

Indexing For Function Approximation

Biswanath Panda¹, Mirek Riedewald¹

Stephen B. Pope², Johannes Gehrke¹, L. Paul Chew¹

¹Dept. of Computer Science
Cornell University

²Dept. of Mechanical & Aerospace Engineering
Cornell University

{bpanda,mirek,johannes,chew}@cs.cornell.edu

pope@mae.cornell.edu

ABSTRACT

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. INTRODUCTION

Studying physical phenomena through computer simulation is an important method of scientific research. Application areas include studies of heat and mass transfer, fluid dynamics, combustion, evaporation and many more [1, 2]. 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 [25], 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 [14, 21, 22].

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 [14, 18].

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 [20, 22, 25]. 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 [22] 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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.

VLDB '06, September 12-15, 2006, Seoul, Korea.

Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

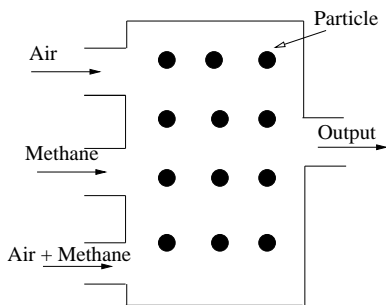


Figure 1: Combustion chamber

from the workloads studied so far in the literature [5, 10].

In summary, this paper makes the following contributions:

- We introduce the novel problem of high-dimensional function approximation as an application of high-dimensional indexing, and we describe an algorithmic framework that abstracts the salient elements of an HFA application (Sections 2 and 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 4).
- We perform a thorough evaluation of a set of candidate index structures from the database community for this application (Section 5).

Section 6 discusses related work and Section 7 concludes the paper.

We would like to emphasize that this paper is just the beginning of 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.

2. PROBLEM FORMULATION

We first introduce the basic problem of high-dimensional function approximation and then show in Section 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.

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). 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 $\phi^p(t) = \langle Y_1^p(t), Y_2^p(t), \dots, Y_s^p(t), h \rangle$, 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 [25].

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

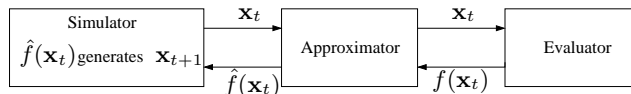


Figure 2: Application model

the rates of the inflow. **(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.

2.2 Application Model

Combustion simulation is representative of the class of applications that our methods apply to in general. Figure 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 function values. The approximator can calculate the exact value¹ 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.

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 $\hat{f}(\mathbf{x}_i)$, $i = 1, \dots, n - 1$. 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

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

into local regions and modeling each region 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.

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 $X = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ of query points, find the smallest set $L = \{l_1, \dots, l_k\}$ 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 [16], 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 $\{R_{f,\hat{f}}(x_1,\varepsilon), \dots, R_{f,\hat{f}}(x_n,\varepsilon)\}$, 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 $X(i) = \{\mathbf{x}_1, \dots, \mathbf{x}_i\}$ and $L(i) = \{l_1, \dots, l_{k(i)}\}$, 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 $L(i)$.

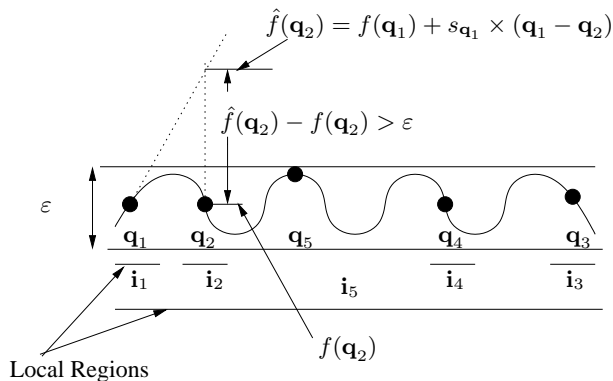


Figure 3: Example for non-competitiveness

A standard performance measure for online algorithms is the *competitive ratio* [6]. It measures the cost of the online algorithm for an input sequence $(\mathbf{x}_1, \dots, \mathbf{x}_n)$ 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 [6].

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 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, \dots, 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 are around points like q_1, \dots, 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.

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 analy-

Algorithm 1 : Framework

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 : Update

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

```

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

```

sis in 2.4 are discouraging, we start with algorithms that we abstracted from the approach of the domain scientists: A greedy heuristic (Section 3.1) and its refinements (Section 3.2). We then summarize the instantiation of choice of this framework by the domain scientists (Section 3.3).

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).

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 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 \mathbf{x} and updates them (Lines 2-6). Finally, only if no existing region can include \mathbf{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.

Algorithm 3 : Update

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

```

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 [22]. 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.

3.3.1 Initializing Local Regions

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

$$f(\mathbf{x}+\mathbf{a}) = \sum_{j=0}^k \left[\frac{1}{j!} (\mathbf{a} \cdot \nabla_{\mathbf{x}})^j f(\mathbf{x}) \right] + \phi_k(\mathbf{x}, \mathbf{a}), \quad (1)$$

where $\nabla_{\mathbf{x}}$ is the gradient $[\frac{\partial}{\partial x_1} \frac{\partial}{\partial x_2} \dots \frac{\partial}{\partial x_m}]$ 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 \mathbf{x} , the value at any point $\mathbf{x}+\mathbf{a}$ “near” \mathbf{x} is usually approximated using the first few terms of the summation in Equation 1. For example, using the first term only, we get a constant approximation $\hat{f}_{0,x}$ of f as follows:

$$f(\mathbf{x}+\mathbf{a}) \approx \hat{f}_{0,x}(\mathbf{x}+\mathbf{a}) \equiv f(\mathbf{x}) \quad (2)$$

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

$$f(\mathbf{x}+\mathbf{a}) \approx \hat{f}_{1,x}(\mathbf{x}+\mathbf{a}) \equiv f(\mathbf{x}) + (\mathbf{a} \cdot \nabla_{\mathbf{x}})f(\mathbf{x}) \quad (3)$$

The errors of the above approximations can be obtained from the remaining terms in the summation of Eq. 1, i.e., those terms not used in the approximation. For small values of $|\mathbf{a}|$, high-order terms in Eq. 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

$$\| (\mathbf{a} \cdot \nabla_{\mathbf{x}})f(\mathbf{x}) \| < \varepsilon, \quad (4)$$

and similarly for the linear approximation

$$\| \frac{1}{2!} (\mathbf{a} \cdot \nabla_{\mathbf{x}})^2 f(\mathbf{x}) \| < \varepsilon. \quad (5)$$

Equation 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

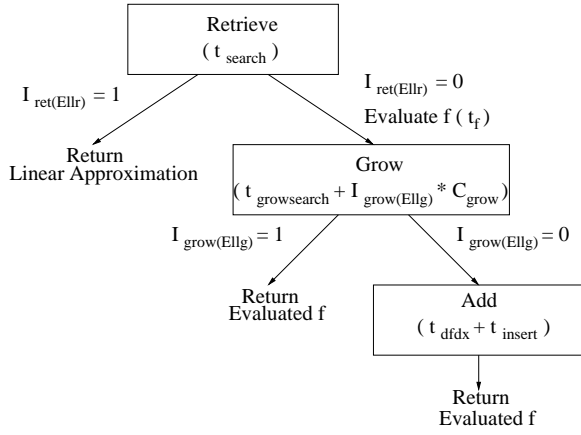


Figure 4: Block diagram

defined by it. However, it can be shown that Eq. 4 defines a hyper-ellipsoid around \mathbf{x} [22]. Therefore, using a constant approximation, the local region around \mathbf{x} is a hyper-ellipsoid.

3.3.2 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. 5 is difficult to compute, we have to start out with the more conservative (and hence smaller) region defined by Eq. 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 \mathbf{x} and an ellipsoid e around it. Suppose, there exists another query point \mathbf{x}' such that \mathbf{x}' lies just outside e but $\hat{f}_{1,\mathbf{x}}(\mathbf{x}')$ is an ε -approximation. Then \mathbf{x}' is assumed to be part of the Local Region of \mathbf{x} ; therefore e is grown to a larger ellipsoid that contains \mathbf{x}' . This simple heuristic of growing has been found to work well in practice [20]. 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.

3.3.3 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 \mathbf{x} .

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

Grow: If the retrieve fails then the algorithm attempts to grow existing ellipsoids in the index to include \mathbf{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).

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.

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 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

$$C_{\text{tot}} = t_{\text{search}} + I_{\text{ret}} \times t_{\text{la}} + (1 - I_{\text{ret}}) \times C_{\text{miss}} \quad (6)$$

Note that the the costs in Eq. 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 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} can therefore be written as

$$C_{\text{miss}} = t_f + t_{\text{growsearch}} + I_{\text{grow}} \times C_{\text{grow}} + (1 - I_{\text{grow}}) \times C_{\text{add}} \quad (7)$$

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
$t_{\text{growsearch}}$	Index search cost during grow
$t_{\text{inellipsoid}}$	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
N_{fpos}	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

Table 1: Index of variables

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

Table 2: Relative costs of the ellipsoid operations

The cost of a grow comprises two parts. $t_{\text{growsearch}}$ 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 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.

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 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 5.1). The additional ellipsoids that are examined by the index are called *false positives*, the number of which is denoted by N_{fpos} . Taking N_{fpos} into account, we can rewrite t_{search} as

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

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 ($t_{\text{inellipsoid}}$). 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, N_{fpos} is a fixed property of the index. However, the function approximation problem provides the flexibility to tune N_{fpos} , 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

$$C_{\text{tot}} = t_{\text{search}} + I_{\text{ret}(\text{Ellr})} \times t_{\text{la}} + (1 - I_{\text{ret}(\text{Ellr})}) \times C_{\text{miss}} \quad (8)$$

$I_{\text{ret}(\text{Ellr})}$ 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, $I_{\text{ret}} \geq I_{\text{ret}(\text{Ellr})}$. 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 $I_{\text{ret}(\text{Ellr})} = 1$ 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: $t_{\text{growsearch}} + I_{\text{grow}} \times C_{\text{grow}}$. 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 $C_{\text{miss}} =$

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

$I_{\text{grow}(\text{Ellg})}$ 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 writ-

ten as

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

Tuning Ellg introduces the following tradeoffs:

Effect 4: Decreasing Ellg decreases $t_{\text{growsearch}}$ 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 $\min\{N_{\text{found}}, N_{\text{growmax}}\}$. 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}).

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

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 $N_{\text{grown}} > 1$ 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 $I_{\text{ret(ElIrr)}}$. 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 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 [22]. 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 approximation 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 (ElIrr), 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.

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 timesteps, 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 execution on a Windows XP machine with a 2.4Ghz processor and 2GB of memory.

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 6); as a representative we selected the robust R*-tree [3]. Bbox Rtree indexes the axis-parallel minimum bounding boxes of the ellipsoids, 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 [10]. 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

