Database Techniques to Improve Scientific Simulations

Biswanath Panda, Johannes Gehrke, and Mirek Riedewald
Cornell University, Ithaca, NY, USA

Synonyms

Indexing for online function approximation

Definition

Scientific simulations approximate real world physical phenomena using complex mathematical models. In most simulations, the mathematical model driving the simulation is computationally expensive to evaluate and must be repeatedly evaluated at several different parameters and settings. This makes running large-scale scientific simulations computationally expensive. A common method used by scientists to speed up simulations is to store model evaluation results at some parameter settings during the course of a simulation and reuse the stored results (instead of direct model evaluations) when similar settings are encountered in later stages of the simulation. Storing and later retrieving model evaluations in simulations can be modeled as a high dimensional indexing problem. Database techniques for improving scientific simulations focus on addressing the new challenges in the resulting indexing problem.

Historical Background

Simulations have always been an important method used by scientists to study real world phenomena. The general methodology in these application areas is similar. Scientists first understand the physical laws that govern the 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 that motivated the line of work discussed in this entry. 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 (10–70 dimensions). 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 called the reaction function, which, given the current composition vector of a particle and other simulation properties, produces a new composition vector for the particle. Combustion

L. Liu, M. T. Özsu (eds.), *Encyclopedia of Database Systems*,
DOI 10.1007/978-1-4899-7993-3_1289-2

simulations usually require up to 10^8–10^{10} reaction function evaluations each of which requires in the order tens of milliseconds of CPU time. As a result, even small simulations can run into days.

Due to their importance in engineering and science, many algorithms have been developed to speed up combustion simulations. The main idea is to build an approximate model of the reaction function that is cheaper to evaluate. Early approaches were offline, where function evaluations were collected from simulations and used to learn multivariate polynomials approximating the reaction function [8]. These polynomials were then used later in different simulations instead of the reaction function. Recently, more sophisticated models like neural networks and self organizing maps have also been used [3]. The offline approaches were not very successful because a single model could not generalize to a large class of simulations. In 1997, Pope developed the In Situ Adaptive Tabulation (ISAT) Algorithm [7]. ISAT was an online approach to speeding up combustion simulations. The algorithm cached reaction function evaluations at certain frequently seen regions in the composition space and then used these cached values to approximate the reaction function evaluations at compositions encountered later on in the simulation. The technique was a major breakthrough in combustion simulation because it enabled scientists to run different simulations without having to first build a model for the reaction function. Until today, the Algorithm remains the state of the art for combustion simulations. Several improvements to ISAT have been proposed. DOLFA [9] and PRISM [1] proposed alternative methods of caching reaction function evaluations. More recently, Panda et al. [6] studied the storage/retrieval problem arising out of caching/reusing reaction function evaluations in ISAT and this entry mainly discusses their observations and findings.

Foundations

Even though the ISAT algorithm was originally proposed in the context of combustion simulations the algorithm can be used for building an approximate model for any high dimensional function (f), that is expensive to compute. This section begins with a discussion of Local Models, that represent the general class of models built by ISAT (section "Local Models"). This is followed by a description of the ISAT Algorithm that uses selective evaluations of the expensive function f to build a local model (section "ISAT Algorithm"). The algorithm introduces a new storage/retrieval, and hence indexing, problem. The section then discusses the indexing problem in detail: its challenges and solutions that have been proposed (sections "Indexing Problem" and "An Example: Binary Tree") and concludes with some recent work on optimizing long running simulations (section "Long Running Simulations").

Local Models

Local Models are used in many applications to approximate complex high dimensional functions. Given a function $f: \mathbf{R}^m \rightarrow \mathbf{R}^n$; a local model defines a set of high dimensional regions in the function domain: $\mathrm{R} = \{R_1 \ldots R_n | R_i \subseteq \mathbf{R}^m\}$. Each region R_i is associated with a function $\hat{f}_{R_i} : R_i \rightarrow \mathbf{R}^n$; such that $\forall \mathbf{x} \in R_i : || \hat{f}_{R_i}(\mathbf{x}) - f(\mathbf{x}) || \leq \epsilon$; where ε is a specified error tolerance in the model and $||$ is some specified error metric such as the Euclidean distance. Using a local model to evaluate f at some point $\mathbf{x}$ in the function domain involves first finding a region ($R \in \mathrm{R}$) that contains $\mathbf{x}$ and then evaluating $\hat{f}_R(\mathbf{x})$ as an approximation to $f(\mathbf{x})$.

ISAT Algorithm

Main algorithm: ISAT is an online algorithm for function approximation; its pseudocode is shown in Fig. 1. The algorithm takes as input a query point $\mathbf{x}$ at which the function value must be computed and a search structure S that stores the regions in R. S is empty when the simulation starts. The algorithm first tries to compute the function value at $\mathbf{x}$ using the local model it has built so far (Lines 2–3). If that fails the algorithm computes $f(\mathbf{x})$ using the expensive model (Line 5) and uses $f(\mathbf{x})$ to update existing or add new regions in the current local model (Line 6). The algorithm is

```
ISAT Algorithm
Require: Query Point x, inde x structure $S$
if $\exists\langle R, \hat{f}_R\rangle \in S$ such that $x \in R$
    Compute $y = \hat{f}(x)$
else
    Compute $y = f(x)$
    Update($S$, x, $f(x)$)
end if
return y
```

Database Techniques to Improve Scientific Simulations, Fig. 1 Pseudocode of the ISAT algorithm

```
Updating a local model
Require: $S$,x,$f(x)$
if $\exists\langle R, \hat{f}_R\rangle \in S$ : x can be included in $R$
    for all $\langle R, \hat{f}_R\rangle \in S$
        if x can be included in $R$
            Update $\langle R, \hat{f}_R\rangle$ to includex
        end if
    end for
else
    Add new $\langle R, \hat{f}_R\rangle$ to $S$
end if
```

Database Techniques to Improve Scientific Simulations, Fig. 2 Pseudocode for updating a local model

online because it does not have access to all query points when it builds the model.

Model updating: ISAT updates the local model using a strategy outlined in Fig. 2. In general it is extremely difficult to exactly define a region R and an associated $\hat{f}_R$, such that $\hat{f}_R$ approximates f in all parts of R. ISAT proposes a two step process to discover regions. It initially starts with a region that is very small and conservative but where it is known that a particular $\hat{f}_R$ approximates f well. It then gradually grows the conservative approximations over time. More specifically, the update process first searches the index S for regions R where $\mathbf{x}$ lies outside R but $||\, \hat{f}_R(\mathbf{x}) - f(\mathbf{x})\,|| \leq \epsilon$. Such regions are grown to contain $\mathbf{x}$ (Lines 2–7). If no existing regions can be grown a new conservative region centered around $\mathbf{x}$ and associated $\hat{f}_R$ is added to the local model (Line 9). The grow process described is a heuristic that works well in practice for functions that are locally smooth. This assumption holds in combustion and in many other applications.

Instantiation: While the shape of the regions and associated functions can be arbitrary, the original ISAT algorithm proposed high dimensional ellipsoids as regions and used a linear model as the function in a region. The linear model is initialized by computing the f value and estimating the derivative of f at the center of the ellipsoidal region:

$$\hat{f}_R(\mathbf{x}) = f(\mathbf{a}) + F'_{\mathrm{a}} \times (\mathbf{x} - \mathbf{a}),$$

where $\mathbf{a}$, is the center of the region R and $F'_{\mathbf{a}}$ is the derivative at $\mathbf{a}$.

ISAT performs one of the following high level operations for each query point $\mathbf{x}$. **Retrieve**: Computing the function value at $\mathbf{x}$ using the current local model by searching for a region containing $\mathbf{x}$. **Grow**: Searching for regions that can be grown to contain $\mathbf{x}$ and updating these regions in S. **Add**: Adding a new region (R) and an associated $\hat{f}_R$ into S.

Indexing Problem

The indexing problem in function approximation produces a challenging workload for the operations on index S in Figs. 1 and 2. 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. There are two main observations which make this indexing problem different from traditional indexing [2, 4]:

- The regions that are stored in the index are not predefined but generated by the add and grow operations. Past decisions about growing and adding affect future performance of the index, therefore the algorithm produces an uncommon query/update workload.
- The framework in which indexes must be evaluated is very different. Traditionally, the performance of index structures has been measured in terms of the cost of a search and in some cases update. There are two distinct cost factors in the function approxima-

tion 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 region 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.

In the light of these observations a principled analysis of the various costs in the function approximation algorithm leads to the discovery of novel tradeoffs. These tradeoffs produce significant and different effects on different index structures. Due to space constraints, only a high-level discussion of the various costs in the algorithm and the associated tradeoffs is included. The remainder of this section briefly describes the tradeoffs and the tuning parameters that have been proposed to exploit the different tradeoffs. The indexing problem is studied here using the concrete instantiation of the ISAT algorithm using ellipsoidal regions with linear models. Therefore, regions are often referred to as ellipsoids in the rest of the section. However, it is important to note that the ideas discussed are applicable to any kind of regions and associated functions.

Tuning Retrieves

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." The additional ellipsoids that are examined by the index are called false positives. For each false positive the algorithm pays to search and retrieve the ellipsoid from the index and to check if the ellipsoid contains the query point. 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, the number of false positives is a fixed property of the index. However, the function approximation problem provides the flexibility to tune the number of false positives, because the expensive function can be evaluated 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. This parameter is denoted by Ellr. Ellr places an upper bound on the number of false positives for a query. Tuning Ellr controls several interesting effects.

- *Effect 1:* Decreasing Ellr reduces the cost of the retrieve operation as fewer ellipsoids are retrieved and examined.
- *Effect 2:* Decreasing Ellr decreases the chances of finding an ellipsoid containing the query point thereby resulting in more expensive function evaluations.
- *Effect 3:* Misses that result from decreasing Ellr can grow and add other 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. This is a more subtle effect unique to this problem.

Tuning Grows and Adds

Just like the retrieve, the grow and add operations can be controlled by the number of ellipsoids examined for growing denoted as Ellg. Since an add is performed only if a grow fails, this parameter controls both the operations. Ellg provides a knob for controlling several effects that affect performance of the index and the algorithm.

- *Effect 4:* The first part of the grow process involves traversing the index to find ellipsoids that can be grown. Decreasing Ellg reduces the time spent in the traversal.
- *Effect 5:* Decreasing Ellg decreases the number of ellipsoids examined for the grow and hence the number of ellipsoids actually grown. This results in the following effects.
  - *Effect 5a:* Reducing the number of ellipsoids grown reduces index update costs that can be significant in high dimensional indexes.
  - *Effect 5b:* Growing a large number of ellipsoids on each grow operation indexes more

parts of the function domain, thereby improving the probability of future retrieves.
  - *Effect 5c:* Growing a large number of ellipsoids on each grow results in significant overlap among ellipsoids. Overlap among objects being indexed reduces search selectivity in many high dimensional indexes.
- *Effect 6:* Decreasing Ellg increases the number of add operations. Creating a new region is more expensive than updating an existing region since it involves initializing the function $\hat{f}_R$ in the new region.

In summary, the two tuning parameters have many different effects on index performance and the cost of the simulation. What makes the problem interesting is that these effects often move in opposite directions. Moreover, tuning affects different types of indexes differently and to varying degrees, which makes it necessary to analyze each type of index individually.

An Example: Binary Tree

The previous section presented a qualitative discussion of the effects that tuning Ellr and Ellg can have on index performance and simulation cost. This section makes the effects outlined in the previous section more concrete using an example index structure, called the Binary Tree. The 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. Figure 3 shows an example tree for three ellipsoids A, B, C and two cutting planes X and Y. We focus on the tree in the top part of Fig. 3 and use it to describe the operations supported by the index.

Retrieve

There are two possible traversals in the index that result in a successful retrieve.

Primary Retrieve. The first called a Primary Retrieve is illustrated 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 the example. When a leaf node is reached, the ellipsoid in the leaf is checked for the containment of the query point. In the example, A contains $\mathbf{q}_2$, and hence, a successful retrieve.

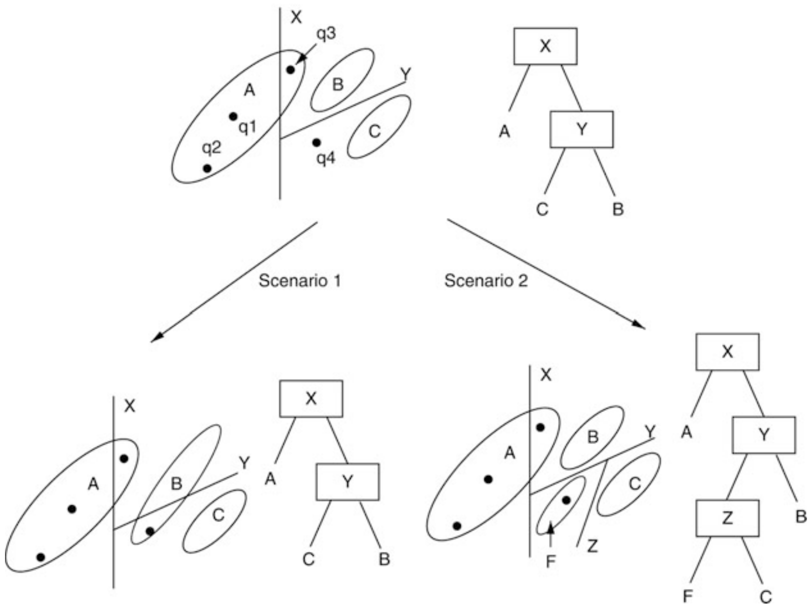
Secondary Retrieve. Since the binary tree only indexes centers, ellipsoids can straddle cutting planes, e.g., A covers volume on both sides of cutting plane X. If ellipsoids are straddling planes, then the Primary Retrieve can result in a false negative. For example, $\mathbf{q}_3$ lies to the right 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 more expensive 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, C is examined first(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.

Update

Scenario 1 (Grow) and 2 (Add) of Fig. 3 illustrate the update operations on the index.

Grow. 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 Fig. 3, 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.

Add. The alternative to growing B is illustrated on the bottom right part of Fig. 3 (Scenario 2).



Database Techniques to Improve Scientific Simulations, Fig. 3 Binary tree

Assume Ellg = 1, i.e., after examining C, the grow search ends unsuccessfully. Now a new ellipsoid F with center $\mathbf{q}_4$ is added 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.

Tuning parameter Ellg affects the Binary Tree in its choice of scenario 2 over 1. This choice, i.e., performing an add instead of a grow operation, reduces false positives for future queries, but adds extra-cost for the current query. Experiments on real simulation workloads have shown that this tradeoff has a profound influence on the overall simulation cost [6].

Long Running Simulations

When ISAT is used in long running combustion simulations ($\geq 10^8$ time steps), updates to the local model are unlikely after the first few million queries and the time spent in building the local model is very small compared to the total simulation time. Based on these observations, Panda et al. have modeled a long running combustion simulation as a traditional supervised learning problem [5]. They divide a combustion simulation into two phases. During the first phase, the ISAT algorithm is run and at the same time $(\mathbf{x}, f(\mathbf{x}))$ pairs are sampled uniformly from the composition space accessed by the simulation. At the end of the first phase, the sampled $(\mathbf{x}, f(\mathbf{x}))$ pairs are used as training data for a supervised learning algorithm that tries to find a "new" local model with lower retrieve cost than the model built using ISAT. This new model is then used for the remainder of the simulation. Their experiment shows that the algorithm adds little overhead and that the new model can reduce retrieve costs by up to 70% in the second phase of the simulation.

Key Applications

The ISAT algorithm and its optimizations have primarily been applied to combustion simulation workloads. However, the ideas are applicable to any simulation setting that requires repeated evaluations in a fixed domain of a function that is locally smooth and expensive to compute are required.

Cross-References

▸ Spatial and Multidimensional Databases

Recommended Reading

1. Bell JB, Brown NJ, Day MS, Frenklach M, Grcar JF, Propp RM, Tonse SR. Scaling and efficiency of PRISM in adaptive simulations of turbulent premixed flames. In: Proceedings of the 28th International Combustion Symposium; 2000.
2. Böhm C, Berchtold S, Keim DA. Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases. ACM Comput Surv. 2001;33(3):322–73.
3. Chen JY, Kollmann W, Dibble R. A self-organizing-map approach to chemistry representation in combustion applications. Combust Theor Model. 2000;4: 61–76.
4. Gaede V, Günther O. Multidimensional access methods. ACM Comput Surv. 1998;30(2):170–231.
5. Panda B, Riedewald M, Gehrke J, Pope SB. High speed function approximation. In: Proceedings of the 2007 IEEE International Conference on Data Mining; 2007.
6. Panda B, Riedewald M, Pope SB, Gehrke J, Chew LP. Indexing for function approximation. In: Proceedings of the 32nd International Conference on Very Large Data Bases; 2006.
7. Pope SB. Computationally efficient implementation of combustion chemistry using in situ adaptive tabulation. Combust Theor Model. 1997;1:41–63.
8. Turanyi T. Application of repro-modeling for the reduction of combustion mechanisms. In: Proceedings of the 25th Symposium on Combustion; 1994. p. 949–55.
9. Veljkovic I, Plassmann P, Haworth DC. A scientific on-line database for efficient function approximation. In: Proceedings of the International Conference on Computational Science and its Applications; 2003. p. 643–53.