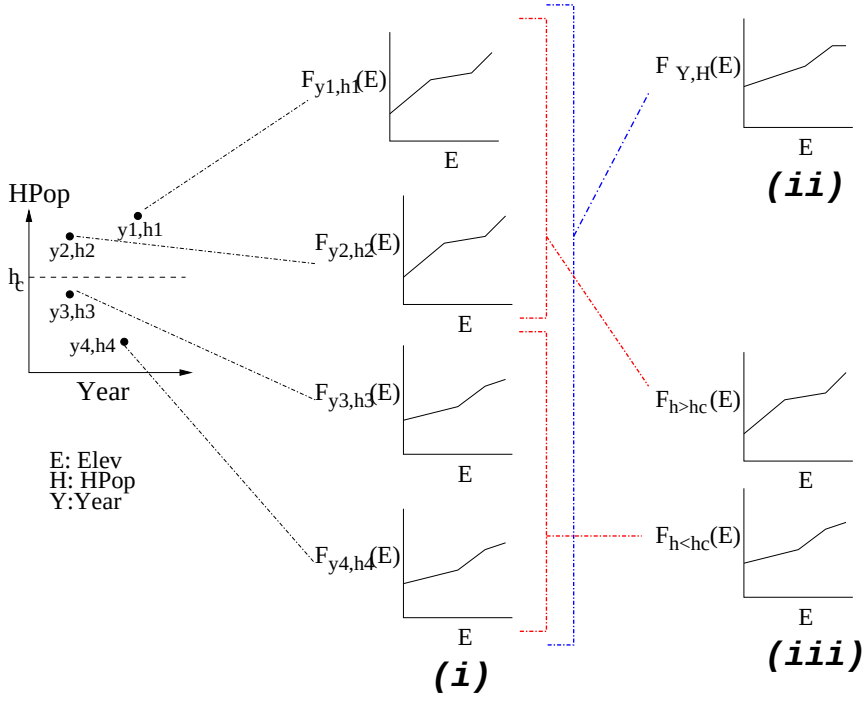

Figure 3.1: Summary Generation

as the *summary attributes*. The remaining attributes in the model are the *non-summary attributes* of this summary. We will also use the terms *attribute* and *variable* interchangeably throughout the paper.

In our elevation analysis example, elevation (Elev) is the summary attribute/variable and year (Year) and human population density (Hpop) are the non-summary attributes/variables. Intuitively, a visualization of the summary will show Elev values on the x-axis, while Year and Hpop do not appear as variables in the summary (see Figure 3.1(i)).

Similarly, if the scientist wanted to study the combined effect of Elev and Hpop on bird occurrence, she would choose these two as the summary attributes, while Year would be the non-summary attribute. The corresponding summary plot would show a two-dimensional function surface.

There is no perfect way of summarizing a high-dimensional function with a lower-dimensional function. Some information will inevitably be lost, no matter which method we choose. However, all accepted methods for summarizing the effect of a given subset of the input variables follow the same fundamental principle of experimental design: *To study the dependency of the output on a variable, one varies only this variable while holding the other variables constant.* Otherwise one would see the mixed effect of varying multiple variables together.

3.2.2 Generating Individual Summaries

Studying the effect of Elev while holding Year and Hpop constant means that we are computing a function $F_{y,h}(\mathrm{Elev}) = F(\mathrm{Elev}, y, h)$, where $y \in \mathrm{Year}$ and $h \in \mathrm{Hpop}$ are a selected year and human population density value. Intuitively, function $F(\mathrm{Elev}, \mathrm{Year}, \mathrm{Hpop})$ is conditioned on a given pair of year and human population density values. The elevation effect might be different for different years and human population densities. Hence scientists usually compute $F_{y,h}(\mathrm{Elev})$ for many different (y, h) pairs.

Assume the scientist wants to compute $F_{y,h}(\mathrm{Elev})$ for a given (y, h) pair and a set of elevation values $\{e_1, e_2, e_3\}$. She would have to evaluate the data mining model for tuples (e_1, y, h), (e_2, y, h), and (e_3, y, h); the corresponding summary of the elevation effect would then consist of the pairs $(e_1, F(e_1, y, h))$, $(e_2, F(e_2, y, h))$, and $(e_3, F(e_3, y, h))$. For another (y, h) pair, she would similarly compute the corresponding summary of the elevation effect. Figure 3.1(i) illustrates the idea. Each plot shows the effect of elevation conditioned on specific values of (y, h). By choosing an appropriate set of (y, h) pairs, the scientist can

obtain a very detailed picture of how elevation might affect bird occurrence. We discuss popular choices next.

3.2.3 Selecting Non-Summary Attribute Values

The (y, h) pairs at which the elevation summaries are computed can be selected in various ways. Often scientists prefer to sample them uniformly from the input dataset, in particular from those data points that were used for training the model. Choosing non-summary attribute value-combinations that actually occurred in the training data reduces possible extrapolation effects from using the model in very sparse regions of the data space [37]. In our example, only (y, h) pairs for which a tuple with this year-human population density combination exists in the training data would be selected.

Choosing non-summary value combinations only from the training data has the drawback that the summaries are limited by biases in the input data. For example, if only a small fraction of bird sightings is reported from regions with very low human population density, then the set of elevation summaries will be dominated by those for (y, h) combinations with larger h. For some years, there might be no reports from very sparsely populated regions at all. To overcome these problems caused by the skewed distribution of non-summary attributes, scientists sometimes select the combinations of non-summary attribute values from a regular grid. In our example, they would select a set of Year values and a set of Hpop values, and then create an elevation summary for each pair in the cross-product of the two sets.

Naturally, whenever computationally feasible, the scientist can generate

summaries using either method and then compare the results.

3.2.4 Aggregating Summaries

Generating a separate summary for each selected combination of non-summary attribute values results in a very large number of summaries, making it difficult to examine and interpret all of them. Therefore, scientists often aggregate these summaries into a single one or a small set of visualizations. This is done by averaging the individual summaries.

In our bird ecology example, the summary of the effect of elevation, taking into account the average affect of year and human population density, is defined as $F_{\mathrm{Year,Hpop}}(\mathrm{Elev}) = \frac{1}{k}\sum_{i=1}^{k} F_{y_i,h_i}(\mathrm{Elev})$. Here the (y_i, h_i) are the selected combinations of year and human population density. Figure 3.1(ii) shows an example. When the (y_i, h_i) pairs are picked by sampling uniformly from the training data, then the summary $F_{\mathrm{Year,Hpop}}(\mathrm{Elev})$ is called a *partial dependence plot* [28].

Aggregate summaries like partial dependence plots are popular not only because they compactly summarize the effect of the summary variables on the output. In some cases the aggregate summary shows the exact dependence. For example, assume $F(\mathrm{Elev}, \mathrm{Year}, \mathrm{Hpop}) = F_1(\mathrm{Elev}) + F_2(\mathrm{Year}, \mathrm{Hpop})$, i.e., F is the sum of two functions where one does not depend on Elev and the other not on Year and Hpop. Then, since $F_{\mathrm{Year,Hpop}}(\mathrm{Elev}) = \frac{1}{k}\sum_{i=1}^{k} F_{y_i,h_i}(\mathrm{Elev})$, we can easily derive $F_{\mathrm{Year,Hpop}}(\mathrm{Elev}) = F_1(\mathrm{Elev}) + \frac{1}{k}\sum_{i=1}^{k} F_2(y_i, h_i)$. Stated differently, the aggregate summary shows the exact dependence of F on Elev, which is $F_1(\mathrm{Elev})$, plus a constant [28].

In general, if F can be expressed as the sum of two functions, one only depending on summary variables and the other only depending on non-summary variables, then F is said to have *no interactions* between summary and non-summary variables.

Even if there are interactions between some summary and non-summary variables, an aggregate summary can still provide valuable information. For example, if $F_{y,h}(\mathrm{Elev})$ shows a certain trend for the vast majority of (y, h) combinations, then this trend will usually show in their average $F_{\mathrm{Year,Hpop}}(\mathrm{Elev})$. Hence the average summary still provides a good compact visualization of the set of individual summaries.

Also, rather than limiting oneself to the extremes—visualize each $F_{y_i,h_i}(\mathrm{Elev})$ individually or visualize the average of all of them —the scientist can explore average summaries for different subsets of (y_i, h_i) pairs. For example, if the scientist suspects an interaction between Hpop and Elev, then she can compute two summaries $F_{\mathrm{Hpop}<h_c}(\mathrm{Elev})$ and $F_{\mathrm{Hpop}\geq h_c}(\mathrm{Elev})$. The former is an average of all $F_{y_i,h_i}(\mathrm{Elev})$ for which $h_i < h_c$; the latter averages all $F_{y_i,h_i}(\mathrm{Elev})$ with $h_i \geq h_c$. Figure 3.1(iii) shows the resulting visualizations for the summaries.

The summaries that we described above are emerging as tools of choice in many applications [28, 35, 47]. The reason for their popularity is that they help a scientist visualize the effects of the model in different slices and dices of the attribute space. Since the summaries are primarily used in visualizations, they are usually computed over one and two attributes only. Summaries over larger subsets of attributes are difficult to visualize, but may also be computed to explore statistics like the range of F values or additivity among subsets of variables. We will introduce summaries and the resulting summary computation problem

more formally in the next section.

3.3 Problem Definition

Our goal is to efficiently compute a set of average summaries for a selected region of the data space. All these summaries are based on the same set of data points for their non-summary attribute values. The formal definition can be found below. For simplicity and without loss of generality, we will focus only on partial dependence plots [28] as our model summaries. All techniques generalize naturally to the other summary types discussed in Section 3.2.

Let $\mathcal{X} = \{X_1, X_2, \ldots, X_{|\mathcal{X}|}\}$ be a set of $|\mathcal{X}|$ attributes with domains dom_1, dom_2,..., $\mathrm{dom}_{|\mathcal{X}|}$, respectively. Let Y be the output with domain dom_y. Let $F : \mathrm{dom}_1 \times \mathrm{dom}_2 \times \cdots \times \mathrm{dom}_{|\mathcal{X}|} \rightarrow \mathrm{dom}_y$ be a data mining model which is trained on a given dataset $D = \{(\mathbf{x}(1), y(1)), (\mathbf{x}(2), y(2)), \ldots, (\mathbf{x}(|D|), y(|D|))\}$, where all $\mathbf{x}(i) \in \mathrm{dom}_1 \times \mathrm{dom}_2 \times \cdots \times \mathrm{dom}_{|\mathcal{X}|}$ and $y(i) \in \mathrm{dom}_y$. Model F maps $|\mathcal{X}|$-dimensional vectors $\mathbf{x} = (x_1, x_2, \ldots, x_{|\mathcal{X}|})$ of input attribute values to the corresponding output value $F(\mathbf{x})$.

Definition 3 *Let $\mathcal{X}$, F and D be as defined above and let $\mathcal{S} \subset \mathcal{X}$ and $\tilde{\mathcal{S}} \subset \mathcal{X}$ be such that $\mathcal{S} \cup \tilde{\mathcal{S}} = \mathcal{X}$ and $\mathcal{S} \cap \tilde{\mathcal{S}} = \emptyset$. The* partial dependence summary *of F on $\mathcal{S}$ is defined as*

$$\forall \mathbf{x}_{\mathcal{S}} \in \bigotimes_{X_j \in \mathcal{S}} \mathrm{dom}_j : \quad \hat{F}_{\tilde{\mathcal{S}}}(\mathbf{x}_{\mathcal{S}}) = \frac{1}{|D|} \sum_{i=1}^{|D|} F(\mathbf{x}_{\mathcal{S}}, \mathbf{x}_{\tilde{\mathcal{S}}}(i)) \tag{3.1}$$

where $\mathbf{x}_{\tilde{\mathcal{S}}}(i) = \pi_{\tilde{\mathcal{S}}}(\mathbf{x}(i))$ is the projection of the i-th data record in D on the attributes in $\tilde{\mathcal{S}}$.

While this definition is particular for partial dependence summaries, it easily extends to other ways of selecting the values of non-summary attributes. All we need to do is replace D with the appropriate set of non-summary value combinations. For example, if the scientist wants to explore summaries OLAP-style in a certain slice or dice of the data space, then D is replaced by some selection $\sigma(D)$. Individual non aggregate summaries correspond to the special case where $|\sigma(D)| = 1$. We will point out additional optimization opportunities for special cases of $\sigma(D)$ where appropriate. We can now define the summary computation problem as follows.

Definition 4 *Let $P = \{p_1, p_2, \ldots, p_{|P|}\}$ be a set of summaries ("plots"). Let $p_i.\mathcal{S} \subset \mathcal{X}$ be the summary attributes for p_i; and let $p_i.V_\mathcal{S} \subseteq \bigotimes_{X_j \in p_i.\mathcal{S}} \mathrm{dom}_j$ be a set of visualization points associated with summary p_i. The* summary computation problem *is to compute $F_{\mathcal{X}-p_i.\mathcal{S}}(p_i.\mathcal{S})$, at each visualization point $\mathbf{v}_{p_i.\mathcal{S}} \in p_i.V_\mathcal{S}$ and for all summaries in P.*

For ease of notation when talking about a single summary we will refer to its summary attributes as $\mathcal{S}$ and visualization points as $V_\mathcal{S}$ dropping the dependence on p.

In this paper we also address a version of the summary computation problem where the set of visualization points for a summary is not predefined, but rather generated in an online manner. This problem which we will refer to as the the *online version of summary computation* arises when a scientist after generating a coarse summary may want to refine it interactively by adding more visualization points.

Algorithm 6: Naive Algorithm For Summary Computation

Require: Model $F(\mathcal{X})$, Summary Attributes $\mathcal{S}$, Non-summary Attributes $\tilde{\mathcal{S}}$, Dataset D, Visualization Points $V_{\mathcal{S}}$

1: **for all** $\mathbf{v}_{\mathcal{S}} \in V_{\mathcal{S}}$ **do**
2: $\quad \text{sum} = 0$
3: $\quad$ **for all** $\mathbf{x} \in D$ **do**
4: $\quad\quad \text{sum} = \text{sum} + F(\mathbf{v}_{\mathcal{S}}, \mathbf{x}_{\tilde{\mathcal{S}}})$
5: $\quad$ **end for**
6: $\quad$ **return** $(\mathbf{v}_{\mathcal{S}}, \frac{\text{sum}}{|D|})$
7: **end for**

3.3.1 Summaries For Black-Box Models

In this section, we describe a simple algorithm for summary computation, which can be used for *any* data mining model. This algorithm is a direct implementation of the definition of a partial dependence summary and is outlined in Algorithm 6 for a single summary. The algorithm is called for each summary in P.

Computing a set of summaries using the naive algorithm is very expensive. The algorithm runs a total of $|V_{\mathcal{S}}| \cdot |D|$ points through the model to compute a single summary. For complex natural processes, good models tend to be complex as well, resulting in high total prediction cost. In the bird ecology domain, a typical analysis, even if limited to one-and two-dimensional summaries for a single region, typically requires several CPU months of computation time. In general, producing the summaries is the computational bottleneck during exploratory analysis. Our goal therefore is to reduce this cost significantly.

3.3.2 Opportunities for Performance Improvement

Our algorithms for speeding up summary computations are based on the following observations.

Repetitive structure among query points: To compute a summary for a given set of visualization points V_S and dataset D, the model is evaluated for all points in $V_S \times D$. (Notice the nested for-loops in Algorithm 6 that implement this cross-product.) For each $\mathbf{v}_S \in V_S$, there are $|D|$ query points that all have the same value vector $\mathbf{v}_S$ for the summary attributes. Similarly, for each $\mathbf{x} \in D$, there are $|V_S|$ points that all have the same value vector $\mathbf{x}_{\bar{S}}$ for the non-summary attributes. This repetitive structure creates a potential for sharing computation across query points.

Aggregation: For a given visualization point $\mathbf{v}_S$, its summary output is the average of the model predictions for all query points in $\{\mathbf{v}_S\} \times D$. Rather than first computing each individual prediction and then averaging them, it might be possible to push aggregation into the model.

Inter-summary commonality: Multiple summaries for the same model can have non-summary or summary attributes in common. Since all summaries in a given summary set P share the same dataset D, there are additional opportunities for sharing computation across summaries. Also, similar sets of visualization points open up even more opportunities for optimization.

Distributed computation: We will also show that summary computations can be scaled to very large data sets and large sets of plots by distributing the computation across a set of processors. Even when distributing computation, we can still take advantage of the above mentioned structural properties of the work-

load.

Notice that if a data mining model is truly a *blackbox*, i.e., we do not know anything about function F other than the input-output combinations for those inputs for which the model was evaluated, then we cannot exploit any of the structural properties of the summary computation workload. Hence techniques for exploiting these properties have to be tailored to the structure of the data mining model. While the high-level approach will be the same—identify common computation and store appropriate results to avoid repeating the same computation; push aggregation into the model—the actual techniques for implementing these ideas will be model-dependent.

In this paper we focus on tree-based models, because they are among the very best prediction models for both classification and regression problems [63, 15], and widely used in practice. Extending our ideas to other data mining and machine learning model types is part of our future work.

3.4 Summary Computation in Trees

We briefly introduce tree-based prediction models and show how to take advantage of their structure to exploit the observations discussed in Section 3.3.2. The algorithms will be more formally introduced in Section 1.3.

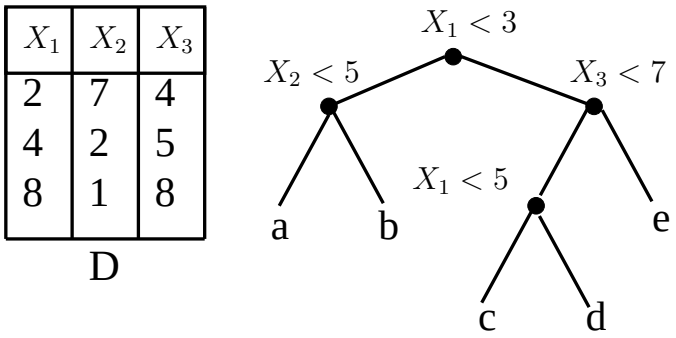

Figure 3.2: Example Tree and Dataset

3.4.1 Tree Models

Classification and regression trees are some of the oldest and most popular predictive models [11]. A tree model partitions the data space recursively, attempting to achieve partitions with high purity, low mean squared error, or similar goals. Each non-leaf node in the tree splits the data space on some attribute; the leaves contain predictions for points that fall into the corresponding region of the data space.

Figure 3.2 shows an example tree model built on three continuous attributes X_1, X_2, and X_3. The root corresponds to the entire data space. It contains a split predicate on X_1 ($X_1 < 3$). The root has two children. The left child corresponds to the "half" of the data space containing all records with $X_1 < 3$, while the right child corresponds to the other "half" of the data space with records satisfying $X_1 \geq 3$. This continues recursively at the children, who can divide their respective sub-spaces further by similarly splitting on any attribute. The leaves of the tree contain predictions, in the example these are some constants a, b, c, d, and e. Nodes are not limited to binary splits, i.e., they can have more than two children. In the rest of the paper, we will refer to a node in a tree model as n, and its children as $\mathrm{c}_1 \ldots \mathrm{c}_\mathrm{k}$, where k denotes the total number of children of n.

When making predictions for a point **x**, the tree is traversed from the root. At each node, the split predicate is evaluated for **x**. This evaluation, which we will refer to as n.TestSplit(**x**) in the rest of the paper, returns the appropriate child where the traversal continues recursively until a leaf is reached. For example, in the tree of Figure 3.2, if $\mathbf{x} = (1, 1, 2)$, then the predicate evaluation at the root results in the traversal of the left child ($X_2 < 5$), after which the prediction a is returned. Missing values in a query point can be handled gracefully by sending partial weights down each sub-tree and then computing the weighted average of the corresponding leaves. Since trees are well-known, we omit a detailed discussion and refer the reader to [11].

Tree models are very popular, because they are somewhat intelligible (less so for large trees), can model complex functions, and can handle missing values and all types of attributes. The only downside is that the predictive performance of single tree models usually is not competitive with more recent machine learning techniques. This disadvantage has been eliminated by ensemble methods like bagged trees [9], boosted trees [59], random forests [10], and Groves [63]. These ensembles consist of many trees and make predictions by adding and/or averaging predictions of all trees in the ensemble. Recent studies have shown that ensembles of trees are among the very best predictive models known today [63, 15] and are widely used in many applications. Our techniques can be applied to all these tree ensembles.

Many variations of trees have been proposed, including some with multivariate splits (split predicates on more than one variable) and with non-trivial prediction functions in the leaves (rather than a constant value). With the advent of ensemble methods, these more "exotic" trees are rarely used. For all our

algorithms in this paper we will therefore focus on trees with univariate splits and constant predictions in the leaves. In Section 3.8 we will briefly outline how our algorithms can be easily extended to the more complex tree types.

3.4.2 Sharing Computation

We show how to implement the basic ideas for speeding up summary computation (see Section 3.3.2) for tree models. We will focus on single trees; all ideas extend to ensembles by applying them to each tree in the ensemble individually. For ease of presentation, the techniques are explained for a concrete example. The general algorithms are discussed in Section 1.3.

Short-circuiting: Our first technique leverages the repetitive structure in the query workload for computing a single summary. Consider the example tree in Figure 3.2 and assume we want to compute a summary on $\mathcal{S} = \{X_1\}$ for dataset D, shown in the same figure. Furthermore, assume the set of visualization points is $V_{X_1} = \{1, 4, 7\}$.

The table in Figure 3.3 shows the set of query points that are run through the model for this summary computation. This set is the cross product of V_{X_1} and $\pi_{X_2,X_3}(D)$. Notice that every combination of (X_2, X_3) values in D occurs $|V_{X_1}|$ many times. Whenever we reach a node with a predicate on X_2 or X_3, if that node was reached before for another query point with the same (X_2, X_3) values, then we can avoid duplicate computation by memoizing previous predicate evaluation results.

We propose to implement this idea by *short-circuiting* the tree for a given

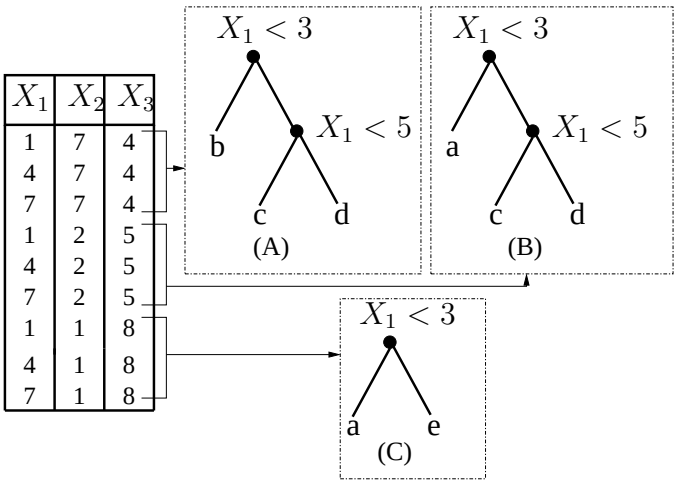

Figure 3.3: Single-point Short-circuit Trees

(X_2, X_3) value combination. To short-circuit the tree, we traverse the tree starting at the root. Whenever we encounter a node that splits on summary attribute X_1, we explore both children. Whenever we encounter a node that splits on one of the non-summary attributes X_2 or X_3, we only explore the corresponding sub-tree for which the predicate evaluated to true. During this traversal, we remove all encountered nodes that split on non-summary attributes X_2 and X_3, and we remove these nodes' sub-trees for which the predicate evaluated to false. Intuitively, for a given (X_2, X_3) value combination, we compress the tree by removing (1) all nodes that split on X_2 and X_3 and (2) all sub-trees that cannot be reached for this (X_2, X_3) value combination, no matter what the X_1 value would be.

Tree (A) in Figure 3.3 is the short-circuited tree for $\mathbf{x}_{X_2,X_3} = \{7, 4\}$. Given this tree, predictions at all points of the form $(X_1, 7, 4)$ can be computed without accessing any X_2, X_3 nodes. In fact, we only need to run the X_1 value through this tree. For example the point $(1, 7, 4)$ avoids the test on $X_2 < 5$ and directly returns the correct prediction b. We will refer to such a tree as a *single-point*

short-circuit tree, because it was created based on a single non-summary attribute value combination (i.e., a single point from D). Trees (B) and (C) in Figure 3.3 are the single-point short-circuit trees for the other (X_2, X_3) combinations in D.

With short-circuiting, we do the predicate evaluation work for the non-summary attribute values of each data record $\mathbf{x} \in D$ only once. The larger the set of evaluation points $V_{\mathcal{S}}$ and the smaller the short-circuited tree, the more these savings will pay off.

Instead of short-circuiting a tree on the non-summary attributes, one could alternatively short-circuit on the summary attributes. However, in practice usually $|\mathcal{S}| \ll |\tilde{\mathcal{S}}|$, because scientists usually care about summaries for visualization ($|\mathcal{S}| = 1$ or $|\mathcal{S}| = 2$) and natural phenomena tend to be complex ($|\mathcal{X}|$ is in the order of hundreds of attributes). This implies that short-circuiting on non-summary attributes will usually result in better tree compression. Therefore, we will always short-circuit on non-summary attributes.

Aggregating short-circuit trees: Short-circuiting reduces redundant computation involving non-summary attributes, but there is still redundant computation for summary attributes. Consider the single-point short-circuit trees in Figure 3.3, where we generated a separate tree for each point in D. There are obvious similarities between these trees. In particular, all trees have the same root node, which tests $X_1 < 3$. A visualization point $X_1 = c$, for any constant c, will perform the same predicate evaluation at the root of each tree, and it will take the same branch in all trees.

This redundant work can be avoided by pushing aggregation into the tree model. For $X_1 = c$, we actually do not need the individual predictions for

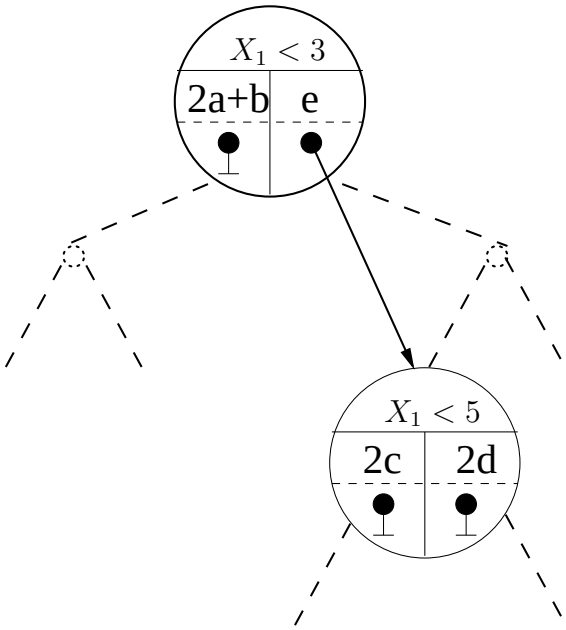

Figure 3.4: Multi-point Short-circuit Tree

query points $(c, 7, 4)$,$(c, 2, 5)$, and $(c, 1, 8)$, but only their average. This allows us to merge the single-point short-circuit trees of Figure 3.3 into a single tree (Figure 3.4) with a new root. We will refer to this merged tree as a *multi-point short-circuit tree*. The details of how this tree is generated and used for computing summary outputs will be discussed in Section 1.3. For a given $\mathbf{v}_S$, a single traversal of this tree will give us the summary output at the visualization point. This avoids the redundant work introduced by generating a separate single-point short-circuit tree for each point in D. For convenience, in the rest of the paper whenever we refer to a short-circuit tree, unless mentioned otherwise, it refers to a multi-point short-circuit tree.

Setting up a multi-point short-circuit tree removes any interaction between D and the visualization points of a summary. This is very useful for the *online* summary computation problem because a stream of visualization points can be processed by a small short-circuited tree.

Inter summary structure: The techniques discussed so far avoid repetitive work in the computation of a *single* summary. In the summary computation problem as introduced in Section 3.3, a scientist might request a large set of summaries, P, for a given dataset D. In this case, any pair of summaries $\{p_i, p_j\} \in P$ can share computation on attributes in $\mathcal{X} - \{p_i.\mathcal{S} \cup p_j.\mathcal{S}\}$.

For example, assume $\mathcal{X} = \{X_1, X_2, \ldots, X_{100}\}$ and P contains the three summaries $\{(X_1, X_2)$, (X_1, X_3) and $(X_1, X_4)\}$. Summaries (X_1, X_2) and (X_1, X_3) can share computation on attributes $X_4, \ldots, X_{100}$. Similarly, summaries $\{X_1, X_2)$ and (X_1, X_4) can share computation on $X_3, X_5, \ldots, X_{100}$ and so on.

The above example illustrates that when multiple summaries have to computed for the same dataset, generating the short-circuit trees for all of the summaries together will help share computation on non-summary attributes across the different summaries. Notice that some summaries in P can also have summary attributes in common. However, in order to exploit any shared computation for summary attributes, the visualization point sets for these summaries should exhibit a particular structure (Section 3.5.3).

3.5 Algorithms

We first introduce short-circuit trees more formally and then present algorithms for creating and querying such trees. For ease of presentation, in the following discussion we will assume that there are no missing values in the set D. Extending the algorithms to support missing values will be addressed in Section 3.8.

3.5.1 Short-circuit Tree Structure

Let T be a given tree for which we want to compute a summary on $\mathcal{S}$. The corresponding short-circuit tree $\mathrm{T}_{\bar{\mathcal{S}}}$ *conceptually* is identical to T augmented by additional information in some of the nodes.

Let n be a node of T that splits on a summary attribute. In $\mathrm{T}_{\bar{\mathcal{S}}}$ this node maintains two types of information about each subtree rooted at its children. Let c be a child of node n. The first type of information is an array of short-circuit pointers, called *shckts*. Each pointer in this array points to a node in the subtree rooted at c with the following properties: (1) the node splits on a summary attribute and (2) none of the node's ancestors that are also descendants of node n splits on a summary attribute. For example in the short-circuit tree of summary X_1 (Figure 3.4), the right child of the root contains a single short-circuit pointer to the node which splits on $X_1 < 5$. The left child of the root has no short-circuit pointers since there does not exist any node in its subtree that splits on X_1. In general, there can be 0, 1, or more pointers in shckts, depending on the tree structure. The second type of information for a subtree rooted at c is a residual prediction, called *res_pred*, which is the sum of the predictions for all points in $\pi_{\bar{\mathcal{S}}}(D)$ that traversed the subtree rooted at c, but reached a leaf without encountering a node that splits on a summary attribute. Consider the left subtree of the root node in our example tree. Since the root splits on a summary attribute, when we traverse the tree for non-summary combinations $\mathbf{x}_{\bar{\mathcal{S}}} \in \pi_{\bar{\mathcal{S}}}(D)$, we pass all these points to both the left and the right child of the root. In the left subtree, all these points reach leaves without encountering another node that splits on summary attribute X_1. Two of these points, $(2, 5)$ and $(1, 8)$, reach the leaf with prediction a, while one point, $(7, 4)$, reaches the one that predicts b. Hence the

Algorithm 7: Point Compute Output

Require: Tree Node n, Visualization Point $\mathbf{v}_{\mathcal{S}}$

1: c = n.TestSplit($\mathbf{v}_{\mathcal{S}}$)
2: sum = c.res_pred
3: **for all** $\mathrm{n}' \in \mathrm{c}$.shckts **do**
4: sum = sum + Point Compute Output(n',$\mathbf{v}_{\mathcal{S}}$)
5: **end for**
6: **return** sum

residual prediction stored at the root for the left subtree is $2a + b$.

If the root node of T splits on a non-summary attribute, then $\mathrm{T}_{\tilde{\mathcal{S}}}$ also has a new root node with the trivial split predicate "true". It maintains an array of short-circuit pointers and a residual prediction computed for the entire tree T, as explained above for nodes that split on summary attributes.

Computing a summary using the short-circuit tree is straightforward. For a visualization point $\mathbf{v}_{\mathcal{S}}$, we traverse tree $\mathrm{T}_{\tilde{\mathcal{S}}}$ starting at the root and then following the appropriate short-circuit pointers. More precisely, we call Algorithm 7 for the root of $\mathrm{T}_{\tilde{\mathcal{S}}}$. The algorithm determines the appropriate subtree by evaluating the split predicate (line 1); it then recursively traverses all short-circuit pointers for this subtree. For all accessed nodes, their residual predictions are added. This sum is equivalent to $\sum_{i=1}^{|D|} F(\mathbf{v}_{\mathcal{S}}, \mathbf{x}_{\tilde{\mathcal{S}}}(i))$, hence we obtain the desired summary output value $\hat{F}_{\tilde{\mathcal{S}}}(\mathbf{v}_{\mathcal{S}})$ by simply dividing this sum by $|D|$.

For a given set of summaries P, a separate short-circuit tree is constructed for each summary. Rather than storing them separately, we "merge" all of them into the original tree T. More precisely, a node has not any more only a single

Algorithm 8: Short-circuit Tree

Require: Tree T, Summary Set P, Dataset D

1: Create new root node new_root
2: **for all x** $\in D$ **do**
3: new_root.Update(Short-circuit Node(T,P,**x**))
4: **end for**
5: **return** new_root

(shckts, res_pred) pair. It now has an array of these entries, one array element for every summary that contains the split attribute of the node as a summary attribute.

3.5.2 Generating Short-circuit Trees

Algorithms 8 and 9 describe the pseudocode for generating all short-circuit trees. For each record in D, they perform a *single traversal of the tree* (instead of $|P|$ traversals). We first discuss Algorithm 9 which performs operations at a single node n in the tree.

To better understand Algorithm 9, let us for now assume that P contains only a single summary on attributes $\mathcal{S}$. In this case the output (called op in the pseudocode) is a single pair consisting of a short-circuit pointer to a node in n's subtree (including n itself) and a prediction value. Exactly one of them is null. Now let us examine how the output is computed.

If n is a leaf, then the algorithm simply returns the node's prediction value (and null for the pointer). Now consider the case when n is not a leaf. If n

Algorithm 9: Short-circuit Node

Require: Tree Node n, Set Of Active Summaries P_{n}, Data Record **x**

if n is a leaf **then**
  for all $p \in P_{\mathrm{n}}$ **do**
    $\mathrm{op}[p] = \langle \mathrm{null}, \mathrm{n.prediction} \rangle$
  end for
else
  $P^s_{\mathrm{n}} = \{p \in P_{\mathrm{n}} \mid \mathrm{n.splitAttribute} \in p.\mathcal{S}\}$
  $\mathrm{c}' = \mathrm{n.TestSplit}(\mathbf{x})$
  for all $i \in \{1, \ldots, \mathrm{k}\}$ **do**
    if $\mathrm{c}_i == \mathrm{c}'$ **then**
      op = Short-circuit Node(c_i,P_{n},**x**)
      for all $p \in P^s_{\mathrm{n}}$ **do**
        $\mathrm{c}_i\,[p]$.shckts.add(op$[p]$.shckts)
        $\mathrm{c}_i\,[p]$.res_pred.add(op$[p]$.res_pred)
        $\mathrm{op}[p] = \langle \mathrm{n}, \mathrm{null} \rangle$
      end for
    else
      $\mathrm{op}^{\mathrm{tmp}}$ = Short-circuit Node(c_i,P^s_{n},**x**)
      for all $p \in P^s_{\mathrm{n}}$ **do**
        $\mathrm{c}_i\,[p]$.shckts.add($\mathrm{op}^{\mathrm{tmp}}\,[p]$.shckts)
        $\mathrm{c}_i\,[p]$.res_pred.add($\mathrm{op}^{\mathrm{tmp}}\,[p]$.res_pred)
      end for
    end if
  end for
end if
return op

splits on an attribute in $\mathcal{S}$, then we need to recursively traverse all children of n. For each child, this traversal returns either a short-circuit pointer to the highest node in that subtree that splits on a summary attribute and was accessed during the traversal for point $\mathbf{x}$ (notice that there is only one such node). Or it returns a residual prediction value for this subtree (if no node splitting on a summary attribute was encountered). The returned pointer or prediction value added to n for the corresponding subtree. Since n splits on a summary attribute, its nearest ancestor that splits on a summary attribute should point to it. Hence the algorithm returns op as a pair containing a pointer to n and null for the residual prediction value.

If n splits on a non-summary attribute, then we only traverse that child for which the split predicate evaluates to true ($\mathrm{n.TestSplit}(\mathbf{x})$). This recursive call returns either a pointer or a residual prediction as described before, but since n splits on a non-summary predictor, it is conceptually deleted from the tree and hence does not store any short-circuit pointers or residual predictions itself. Instead, it simply returns what ever it received from its subtree to its ancestor.

Algorithm 9 efficiently implements this procedure we just discussed, but now for an entire set of summaries together. The set of active summaries encodes the set of all summaries that reach node n. Notice also the branches in lines 10–15 and 17–21. For the child c' that was selected by the split predicate evaluation, all summaries that were active at n remain active. For the other children, only those summaries for which the split attribute is a summary attribute will be active. Similarly, as lines 11 and 18 indicate, we only update n's pointers and residual prediction values for those summaries that contain n's split attribute as a summary attribute. Line 14 ensures that for these summaries, we

return a short-circuit pointer to n to n's ancestors. For all other summaries, i.e., those for which n's split attribute is a non-summary attribute, the algorithm simply outputs the pointer or residual prediction value it received from traversing the subtree rooted at c'.

Algorithm 8 calls Algorithm 9 on the root node of tree T with the active set of summaries set to P, for every point in D. The return array from the call to Algorithm 9 contains information to update the short-circuit pointers and residual prediction for the root of the short-circuit tree for each summary in P.

3.5.3 Generating The Summary Output

Once the short-circuit tree for a set of summaries has been generated, computing the summary output is straightforward: we use Algorithm 7 as before, but now include a summary identifier in each set of visualization points to access the correct short-circuit tree for a summary. In the remainder of this section we will introduce a technique for speeding up querying the short-circuit tree by taking advantage of structure in the set of visualization points.

Since summaries are primarily created for visualization, scientists often define $V_{\mathcal{S}}$ as a grid of summary values, i.e., $V_{\mathcal{S}} = V_{X_1} \times \ldots \times V_{X_{|\mathcal{S}|}}$, where each V_{X_i} is a set of values selected independently for X_i. This creates repetitive structure within the set of visualization points.

Algorithm 10 exploits this structure in $V_{\mathcal{S}}$ to speed up querying the short-circuit trees with the evaluation points. Intuitively, rather than pushing the cross product $V_{X_1} \times \cdots \times V_{X_{|\mathcal{S}|}}$ through the tree, we traverse the tree with the individual

Algorithm 10: Set Compute Output

Require: Tree Node n with X_i as its split attribute, Sets Of Visualization Points $V_{X_1}, \ldots, V_{X_{|\mathcal{S}|}}$, Summary p, Summary Attributes $\mathcal{S}$

1: $(V_{\mathrm{c}_1,X_i}, \ldots, V_{\mathrm{c}_\mathrm{k},X_i})$ = SplitV(V_{X_i})
2: **for all** $j \in \{1, \ldots, \mathrm{k}\}$ **do**
3: **for all** $\mathbf{v}_\mathcal{S} \in V_{X_1} \times \cdots \times V_{X_{i-1}} \times V_{\mathrm{c}_j,X_i} \times V_{X_{i+1}} \times \cdots \times V_{X_{|\mathcal{S}|}}$ **do**
4: UpdateOutput($\mathbf{v}_\mathcal{S}$,c_j [p].res_pred)
5: **end for**
6: **for all** $\mathrm{n}' \in \mathrm{c}_j$ [p].shckts **do**
7: Set Compute Output($\mathrm{n}', V_{X_1}, \ldots, V_{X_{i-1}}, V_{\mathrm{c}_j,X_i}, V_{X_{i+1}}, \ldots, V_{X_{|\mathcal{S}|}}$)
8: **end for**
9: **end for**

sets V_{X_i} kept separate. While update cost at each node is still proportional to the size of the cross product (lines 3, 4), splitting cost (line 1) and parameter passing cost (line 7) are reduced to being proportional to the size of an individual set V_{X_i} and the sum of the sizes of all individual sets, respectively. (Without the optimization, these costs would have been proportional to the size of the cross product as well.) Hence, while the overall evaluation cost asymptotically still depends on $|V_\mathcal{S}| = |V_{X_1} \times \cdots \times V_{X_{|\mathcal{S}|}}|$, the optimization significantly reduces the constant factor, leading to a slower cost increase with increasing $|V_\mathcal{S}|$.

For a given summary $\mathcal{S}$, the algorithm traverses the entire short-circuit tree once. At a node n that splits on attribute X_i, it splits the set of visualization values for this attribute, V_{X_i}, into V_{c_1,X_i},..., $V_{\mathrm{c}_\mathrm{k},X_i}$ based on the evaluation of the split predicate for each value in V_{X_i}. For every child c of n, the algorithm first updates the summary output for all visualization points in $V_{X_1} \times \cdots \times V_{X_{i-1}} \times V_{\mathrm{c},X_i} \times V_{X_{i+1}} \times \cdots \times V_{X_{|\mathcal{S}|}}$ with the residual prediction at c for summary $\mathcal{S}$. It

then makes a recursive call for every node in c's list of short-circuit pointers for summary $\mathcal{S}$, passing the appropriate V_X sets. The above routine is called on the root of the short-circuit tree for a summary and the sets of visualization values for each attribute.

As we pointed out in Section 3.2, the set of points from which values for non-summary attributes are derived may also be defined by a grid. If that was the case, we could use a technique like Algorithm 10 for generating the short-circuit trees. We do not describe the details here in the interest of space.

Notice that if different summaries have the same visualization values for attributes they share, then we can achieve additional inter-summary cost savings. For example, if multiple summaries with summary attribute X_i use the same set V_{X_i}, then we can cache the split result of the V_{X_i} set at node n. If other summary computations reach the same node, they can re-use the cached splits. In our experiments, however, this optimization *did not* provide significant speedups because for one-and two-dimensional summaries the chance of sharing summary attributes is much smaller than for sharing non-summary attributes.

Extension to ensembles: Extending the above algorithms to an ensemble of trees, denoted $\mathcal{T}$, is straightforward. In the first step, for each tree $\mathrm{T} \in \mathcal{T}$, we generate the short-circuit trees for all summaries. In the second step, we compute summary output for a visualization point by running the point through all the $|\mathcal{T}|$ short-circuit trees for each summary. The results are then appropriately averaged or summed, depending on the ensemble type.

3.6 Distributed Computation

Despite all cost improvements from taking advantage of the structure of summary computation workloads, we have to parallelize computation to make it scale to realistic scientific applications. In this section, we propose algorithms that allow us to scale in all important input parameters of summary computation: size of the dataset $|D|$, number of summaries $|P|$, size of the ensemble $|\mathcal{T}|$, and number of visualization points $|V_S|$ per summary. We say that our algorithm is (linearly) *scalable* in a parameter, if we can achieve the following: With c times as many machines of equal capability, we can process a c times larger job (i.e., parameter scaled up by a factor of c) without suffering a significantly higher response time.

While it is possible to implement a distributed framework specific to summary computations, we decided to use the recently proposed MapReduce [21]. Two factors prompted this design decision: First, the requirements for distributing summary computations fit well with the programming abstractions supported by MapReduce. Second, using a framework like MapReduce makes it easy to run the distributed algorithms on a cluster without having to worry about low-level issues usually associated with implementing distributed algorithms.

3.6.1 Map Reduce

Before discussing our algorithms, we briefly introduce MapReduce. The MapReduce framework can be used to implement a two-phase distributed com-

putation on a very large input dataset, which we denote as I. The first phase, *Map*, partitions I into a set of disjoint units. A user-specified map function is then applied to each unit in parallel by a set of machines, called the mappers. The output of map is a set of $< \text{key}, \text{value} >$ pairs. A single map function can produce many different keys and values. The second (and optional) phase, *Reduce*, works on all the key-value pairs produced by the mappers. Conceptually, these pairs are grouped by their key; then each group is processed by a single reducer. This happens in parallel on many reducer machines. The output produced by all the reducers is the final output of the distributed computation. Further details on MapReduce can be found in [21].

3.6.2 Algorithms

As we show in this section, MapReduce is a good fit for distributing summary computations. Obviously, there are many different ways to implement our problem in MapReduce. To be scalable in the number of summaries, we can implement a map phase that distributes the summaries across mappers and computes summary outputs in parallel. The summary output for a visualization point can be computed on subsets of D and $\mathcal{T}$ in parallel (map) and then aggregated (reduce). To be able to scale in other problem parameters as well, we set up map and reduce tasks as follows.

Input and output: The input dataset to our MapReduce algorithms is the cross product $P \times D \times \mathcal{T}$. Hence each mapper will work on some subset $P' \times D' \times \mathcal{T}'$, where $P' \subseteq P$, $D' \subseteq D$, and $\mathcal{T}' \subseteq \mathcal{T}$. Additionally, each summary may have a set of predefined visualization points, in which case the output of the

Algorithm 11: Map

Require: set of $(P', \mathcal{T}', D')$ triplets, containing some summaries $P' \subseteq P$, some trees $\mathcal{T}' \subseteq \mathcal{T}$, and some data points $D' \subseteq D$

1: **for all** $\mathrm{T} \in \mathcal{T}'$ **do**
2: Short-circuit Tree(T, P', D')
3: **end for**
4: **for all** $p \in P'$ **do**
5: **for all** $\mathbf{v}_\mathcal{S} \in p.V_\mathcal{S}$ **do**
6: sum = $\sum_{\mathrm{T} \in \mathcal{T}'}$Point Compute Output(T, $\mathbf{v}_\mathcal{S}$)
7: Output((p, $\mathbf{v}_\mathcal{S}$), (sum, $|D'|$))
8: **end for**
9: **end for**

Algorithm 12: Reduce

Require: Key=($p, \mathbf{v}_\mathcal{S}$), Value=$\{(\mathrm{sum}_1, |D'|_1), (\mathrm{sum}_2, |D'|_2), \ldots\}$

1: t_sum=0; cnt=0;
2: **for all** (sum, $|D'|$) $\in$ Value **do**
3: t_sum += sum; cnt += $|D'|$
4: **end for**
5: Output((p, $\mathbf{v}_\mathcal{S}$), $\frac{\mathrm{sum}}{\mathrm{cnt}}$)

MapReduce computation will be the final summary outputs. If the visualization points are not specified (the online problem), then the output of the MapReduce computation will be the short-circuit trees for all summaries in P.

Predefined $V_\mathcal{S}$: When the set of visualization points is predefined, then this set is loaded into each mapper before executing the map function. Each mapper then computes for each visualization point of a summary in P', the total contri-

bution that D' and trees in $\mathcal{T}'$ make to the summary output at the visualization point. The corresponding map function is described in Algorithm 11. Each reduce function receives as key a $(p, \mathbf{v}_\mathcal{S})$ combination and as values a set of partial summary outputs computed over subsets of $\mathcal{T}$ and D. The reduce function performs a simple aggregation of this set to produce the final summary output for the $(p, \mathbf{v}_\mathcal{S})$ combination (Algorithm 12). If the ensemble computes predictions by weighting trees differently, Line 6 in Algorithm 11 needs to be modified to compute the corresponding weighted sum.

If the set of visualization points per summary is very large, the above algorithm may have two bottlenecks. First, each mapper processes all visualization points for a summary. Second, the set of keys generated by the map will be very large, making the grouping by key operation expensive. To avoid this, and guarantee scalability in the number of visualization points, we can partition the set of visualization points of a summary. For each subset of visualization points, we add a new summary with a new summary id to P, but with the same $\mathcal{S}$. To reduce the number of keys, each mapper produces the summary (p) as the key and a reducer aggregates outputs for all visualization points associated with a summary.

The computation of the summary output for a visualization point, given D and $\mathcal{T}$, involves only sum and count aggregates. Hence it is easy to show that the distributed algorithm computes the correct result.

No predefined $V_\mathcal{S}$; generating short-circuit trees: Our second MapReduce algorithm is for the case when the set of visualization points for a summary is not part of the input. Hence we want to compute the short-circuit trees for all summaries in P and all trees in $\mathcal{T}$, so that when visualization points are later

presented to the model, it can efficiently produce summaries from the short-circuited models. The basic idea is to generate a multi-point short-circuit tree for a summary p and a tree T by merging the singlepoint short-circuit trees generated for each point in D. As before, each mapper loads a triplet $(P', D', \mathcal{T}')$, but different from Algorithm 11, the mapper in this case outputs for each pair $(p \in P', \mathrm{T} \in \mathcal{T}')$ (the key), the set of single point short-circuit trees generated for each point in D' (the value). The reduce function receives a single (p, T) pair as the key and sets of single-point short-circuit trees generated from the different mappers as values. The reduce function merges all these single-point short-circuit trees into the multi-point short-circuit tree for the (p, T) pair.

The main challenge here is to efficiently merge the single-point short-circuit trees. There are many ways of implementing the merge process. We propose an approach that works well in practice. The main observation is that the actual single-point short-circuit trees are not necessary for generating a multi-point short-circuit tree. All we need is, for every leaf in the original tree, a count of the number of single-point short-circuit trees that the leaf is part of. Once these counts are known, a single pass on the tree can be made to generate the multi-point short-circuit tree with the short-circuit pointers and all residual predictions. Consider the example in Figure 3.3. Knowing that $\mathcal{S} = X_1$, that the leafs with predictions a, c, and d are part of two single-point short-circuit trees, and that b and e are part of one, is enough information to produce the multi-point short-circuit tree of Figure 3.4.

With this observation, the merge process becomes very simple. A mapper produces for a particular (p, T) pair (the key), a set of counts, one for every leaf in T (the value). For a leaf, the count represents the number of single-

point short-circuit trees generated from D' that the leaf is part of. The reducer aggregates these counts to obtain the number of single-point short-circuit trees that a leaf is part of for the entire set D. The reducer then loads the original tree T and makes a traversal of T to generate the multi-point short-circuit tree for the (p, T) pair that is the key to the reducer.

Scalability: The proposed MapReduce algorithms can scale in all the inputs of the summary computation problem, namely P, D, $\mathcal{T}$. On the map side, the space of $P \times D \times \mathcal{T}$ can be partitioned to any granularity and distributed across a set of mappers. On the reduce side, when the sets of visualization points are known, reducers work on the $P \times V_S$ space, which can again be partitioned to any granularity across the reduce machines. When the reducers are generating short-circuit trees, they work on the $P \times \mathcal{T}$ space which is again partitioned across a set of reduce machines. The exact method used for generating the partitions for a given problem is discussed in Section 3.7.2.

3.7 Experiments

We study the benefits of short-circuiting and the other proposed optimizations over the naive algorithm. Then we demonstrate the scalability of our distributed algorithms. The presented results are representative for the results of our extensive study and they capture what one could typically expect to see in terms of cost reduction in practice.

We report results for a real bird ecology dataset obtained from the Avian Knowledge Network (www.avianknowledge.net), Project FeederWatch, that was joined with datasets containing geographical features of observation lo-

cations. It covers a geographical region in North America and contains about 90,000 observation records, described by 155 continuous attributes (e.g., time, location, habitat features, climate, census features, elevation). We use 60,000 records to train a model for predicting the probability of observing the Dark-eyed Junco. The model is a bagged tree model consisting of 10 trees and is trained using the IND package [13]. The trees were grown using the information gain splitting criterion without any kind of pruning, and due to bagging it was among the very best predictive models we were able to train for this dataset. Each tree in the ensemble had on average 10,300 non-leaf nodes.

For our experiments, we use the entire 60,000 training records as the D set in summary computations. The training data had missing values on some attributes, which we filled in using values randomly selected from the attribute's domain. We verified that all leaves of the tree contained data points to guard against degenerate cases. Not handling missing values in the short-circuited trees is not an inherent limitation of our approach, as discussed in Section 3.8, just a limitation in our current implementation. The choice of the actual D set does not matter much, because our experiments are only aimed at evaluating the speedups we obtain in summary computations.

3.7.1 Single Machine Experiments

Our algorithms are implemented in Java and the experiments reported in this section were run on a Linux machine, with a 2.66GHz processor and a JVM heap size of 3GB. All reported times are in seconds. Standard deviations in the reported times were negligible and hence are not reported.

Table 3.1: Summary Computation Time (sec): Frequent Attributes

$\|V_S\|$	Naive	ShCkt
100	85.0	3.02 (= 2.96 + 0.06)
400	311.5	3.17 (= 2.97 + 0.20)
625	469.8	3.29 (= 2.96 + 0.33)

Table 3.2: Summary Computation Time (sec): Infrequent Attributes

$\|V_S\|$	Naive	ShCkt
100	84.8	2.1 (= 2.1 + 0.001)
400	324.5	2.1 (= 2.1 + 0.001)
625	462.7	2.1 (= 2.1 + 0.002)

Naive vs. short-circuiting: We begin our evaluation by comparing the benefits of the short-circuiting algorithm (Section 1.3) over the naive algorithm (Section 3.3.1). The first summary, which we refer to as the *frequent attributes summary*, is on a pair of attributes that are frequently used as split attributes in the ensemble (Table 3.1). The second summary is on a pair of attributes used infrequently in the ensemble (Table 3.2). The tables report the time taken to compute the summaries using the naive algorithm (Naive) and time taken to compute summaries using short-circuiting (ShCkt). For short-circuiting we also report how much of the total time went into generating the short-circuit tree versus how much into querying these trees and generating summary outputs. The above times are reported for sets of visualization points of different sizes.

For the frequent attribute summary, the summary attributes occur in the split predicates of 12% and 11% of all non-leaf tree nodes, respectively. In this case, we see that short-circuiting is about 30 to 140 times faster than naive. The

speedup improves with increasing $|V_S|$ (Table 3.1). The naive algorithm makes a pass on the entire D set per visualization point, revisiting and re-evaluating node predicates in the trees multiple times. The short-circuiting algorithm on the other hand pays a fixed cost for generating the short-circuit trees in a single pass over D. It then runs visualization points through the smaller trees with their pre-computed residual predictions. Therefore, it is natural to see greater savings from short-circuiting as we increase the number of visualization points in a summary. As the cost breakup for ShCkt shows, the cost of generating the short-circuit trees is independent of V_S (2.9 secs), and the cost of running visualization points through them is approximately linear in the number of visualization points.

Table 3.2 reports the results for a summary on infrequently used split attributes, which together accounted for less than 1% of the splits in the ensemble. Here short-circuiting provides even greater benefits, showing 40 to 200 times speedup over naive. The naive algorithm does the same amount of work for any summary and hence the numbers in the Naive column in Tables 3.1 and 3.2 are similar. However, the short-circuiting algorithm is able to exploit the fact that the summary attributes of interest are not used often in the tree. The time spent in creating the short-circuit model is down to 2.1 secs since fewer nodes in a tree split on summary attributes and hence less work is done in setting up short-circuit pointers and residual predictions. The resulting short-circuit trees are also smaller and therefore there is a significant reduction in the time taken to compute summary outputs. In fact, the short-circuit trees are so small that we hardly see any measurable increase in the cost of generating summary outputs as we increase V_S. The less frequently the summary attributes occur as split attributes in the tree, the greater the expected speedup from short-circuiting.

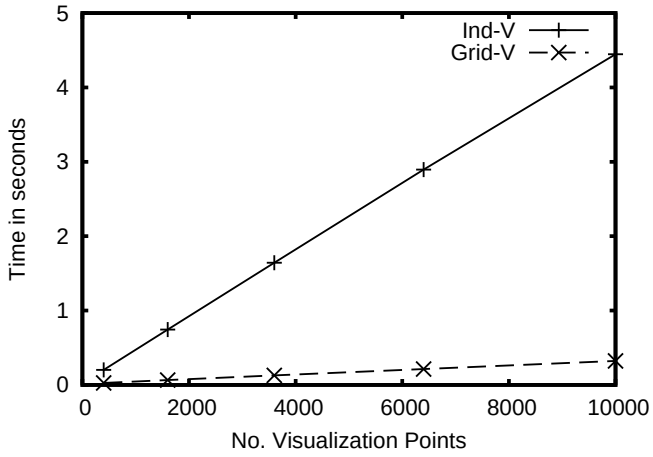

Figure 3.5: V_S On A Grid

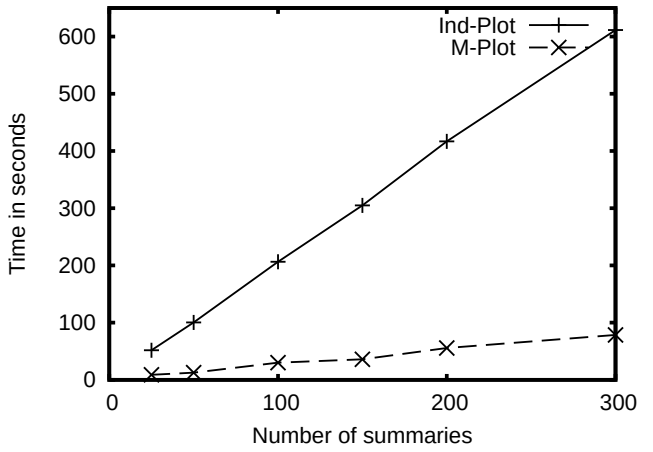

Figure 3.6: Mutiple Summaries

V_S on a Grid: The next experiment measures the benefits when the visualization points are defined on a grid as described in Section 3.5.3. Given the short-circuit tree for the frequent attribute summary, Figure 3.5 plots the time taken to run a set of visualization points through the short-circuit tree and produce the summary outputs. The set of visualization points are selected on a two-dimensional grid and obtained by selecting values for each attribute independently and uniformly between the maximum and minimum values of the

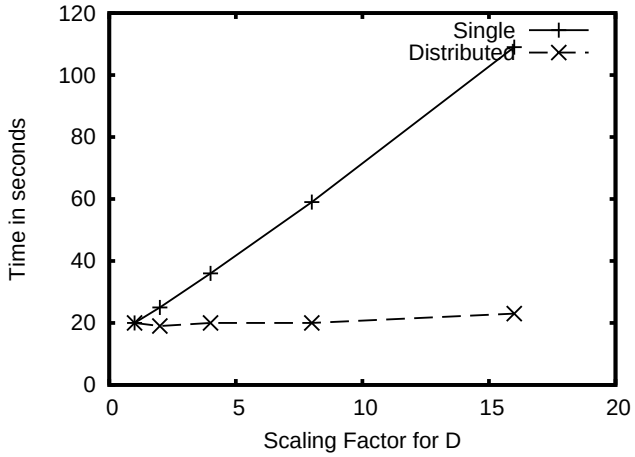

Figure 3.7: No Predefined V_S: Scaling In $|D|$

attribute's domain.

Ind-V plots the times when the visualization points are run through the short-circuit model one-by-one. Grid-V is the time taken for computing summary values using Algorithm 10. The graph shows that the costs for Ind-V and Grid-V both increase linearly with the number of visualization points, but at a much lower rate for Grid-V, as we discussed in Section 3.5.3.

Multiple Summaries: The last experiment in this section measures the benefits of generating the short-circuit trees for a set of summaries together (M-Plot) rather than one summary at a time (Ind-Plot). Summary workloads for this experiment were generated as follows. We first generated the set of all possible one- and two-dimensional summaries over the attributes used by the trees in the ensemble as split attributes. Summary workloads of different sizes are obtained by randomly sampling this set.

Figure 3.6 shows that generating short-circuit trees for multiple plots together is up to 10 times faster than generating them one summary at a time.

These benefits increase with increasing number of summaries. M-Plot makes a single pass on D, evaluates node predicates for each point in D exactly once, and for a point in D, makes all updates to short-circuit pointers and residual predictions in one visit to a node. Ind-Plot generates the same short-circuit trees as M-Plot, but it uses $|P|$ passes on D, repeatedly accessing the same nodes and evaluating predicates in them for every pass on D.

An interesting observation in Figure 3.6 is how the costs increase for Ind-Plot and M-Plot. For Ind-Plot the cost increases linearly as expected. M-Plot on the other hand shows also non-linear trends, which can be explained as follows. Recall from Algorithm 9 that a node n is visited by M-Plot only if $|P_{\mathrm{n}}| > 0$. As P increases, more nodes in the tree will have $|P_{\mathrm{n}}| > 0$, and hence more nodes will be visited, resulting in higher cost. The exact increase depends on how many additional nodes end up being accessed. Once P is large enough, such that for all nodes in the tree $|P_{\mathrm{n}}| > 0$, further increasing P will cause costs to increase linearly due to the extra short-circuit trees generated, while the non-linear effects from reaching additional nodes disappear.

The only obvious drawback of M-Plot is that we may run out of memory when generating short-circuit trees for large P. In that case we can use either the distributed solutions (Sec 3.6) or partition P into smaller subsets and only processing these subsets together. The cost of this approach will be somewhere between Ind-Plot and M-Plot, depending on the size of the subsets of P.

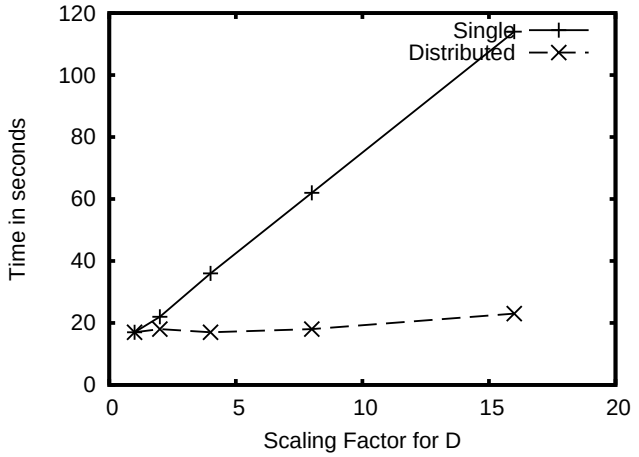

Figure 3.8: Predefined V_S: Scaling In $|D|$

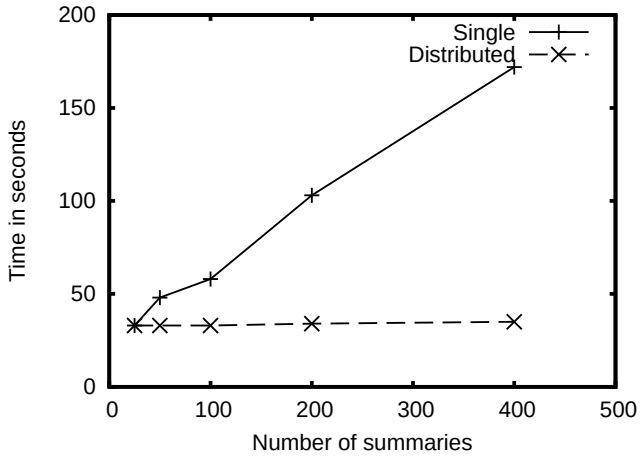

Figure 3.9: Predefined V_S: Scaling In $|P|$

3.7.2 Distributing Summary Computations

In this section we evaluate how the MapReduce algorithms proposed in Section 3.6 scale in the different input parameters of summary computations. The experiments were run on a cluster with 20 machines. Each machine in the cluster had a 2.66Ghz processor and 8GB RAM and the cluster was running Hadoop v0.18 [71], the open source implementation of MapReduce configured with all

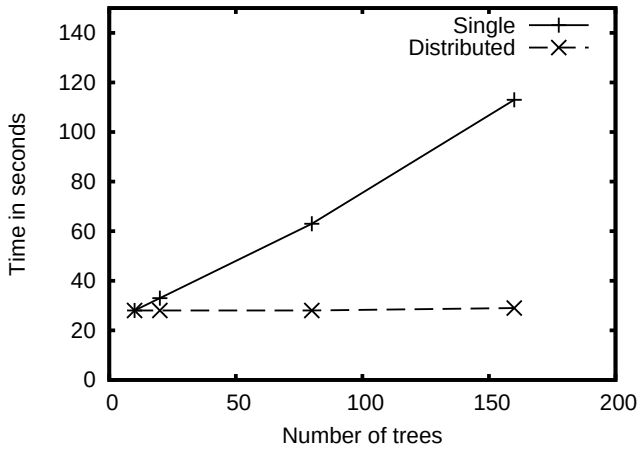

Figure 3.10: Predefined V_S: Scaling In $|\mathcal{T}|$

the default settings. The times that we report are job completion times reported by the Hadoop framework and include all costs such as job setup and teardown times. Deviations in the measured times were small and hence not reported.

In Section 3.6 we defined the map input as the $P \times D \times \mathcal{T}$ space and mentioned that each mapper works on some chunk of this space. Here we now specify explicitly how we generate these chunks to achieve scalability. To scale in a given parameter, we partition the space along this parameter and do not partition along the other parameters. For example, to scale in D, we create chunks $P \times D' \times \mathcal{T}$, where $D' \subseteq D$. Each mapper therefore processes a subset of D, but the complete P and $\mathcal{T}$.

Predefined V_S: Figures 3.8, 3.9 and 3.10 show how the MapReduce algorithm scales in D, P, and $\mathcal{T}$ respectively, when the set of visualization points is part of the input. (The graph for scalability in $|V_S|$ looks virtually the same, and hence is omitted.)

Figure 3.8 reports scalability in D. We fixed the number of summaries to 1

(the frequent attributes summary) with 900 visualization points. The model is the same 10-tree ensemble as before. The number of reducers was set to 1. We then computed the summary for D of varying size. The larger datasets were generated by simply replicating the D set used in the previous section. This ensures that as the datasets are scaled up, access patterns in the tree remain the same and any increase in cost is only due to processing additional data points. The scaling factor is the size multiplier for D. Line Single shows job completion times when the entire computation is done on a single mapper and, as expected response time increases linearly. For the Distributed graph, we increased the number of mappers in proportion to the scaling factor of the dataset. As we can see, response time remains constant with increasing $|D|$, showing that the framework scales well in D.

Figure 3.9 shows the results for scalability in P. We fixed the dataset D to the 60,000 points used in previous experiments, used the same 10-tree ensemble, and fixed the number of visualization points at 900. Notice that when partitioning on P, each mapper works on a subset of P, but the entire D and $\mathcal{T}$. Hence we do not need a reduce phase and the number of reducers was set to 0. We use the same method as described above for generating summary workloads. The line marked Single measures job completion time when the computation uses 1 mapper; for the Distributed line we added mappers proportional to the increase in the number of summaries. The algorithm scales well in P. Note that the costs for Single shows non-linear trends because of the interaction that we discussed in the previous section between the size of P and nodes accessed in a tree.

Figure 3.10 reports scalability in the size of the ensemble. We fixed the number of summaries to 1 (the frequent attributes summary) with 900 visualization

points, and used the usual dataset of 60,000 points. Like the experiment for scaling in D, the number of reducers was set to 1.

No predefined V_S; computing short-circuit trees: The last experiment evaluates our MapReduce algorithm for computing the short-circuit trees. We only show results for scaling the computation in D; the results for the other parameters were similar. Figure 3.7 shows the results for the exact same experimental setup described for the scaling in D experiment earlier. We also used a single reducer which generated the short-circuit trees for all 10 trees in the ensemble. The experiment shows that the framework scales well in D, even for computing short-circuit trees.

Our experiments show that our MapReduce algorithms scale in *each* dimension of the summary computation problem. When the user has access to a limited number of machines and wishes to scale in *all* of P, D and $\mathcal{T}$, one may need to evaluate which inputs are more important to partition. However, given that the algorithms scale similarly in all the input parameters, the exact choice of partitioning will not make a big difference.

3.8 Extensions

For the sake of clarity, we made a few simplifying assumptions in previous sections. Our algorithms generalize naturally and can be extended with additional functionality, as we briefly discuss in this section.

Overlapping D_i: We pointed out in Section 3.3 that our algorithms are directly applicable to any arbitrary data set D_i that is used for obtaining non-summary

attribute values, including selections of regions of D. When the user creates a set of summary computation problems, each for a different D_i, we process them as independent instances of the summary computation problem. However, if there is significant overlap between these D_i sets, then we can do much better. Algorithm 8 can exploit this additional commonality by defining P as the union of the individual summary sets for the different D_i, and D as the union of the D_i's. The algorithm makes one pass on D, but for an $\mathbf{x} \in D$, the set of active summaries at the root is not P, but a subset of P, which contains only those summaries for which $\mathbf{x}$ is contained in the corresponding D_i.

Confidence Intervals: For aggregate summaries, scientists are also interested in obtaining the standard deviation. The residual prediction for a child c of a node n in the tree represents a part of the summary output, obtained as the sum of some leaf predictions. For computing standard deviation, we would also store for c the sum of squares of the leaf predictions.

Missing Values: Many tree types handle missing values in a query point gracefully by sending partial weights down each sub-tree of a node that splits on an attribute whose value is missing; then computing the weighted average of the corresponding leaves. Our algorithms, which were described for the case that there are no missing values in D, can be extended to support this behavior. We extend Algorithm 9 by associating a weight with each active summary at a node. At the root all weights are 1. When Line 7 in Algorithm 9 fails because $\mathbf{x}$ is missing the value for the split attribute, all children are recursively visited with P_{n} as the active set. However, when visiting a child c, the weight for a summary $p \in P_{\mathrm{n}} - P_{\mathrm{n}}^{s}$ is modified to the current weight of p times the fraction of training cases that went into the subtree of c. Line 3 returns the prediction in the leaf

multiplied with the weight of the summary. Also note that op[p].shckts could now contain multiple node pointers and not just one.

Complex Trees: Our approach also generalizes to tree types with multivariate splits at non-leaf nodes and non-trivial functions as leaf predictions. For these trees, we usually cannot generate multi-point short-circuit trees because the prediction made by a leaf and predicate evaluation at a non-leaf node may require the values of both the visualization point and some non-summary attributes. However, we can still generate single-point short-circuit trees that modify each non-leaf and leaf node to contain functions only on summary attributes, by replacing the non-summary variables with the values of the given point $\mathbf{x}_{\tilde{S}} \in D$. The multisummary optimization can also still be applied.

3.9 Related Work

Several papers discuss partial dependence plots [28, 47] and functional ANOVA methods [57] for summarization of complex models. In more recent work, Hooker points out extrapolation problems that may arise when generating summaries from sparse data sets [36].

Friedman [28] proposes a technique for computing *approximate* partial dependence plots in tree models. This method gives no approximation quality guarantees, and it produces accurate summaries only when strong independence assumptions hold. Our focus is on computing the *exact* same summaries as the naive algorithm, no matter what the attribute distribution. Our work is the first to address the computational challenges that arise in generating low-dimensional summaries without introducing any approximation errors com-

pared to the exhaustive naive algorithm.

Research on tree models usually concentrates on improving prediction quality [11, 9, 59, 10] or on scalable algorithms for *training* tree models from large data sets [30]. In general, little work has been done to address the performance issues that arise when using complex data mining models, not only trees, for *making predictions*. Bucila et al [12] propose model compression to reduce model size and computational cost for making predictions. Model prediction time has also been studied in the context of scientific simulations [53]. However, in both cases the original model is approximated, and elimination of redundant computation for summaries is not considered. Our approach is orthogonal in the sense that we would speed up summary computation for such approximate models further by eliminating redundant computation.

The database community has started to explore efficient data management for models [23, 65]. The focus of that work is very different from our work on computing low-dimensional summaries of complex models.

Multi-query optimization has been studied in many different contexts like relational databases [61] and stream processing [2, 22]. The central idea is to optimize a set of queries by reusing query results or by leveraging common sub-expressions. While our algorithms have a similar theme of sharing computation, the structural properties that we exploit are very different.

3.10 Conclusions

In this paper, we motivated and introduced a new data management problem that arises when generating low-dimensional summaries from complex data mining models. We identified various types of structure in summary computation workloads, which we leveraged in order to develop algorithms for speeding up summary computations in tree-based models. Our algorithms produce the exact same results as the naive approach for summary generation, but are in some cases more than 100 times faster than naive. Tree-based models are widely used and hence the algorithms in this paper help address the computational challenges in generating summaries in many applications. We are currently exploring how the ideas of taking advantage of structure presented in this paper can be applied to other complex learning models.

BIBLIOGRAPHY

[1] S. Arya, D. M. Mount, and O. Narayan. Accounting for boundary effects in nearest neighbor searching. In *Proc. Symposium on Computational Geometry*, pages 336–344, 1995.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. ACM Symposium On Principles of Database Systems (PODS)*, pages 1–16, 2002.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD International Conference On Management Of Data*, pages 322–331, 1990.

[4] J. B. Bell, N. J. Brown, M. S. Day, M. Frenklach, J. F. Grcar, R. M. Propp, and S. R. Tonse. Scaling and efficiency of PRISM in adaptive simulations of turbulent premixed flames. In *Proc. International Combustion Symposium*, 2000.

[5] S. Berchtold, C. Böhm, D. A. Keim, and H. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proc. ACM Symposium On Principles of Database Systems (PODS)*, pages 78–86, 1997.

[6] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. International Conference On Very Large Databases (VLDB)*, pages 28–39, 1996.

[7] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[8] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 2005.

[9] L. Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996.

[10] L. Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

[11] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and Regression Trees*. McGraw-Hill, 2000.

[12] C. Bucilu, R. Caruana, and A. Niculescu-Mizil. Model compression. In *ACM SIGKDD*, pages 535–541, 2006.

[13] W. Buntine and R. Caruana. *Introduction to IND and Recursive Partitioning*. Technical Report FIA-91-28, NASA Ames Research Center, 1991.

[14] C. J. C. Burges, J. C. Platt, and J. Goldstein. Identifying audio clips with RARE. In *Proc. ACM International Conference On Multimedia*, pages 444–445, 2003.

[15] R. Caruana and A. Niculescu-Mizil. Data mining in metric space: An empirical analysis of supervised learning performance criteria. In *Proc. ACM SIGKDD International Conference On Knowledge Discovery*, pages 69–78, 2004.

[16] P. Chaudhuri, M. C. Huang, W. Y. Loh, and R. Yao. Piecewise-polynomial regression trees. *Statistica Sinica*, 4:143–167, 1994.

[17] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[18] J. Y. Chen, W. Kollmann, and R. W. Dibble. Pdf modeling of turbulent non-premixed methane jet flames. *Combustion Science and Technology*, 64:315–346, 1989.

[19] P. Ciaccia and M. Patella. Bulk loading the m-tree. In *Proc. Australasian Database Conference*, pages 15–26, 1998.

[20] B. Cui, B. Ooi, J. Su, and K. Tan. Contorting high dimensional data for efficient main memory processing. In *Proc. ACM SIGMOD International Conference On Management Of Data*, pages 479–490, 2003.

[21] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications Of ACM*, 51(1):107–113, 2008.

[22] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *Proc. International Conference on Extending Database Technology (EDBT)*, pages 627–644, 2006.

[23] A. Deshpande and S. Madden. Mauvedb: Supporting model-based user views in database systems. In *Proc. ACM SIGMOD International Conference On Management Of Data*, pages 73–84, 2006.

[24] A. Dobra and J. Gehrke. Secret: A scalable linear regression tree algorithm. In *Proc. ACM SIGKDD International Conference On Knowledge Discovery*, pages 481–487, 2002.

[25] C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of r-trees using the concept of fractal dimension. In *Proc. ACM Symposium On Principles of Database Systems (PODS)*, pages 4–13, 1994.

[26] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. Vector approximation based indexing for non-uniform high dimensional data sets. In *Proc. ACM International Conference On Information And Knowledge Management (CIKM)*, pages 202–209, 2000.

[27] J. H. Friedman. Multivariate adaptive regression splines. Technical report, Stanford University, 1988.

[28] J. H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2001.

[29] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[30] J. Gehrke, V. Ganti, R. Ramakrishnan, and W. Loh. BOAT-optimistic decision tree construction. In *Proc. ACM SIGMOD International Conference On Management Of Data*, pages 169–180, 1999.

[31] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. International Conference On Very Large Databases (VLDB)*, pages 518–529, 1999.

[32] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD International Conference On Management Of Data*, pages 47–57, 1984.

[33] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer, 2003.

[34] J. D. Hedengren and T. F. Edgar. In situ adaptive tabulation for real-time control. *Industrial and Engineering Chemistry Research*, 44:2716–2724, 2005.

[35] W. M. Hochachka, R. Caruana, D. Fink, A. Munson, M. Riedewald, D. Sorokina, and S. Kelling. Data-mining discovery of pattern and pro-

cess in ecological systems. *Journal of Wildlife Management*, 71(7):2427–2437, 2006.

[36] G. Hooker. Diagnosing extrapolation: Tree based density estimation. In *Proc. ACM SIGKDD International Conference On Knowledge Discovery*, pages 569–574, 2004.

[37] G. Hooker. *Diagnostics and Extrapolation in Machine Learning*. PhD thesis, Stanford University, 2004.

[38] H. V. Jagadish, B. C. Ooi, K. L. Tan, C. Yu, and R. Zhang. idistance: An adaptive B+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems (TODS)*, 30(2):364–397, 2005.

[39] David S. Johnson. The NP-completeness column: An ongoing guide. *Journal of Algorithms*, 3(2):182–195, 1982.

[40] M. I. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation*, 6:181–214, 1994.

[41] I. Kamel and C. Faloutsos. On packing r-trees. In *Proc. ACM International Conference On Information And Knowledge Management (CIKM)*, pages 490–499, 1993.

[42] A. Karalic. Linear regression in regression tree leaves. In *Proc. European Conference On Artificial Intelligence (ECAI)*, pages 440–441, 1992.

[43] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *ACM Symposium On Theory Of Computing (STOC)*, pages 599–608, 1997.

[44] C. A. Lang and A. K. Singh. Modeling high-dimensional index structures using sampling. In *Proc. ACM SIGMOD International Conference On Management Of Data*, pages 389–400, 2001.

[45] S. Lawrence, A. C. Tsoi, and A. D. Back. Function approximation with neural networks and local methods: Bias, variance and smoothness. In *Proc. Australian Conference on Neural Networks*, pages 16–21, 1996.

[46] K. I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. Technical report, University Of Maryland, College Park, 1994.

[47] O. Linton and J. P. Nielsen. A kernel method of estimating structured nonparametric regression based on marginal integration. *Biometrika*, 82(1):93–100, 1995.

[48] B. J. D. Liu and S. B. Pope. The performance of *In Situ* adaptive tabulation in computations of turbulent flames. *Combustion, Theory and Modelling*, 9(4):549–568, 2005.

[49] S. Mazumder. Adaptation of the in situ adaptive tabulation (isat) procedure for efficient computation of surface reactions. *Computers and Chemical Engineering*, 30(1):115–124, 2005.

[50] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[51] A. Ning, J. Jin, and A. Sivasubramaniam. Analyzing range queries on spatial data. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 525–625, 2000.

[52] B. Pagel, F. Korn, and C. Faloutsos. Deflating the dimensionality curse using multiple fractal dimensions. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 589–598, 2000.

[53] B. Panda, M. Riedewald, J. Gehrke, and S. B. Pope. High-speed function approximation. In *Proc. IEEE International Conference On Data Mining (ICDM)*, pages 613–618, 2007.

[54] B. Panda, M. Riedewald, S. B. Pope, J. Gehrke, and L. P. Chew. Indexing for function approximation. In *Proc. International Conference On Very Large Databases (VLDB)*, pages 523–534, 2006.

[55] S. B. Pope. Computationally efficient implementation of combustion chemistry using *In Situ* adaptive tabulation. *Combustion Theory Modelling*, 1:41–63, 1997.

[56] M. Riedewald, D. Agrawal, A. El Abbadi, and F. Korn. Accessing scientific data: Simpler is better. In *Proc. International Symposium on Spatial and Temporal Databases (SSTD)*, pages 214–232, 2003.

[57] C. Roosen. *Visualization and Exploration of High-Dimensional Functions Using the Functional Anova Decomposition*. PhD thesis, Stanford University, 1995.

[58] S. Schaal, C. Atkeson, and S. Vijayakumar. Real-time robot learning with

locally weighted statistical learning. In *Proc. IEEE International Conference On Robotics and Automation*, pages 288–293, 2000.

[59] R. Schapire. The boosting approach to machine learning: An overview, 2001.

[60] Towards 2020 science. Microsoft Research, Cambridge, 2005.

[61] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, 1988.

[62] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proc. International Conference On Very Large Databases (VLDB)*, pages 507–518, 1987.

[63] D. Sorokina, R. Caruana, and M. Riedewald. Additive groves of regression trees. In *Proc. European Conference On Machine Learning (ECML)*, pages 323–334, 2007.

[64] Y. Theodoridis and T. Sellis. A model for the prediction of R-tree performance. In *Proc. ACM Symposium On Principles of Database Systems (PODS)*, pages 161–171, 1996.

[65] A. Thiagarajan and S. Madden. Querying continuous functions in a database system. In *Proc. ACM SIGMOD International Conference On Management Of Data*, pages 791–804, 2008.

[66] L. Torgo. Kernel regression trees. In *Proc. European Conference On Machine Learning (ECML)*, pages 80–89, 1997.

[67] S. R. Turns. *An Introduction to Combustion: Concepts and Applications*. McGraw-Hill Science/Engineering/Math, 2000.

[68] I. Veljkovic, P. Plassmann, and D. C. Haworth. A scientific on-line database for efficient function approximation. In *Proc. International Conference In Computational Science And Its Applications (ICCSA)*, pages 643–653, 2003.

[69] http://www.fluent.com.

[70] http://www.femlab.com.

[71] http://hadoop.apache.org/core/.

[72] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. International Conference On Very Large Databases (VLDB)*, pages 194–205, 1998.

[73] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. International Conference On Very Large Databases (VLDB)*, pages 194–205, 1998.

[74] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 516–523, 1996.