Fall 2004

Granite: A scientific database model and implementation

Philip J. Rhodes
University of New Hampshire, Durham

GRANITE: A SCIENTIFIC DATABASE MODEL AND IMPLEMENTATION

BY

Philip J. Rhodes

BA, University of Virginia, 1991

MS, University of Rhode Island, 1995

DISSERTATION

Submitted to the University of New Hampshire

in Partial Fulfillment of

the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science

September, 2004

This dissertation has been examined and approved.

[illegible]

Dissertation Director, R. Daniel Bergeron, Professor

[illegible]

Dissertation Director, Ted M. Sparr, Professor

[illegible]

Georges G. Grinstein, Professor

[illegible]

Philip J. Hatcher, Professor

Elizabeth Varki

Elizabeth Varki, Associate Professor

6/24/04

Date

This dissertation has been examined and approved.

Dissertation Director, R. Daniel Bergeron, Professor

Dissertation Director, Ted M. Sparr, Professor

Georges G. Grinstein, Professor

Philip J. Hatcher, Professor

Elizabeth Varki, Associate Professor

Date

DEDICATION

To my parents, Christopher and Josephine, who have always been supremely supportive and inspiring.

ACKNOWLEDGEMENTS

I must first thank my dissertation directors, Dr. R. Daniel Bergeron, and Dr. Ted M. Sparr. They have dedicated an enormous amount of time and effort in guiding me through this work, teaching me the value not only of hard work but also creative thought. My gratitude for their work and time is profound.

My committee members, Dr. Georges Grinstein, Dr. Philip Hatcher, and Dr. Elizabeth Varki have all offered valuable advice and suggestions. They have done a lot of reading and editing on my behalf, and I very much appreciate their efforts.

I've had the pleasure of working with a collection of very capable Master's students over the years, without whom the Granite system would not exist. My thanks to Xionghui Chen, Lorna Ellis, Joseph Hempfling, Feng Jiang, Wenhui Li, Gang Lu, Denise Mitchell, Xuan Tang, and Li Ye.

I would also like to thank my friends from New Hampshire, Rhode Island, and Babcock Hall, of which there are too many to list here. At the risk of leaving someone important out, let me thank Maria Agorastou, Lexi Barnett, Nat Barnett, Cris Caruso, Beth Demers, Genevieve Ellison, Susan Engel, Rada Filip, Lauren Howard, Zorana Ivcevic, Deepak Jadhav, Mike Kern, Bob Laramee, Nina Laramee, Denise Liliedahl, Pete Lisichenko, Kate McCafferty, Kate Oparowsky, Edin Pasic, Todd Patenaude, Renee Percy, Khrystyna Pisareva, Lou Ramirez, Mark Randall, Lincoln Ross, Libby Ryan-Kern, Sally, Lara Skinner, Mary Sytek, Mikkel Thrane, Anne Tocker, Nathan Vooge, and Dale Wright. Some of you have lent me particular support during tough times, and you know that I will always be especially grateful for that. However, you are all mentioned here not just because you have been a fun and enjoyable friend, but because I have felt some connection with you that helped me through the labors that this document represents.

My thanks also to Linda Spring Andrews, Andy Evans, Gerry Pregent, and Gina Ross for providing a pleasant and efficient working environment. By clearing many little obstacles out of my way, you have helped me accomplish my goal.

Lastly, I would like to thank the National Science Foundation, which supported my work under grant IIS-0082577 and also the University of New Hampshire, which supported me with a Summer Fellowship for Teaching Assistants. The work presented here could not have been done without this funding.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

GRANITE: A SCIENTIFIC DATABASE MODEL AND IMPLEMENTATION

by

Philip J. Rhodes

University of New Hampshire, September, 2004

The principal goal of this research was to develop a formal comprehensive model for representing highly complex scientific data. An effective model should provide a conceptually uniform way to represent data and it should serve as a framework for the implementation of an efficient and easy-to-use software environment that implements the model. The dissertation work presented here describes such a model and its contributions to the field of scientific databases. In particular, the Granite model encompasses a wide variety of datatypes used across many disciplines of science and engineering today. It is unique in that it defines dataset geometry and topology as separate conceptual components of a scientific dataset. We provide a novel classification of geometries and topologies that has important practical implications for a scientific database implementation. The Granite model also offers integrated support for multiresolution and adaptive resolution data. Many of these ideas have been addressed by others, but no one has tried to bring them all together in a single comprehensive model.

The datasource portion of the Granite model offers several further contributions. In addition to providing a convenient conceptual view of rectilinear data, it also supports multisource data. Data can be taken from various sources and combined into a unified view.

The rod storage model is an abstraction for file storage that has proven an effective platform upon which to develop efficient access to storage. Our spatial prefetching technique is built upon the rod storage model, and demonstrates very significant improvement in access to scientific datasets, and also allows machines to access data that is far too large to fit in main memory. These improvements bring the extremely large datasets now being generated in many scientific fields into the realm of tractability for the ordinary researcher.

We validated the feasibility and viability of the model by implementing a significant portion of it in the Granite system. Extensive performance evaluations of the implementation indicate that the features of the model can be provided in a user-friendly manner with an efficiency that is competitive with more ad hoc systems and more specialized application specific solutions.

CHAPTER 1

INTRODUCTION AND OVERVIEW

1.1 Introduction

Building effective tools for handling scientific data presents many challenges to system designers. Scientific data is often extremely large, and comes in a variety of types and formats. Traditional database systems simply do not effectively support large scientific data. In particular, they cannot efficiently represent the structure that is implicit in the geometric relationships between the data points. In contrast to traditional databases where relationships among items are explicitly known, a scientific database should assist the researcher in discovering the relationships hidden among the data.

The principal goal of this research is to develop a formal comprehensive model for representing highly complex scientific data. An effective model should provide a conceptually uniform way to represent different kinds of data and it should serve as a framework for the implementation of an efficient and easy-to-use software environment that supports the model. The dissertation work presented here describes such a model, called the *Granite Model.* This model encompasses a wide variety of datatypes and data organizations used across many disciplines of science and engineering today. We validated the feasibility and viability of the model by implementing a significant portion of it in the *Granite System.* Extensive performance evaluations of the implementation indicate that the features of the model can be provided in a user-friendly manner with an efficiency that is competitive with more ad hoc systems and more specialized application specific solutions.

The Granite Model consists of the *Lattice Model* and *Datasource Model,* implemented as separate layers in the Granite System. The rest of this chapter summarizes these major components and the contributions of the research.

1.2 Scientific Data Model

We define scientific data as a collection of sample values that represents some natural phenomenon [HIBB94]. We refer to the phenomenon being sampled as a function ϕ defined over a domain D. The *dataset* consists of a sampling of ϕ taken at a finite set of points $\Delta \in D$. Our model is specifically tailored to handle data defined over a continuous n-dimensional domain. We use the term *dimensional* to identify such datasets.

1.3 Lattice Overview

The Lattice layer of the model is the most general, and supports both *uniform* and *unstructured* data. An example of uniform data is *rectilinear* data. Uniform data is uniformly distributed in the geometry and has neighbors to the north, south, east, west, etc. Elements of unstructured data sets are placed arbitrarily throughout the dataset domain, and have some arbitrary number of neighbors that must be explicitly specified. Figures 1.1.a and 1.1.b show two and three dimensional rectilinear data, while figure 1.1.c shows a small two dimensional unstructured dataset.

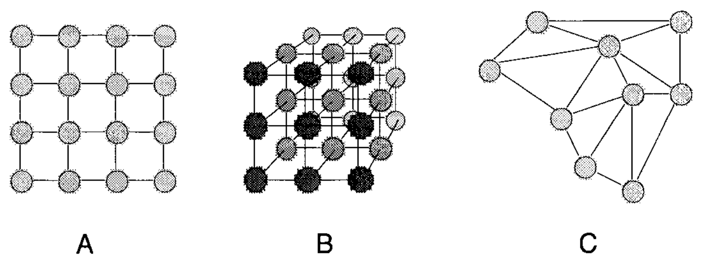


Figure 1.1. a) 2d Rectilinear data. b) 3d rectilinear data. c) 2d unstructured data

Our Lattice model is unique in that it separately represents *geometry* and *topology*. Geometry refers to the placement of sample points within a *domain*. Topology refers to the neighborhood relationships between points, regardless of their placement. *Cells*, an important topological concept, are often defined as regions bounded by arcs connecting neighboring points. Through the combination of different kinds of geometries and topologies, the lattice model can accommodate a wide variety of data formats within a single conceptual framework.

The Lattice itself contains several components. In addition to the geometry and topology, it also includes a *value space,* which specifies the set of values found in the data, and an approximating function, which is used to provide values for lattice locations that do not correspond to sample points.

The model allows users to access lattice data either geometrically or topologically. With geometric access, the user specifies a location in the domain from which data is returned. With topological access, the lattice provides an *iterator* that returns successive points or cells for user processing.

The current Lattice implementation supports two dimensional unstructured data with multiresolution, as well as n-dimensional rectilinear multiresolution data with the help of the datasource layer, described below. The implementation is very general and is easily extended to higher dimensionalities. However, Lattice development is sufficiently advanced to validate the feasibility of our model for scientific data.

1.4 Multiresolution

The lattice model provides support for both *multiresolution* (MR) and *adaptive resolution* (AR) data, in which a dataset is represented at several levels of detail, allowing a lattice user to access only the most interesting data at the finest resolution. These formats avoid storage or processing costs associated with uninteresting and unnecessary data. Granite builds support for these formats directly into the model, while other systems may require an experimenter to devise more *ad hoc* support.

A *multiresolution hierarchy* is a stack of lattices viewing the same n-dimensional volume at different resolutions. Typically, we think of these lattices as ordered vertically from the most detailed on the bottom to the least detailed on top. The spatial overlap of these lattices facilitates the correlation of coarse and fine views of the same regions. We use these spatial semantics to map a sub-volume vertically through the hierarchy using *support* and *influence.* Each neighboring set of points or cells in a coarse view is related to a (larger) set of neighboring points and/or cells in a finer view; this set forms the *support* for the items in the coarser view. Each point or cell in the finer view participates in the support for a set of items in the coarse view; this set in the coarse view is its *influence.*

An adaptive resolution representation allows resolution to vary within a single lattice. The resolution near a point may depend on the behavior of the sampling function, on the behavior of the error function, or on the nature of the domain in the neighborhood of the point. An AR representation is a coarse view with interesting regions replaced with data often taken from more detailed views acquired by drilling down an MR hierarchy. The

AR representation approximates the functional accuracy of the finer view with the memory cost of the coarser view.

It is possible to define a hierarchy of adaptive resolutions on the same data. Typically, each coarser level of this hierarchy is created using successively relaxed error tolerances. Because an AR hierarchy contains multiple resolutions within each level, it has the potential to achieve a representation with the same accuracy as MR using less storage. Alternatively, for a given amount of memory, it can retain increased detail and accuracy in important regions of the domain.

1.5 DataSource Layer

The DataSource Layer assists the Lattice layer in the handling of rectilinear data, though it can also be used alone. Conceptually, the datasource model represents rectilinear topologies using an array. The array axes form an *index space,* and each element of the array contains a single *datum*, which has one or more *fields* or *attributes. Physical* datasources may be directly associated with a file or network stream. A *composite* datasource combines one or more component datasources. For example, the *AttributeJoinDataSource* can join one or more attributes taken from each of several component datasources to produce a single, unified representation of the several component datasets. Similarly, the *BlockJoinDataSource* can form a single view of several component datasources by joining their index spaces. These two datasources form the core of our support for *multisource* data, in which data is combined from several different sources.

The datasource model also supports adaptive resolution for cell oriented rectilinear data. The *ARRCellDataSource* uses various tree data structures to present a single representation of the data that contains different resolutions for different regions of the index space.

1.6 Spatial Prefetching

Physical datasources must address the problem of efficiently reading data from a one dimensional file in order to populate an n-dimensional index space. The *rod storage model* has proven an effective platform for developing solutions to this problem. Coupled with the development of several kinds of iterators, the rod storage model has also led to significant research on caching and prefetching. This work culminated in the development of the *spatial prefetching* technique, described in chapter 5. Spatial prefetching not only greatly accelerates access to data on disk, it also brings very large datasets within reach of the Granite system. For example, at the end of chapter 5 we describe an interactive Granite application used with the 39GB *Visible Female* dataset, provided by the National Institutes of Health.

1.7 Contributions

This document describes a collection of important contributions to the field of scientific database systems. The Granite model is unique in that it defines dataset geometry and topology as separate conceptual components of a scientific dataset. We provide a novel classification of geometries and topologies that has important practical implications for a scientific database implementation. Unlike the systems commonly in use today, the Granite model also offers integrated support for multiresolution (MR) and adaptive resolution (AR) data. AR and MR formats attempt to reduce the cost of representing or processing data that has resolution higher than required for the task. Many of these ideas have been addressed by others, but no one has tried to bring them all together in a single comprehensive model.

The datasource portion of the Granite model offers several further contributions. In addition to providing the user with a convenient conceptual view of rectilinear data, it also offers support for multisource data. Data can be taken from various files or network sources and combined using an *attribute join* or *block join*, still providing a unified view of the combined data.

The *Granite System* is our implementation of the Granite model, and is not only a working system that provides useful and novel functionality, but also serves to validate the effectiveness and feasibility of the model. The system supports both unstructured trimesh datasets and n-dimensional rectilinear datasets. With the help of the datasource layer, the Granite system also handles adaptive resolution for rectilinear cell and point based data.

The *rod storage model* is an abstraction for file storage that has proven an effective platform upon which to develop efficient access to storage. Our *spatial prefetching* technique is built upon the rod storage model, and demonstrates very significant improvement in access to scientific datasets. It not only speeds access to datasets, it also allows machines to access data that is far too large to fit in main memory. These improvements bring the extremely large datasets now being generated in many scientific fields into the realm of tractability for researchers using conventional machines.

The remainder of this document begins with an overview of background and related work, followed by a description of the Lattice layer, both the model and implementation. The next two chapters address the model and implementation of the Datasource layer, and *spatial prefetching,* respectively. We end with conclusions and a discussion of future work.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Introduction

Although traditional databases have been around for many years, they are not well suited for scientific data. They do not handle the tremendous size of scientific datasets well, and there is a fundamental mismatch between the design of traditional databases and the operations scientists want to perform on scientific data. One important weakness is in the role of metadata. In traditional databases, metadata is mainly *structural*, describing the types and relationships of the various attributes. In scientific databases, the relationships within the data are initially unknown; it is these relationships that the scientist hopes to discover through an exploration of potentially huge datasets. Appendix B contains a discussion of metadata issues. For a discussion of the unique features of scientific data, see [PFALTZ98].

The process of exploring scientific data should be as interactive as possible, even though large volumes of data can make this difficult. Recently, researchers have been studying the advantages of multiresolution (MR) representations for scientific data [CIGNO94, CIGNO97, DEBE98, HECK97,STOLL96, WONG95]. MR representations allow the same data to be examined different resolutions. Examining a coarse representation of the data can provide enormous savings in processing and storage costs, thereby enhancing interactivity. These advantages can be extended further by employing a distributed and/or parallel processing system to increase the speed at which data is retrieved, visualized, and manipulated. Appendices C and D contain further discussion regarding both MR and distributed processing.

The remainder of this chapter reviews the field of scientific databases and scientific data, providing the necessary background for the rest of this dissertation. The appendices contain a much expanded discussion of these same issues as well as additional areas that are not strictly necessary for understanding the remainder of this document.

2.2 Scientific Data

A scientific database should be able to represent or model scientific data gathered either from the real world, or from simulation. In other words, a set of scientific data is a collection of sample values that represents some "natural" phenomenon [HIBB94].

We refer to the phenomenon being sampled as a function ϕ defined over a domain D. The *dataset* consists of a sampling of ϕ taken at a finite set of points $\Delta \in$ D. A *mesh* consisting of the points of Δ along with connecting edges generally spans the domain D. Cignoni, *et al.* postulate a function f which interpolates values of ϕ for domain points not in Δ [CIGNO97]. The mesh assists the approximating function, since edges of the mesh connect points, and also form regions within which values can be approximated or interpolated.

Notice that in order for this model to be useful, the domain D must be defined over 1 or more *dimensions*. For example, in 3D Cartesian space these dimensions would correspond to the x, y, and z axes.

We focus our research on data that can be meaningfully represented in a continuous k-dimensional data space. Practically speaking, if one or more independent attributes of the data can be mapped to the set of real numbers $\Re$, then the data is dimensional for our purposes. In the event that a researcher wishes to use non-metric data as a dimension in a scientific database, Kao [KAO97] has developed techniques for imposing a metric on data that would otherwise be considered categorical or nominal.

2.3 Scientific Databases

It is the job of the scientific investigator to develop hypotheses that explain the natural world. An important part of a scientist's work is to collect data either from the real world or from simulation, and compare this data with values predicted by the hypothesis. Since it is important not to contaminate the collected data in any way, scientific datasets are not usually modified once they have been loaded into the database system. Data may be viewed in different ways, but the values themselves are not changed, although new derived datasets are often created. In contrast, an important part of traditional databases is the *update* operation, which changes existing values [PFALTZ98].

2.4 The Multiresolution Representation

A Multiresolution (MR) dataset contains several representations of the same data at different resolutions. These different representations are referred to as *levels*. According to Cignoni *et al.*[CIGNO97], the number of levels in a true MR dataset depends upon the size of the original data. If the number of levels is constant regardless of data size, we have a *Level of Detail* (LoD) representation [CIGNO97]. Conceptually, there is a correspondence between a value v at level i and one or more values $\{v_0..v_i\}$ at level $i-1$. For this reason, authors sometimes use the term *hierarchical* to describe MR techniques. The precise nature of this correspondence between levels depends upon the particular MR method used.

2.5 Adaptive Resolution

MR techniques can be divided into two groups. In *non-adaptive MR*, the resolution used to represent the dataset is constant throughout the domain for any one level. Adaptive techniques, which are usually called Adaptive Resolution (AR) are able to vary the resolution of an area of the domain depending on the behavior of the data within that area [CIGNO94, LARAMEE02]. Local resolution may be reduced if this coarser resolution still represents the area with an acceptable amount of error. Another important application of AR is to reduce resolution in areas that are considered uninteresting in some sense.

A multiresolution representation allows a researcher to view data using resolutions ranging from low (very coarse) to high (the original data). Using a low resolution can vastly reduce the size of the data that needs to be stored, manipulated, and displayed. It also serves as an overview of the entire dataset, allowing the researcher to pick out regions of interest without examining the original data directly. Once an interesting region has been determined, the researcher may examine it at higher, more detailed resolutions, perhaps even descending to the original data. Note that higher resolutions are thought of as being below lower levels, with the original data on the very bottom. Descending through this hierarchy is called "drilling down" [OLAP]. Drilling down allows the researcher to examine only data of interest at high resolution, minimizing processing and display costs.

By using Adaptive Resolution, we can save storage space as well as processing costs. In this case, the hierarchy is much the same as before, except that a single level can contain data at different resolutions where uninteresting areas are represented as coarsely as possible.

2.6 Multidimensional Access Methods

When processing scientific data, it is important to consider *inter-instance relationships*. While relational databases are good at representing relationships between attributes, they are not well suited for representing relationships between instances of the same attribute.

Spatial Data Models use a different approach. Here, data points are represented using coordinates in a vector space [KAO97]. If data is *dimensional* it can be represented using a spatial data model. Because spatial data models are so different from the relational data model, their use introduces a new set of problems and techniques.

Gaede *et al.* list several kinds of multidimensional (MD) queries [GAEDE98]. *Exact Match, Intersection, Enclosure, Containment, Nearest Neighbor* and *Adjacency Queries* all take the spatial extent of an object *o*, and return the set of objects that properly answer the query. For example, the containment query returns the set of objects contained in the extent of *o*. The *Point Query* takes a single point as argument, and returns the objects that intersect that point. The *Window Query* returns the set of objects that intersect with the *d*-dimensional interval $I^d = [l_1, u_1] \times [l_2, u_2] \times \cdots \times [l_1, u_1]$. I^d should be aligned with the domain axes, or *iso-oriented.* The *Nearest Neighbors Query* returns the set of objects with minimum distance from *o*.

2.7 Access to Large Datasets

Providing efficient access to huge scientific datasets is a challenging problem, and has attracted a lot of attention from both operating system and scientific data management communities. Work has focused on either providing comprehensive scientific data and metadata management systems, or optimizing file systems using techniques like prefetching, caching and parallel I/O.

2.7.1 File Access

Reorganizing datasets on disk to speed access has been explored by a number of researchers. Sarawagi and Stonebraker [Sarawagi94] describe *chunking*, which groups spatially adjacent data elements into n-dimensional chunks which are then used as a basic I/O unit, making access to multidimensional data an order of magnitude faster. They also arrange the storage order of these chunks to minimize seek distance during access. Following this work, many other reorganization methods have been developed. More and Choudary [More00] reorganize their data according to the expected query type, and the likelihood that data values will be accessed together. The Active Data Repository (ADR) uses chunking to reduce overall access costs and to achieve balanced parallel I/O [CChang00, CChangADR].

2.7.2 Prefetching and Caching

Software prefetching has been used by many researchers to hide or minimize the cost of I/O stalling. In the file systems arena, approaches to this problem can be distinguished by whether or not prefetching is guided by explicit information about the access pattern. Albers *et al.* [Albers98] describe an algorithm that produces an optimal schedule for prefetching and discarding cache blocks when the entire access pattern is given in advance. Other researchers have explored the case where the access pattern is disclosed less completely in the form of *hints*. Patterson *et al.* [Patterson95] developed a framework for informed caching and prefetching based on a cost-benefit model. This model has been extended to account for storage devices with very different performance characteristics [Forney02]. Cao *et al.* have had success by letting applications have control of data cache replacement strategy in their share of cache blocks [Cao96].

When no explicit information about access pattern is available, the history of prior accesses can be used to predict future accesses. Amer et al. group files together based on historical file access patterns [Amer02]. Other researchers have used probability trees or graphs to represent the likelihood of future block accesses given past and current block accesses [Vellanki99, Highley02, Highley03, Griffioen94]. Madhyastha *et al.* use a hidden Markov model to automatically predict file access patterns over time; the file system adaptively selects appropriate caching and prefetching policies according to the detected pattern [Madhyastha96, Madhyastha97].

At the application level, Chang [Chang01] adds a separate thread to the user program that performs prefetching by mimicking the I/O behavior of the main thread and preloading data. The VisTools [Nadeau] system is most similar to our approach. It provides an application level data prefetching and caching service for huge multidimensional datasets using the Paged-Array schema. It reads formatted pages of elements from the underlying files when the first element in the page is requested. Then, the formatted pages are stored in a *page cache* for fast future re-access. When the cache size limit is reached, the paged-arrays are deleted or written to a swap file. Like our own work, paged-arrays also support intelligent prefetching guided by iterators that have an *n*-dimensional view of the dataset. However, the one dimensional nature of pages fails to take into account the proximity of elements in N-dimensional space. By using pages as its unit of cache storage, VisTools and other page based methods may make poor decisions about what data to retain or discard.

2.8 Conclusion

We have presented a brief survey of the issues relevant to scientific databases, and to the design and implementation of the Granite system. For a more in-depth examination of these issues, the reader is invited to look at appendices B, C, and D at the end of the document.

CHAPTER 3

THE LATTICE LAYER

3 Introduction

Our formal scientific data model provides a conceptual framework for defining and processing a wide range of scientific data. This chapter describes those aspects of the model that are encapsulated in the Lattice layer of the model and summarizes the current state of the lattice implementation in the Granite system.

3.1 Dimensional Data

A scientific database should be able to represent or model data gathered either from the real world, or from simulation. We focus our research on data that can be meaningfully represented in a continuous k-dimensional data space. If a dataset consists of some attributes that are ordinal[1] , independent, and defined on a continuous value range, we can

[1] An attribute is ordinal if its values have a complete ordering.

say that the dataset contains *dimensional data*, and that these attributes are *dimensional.*

It is not necessary for all attributes to be dimensional. If we view the data as a function, some of the dimensional attributes define the domain of this function, and the remaining attributes define the range. Our function therefore maps any point in the domain defined by the dimensions to a particular range value. Choosing which attributes should be used as dimensions is up to the researcher using the system, and can be an important part of the data exploration process. We call each possible combination of dimensions a *view* of the data: a notion similar to the "view" found in traditional databases. So, for data with *k* dimensional attributes, there are 2^k-1 possible views. Each view affords a different way of looking at the same data.

A natural example of dimensional data is *spatial data* such as satellite images and fluid flow datasets. Here, the data represents an actual physical space. However, it is possible for a dataset to be dimensional without being spatial. For example, data from a Greenland ice core sample might contain readings for calcium, nitrogen, and carbon concentrations at different times in the Earth's history. Even though this data does not correspond to a real space, it may be very beneficial to visualize the data as if it were spatial, since humans find this representation familiar and easy to grasp. For this reason, we often use the word "spatial" in this document, even when referring to data which does not represent a physical space.

It may be convenient to treat a set of attributes as if they are dimensional attributes even though they may not satisfy all the conditions for dimensional data. In particular, we often don't know exactly which attributes are independent of each other, but we might want to assume they are independent for exploration purposes with the goal of either validating or disproving that assumption.

3.2 Geometry and Topology

Granite's scientific data is n-dimensional, and like most systems that manipulate such data, we must develop efficient ways of mapping between data that is fundamentally multidimensional and a storage scheme such as a file or array that may not directly reflect the dimensionality of the data.

With this in mind, it is useful to examine the complexity of the mapping between the dataset and an n-dimensional array. We call the space formed by an array an *index space*, a concept used extensively in the datasource layer, described in the next chapter. The *geometry* of a dataset consists of the dataset spatial domain *D*, and the placement of the set of sample points within that domain. Generally, if the sample points are distributed throughout this geometry with uniform spacing, we say that the data is *uniform.* Figure 3.1 shows three uniform geometries in one, two, and three dimensions.

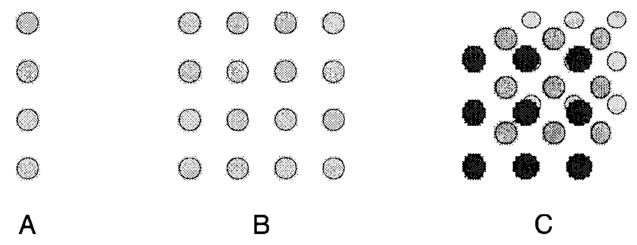


Figure 3.1. Three uniform geometries in one, two and three dimensions

The mapping from index space to geometry can be defined as a function:

$$f(I,W) \rightarrow p \in D$$

where I is the index space, D is the geometry domain, p is a location in D, and W is the auxiliary information consisting of a set of numbers required to perform the mapping. Geometries can be classified by examining the size of this set. For uniform geometries, the only extra information needed is the spacing between the points for each dimension. In this case, the size of W (i.e. $|W|$) is O(d), where d is the dimensionality of an index space.

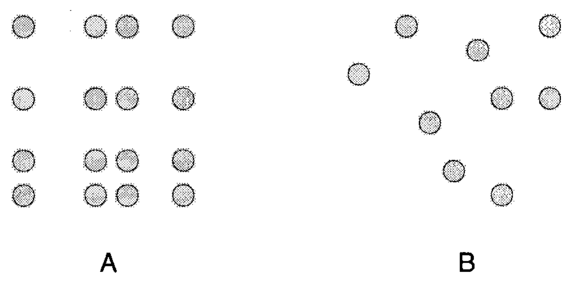


Figure 3.2. Two examples of non-uniform data. The left example is sometimes called a *perimeter lattice*, while the one on the right is known as *unstructured data*.

If $|W|$ is not O(d), we say the geometry is *non-uniform.* Figure 3.2 shows two examples of non-uniform geometry. However, important distinctions can still be made between different kinds of non-uniform geometries. In figure 3.2a, the spacing between the points is consistent between each row and column, so we can perform the mapping using just two arrays, each storing the spacing between each row and column. The length of these two arrays is related to n, the number of points. For example, if the dataset is square, $|W|$ will be $O(\sqrt{n})$. In contrast, the geometry in figure 3.2b has no pattern whatsoever, and requires that the position of each point be given explicitly. In this case, where $|W|$ is O(n), we say the geometry is *unstructured.* Accordingly, the less difficult situation shown in figure 2a is an example of a *semi-uniform* geometry, meaning $|W|$ is greater than O(d) but less than O(n).

The *topology* of a dataset refers to the way that points are connected to each other. A dataset's topology is a graph, with data points as nodes and arcs between nodes representing a *neighbor* or *adjacency* relationship. Often, the researcher wants to view the data using the geometry, but the system most efficiently accesses the topology, since it is the topology that gives a dataset its structure. Figure 3.3 gives examples of some different topologies.

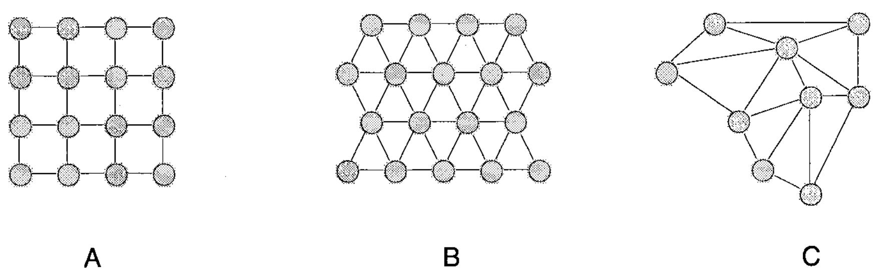


Figure 3.3. Various topologies

As with geometries, we can also classify topologies according to an index space mapping. Since the topology is a representation of the neighborhood relationship, this mapping function should produce a set of neighbor nodes for any node represented by a point in the index space. That is,

$$f(I,W) \rightarrow N$$

where W is defined as before, but N is a set of neighbor nodes. As before, a topology for which $|W|$ is O(d) is uniform. If $|W|$ is O(n), the topology is unstructured, and we use *semi-uniform* for the cases in between. Figures 3.3.a and 3.3.b are both examples of uniform topologies, while figure 3.3.c is an unstructured topology.

A particularly important kind of uniform topology is the *rectilinear* topology. In a two dimensional rectilinear dataset, the topology of the data points is a rectangular grid. For three dimensions, the topology is a hexahedral (e.g. cubic) mesh. Figure 3.4 shows several rectilinear topologies. Notice that figures 3.4.d and 3.4.e do not have uniform geometries. Various combinations of geometric and topological types are possible.

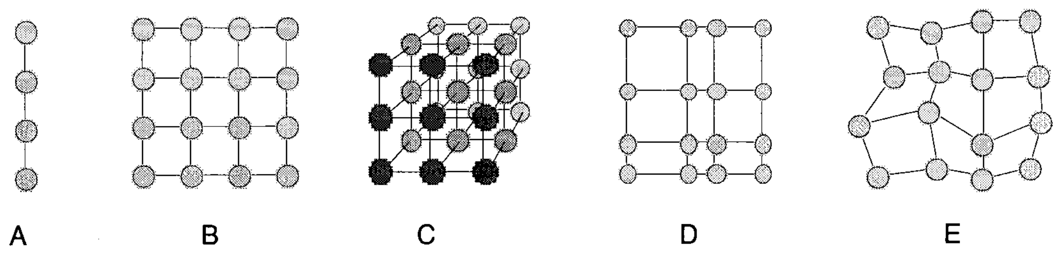


Figure 3.4. Some rectilinear topologies

Within an array, if two elements of an array have an index that differs by only one, we consider the elements to be neighbors. Therefore, we can exploit the natural topology of the array to represent the topology of rectilinear data.

In cases where both the topology and geometry is unstructured, the array representation can offer only storage space. For two dimensional datasets of this kind a mesh of triangles can be created such that the vertex of each triangle is a known data point. These meshes are commonly called *trimeshes*. Figure 3.3.c shows a small trimesh for a non-uniform dataset. This approach can also be used to handle three dimensional surfaces, where the vertices now have three coordinates instead of two. For true three dimensional volume data sets, the triangle is replaced with a tetrahedron. This process can be extended to handle dimensions greater than three.

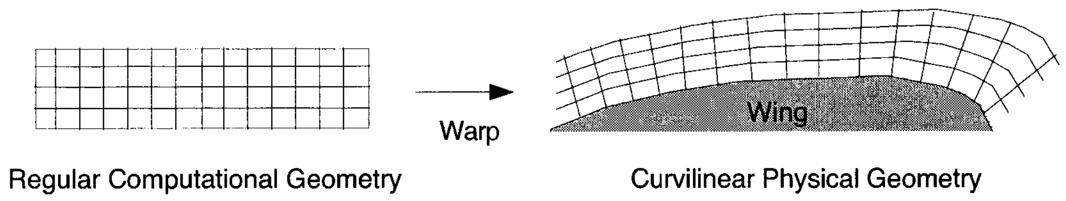


Figure 3.5. Warping a regular grid to a curvilinear dataset

It is sometimes possible to map a rectilinear grid to a set of points that is not uniformly distributed in the geometry. For example, a fluid flow simulation of air velocities over the top surface of an airplane wing might produce samples that lie in concentric curves echoing the shape of the wing. We say this arrangement of points is *curvilinear.* However, a rectilinear grid can be *warped* to the physical space to provide a dataset that is regular in computational space. Figure 3.5 illustrates the warping transformation.

3.3 Periodic Tilings and Data

The study of tilings (tessellations) has some relevance to our research since topologies often define a tiling. A tiling is an arrangement of contiguous shapes that cover a domain. A review of this field can be found in [Schatt97].

If a tiling is *periodic*, then it is possible to duplicate the tiling, translate it some distance, and place it down again so that it matches exactly with the original copy. That is, the tiling consists of a number of translated repetitions of some pattern of tiles. An important and related property of periodic tilings is that there exists a subset of the space S that can be repeatedly copied and translated throughout the space to complete the tiling. A minimal subset of this kind is called a *fundamental domain* or *generating region.* A *regular tiling* is a periodic tiling made up of identical regular polygons [Schatt97]. The three tilings shown in figure 3.6 are the only regular tilings for 2D space. Other shapes do not meet the mathematical constraints required for a single shape to tile the plane.

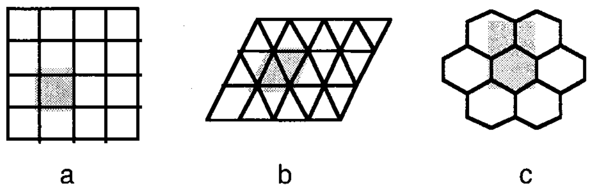


Figure 3.6. The fundamental domains of the 2D regular tilings

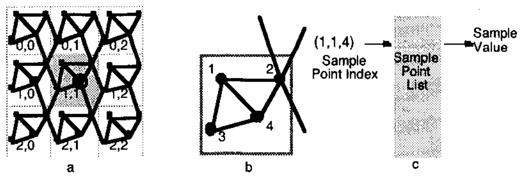


Figure 3.7. A possible supercell implementation.

Although tilings are defined entirely in terms of geometry, the concepts can also be applied to our notion of topology. We use the notion of the *supercell* to represent periodic sampling topologies. As shown in figure 3.7, a supercell represents a generating region for the topology, allowing the entire topology to be conceptually represented by a grid of repeated supercells while only storing a single supercell definition. If we can find where in the grid a point lies, we can very easily form a search key from the position in the grid (i.e., supercell identifier), and the position of the point within the supercell (i.e., point identifier). Such a technique promises a quick way to access a point's data given its geometric position.

3.4 Neighborhood

Many scientific applications require access to the *neighborhood* of a point. Generally, the neighborhood of a point p is a contiguous set of points containing p and points that are near p. Deciding which points are near p depends on whether we are computing a *geometric neighborhood* or a *topological neighborhood.* A typical geometric neighborhood might include all points that are within distance d of p in the geometric space. On the other hand, a topological neighborhood might include all points that are within n arcs of p. Notice that these two kinds of neighborhood are not normally equivalent. For example, if we map a geometric neighborhood to the dataset's topology, the result need not be a topological neighborhood. The corresponding fact is also true when we map a topological neighborhood to the dataset's geometry.

Despite this, the dataset topology may still be useful in generating a geometric neighborhood. In order to retrieve sample points near p in geometric space, the system may traverse the dataset topology and test the geometric location of sample points to see if they should be included in the geometric neighborhood.

3.5 A Model for Scientific Data

In order to develop a multiresolution (MR) model of scientific data, we must first define a model for scientific data. We define scientific data as a collection of sample values that represents some natural phenomenon [HIBB94]. We'll view this phenomenon as a function over a domain D, which might be time, space, radio frequency, etc. or some multidimensional combination. In the next two sections, we describe our general requirements for representing scientific data, followed by a description of the *lattice* representation. First developed by Bergeron and Grinstein [BERG89], the lattice satisfies our requirements for a data representation.

3.5.1 Data Representation

An investigator using a scientific database should be able to represent hypotheses in the system, and determine whether the data supports the hypothesis. A *hypothesis model* is represented as a function H:

$$H: D \rightarrow V$$

That is, the hypothesis model maps every point in the domain D to a point in a value space V. The elements of a value space are defined on arbitrary n-tuples of values.

The goal of the investigator is to find a hypothesis that describes the natural phenomenon as closely as possible. Therefore, we need to represent the phenomenon within the scientific database. This representation is called a *data representation.* Of course, since the domain D is continuous, it contains an infinite number of points. Therefore a data representation represents the phenomenon at a finite set of points within the domain [HIBB95]. This set of points, known as *sample points*, constitutes our scientific data. We represent the sample points with a *sampling function* [KAO97]. We'll denote the finite set of domain points as $\Delta \subset D$. A sampling function f_Δ maps Δ to a value space V :

$$f_\Delta: \Delta \rightarrow V$$

Notice that in order to implement a sampling function, we only need Δ, the set of sample points, and a set of data values *v*, such that each $v_i \in v$ is the value measured at a corresponding $d_i \in \Delta$. We also require a localized error function E_Δ, known as the *sampling error*, that indicates the error for any sample point by mapping Δ to an error space E:

$$E_\Delta: \Delta \rightarrow E$$

Together, f_Δ and E_Δ model the behavior of a measuring instrument as it gathers data from the real world. They may also model a resampling from another dataset. In many cases, the error (or accuracy) of the values is not known, so E_Δ may default to zero. In other cases, an instrument or dataset has a known accuracy and precision, so a better error estimate can be made.

It is convenient to package the domain and data values into a single item. We define a *data representation* as:

$$R = \langle D, V, \Delta, f_\Delta \rangle$$

That is, a data representation consists of the domain and value space for the data, along with the data values, and sampling function. Our definition of a data representation is quite general. In the next section, we discuss the *lattice,* an elaboration on the representation presented here.

So far, we can only give values and error for any point $d \in \Delta$. In order to generate values for points that may not be sample points, we need an *approximating function.* An approximating function is defined as:

$$f_A : D \rightarrow V$$

The approximating function takes any point in the domain, and returns a value that approximates the value of the phenomenon at that point, based on the sampling function. An important class of approximating functions is the interpolation functions, which must satisfy the following condition:

$$\forall d \in \Delta,\ f_A(d) = f_\Delta(d)$$

That is, for a point in the sample domain Δ, the interpolation function and the sampling function must produce the same value. Usually, the approximating function is an interpolation function. For any data set, a large number of approximating functions are possible. Some are better for a selected task than others. Hypothesis development can be viewed as the process of discovering increasingly better approximating functions.

We are now ready to define a *data model.* A data model consists of a data representation and error functions E_Δ and E_D along with the approximating function f_A.

$$M = \langle R, E_\Delta, E_D, f_A \rangle$$

The error representation E_D should be defined over the entire domain D. That is, for every $d \in D$, $E_D(d)$ provides an estimate of the data model error associated with the point d. When a data model is first produced from the original data, we have E_Δ, the error associated with the sampling function, which is defined only over Δ. One possible definition of the initial E_D could be f_{E_Δ}, which uses the approximation function to find values E_D from the values of E_Δ. Data models can also be derived from other data models through a function that introduces additional error. We'd like E_D to record the total *cumulative* error in the data model.

3.5.2 The Lattice Representation

The *lattice* model can be extended to meet our requirements for a data representation. The lattice includes both topological and geometric views of the dataset. We can refer to the *dimensionality* of a topology. Intuitively, this is the dimensionality of the space required to contain the topology graph without having any arcs intersect. The first three diagrams of figure 3.4 in section 3.2 illustrates one, two, and three dimensional topologies of the simplest kind.

As described in [BERG89], a lattice L_n^k has k dimensions that define a space, and n attributes for a point located in that space. We say that k is the dimensionality of the lattice. It is also the dimensionality of the lattice topology. So, a 0-dimensional lattice is simply an unordered set, while a 1-dimensional lattice is an ordered list, a 2-dimensional lattice defines a plane, and so on. Notice that the lattice geometry does not need to have the same dimensionality as the lattice topology. As an example, Kao points out that a 2D lattice can be mapped to a surface that exists in three dimensional space [KAO97].

Perhaps the simplest form of lattice is the rectilinear lattice, which has a rectilinear topology, as described in section 2.2. Of course, such a lattice can easily be represented using a rectangular array. Kao defines a lattice to be a function from an *index space* to a value space [KAO97]. Given indices into the array, we can easily retrieve the value stored there. In addition, we need a way to map points in our geometric space to index space. Kao calls this mapping a *logical transformation*. Although the logical transformation for a regular lattice is trivial, it is not nearly so apparent for data that is non-uniform with respect to geometry. We'd like to extend the lattice model to handle a much wider variety of data formats, and redefine a lattice to be a function that maps the geometric space to the value space. That is, we'd like to include the logical transformations in the lattice itself. More formally, a lattice is defined as:

$$L = \langle D, V, \Delta, \tau, f_\Delta \rangle$$

where D is the geometric domain, V is the value space, Δ is the location of the data points within D, τ is the lattice topology, and f_Δ is the sampling function, which produces points in V. Note that the lattice meets the requirements for a data representation, and also explicitly includes the lattice topology. For a lattice L, we denote a member of the lattice using dot notation. For example, $L.D$ refers to the domain of a lattice L, and $L.V$ refers to that lattice's value space.

3.5.3 Lattice Transformations

We need to define transformations that can be applied to lattices. We can characterize such transformations by noting their effect on the lattice value space, geometry, and topology. These transformations normally do not change the lattice they are applied to. Instead, a new lattice representing the result of the transformation is created.

We call transformations that change data *value transformations,* described as a function T_V:

$$T_V\colon L \rightarrow L' \text{ where } L.f_\Delta \neq L'.f_\Delta$$

For example, suppose a transformation simply normalizes values to the range [0…1]. We create a new lattice L' and apply the normalization operation to the sample values of the old lattice, producing the new sample values for L', contained in $L'.f_\Delta$.

It is important to realize that a value transformation does not necessarily just map one value space to another. A value transformation may use the location of a value in the domain and its relationship with surrounding values. For example, consider a value transformation that produces a value 1 if a domain location corresponds to a local maximum for attribute *A,* and 0 otherwise. The value 5 may occur many times in a lattice, but it may be a local maximum only once. Our transformation can't just map the value 5 to a single value. Rather, it must map one occurrence of 5 to 1, and the others to 0, and can only do so by looking at values surrounding an occurrence of 5. Therefore, this transformation must examine values contained by subdomains of the lattice. On the other hand, this is not true of all value transformations.

A transformation that changes the lattice geometry is really changing the lattice domain, so we call such transformations *domain transformations*, described as a function T_D:

$$T_D{:}L \rightarrow L' \text{ where } L.D \neq L'.D$$

That is, the transformation acts upon a domain D and produces a new domain D'. Creating a new domain implies changes in $L'.f_\Delta$ as well. Examples of domain transformations include *affine* transformations like scaling, rotation, translation and shear. Bergeron [BERG89] also describes *extensions* and *restrictions*. An extension is a mapping to a higher dimensional space. Restrictions include projection to a lower dimensional space, and also the generalized subset operation. The subset operation is normally considered to be purely geometric. However, a subset can be computed either on the basis of domain shape, or through an examination of data values. Computing a subset via data values is an important tool for reducing the amount of uninteresting data contained in a lattice.

Lastly, a *topological transformation* is defined as

$$T_\tau : L \rightarrow L' \text{ where } L.\tau \neq L'.\tau$$

For example, extensions and restrictions can apply to the topology as well as geometry. Also, any subset operation that removes points or arcs from the lattice can be characterized as a topological transformation.

3.6 Multiresolution Representations for Data

This section presents our model of multiresolution (MR) data, and describes an important subclass known as *adaptive resolution.*

3.6.1 The MR model

The key to our description of a multiresolution model is a *reducing operator* R:

$$\mathsf{R} : M \rightarrow (M', E_{\mathsf{R}})$$

This function takes a data model *M* as an argument and returns a new data model *M'*, along with associated localized error E_R that was introduced into *M'* by the operator. Remember that any data model contains a data representation, error functions E_Δ and E_D, and an approximating function. It is important to note that the domains of *M* and *M'* are both D, the domain of the natural phenomenon. $M'.E_D$ is a composition of $M.E_D$ and E_R, to reflect the error that R introduced. The data representation *M'.R* and sampling error $M'.E_\Delta$ must also differ from their counterparts in *M* since it is a requirement for any reducing operator that:

$$|\Delta'| < |\Delta|$$

In other words, $M'.R$ should contain fewer sample points than $M.R$. It is for this reason that R is called a reducing operator. This change in Δ causes $M'.f_A$ to differ from $M.f_A$ since the reduction in data points is likely to change the approximating function.

An MR hierarchy $\mathcal{M}$ is a pair of items:

$$\mathcal{M} = \langle \Lambda, \mathrm{P} \rangle$$

where Λ is a sequence of *levels* $\{\Lambda_0 \ldots \Lambda_n\}$ and P (*rho*) is a sequence of reducing functions $\{\mathsf{R}_0 \ldots \mathsf{R}_{n-1}\}$. Each Λ_i in Λ is defined to be a pair containing a data model and associated localized reduction error:

$$\Lambda_i = \langle M^i, E_i \rangle$$

Each R_i in P is defined as:

$$\mathsf{R}_i : \Lambda_i \rightarrow \Lambda_{i+1}$$

That is, the reducing function R_i maps the finer level Λ_i to the coarser level Λ_{i+1}. The MR hierarchy is formed by repeated applications of reducing operations. First, the original data is stored in M^0 which is then stored in level Λ_0. Since no reduction has yet occurred, E_0 is assumed to be null. Next, a reducing operator is applied to M^0 to form M^1 and an error E_{R} which now becomes E_1. M^1 and E_1 make up level Λ_1. The process is repeated an arbitrary number of times, typically depending on whether the size of the data has been reduced sufficiently, or further reductions would produce too much error.

3.6.2 Resolution

In some MR hierarchies, the reducing operator works uniformly over D. We call these methods *non-adaptive MR*, since the reducing operator doesn't change or adapt over the domain. For example, if the points in Δ are uniformly distributed in D, R might discard alternate data points so that Δ' is half the size of Δ. Such a reduction would affect the accuracy of the new data model similarly over all of D. Of course, it is most appropriate to use a uniform reducing operator if the original data is evenly distributed, and of equal interest. On the other hand, if the data distribution is uneven, or some data is more interesting to the researcher than the rest, another kind of reducing operator may be more suitable.

3.6.3 Adaptive Resolution

A reducing operator that behaves differently over parts of D is a hallmark of a subset of MR known as *adaptive resolution* (AR). An *AR hierarchy* is a kind of MR hierarchy in which an AR reducing operator is used. An AR reducing operator still needs to reduce the size of Δ, but it is more sophisticated in how it chooses to do so. For example, an AR reducing operator might examine the cumulative error for a data model, and attempt to reduce resolution in areas with lowest error when forming the data model for the next level. Alternatively, it might try to preserve resolution in areas of rapid value change, and instead reduce resolution from less volatile areas. In general, an AR reducing operator's behavior over the domain is determined by the data and the requirements of system designers or perhaps the experimenter.

3.6.3.1 AR Representations vs. AR Hierarchies

When an AR reducing function is used to produce a series of levels, the result is an AR hierarchy. On the other hand, a representation of a domain with resolution that varies in response to data values is known as an *AR representation.* The levels of an AR hierarchy are actually AR representations. An AR representation is *not* a hierarchy; any point or region of the domain is represented only once. In fact, an AR representation can be formed by navigating a non-adaptive MR hierarchy, and choosing suitable nonoverlapping regions from different levels. These distinctions are shown in figure 3.8.

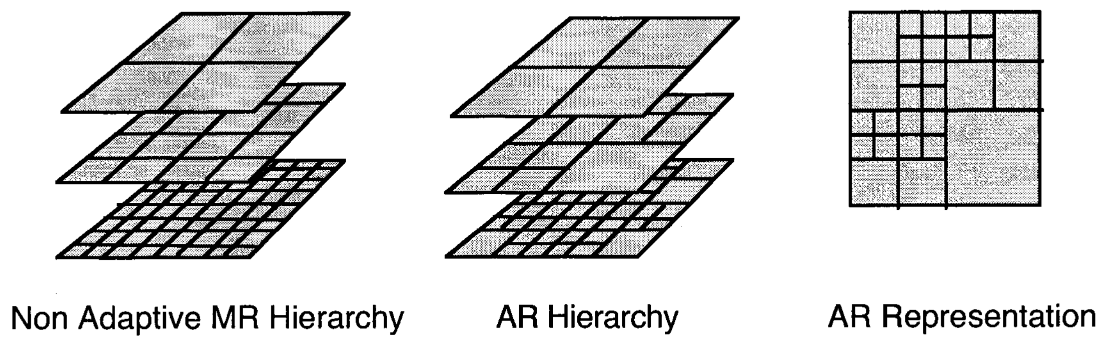


Figure 3.8. Non-adaptive MR and AR hierarchies, and an AR representation

3.6.3.2 Locally Monotone Reductions

Consider two levels of a hierarchy Λ_a and Λ_b, where Λ_a is coarser than Λ_b. The only restriction our model specifies is that Λ_a must contain fewer points than Λ_b. This leaves open the possibility for some subdomain to be represented with *more* points in Λ_a than Λ_b, even though Λ_a contains fewer total points. For example, there may be two regions that have reduced resolution in Λ_a when compared to Λ_b, but one region where resolution in Λ_a is higher. Overall, this still satisfies the model. However, we feel that most of the time every region of the domain will be represented with either the same or less resolution in Λ_a as in Λ_b. In this case, we'll say that the reducing function that produced Λ_a from Λ_b is *locally monotone.* A rigorous definition of *locally monotone* requires a rigorous definition of what *locality* means. One possible definition is based on a partitioning of D. That is, given a partitioning of the domain, a reducing function is locally monotone with respect to the partitioning if it either reduces or preserves the number of points within every partition, but never increases it.

3.6.4 Support and Influence

Our model for MR is very general. In practice, most MR hierarchies place further restrictions on the reducing function. Recall that reducing functions relate a level Λ_i to level Λ_{i+1}, where Λ_{i+1} is a coarser representation of the same domain as Λ_i. Most MR methods require that the reducing function be spatially coherent. That is, any contiguous set of points in Λ_i should map to a contiguous set of points in Λ_{i+1}. More formally, we could characterize a reducing function R as a collection of functions $\{r_0 \ldots r_n\}$ such that each r_i maps a contiguous set of points $S_i \subset \Delta$ in level Λ_i to a contiguous set of points $S_j \subset \Delta$ in level Λ_{i+1}. We say that S_i is the *support* for S_j. The union of all S_i for any level should equal Δ. Notice that this allows the domains of each r_i to overlap, meaning that a point p in Λ_i might belong to the support for several different points in Λ_{i+1}. We call the set of these points the *influence* of p. Both support and influence are shown in figure 3.9. A common example of this overlapping support is when the reducing function computes weighted averages for several points of Λ_i, to produce a new point for Λ_{i+1}.

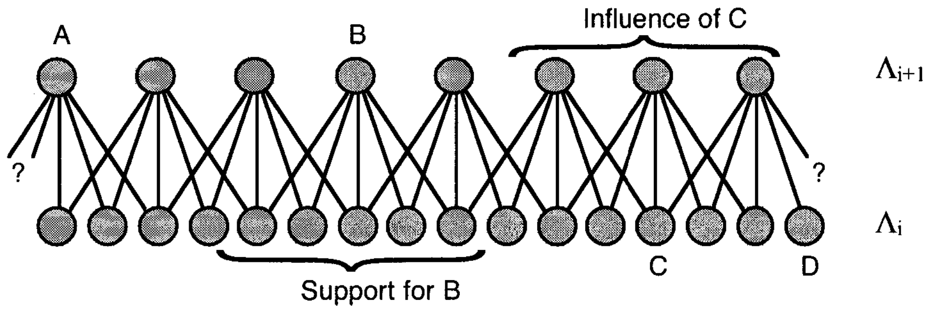


Figure 3.9. Support and Influence

Such overlapping support can introduce problems at the boundaries of the domain. For example, consider a reducing function in which each point in Λ_{i+1} is supported by overlapping sets of points in Λ_i, as seen in figure 3.9. Points like *A*, located at the edge of Λ_{i+1} may be missing some support from Λ_i. For the same reason, points like *D*, located at the edge of Λ_i will be under-represented in Λ_{i+1} because they influence only one or two points in Λ_{i+1} instead of the expected three.

3.6.5 MR for Regular Data

A common implementation of non-adaptive MR uses an array of points to represent the data set function S. Since Λ_0 is the original data, we expect that Λ_1 represents the same information at a coarser resolution, i.e. with fewer data points. A simple way to do this is to have each point in Λ_i represent 2^d points of Λ_{i-1}, where d is the dimensionality of the dataset. So, for a one dimensional dataset, the first point of Λ_1 should represent points 0 and 1 of Λ_0, the second point should represent points 2 and 3, and so on. So, Λ_1 is half the size of Λ_0, and Λ_2 half the size of Λ_1, etc. This approach can be extended to any number of dimensions. For example, in three dimensions, each point of Λ_1 represents eight points of Λ_0, which is analogous to the familiar octtree data structure used commonly in computer graphics.

Another important issue is how to combine two or more points into a single point for the next level. The method used depends upon the application. In the simplest case, where each point represents one value, we might just average points together to get a single value. However, it might be desirable for a point in Λ_i to keep track of attributes like minimum and maximum value, and standard deviation for all the points it represents in Λ_{i-1}. Li *et al.* [LI98] store a probability density function for each point in their MR hierarchy.

3.6.6 MR for Irregular Non-Rectilinear Data

With regular data, the reducing operator can easily do something simple like removing every other point. Researchers [DEBE98] have also developed non-adaptive MR techniques that work on triangular meshes. However, the implementation of non-adaptive reducing functions for triangular meshes is not trivial. For this reason, we envision that such data will more commonly be represented using AR, while uniform data may use either AR or non-adaptive MR.

For example, Cignoni *et al.* [CIGNO97] outline a method for three dimensional data represented as a tetrahedral mesh. Their method produces a new, coarser level by removing the tetrahedron from the current level that causes the least error. Since this method looks at the data values (and their error) when deciding where to reduce resolution, this is an example of adaptive resolution. Notice that each level differs from the last by only one tetrahedron. This means that the number of levels will be large compared to a non-adaptive technique that removes half the points with each level.

3.6.7 Advantages of MR for Scientific Data

The principal advantage of the MR model is that it allows a researcher to examine a low resolution summary or overview of a dataset. Using this overview, the researcher can decide which areas of the data require more detailed examination. Because MR provides several levels of increasingly fine resolution, the area of interest can be progressively refined at each level. This process is known as "drilling down". It's very possible that the researcher's needs are met by a level of the MR hierarchy that is coarser than the original data. If so, there is likely to be a significant savings in network and visualization costs, since the coarser representation should be much smaller than the original data. If the researcher's needs are always met by levels above the original data, then storage costs are also saved, since there is no need to store the original data. Even if the researcher does need to descend to the bottom level of the hierarchy, the refinement of the area of interest as he or she descends means that less of the original data is required.

3.6.7.1 Noise Filtering with MR

Some kinds of MR hierarchies specify that each value of a level Λ_{i+1} is related to a small set of neighboring values from Λ_i. Notice that our definition of a reducing function allows this, but does not require it. However, if a reducing function averages values over some area of the domain, it may not only reduce the size of the next level's data set, but also eliminate noise. The reducing function is effectively working as a low-pass filter of the sort used in image processing. If imprecise instruments generate high frequency noise during data collection, a coarser level of the data might provide a better view than the original data.

3.6.7.2 Advantages of Adaptive Resolution for Scientific Data

Using an AR hierarchy yields all the advantages described in the previous paragraphs, along with some others. An AR hierarchy uses reducing functions that respond to the data, preferring to reduce resolution over uninteresting areas of the domain. In a way, this is an automation of the process the researcher would go through when drilling down through a non-adaptive MR hierarchy. If the researcher is allowed to choose the reducing functions used, we can say that choice is communicating his or her idea of what constitutes "interesting" data. Therefore, the AR hierarchy contains information about which areas of the domain are interesting to the researcher. Such information is very useful when trying to distribute data over several machines, since good load balancing should place a roughly equal amount of interesting data on each machine. Similarly, in a parallel environment, this information can be used to help minimize communication between processors.

3.7 Representing Domains

Our model requires a flexible representation for domains and subdomains, since they play an integral role in the representation of domain relative data. Since we model scientific data as dimensional data, we must represent the number and kind of dimensions that make up the domain space. Often, a dataset is defined within an infinite space such as $\Re^3$, which cannot be directly represented in the database. Therefore, we must represent this information symbolically in a *universal domain*. A universal domain defines the space that a dataset resides in by specifying the number and type of dimensions, along with a distance metric. The universal domain can be considered structural metadata. Notice that the domain of a dataset must be a finite subset or *subdomain* of this universal domain.

We clearly need a way to represent the domain of a dataset, but there are other applications that are less obvious, as we demonstrate in our discussion of semantic metadata. In particular, we'd like to represent subdomains in a very expressive way.

3.7.1 Stencils

A subdomain is a subset of some larger domain. Domain relative semantic metadata, described in section 6.3, lends support to the classification of regions and feature identification within the data. To support this application, our subdomain representation must be able to handle complex and irregularly shaped and perhaps disjoint subdomains, and even subdomains with fuzzy boundaries. In our model, the *stencil* is the object used to represent this wide variety of subdomains. Like a cardboard stencil used in painting letters and figures, our stencil represents the shape of a subdomain carved out of some larger, enclosing domain. A stencil may be a subdomain of the universal domain, or of another stencil. In either case, a stencil's *base domain* is the domain from which the stencil is taken. If the stencil is the result of a *filter,* described in section 5.4, the base domain is the domain that was examined by the filter. Regardless of how it is produced, a stencil is always a subset of this base domain. Historical metadata, described in section 6, can be created that relates the stencil to its base and universal domains, including the process through which it was selected from the base domain. The universal domain is needed in order to make sure that certain stencil operations are sensible. Such operations include the union, intersection, and subtraction operators, as well as transformations like scaling and translation. If two stencils are defined on entirely different domains, it probably isn't sensible to combine them with a union operation. Similarly, there should be restrictions on translation and scaling. For example, if one dimension of a domain is a probability, no subdomain should be translated along that axis so that it exceeds the range [0...1].

3.7.2 Stencil Representation

The stencil object must be able to represent a wide variety of different domain shapes. It is unlikely that a single representation can give such flexibility, so several underlying implementations are needed. In choosing an implementation, we must decide what kind of queries a stencil object must answer. Certainly, any stencil must be able to answer a query of the form:

$$Q: p \in D_{base} \rightarrow [0 \ldots 1]$$

That is, given a point p in the base domain of the stencil, return a value in the range [0...1] that indicates whether the point is outside (0) or inside (1) the subdomain represented by the stencil. In the case of *crisp stencils*, only 0 or 1 is returned, meaning that a point's membership in the subdomain is either total or none. For *fuzzy stencils*, intermediate values may be returned, indicating some uncertainty about a point's membership. Such uncertainty may be due to the criterion used to create the subdomain or to error in the data to which the criterion was applied.

Representing crisp stencils can be done efficiently by using a multidimensional (MD) access method, as discussed in [GAEDE98]. Since subdomains are essentially regions within an enclosing domain, we want to use an MD access method designed to store regions, rather than points. In order to answer the query above, a crisp stencil must determine whether it holds a region that contains the point and return 1 if such a region is found, and 0 otherwise. Regions can often be represented with dramatically less storage than a collection of points with the same spatial extent. For example, rectangular regions can be represented with just a pair of coordinates. Even complex curved regions can be represented with comparatively little storage using parametric curves or surfaces.

Fuzzy stencils may require more storage space, since they don't have sharp borders. One possibility is to use a representation similar to methods used for data. A lattice of points accompanied by an approximation function would certainly be capable of representing a fuzzy domain. Other methods may also have the advantage of saving storage space. For example, we should be able to use an AR representation to store border information in great detail, while saving storage space in more homogeneous areas.

3.7.3 MR Stencils

Imagine a researcher is developing a feature recognition algorithm. The researcher might generate a stencil that identifies areas of the domain where the feature exists by running the algorithm with low resolution data. This result could be compared with another stencil that was generated via high resolution data to see how well the algorithm performs with the low resolution information. In such a scenario, an MR stencil might be useful, since it would demonstrate the difference that resolution makes to the recognition algorithm's performance. An MR stencil could be even more useful if the recognition algorithm is trainable like a neural net. In this case the algorithm's success at low resolution could be enhanced by reward or punishment based on high resolution information.

Clearly, there are times when it is beneficial to always store stencils using MR, and therefore store domain information at different levels of resolution. For fuzzy stencils this is especially true, since membership in the subdomain is represented in a continuous fashion over the enclosing domain. For crisp stencils the domain can be stored very efficiently as regions using an MD access method, so we would only use MR if the application required it. In many cases, the researcher is interested in having the best possible information at his or her disposal. In such situations, the stencil may have only a single resolution that represents this best available information, perhaps taken from several resolutions of the data. In other situations the researcher may want to store even discrete stencils using MR for reasons having to do with the stencil's application.

3.7.4 Making Stencils

Since representing domains is a fundamental task in a scientific database, we must provide several ways of producing stencils. The simplest way of generating a domain is to specify its extent within the universal domain. Note that stencils are fixed within their base domain, although new stencils with a different placement can be created using the operations described in the next section. For example, a simple rectangular stencil can be made by specifying its corner points. Of course, a stencil is not restricted to rectangles, and can be any polygon, polyhedron, or curved shape. Such stencils are called *shaped stencils.* Shaped stencils are used to represent the domain of a dataset, and also for selecting areas or volumes of data defined on a larger domain.

In contrast, other stencils are produced by examining data values in a dataset. This examination is conducted by a *filter* which accepts or rejects areas of a domain depending on the data values found there. The output of the filter is stored in a *result stencil.* Normally we use *historical metadata,* described in section 6, to relate a stencil, filter, and result stencil. Such information can also be used to support *delayed evaluation.* That is, we don't require that a stencil be fully materialized upon creation. Instead, we can define the parameters of the stencil at creation time, and only compute the stencil's domain when it is actually needed. Delayed evaluation should be especially helpful with complex queries involving several filters. For example, consider a query on a satellite data set that asks for areas of land not covered by clouds that appear to contain urban development. Let's assume that filters are available to identify clouds, land, and urban development. One way of answering the query is to create three stencils that each refer to a filter. (Notice that the cloud stencil has to be negated.) The final result is formed by computing the intersection of the three stencils. By delaying the evaluation of the stencils until the entire query is answered we have an opportunity for optimization. Suppose the land filter is the most selective, and eliminates 80% of its base domain. It makes sense to evaluate this filter first, and feed the results to the cloud filter. By evaluating the most selective filter first, we can prevent the cloud filter from being run on areas that turn out to be ocean anyway.

There are two wrinkles with this sort of optimization. First, we must have at least an estimate of how selective a filter is likely to be. This could be computed by running the filter on a very low resolution version of the data to get a rough measure of how selective it is. The other possibility is to get an estimate from the experimenter. In either case, the estimate can be updated as the filter is used. The second wrinkle is that some filters may require other filters to be run first. For example, if the land filter in the last example performs very poorly if clouds are left in the data, we'd have to evaluate the query differently.

3.7.5 Stencil Operations

In addition to representing domains, stencils play an important role in the representation of domain relative information. For this reason, stencils must support a variety of operations including set operators like union and intersection, transformations like scaling, rotation, and translation, and also operations to allow domains to be converted from one type to another.

Set operations include union, intersection, subtraction, and negation. For binary operations, it is important to check that the two stencils have the same base domain. If they share a base domain, the result is a new stencil with the expected value.

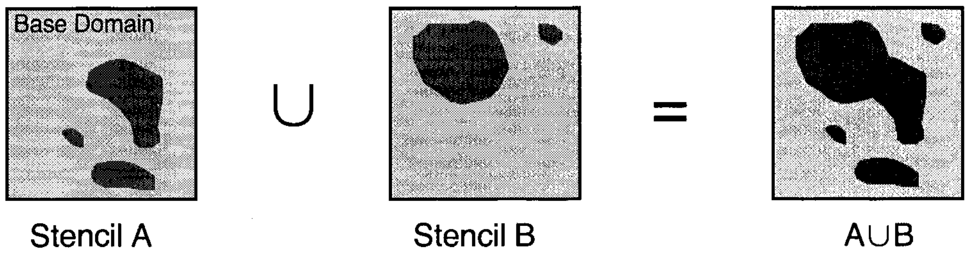


Figure 3.10. A simple stencil union

For example, the union of two stencils sharing a base domain should be a stencil with this same base containing all regions contained in the operand domains. We may choose to combine overlapping regions into a single region, but this is left up to the implementation.

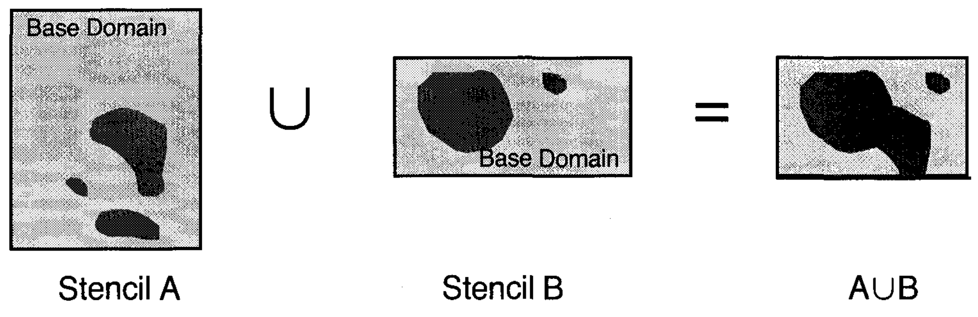


Figure 3.11. Union of stencils with different base domains.

The situation grows more complicated if two stencils do not share a base domain. For example, consider the union operation performed on two stencils with base domains that overlap, but are not the same. The proper behavior really depends on what the stencils represent, and what the user wants. In figure 3.11, let us assume that stencil A indicates areas where property A is found within the base domain of stencil A. The lighter area of the stencil denotes the region in which property A was found not to exist. The corresponding statements can be made about stencil B and property B. For the result stencil, the lighter area should denote places where neither property A nor B was found. In order for this to work properly, we have to trim the base domain for the result stencil so that it only contains the area that was examined for both property A and property B. In other words, the base domain of the result should be the intersection of the base domains of the operands. This example is only one way of handling different base domains in set operations. If the stencils represent something besides presence of a property, a different behavior may be required.

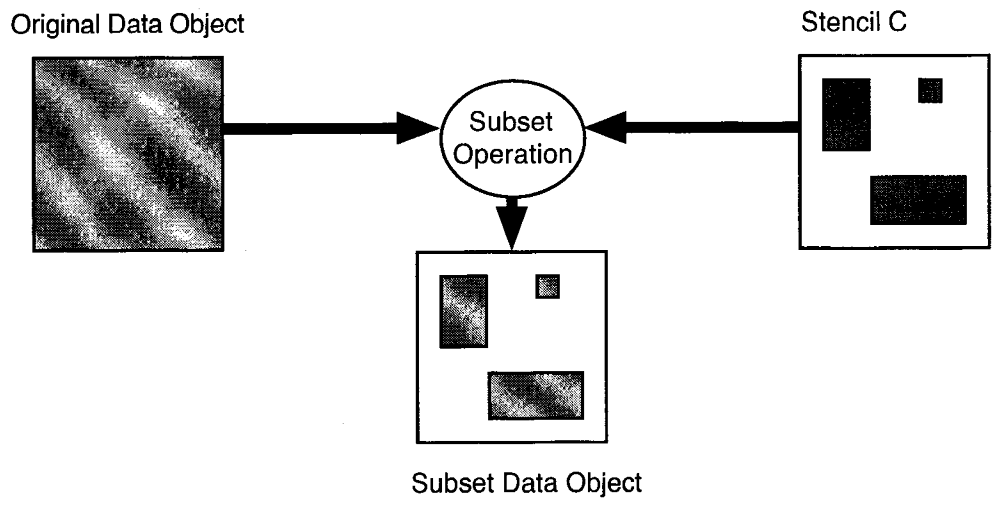


Figure 3.12. A subset operation

Consider a stencil *C* which identifies areas of a data object's domain that meet some criterion. As shown in figure 3.12, we can make a new data object that has a domain formed by the intersection of the original object's domain and the stencil *C*. We can say that the new data object is a subset of the original data. This subset operation is very useful, because it allows researchers to reduce the size of the data being manipulated and allows them to focus only on data which they consider interesting.

Transformations like scaling, rotation, and translation are used to move a stencil within the universal domain. This functionality is useful when a stencil is used to extract data from different places in a dataset, or when a new dataset is constructed from two or more existing datasets. As an example of extraction, consider a stencil which is meant to indicate areas containing urban development. The stencil is originally defined with a square base domain with sides of length 1 centered at the origin. As it stands, only areas within this base domain will be examined for urban development. However, with a translation operation, we can create a new stencil with a base domain of the same size, but at a different location in the dataset's domain. Similarly, we might want to scale the stencil to a size that encompasses the entire dataset domain so the entire dataset can be examined at once.

When constructing a new database from one or more existing databases, two issues must be addressed. The first problem is the placement of the old domains within the new domain. For projections, the new domain has fewer dimensions than the old. The projection must therefore specify how the dimensionality of the space will be reduced. Even if the dimensionality of domains is the same, several things must still be specified. The axes of the old domain may need to be aligned with axes of the new domain using rotation. If the domains use different units of measurement or points of origin, a scaling and translation operation may be required. Next, the universal domains of the existing databases must be checked to see if they can be sensibly represented within the new universal domain. We must verify that an old dimension can be represented by the corresponding (aligned) dimension in the new domain.

3.8 Representing Derived Data and Metadata

Although the representation of data in a scientific database is of great importance, an investigator needs an effective way of managing information about the data. This information could come from the researcher's pool of expert knowledge, from some previous exploration, or it could be generated semi-automatically by the system itself. Such information is commonly referred to as "metadata". However, the term "metadata" is vague. For example, if A is a set of facts about data, A is clearly metadata. But, if we use A to generate B, a set of facts about A, then is A still metadata? Should we call B meta-metadata? The problem is that classifying information as data or metadata depends upon the context, which leads to confusion. We can use the very general term *derived* *data* to refer to information that was somehow derived from another dataset. This term can be used even when it is not clear whether "metadata" is an appropriate label.

The following sections describe our thoughts on how to represent derived data and metadata. First we explain our terminology, and then describe how we represent these different kinds of information, and how they interact within the database.

3.8.1 Terminology

Within our model, we have decided to make a distinction between *semantic metadata, structural metadata,* and *historical metadata.* Semantic metadata represents information that the user has extracted from raw data through the application of expert knowledge and adds meaning to the data to which it refers. This metadata is directly available for the researcher to use in the production of further information. Structural metadata represents, among other things, information about the type, size, organization and source of a dataset, as well as the domain upon which it is defined. Notice that it is possible for such information to be specified before a dataset is populated with data. Structural metadata is needed to determine what operations and applications of data are sensible. Historical metadata keeps track of operations performed on data and how data is used. It primarily describes how a dataset was formed from one or more datasets in the database. This information establishes a chain that records the complete history of a data object, extending back to the original dataset it was derived from. Notice that historical metadata does not by itself add meaning to the data, so it should not be considered semantic metadata. Also, since it records the relationship between two or more data objects rather than a single data object, it does not fit well as structural metadata.

Our model for scientific data is restricted to data that can be represented dimensionally. That is, our data model M_D is defined over a domain space D that consists of an arbitrary number of dimensions. A data value at a location in D is an attribute of that location in the domain. We therefore use the term *domain relative* to describe information that refers to a point or region in a domain. Notice that this term can be applied to data and either structural or semantic metadata. Since we model scientific data as dimensional data, much of the data in the system is domain relative. We commonly think of historical metadata as not being domain relative, but this need not always be so. Semantic metadata may or may not be domain relative. However, we feel it is important for a scientific database to support domain relative semantic metadata since this is the kind of information used to represent identification of features or patterns within the data, and the classification of regions of the domain.

3.8.2 Requirements for Domain Relative Derived Data

In the previous section, we mentioned that the term "derived data" can be used to describe information when it is ambiguous whether the data should be considered metadata. Generally, derived data is information that was produced through the application of some operation on an initial data object. The value space of the derived data may be different from the value space of the initial data. For example, if the initial data has a value space [0...100], and we produce from it a new data object that is normalized to the range [0...1], we would say that the second data object is derived from the first. By itself, derived data need not have any more semantic value than the data it is derived from. However, it may be a component of a piece of semantic or historical metadata.

We should use an MR hierarchy for derived data as well as data, and for similar reasons. Derived data is distinguished only in how it is produced, so there is no need to represent it differently from other data. In fact, all the advantages of the MR model for data apply to derived data as well. In addition, using the same representation makes the association between different resolutions of data and derived data clear. It is convenient to have levels of derived information that correspond easily to the data they were derived from. Also, if we know how to distribute data between multiple machines or processors, the same distribution can be used for the derived data.

A third requirement for derived data is that we be able to produce yet more derived data from existing derived data. More specifically, the system should be able to take one or more MR data objects and perform an operation upon them that results in a new MR object. If we represent data and derived data in the same way, this ability follows naturally.

It is important to realize that producing derived data is conceptually equivalent to adding a new attribute to the domain. For example, suppose a domain is defined by three dimensional attributes *X,Y,* and *Z*. A data object maintains attributes *A,B,* and *C* for points within the domain. Now we produce a derived data object containing attribute *D*, defined over the same domain. The points in the domain space now have four attributes *A,B,C,* and *D*. Whether these four attributes are stored in a single data object or spread over two objects does not make any difference to their meaning. One of the roles of historical metadata is to allow the retrieval of attributes defined over a given domain.

Lastly, levels of derived MR data may be produced in either of two ways. The derived levels can be produced directly from the lowest level of the original data. We call this *bottom-up* construction. This yields the most accurate results, but it may be expensive to access the original data because of its size, or because it is not stored locally.

In such situations, it may be desirable to construct the different levels of the derived data object from the levels of the original data. That is, Λ_i of the derived data is not constructed from its own Λ_{i-1}, but from Λ_i of the original data. We call this *sideways* construction. A sideways construction yields considerable savings by reducing access to the original data, but the results may not be as accurate, depending on the method used to generate the derived data. We must also take care that the derived data still satisfies the resolution constraints of an MR hierarchy. Figure 3.13 shows sideways construction.

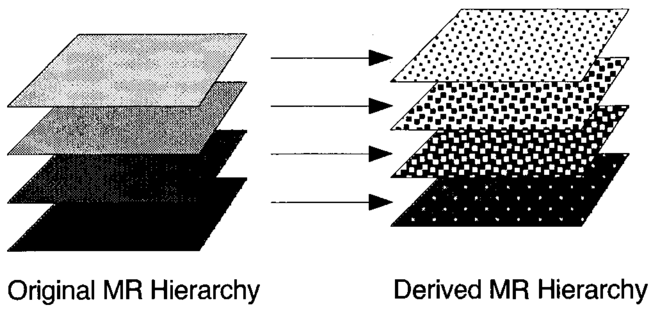


Figure 3.13. Sideways construction of derived data.

3.8.3 Representing Domain Relative Semantic Metadata

There are two parts to the job of representing semantic metadata that is domain relative. First, we must represent the semantic information, and secondly we must represent the domain or domain relative data to which the semantic information refers. The semantic information has some meaning to the experimenter, and may be as simple as a label or short string of text. This could be stored in a standard relational database.

Although we allow semantic metadata to refer to MR derived data, we expect that it will most commonly refer to a domain represented by a stencil. The combination of one or more stencils and semantic metadata is known as a *map*. The metadata gives meaning to the domains delineated by the stencils. Notice that the stencil may be formed from a detailed representation of data without greatly affecting the storage size of the stencil. This is particularly true for crisp stencils, because they can be represented with a collection of regions. It is usually much cheaper to store a region than to store the points enclosed by it. Therefore, even if a researcher finds the size of high resolution data prohibitive, he or she is still able to use metadata produced from a high resolution representation.

Many of the operations supported by stencils will be very useful when the stencil is part of a map. Certainly, all the set operations described in section 3.7.5 can be used to support similar operations with maps. The only complication is that if two maps are combined using a set operator, e.g. union or intersection, the semantic metadata of the result map must also be computed somehow. One option is to use the historical metadata to form the semantic metadata. For example, suppose a new map is formed from two existing maps named "UV<0.5" and "IR>0.2" using an intersection operator. The resulting map indicates areas where ultraviolet reflectivity is less than 0.5 *and* infrared reflectivity is greater than 0.2. The historical metadata for the result would indicate that the new map was formed by the application of the intersection operator to the two operands. So, if we automatically generate semantic metadata for the result map, the name will probably be similar to "UV<0.5∩IR>0.2". However, the expert researcher may know that the result map corresponds to areas containing vegetation. Therefore, he or she may decide to name or label the result map's domain to be "vegetation". In doing so, the researcher is encoding expert knowledge into the database in the form of a map label.

Notice that a map may contain more than one stencil, each with a separate label. This allows related domain relative metadata to be grouped together into a single object. The stencils must all have the same base domain, since the map as a whole is required to refer to a single domain. Also, the map must have a piece of metadata referring to each stencil. As a metaphor, consider a loose-leaf binder filled with labeled transparencies. Each label refers to a colored region drawn on a transparency. Let's also have a piece of paper the same size and shape as the transparencies that represents data values over a domain. When one or more transparencies are laid over the paper, they classify or annotate areas of the domain and the data values found there.

In addition to clarifying the functionality of maps, the preceding metaphor also suggests a way of visualizing maps. The stencil domains could be used to tint the visual representation of the underlying data values, giving a clear visualization of the domain relative metadata. In fact, maps can play an important role in the user's interaction with the database because they can be so effectively visualized.

3.8.4 Access Maps

We mentioned earlier that historical metadata refers to information about operations performed on data and how data is used. This metadata does not need to be domain relative, but it certainly can be. The *access map* is an example of domain relative historical metadata. It consists of a fuzzy stencil and fixed metadata that identifies it as an access map referring to a particular data object. The access map keeps a record of what parts of an MR data object have been most frequently accessed by the experimenter. The base domain of an access map is the same as the base domain of the data object to which it refers. When the map is created, its stencil domain is empty. When a subdomain of the data object is accessed by the experimenter, the membership value of the corresponding subdomain of the access map is adjusted to record the visit. Over time, the fuzzy domain in the stencil indicates what areas of the domain are most frequently accessed. The most frequently accessed areas would have a membership value near 1, while areas which were only occasionally visited would have a value near 0.

This information can be used to help distribute data efficiently among several machines or processors. Also, a subdomain's frequency of access is a measure of how interesting that area of the data is to the researcher. Such information could potentially be transformed into a normal map, making it visible to the user.

3.9 Operations on MR Data and Metadata

We have already described how a researcher would explore an MR hierarchy by hand. The process consists of scanning a coarse overview of a region, selecting interesting areas, and then drilling down into a finer resolution representation of those areas. Even though an MR hierarchy allows the researcher to work with large volumes of data in this manner, it is still desirable to have tools that at least partially automate this process. That is, if the researcher can define with some rigor what is "interesting", the database system should be able to find regions of the domain that match that definition. Also, the researcher may want to apply an operation to every location in the domain. Therefore, traversing an MR data object is an important operation for our model.

3.9.1 Traversing MR

Several things are necessary for traversal of an MR hierarchy. The *traverse* operator controls which regions of a single level are visited, and in what order. Its implementation depends partly on the format of the MR data, and partly on the purpose of the traversal. The *descend* and *ascend* operators are given a region of the domain, and return the next finer or coarser level, respectively. The implementation of descend and ascend operators depends solely on the data format. It is possible to design an *MR iterator* consisting of a *traverse* operator and a *descend* function. Next, some lattice transformation *T,* described in section 3.3, must be applied to the subdomains visited during the traversal. Different definitions of *T* are used to implement the various MR operations.

The descend and ascend operators may use a predicate that examines the domain, and decides, for example, whether descending is warranted based on the data. We'll call the predicates used by the descend and ascend operators *finer* and *coarser*, respectively. The behavior of these predicates is very specific to the purpose of the traversal. If the transformation should be applied to all levels of a hierarchy, the *finer* predicate should return *true* in all cases except for the bottom-most level. This is commonly the case when the transformation being applied is a value transformation that produces a new derived data object. In other cases, the *finer* predicate must decide whether an area is sufficiently interesting to warrant being visited at finer resolution. For example, a domain subset transformation may not need to descend to a finer level if a subdomain can be eliminated based on a coarser representation. A *filter* can be implemented as an iterator that applies a subset transformation to the domain of an MR hierarchy.

Similarly, the *traverse* operator may behave differently depending on the circumstances. For example, the traversal could be depth-first, breadth-first, or best-first, depending on the application. A best-first ordering implies that different subdomains can be ranked by how likely they are to contain some feature or meet some other constraint.

3.9.2 Search

For any search, the researcher specifies the domain to be searched and the criteria for the desired data, and in reply expects a subdomain containing values that meet the criteria. The search criteria can be specified with a subset transformation, as described in section

3.5.3.

If suitable maps exist in the system, a search through a dataset can be greatly accelerated in certain circumstances. For example, a researcher might ask, “Find a place where the temperature is over 30°C and oxygen is less than 20%.” If a map has been built which contains the maximum and minimum temperatures for regions of the domain, then the search can avoid regions for which it is known that 30°C falls outside the indicated range. We call this process *pruning the search domain.* If a similar map exists for oxygen percentages, the search domain can be pruned even further.

3.9.2.1 “Find One” Searches

The easiest search to perform is one in which only a single example of an interesting feature is required. After the pruning process, it may be possible to rank regions of the search domain so that the most promising regions are searched first. Such a ranking may use a simple heuristic, or may require expert knowledge of the experimenter's field.

We hope that ranking the search domain will improve search performance in cases where data meeting the search criteria actually does exist. Note that if no such data exists, the entire search domain will be examined, and the ranking won't have made any difference. Even in this worst case, though, search domain pruning should still be beneficial.

3.9.2.2 “Find All” Searches

If a researcher wants to find *all* instances of data meeting certain criteria, there’s no point in ranking regions of the search domain. However, pruning the search domain is still helpful, since there is no point in searching regions that cannot meet the search criteria. Of course, whether pruning can be done at all depends on the metadata available. Pruning a region from the search domain is only appropriate if it is impossible for that region to contain data meeting the search criteria.

There are at least two different ways to represent the result of a find-all search. The most obvious way is to use a discrete stencil with a domain containing only the data that meets the search criteria. Such an approach is most useful when the distinction between "matching" and "not matching" is sharp and discrete. However, some searches use criteria that are fuzzier. In such cases it would be useful to assign a value to each region of the domain indicating how well that region matches the search criteria. The fuzzy stencil provides just such a representation.

3.10 Distributed and Parallel Computing

Because of the enormous size of scientific data, it is very desirable to split the burden of storing and manipulating data across several processors or machines. For our purposes, *distributed computing* is the storage and manipulation of data on machines connected by a network. These machines might all be in the same room, connected by a LAN, or might be located on different continents, reachable through the Internet. In contrast, parallel computing refers to computation on a single machine that has more than one processor. In practice though, the line between distributed and parallel computing is not clear cut. Many algorithms for parallel computation can be run on a collection of machines connected via network, forming a distributed (virtual) parallel machine. For this reason, many of the issues that arise in parallel computation also apply to a distributed environment.

3.10.1 How MR Can Help With Load Balancing

Load balancing is an important part of using parallel machines effectively. One of the problems with scientific data is that it may contain large areas of uninteresting information. These areas are unlikely to require much processing by the database system, whereas other areas may represent a very significant amount of work. It is important that each processor in a parallel system get roughly the same amount of work to do. We can see at least three ways that our model might assist with balancing processor load.

AR hierarchies are clearly useful for our purposes, because their reducing functions say something about what kind of data the user finds interesting. One possibility is to examine the data at a coarse resolution and use it as a guide to aid in distribution of lower levels of the data object. (In this case it would be beneficial if all processors have copies of the coarsest levels.)

Stencils should prove particularly useful, since they can explicitly eliminate areas of the domain that are not considered interesting. If the remaining areas are evenly divided among processors, reasonable load balancing should result. Since stencils are widely used in our model, this represents a significant bonus.

Access maps should also be useful in this context. If a user has been working with a particular region of the domain extensively, then the distribution of data can be adjusted to reflect the user's focus. We can envision a system where a user's access map is saved between sessions, so that when they log on again, data is distributed in the appropriate way. If multiple users of the same data are likely to use the data differently, several access maps can be saved, and a compromise distribution generated from this information.

3.10.2 Problems at Boundaries

In section 3.6.4, we mention that boundaries can cause difficulties for region based data with overlapping support. Such problems are exacerbated with parallel computation because the distribution of the domain among several processors creates many more boundaries. If data is distributed naively, some of the data in Λ_i providing support for a value in Λ_{i+1} may be located on another processor. Since we have seen that interprocessor communication is expensive, we would like to either eliminate or at least minimize the cost of this communication. Since more than one processor is likely to need access to data at the boundaries, it seems desirable to duplicate information at the edges. This could be done either during the initial distribution of the data, or perhaps dynamically. If it is done dynamically, a processor should try to send one large block of data to its neighbor, rather than several small blocks so the fixed communications cost is incurred only once.

3.10.3 Distributing Data

Consider a system in which data is stored on several machines in a network, but processing is always done on a local machine sitting in front of the user. The MR data model is very well suited for such a system, since the highest resolution levels can be stored on a large server somewhere on the Internet, while coarser levels are stored locally.

Deciding where to store a particular level of an MR object is a classic time-space tradeoff. Clearly, it is not feasible for most users to store a terabyte of data on their local workstation. Even if a user has the storage capacity, it is very likely that most of the data is not interesting and will never be used. Instead, a low resolution version of the dataset is stored locally, and the high resolution representation is stored on the remote server. As the user drills down into the MR data, subsets of the higher resolution levels are downloaded to their workstation for visualization and manipulation. Of course, the cost of this storage efficiency is time, since it usually takes longer to transfer data over a LAN or the Internet than it would to retrieve the data from a local disk. The users may want to make some decisions about where data is stored, depending on what data they expect to be most interesting, or perhaps a caching algorithm could be employed to manage the space on the local machine automatically. Once again, since history maps indicate which data the user has found interesting in the past, it might be possible to configure data in an efficient manner based upon this history. That is, when a user logs on to the system in the morning, data could be distributed in a pattern that supports what he or she was doing the night before.

3.11 Lattice Implementation

The Lattice layer of the Granite system is responsible for providing the user with a uniform view of a wide range of data formats. This is accomplished with the help of various components that are tailored to a particular kind of data. By constructing a Lattice object with the appropriate components, the Granite system is able to provide the user with a convenient abstraction of the underlying data, regardless of the format.

This section describes the implementation of the Lattice components and their role in the Lattice layer and also addresses Granite's way of handling multiresolution data.

3.11.1 Geometry

The implementation of the Geometry class is very straightforward. This class has two main jobs. First, it must represent the extent of the Lattice domain, which is easily done with the help of a GBounds data member. Secondly, the geometry class maintains a spatial partitioning that is used to facilitate searching within the domain. In general it is expected that a geometry will use a rectilinear partitioning because that is most likely to provide the best search support. However, this is not required.

The value of the partitioning is most obvious when dealing with unstructured data. In this case it is particularly difficult to map a point in the geometry to the sample points and cells defined in the topology. It is easy, however, to map a point in the geometry to a partition identifier, especially for a rectilinear partition. This partition identifier can then be passed to the Topology class which uses the identifier to greatly accelerate the search for relevant sample points. This implementation allows the geometric location of sample points to be stored with their data values, which reduces the complexity and cost of access to disk.

The mapping between geometric and index spaces is done with the help of the *Partitioning* class. This class stores the dimensions of each partition, and uses these values to perform the mapping with some simple arithmetic operations. For a uniform rectilinear partitioning it is only necessary to store a single value for each axis of the space, since all partitions are the same shape. To represent partitionings with various shapes, more information must be stored, but the mapping is still very straightforward.

3.11.2 Topology and Cells

The Lattice contains a *Topology* component which is responsible for retrieving data values from disk. For rectilinear data, this can be done very easily by passing the index space location produced by the geometry directly to a DataSource, which then retrieves the data. For unstructured data, the topology uses a partitioning corresponding to the one used in the geometry. The partition identifier computed by the geometry can then be used to help navigate the topology. For both kinds of data, the topology can be viewed as a contiguous collection of cells that span the Lattice domain. Cells are particularly useful for computing approximated values for locations in the domain that do not correspond to sample points.

3.11.3 Cell Implementation

The Cell class defined in the Lattice layer consists simply of two arrays containing Point and Datum objects that correspond to the cell vertices. Important methods include *containment* operations that indicate if a cell contains a Point, and the *cell/GBounds intersection* operation, which indicates if a Cell intersects with a GBounds. This form of intersection is used extensively during construction of unstructured topologies to determine which partition to place a cell in. The containment operation is used to determine which cell should be used when computing an approximate value for a point that is not a sample point.

For rectilinear data, these operations are trivially implemented with a few comparisons. For unstructured cells, these operations are more complex. Intersection with GBounds is implemented with the help of *outcodes*, bit strings that help to eliminate line intersection tests by identifying trivial cases. Containment is decided by drawing a line through a point and counting the number of intersections between the line and the cell boundary.

Currently, the Datasource and Lattice layers have their own separate implementations of the Cell class. However, these implementations are extremely similar, so it is likely that they will be merged and moved to the *Common* package at some future time, a location where the class will be available to the rest of the system.

3.11.4 Out of Core Unstructured Topologies

The datasource layer, which handles rectilinear data, is an "out of core" implementation, meaning that we do not have to load the entire dataset into memory at once. However, with unstructured data the Lattice layer cannot simply depend upon the datasource layer, so more work is required to achieve an out of core implementation.

The key feature of our out of core support for unstructured data is the partitioning described earlier. If we can determine which cells lie in each partition and store those cells together on disk, we can often greatly reduce the amount of memory needed at one time by only loading partitions that are actually being used.

The motivation for out of core methods is to be able to process data that is too large to fit in memory at once. It would defeat this goal if the cell lists associated with each partition were kept in memory until the end of processing. Instead, we write sections of each list out to disk as it is being built. Writing the list in sections instead of directly to disk reduces I/O operations, and also allows us to close partition files between operations. If we did not close these files, the number of partitions would be limited by the number of files the operating system allows us to keep open at one time

This partitioning process is reasonably efficient, mainly because we use the partitioning to greatly reduce the number of intersection tests that must be performed. Since partitions are meant to contain a fairly large number of cells, it is usually just a single partition that needs to be tested for intersection. However, it is not at all unusual for a cell to intersect several partitions. This situation could be handled by simply making duplicate entries in the cell list of each partition that intersects the cell. This approach is simple, but increases disk space usage, especially if the fineness of the partitioning is increased. On the other hand, it is important that each partition have a complete list of the cells that it intersects with. Otherwise, we would later be unable to answer queries for domain locations that map to a partition, but for which no containing cell can be found.

Our solution to this problem is a compromise in which the point information for all intersecting cells is stored in each partition's cell list, but the data associated with the cell may be stored only in one partition. This partition is called the *owner* of the cell. All cells have exactly one owner. Partitions that intersect with a cell that is owned by another partition are *borrowers* with respect to that cell. A cell may have an arbitrary number of borrowers, always one less than the number of partitions it intersects with. If a cell has no borrowers, then it must be entirely contained within its owner partition. Borrowed cells are represented on disk using only the vertex indices necessary for representing the cell, and an identifier that denotes the owner partition and position in the owner cell list. Since all cells have exactly one owner, the data associated with the cell need only be stored once. Scientific datasets may contain a large number of attributes for each point, so this can result in significant space savings over a duplication method.

3.11.5 Queries and Out of Core Topologies

The partitioning process described above only needs to be performed once, after which it is ready for repeated use. When the lattice receives a user query for some location *d* in the domain, the lattice geometry first maps *d* to a partition. This information is passed to the topology, which can then check to see if this partition is already in memory and load it if necessary[2].

The relevant partition is now asked to retrieve the cell that contains *d*. This cell could be either an owned or borrowed cell. If it is owned, then the data associated with the vertices is available, and an approximation can be directly applied to produce a value associated with *d*. If the cell is a borrowed cell, then the partition does not have the vertex data. In this case, upon receiving the borrowed cell, the topology will check to see if the owner is in-core, and load it if necessary. The vertex data for the borrowed cell is now available, so a value for *d* can be computed.

[2] The list of in-core partitions is maintained in a simple LRU fashion.

3.11.6 Iteration over Unstructured Topologies

In addition to accessing data through the Lattice, the Granite user is allowed to access the topology directly through a *CellIterator*. The CellIterator is itself a kind of Cell, with a value equal to one of the cells in the topology. With each invocation of the *next()* method, the Granite user causes the CellIterator to take on the value of the next cell, making it available for processing or rendering. The topology is able to avoid duplicating cells in the iteration by only iterating through the owned cells for each partition.

Currently, the CellIterator does not allow the user to specify any particular order for the iteration, but the interface is general enough to allow future implementations which perform variations such as depth or breadth first search, or perhaps iteration based on data value.

3.11.7 Multiresolution Support

The Lattice layer supports multiresolution through the *MRLattice* class. This class is essentially a list of lattice objects ordered with respect to level of resolution. MRLattice maintains a *current resolution level*, which is used to select which of the component lattice objects should be used to satisfy queries. The user can decide the appropriate level of resolution based on application parameters and communicate that information to the MRLattice.

MRLattice functionality should prove particularly useful for applications requiring Level of Detail (LOD) functionality. For example, when visualizing a terrain, areas distant from the camera point are rendered very coarsely to the screen. Retrieving fine resolution data is wasteful in this case. Instead, the application should decide the appropriate resolution for various regions of the terrain, and use MRLattice to choose the resolution of the data to be rendered.

CHAPTER 4

THE DATASOURCE LAYER

4.1 Datasources

Lattices provide the scientist with a conceptual view of his or her data that should be consistent with the operations that need to be applied to the data. In principle, this conceptual view is reflected in the organization of the physical data. In practice, however, this is often not feasible. The scientist may need different views of the same data and the data may be too large to replicate and reorganize to match each desired view. In general, multisource data and distributed computing require sophisticated ways of dividing large files into smaller pieces while maintaining a simple view of the distributed data. The datasource layer helps the lattice perform these tasks, but also provides useful functionality as a stand-alone tool.

4.2 Mapping Lattices to Data

A lattice is able to map locations in the geometry to locations in the topology. It remains to map topological locations to offsets in file or network streams. A *datasource* provides the lattice with a single, unified view of multisource data. This simplifies the mapping from topological locations to file and network stream offsets.

Some datasources are directly associated with a local file or remote source, and are known as *physical* datasources. Other datasources are *composite*, meaning they are made up of more than one component datasource. For example, a datasource that performs an attribute join would be composite. It is possible to perform very complex operations by combining several datasources together in a tree structure, with the *root datasource* at the top of the tree providing the lattice with an abstract, cohesive view of the data.

4.3 Datasource Model

A datasource can be modeled as an n-dimensional array containing a set of lattice sample points Δ. We think of arrays as an *index space I* paired with a collection of associated data values. An index space can be expressed as the cross product of several indices, each defined as a finite subset of the integers:

$$I = I_1 \times I_2 \times \ldots I_n$$

where each I_k is an integer in the range $[a_k \ldots b_k]$. When a datasource is used as a lattice component it is necessary to define a mapping of the index space *I* to the lattice domain *D*. It is not necessary for the dimensionalities of *I* and *D* to match. If these dimensionalities do match, then the neighborhood relationships present in the lattice may be reflected in the adjacencies present in the datasource index space. In other cases, there may be no simple pattern in the distribution of Δ in D, so more effort is needed for the lattice topology to map points between *D* and *I.*

Datasources must handle two basic kinds of queries. A *datum query* specifies a single location in the index space, and is satisfied by the return of a single datum. A *subblock* *query* specifies an n-dimensional rectangular region of the index space, and is satisfied by the return of a *data block*, which is conceptually an array of datums with a dimensionality matching the datasource.

The remainder of this chapter describes the various conceptual and design issues relevant to physical and composite datasources. Finally, we examine Granite support for rectilinear adaptive and variable resolution data at the datasource level.

4.4 Attribute Join Datasource

An *attribute join datasource* is a composite datasource for which each sample point is composed of attributes taken from two or more component datasources. If A is the attribute set of an attribute join datasource, then we say:

$$A = \bigcup_{i=1\ldots n} A_i$$

where A_i are the attribute sets of the component datasources.

For example, suppose *ds1* is a datasource with attributes {*salinity, pH, oxygen*} and *ds2* is a datasource with attributes {*temperature, depth*}. If these two datasources are combined by an attribute join datasource *ds3*, each point in the index space of ds3 has attributes {*salinity, pH, oxygen, temperature, depth*}. Such an operation is particularly useful when data has been organized into separate files, perhaps because it was gathered by different instruments. However, an attribute join can only be applied if the component datasources have compatible index spaces. For example, the index spaces of all components may be identical. Alternatively, congruent subsets could be taken from each component and used for the join.

4.5 Blocked Datasource

A *blocked datasource* is a composite datasource in which the index spaces of the component datasources are joined together to form a single index space. The components must have compatible attributes. For example, consider four datasources *ds1* through *ds4* that might represent several contiguous satellite image files, as shown in figure 4.1. Their index spaces can be joined together in the fashion shown by *ds5*, a blocked datasource, producing a single index space that can be manipulated as a single entity. Of course, a blocked datasource can have an arbitrary number of component datasources, allowing large amounts of data to be viewed as a single entity, but stored and accessed in a distributed fashion.

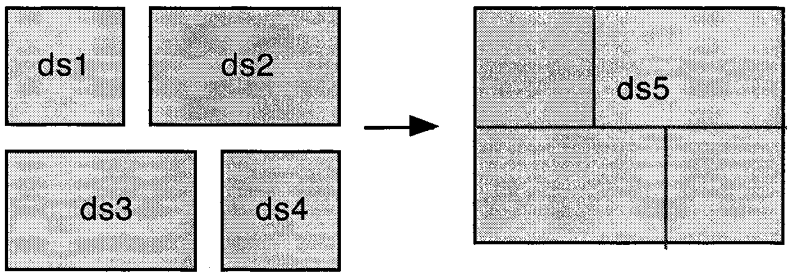


Figure 4.1. Four datasources joined by a blocked datasource.

4.6 Physical Datasources

The Granite system employs an abstract model of storage for multidimensional data that facilitates the development of efficient data access schemes. Researchers have been working for years to reduce the costs associated with disk access, but the Granite model allows the user to concentrate on the task at hand without worrying about the details of efficient data access. It is also the framework upon which *spatial prefetching* is built, as described in the next chapter. Taken as a whole, the Granite approach to I/O allows a user to access data according to the science being done, rather than the way it is stored on disk.

Figure 4.2 is a conceptual diagram of the relationship between the datasource data model and the organization of the data file on disk. The datasource is the representation seen by a Granite user, and uses a *storage model* to help translate the n-dimensional

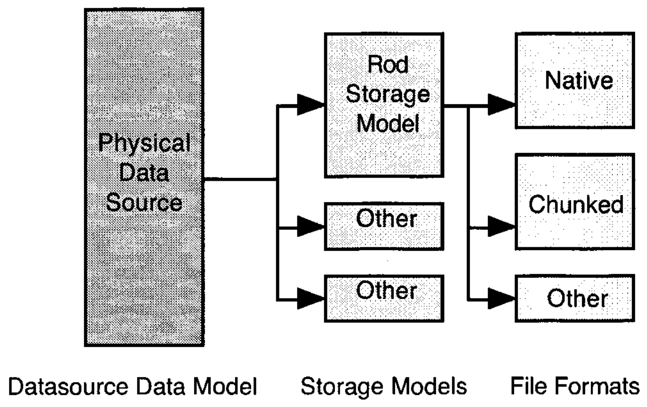


Figure 4.2. Granite users interact with the Datasource data model, which employs a storage model to help map user operations to operations performed on the file residing on disk.

data space to the one dimensional file space. The storage model can work with more than one *file format*. For example, the rod storage model discussed in the next section represents both chunked data files and files that have been left in their native plane-row-column order.

4.7 The Rod Storage Model

While the file is a one-dimensional entity, a datasource has an index space that is n-dimensional. The datasource is responsible for satisfying queries expressed in its index space by reading data from the file. It must therefore map its index space to file offsets. It does this with the help of an *axis ordering,* which is simply a ranking of axes from *outermost* to *innermost*. "Innermost" and "outermost" suggest positions in a set of nested *for* loops. Axes are labeled with numbers, so an axis ordering is really just a list of integers. The axis ordering associated with a physical datasource is called the *storage ordering*. The innermost axis of a storage ordering changes most frequently and is called the *rod* axis. For example, the storage ordering for figure 4.3 would be {1,0} if axis 0 is vertical and axis 1 is horizontal. The rod axis is always the rightmost axis in the ordering, so in this example, the rod axis is axis 0.

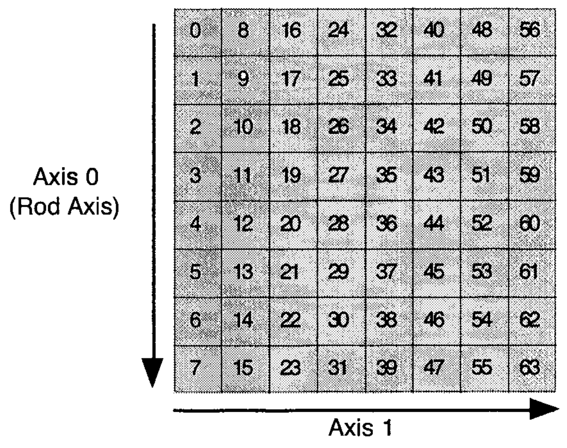


Figure 4.3 The numbers represent the offset of each element in the 1 dimensional file space. For this two dimensional datasource, the storage ordering is {1, 0}, with axis 0 as the rod axis.

I/O performance depends on the number of separate read requests made to the storage device. It is important to minimize the number of reads from disk when satisfying a subblock query. Toward this end, the rod storage model views the datasource as being conceptually composed of *rods*. A rod is a one dimensional sequence of elements that are contiguous in the index space as well as the file space. Consequently, rods are always aligned with the rod axis. Rods can be accessed in a single read operation. When a subblock query is processed, the requested region of index space is decomposed into a collection of the rod subsets contained entirely within the region. We then retrieve the subblock data from disk in rod-by-rod fashion where each rod is read with a single I/O operation. In the case where a set of rods is itself contiguous (or nearly so) in the file, we issue only one read and retrieve many rods in one disk operation.

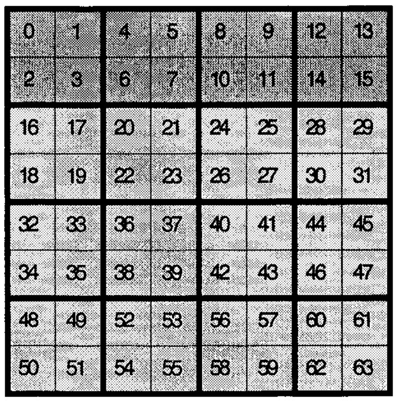
It is important to note that the rod storage model is a conceptual view of an n-dimensional dataset stored in a one dimensional file. It does not require any reordering or reformatting of the data on disk. The main function of this model is to provide a conceptual foundation for the prefetching technique described in chapter 5.

In the case where this set of rods is itself contiguous (or nearly so) in the file, we issue only one read and retrieve the entire set of rods in one disk operation. Although this may mean that some unneeded data is read, the savings in disk latency costs outweighs the cost of reading a surprising amount of extra data. On the other hand, a very long read operation can monopolize the system bus for a long period, which can be inconvenient for applications like animation. The point at which it becomes undesirable to read several rods at once therefore depends upon a mix of system characteristics (e.g., average seek time and bandwidth for the disk) and also the application that Granite is supporting. For these reasons, the Granite system allows the user to control two system parameters. First, the user may limit the maximum amount of unneeded data that can be read in order to eliminate separate read operations. Second, the user may set the maximum amount of data that will be read in a single operation. These parameters can be given default values that are tuned to a particular installation.

4.8 File Formats and the Rod Storage Model

The rod storage model can be applied to more than one file format, as long as certain basic requirements are met. Consider a rectilinear partitioning of the n-dimensional index space. The rod storage model can be applied if partitions that are adjacent in some axis can be read from disk with a single read. Files which are stored on disk in simple plane-row-column format trivially satisfy this condition if we consider a partitioning in which each partition contains only a single datum. We call this the *native file format*, because the file has probably not been preprocessed in any way to increase access efficiency.

The other file format that satisfies the conditions for the rod storage model is the *chunked file format*. Chunked files are preprocessed using a partitioning so that each partition is stored as a contiguous chunk of data in the file, as shown in figure 4.4. Chunked files are widely used in the scientific computing community because they significantly accelerate block access to the data file, especially when the chunk shape matches the shape of the block query. The chunked format also greatly reduces the performance penalty associated with accessing the file with different orderings.



0	1	4	5	8	9	12	13
2	3	6	7	10	11	14	15
16	17	20	21	24	25	28	29
18	19	22	23	26	27	30	31
32	33	36	37	40	41	44	45
34	35	38	39	42	43	46	47
48	49	52	53	56	57	60	61
50	51	54	55	58	59	62	63

Figure 4.4 The numbers are file offsets for a 2D file organized into 2x2 chunks.

A brief examination of Figure 4.4 shows how the rod storage model can be applied on top of the chunked file format. The shaded top row of chunks can be read from disk using a single read operation that loads elements at offsets 0 through 15. We can therefore regard this row of chunks, and others like it, as a rod suitable for use within the rod storage model.

4.9 Adaptive and Variable Resolution

The datasource layer supports rectilinear adaptive resolution data, and also supports the ability to view uniform resolution data at a different, but still uniform resolution. Although this functionality could be handled at the Lattice level, handling the rectilinear case at the datasource level produces increased performance and ease of implementation.

The *VRDataSource* allows the user to specify the resolution at which *datum* or *subblock* queries will be satisfied. If the resolution is finer than what is available on disk, an approximation technique is applied to generate intermediate points. Approximation may be as straightforward as a simple duplication of existing points, or could involve averaging or linear interpolation. When the requested resolution is coarser than the original resolution, we support the request by resampling. This may be just decimation, in which some points are simply left out of the query result, or possibly a more complex approximation method.

Support for point based rectilinear adaptive resolution is provided by the combination of BlockedDataSource and VRDataSource. For each component of the BlockedDataSource, a VRDataSource is placed on top, ensuring that the resolution seen by the BlockedDataSource is uniform across all components.

4.9.1 In Core Support for Cell Based Adaptive Resolution Rectilinear Data

The Datasource layer also provides support for cell and point based adaptive resolution rectilinear (ARR) data through the *ARRCellDataSource*, *ARRCell*, *URCellDataSource*, *URCellBlock*, and *ARRCellBlock* classes [Ye04]. Support for in core adaptive resolution data is implemented as a tree with leaf nodes containing blocks of uniform resolution cell data contained in a *URCellBlock* object. *URCellBlock* internally maintains the cell data using a plain "point based" *DataBlock*, and is responsible for mapping the cell index space to the point based index space of the *DataBlock*.

Figure 4.5 shows an *ARRCellDataSource* and the corresponding *region n-tree* used to access the various blocks of uniform data. The root node *A* corresponds to the point

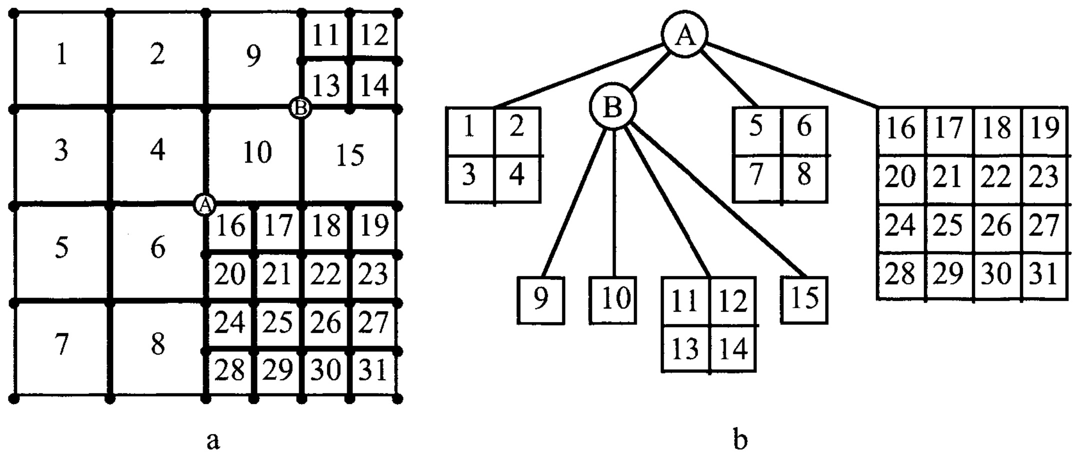


Figure 4.5. a) An ARR Cell Data Source, and b) the corresponding region quad tree.

in the middle of the space which divides the entire domain into four equal quadrants. Three of these quadrants contain data of uniform resolution, and are represented with leaf nodes containing *URCellBlocks*. The remaining quadrant contains data of two different resolutions, so it is divided again using the internal node *B* and the corresponding point in the domain.

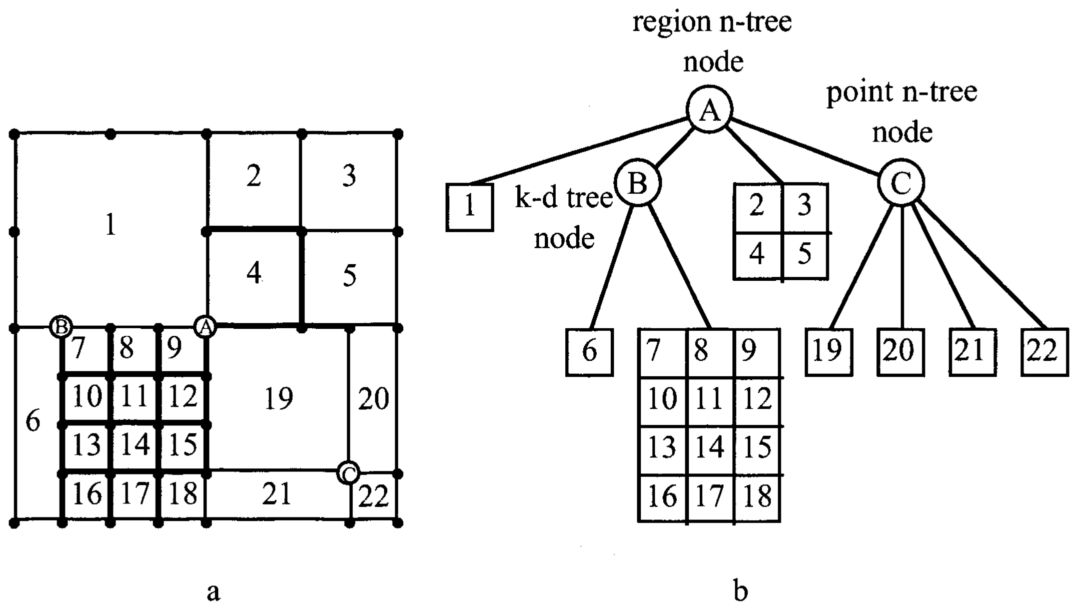


Figure 4.6. a) An ARR Cell Data Source, and b) the corresponding tree containing three different kinds of tree nodes.

More than one kind of tree can be used to construct ARR data. In addition to the region n-tree, the *point n-tree* and *k-d tree* can also be used to represent ARR data. As described in appendix B, the point n-tree differs from the region tree in that internal nodes can be split into unequal regions, according to a split point. The k-d tree divides divides internal nodes into two parts along one dimension only. The split dimension varies with each level of the tree. Modified versions of both n-tree varieties are able to merge adjacent regions of the same resolution into a single *URCellBlock.*

Lastly, we support the ability to mix different kinds of internal nodes in a single tree. Figure 4.6 shows a cell dataset along with the corresponding hybrid tree containing all three kinds of internal nodes. Supporting these different node types in a single tree greatly increases the expressiveness of Granite's ARR implementation.

4.9.2 Out of Core Support for Cell Based Adaptive Resolution Rectilinear Data

ARR trees can be written to disk with the aid of three simple data structures stored on disk. First, the cell data is stored in a one dimensional file which we call the *data array file*. Next, a *data array index file* stores contiguous sequences of indices into the data array file. Lastly, the *ARR Tree File* is a table stored on disk recording several pieces of information for each node in the tree, including node type, parent id, domain location, and size. It also stores an offset into the data array index file at which a contiguous sequence of indices will be found. These indices will be used to access the data array file when filling a URCellBlock with data.

4.10 Stencils

We have a prototype implementation of a stencil working in the datasource layer. As described earlier in this document, a stencil denotes locations in the domain that are of interest to the user, or for which some pre-defined property holds. At the datasource level, the domain in question is an index space.

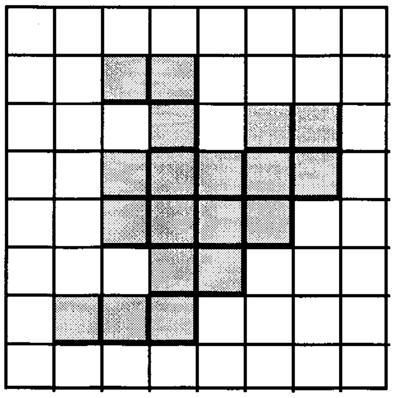


Figure 4.7. An example of our prototype Stencil implementation. The shaded partitions belong to the stencil.

Our implementation uses a partitioning to divide the index space into some number of partitions. The stencil itself is simply a list of the partitions that satisfy the stencil property. The stencil is constructed by repeated calls to the *set()* method, which adds a partition to the list. After construction, the stencil can be used with a *StencilIterator*. This iterator takes on successive values equal to the bounds of each partition in the stencil. The user is then able to use this bounds as an argument for a datasource query.

We have tried the stencil on some test datasets, including CT and MRI data, and found it reasonably effective. Using a coarse partitioning can be advantageous for visualization because the data surrounding the location of interest provides context for the user. On the other hand, a finer partitioning more narrowly identifies the areas of interest, and may reduce the costs associated with loading and processing unnecessary data.

4.11 Encapsulation and Performance Issues

Since Granite is meant to be used with large datasets, maximizing performance is an important goal. Some issues are specific the Java language, but most are applicable in other environments. Unfortunately, we must sometimes make small compromises in design in order to achieve this performance. For example, several classes in Granite have constructors that may take arrays as arguments. It is much faster to use an argument array directly in the object, rather than copying its contents to a separate internal array. Unfortunately, using the argument directly breaks encapsulation, since it means that the environment outside of the object has a reference to an internal data member.

Another example involves the checking of arguments for correctness. Checking may include looking for "out of bounds" conditions, and that objects destined to receive data have adequate space. Since Granite methods are typically implemented using other

Granite methods, it is unacceptably expensive to perform such checking with every method call.

As a compromise, we draw a distinction between public and package access methods. Public methods are visible to the Granite user, while package access methods are only visible to the package programmer. That is, they can only be called from inside their own package. This allows us to check arguments for correctness only when the public method is called. The public method then performs its task using only package access methods, which perform no checking. Similarly, objects that use references to external arrays and other argument types can only be called from within their package. This design results in a reduced level of safety within the Granite core, but the performance gains justify this cost.

Another implementation issue that has had a profound effect on performance is the difference between query methods that take a reference to a datablock or datum as an argument, and methods that return a new datablock or datum [JIANG02]. Although the problem exists for both datum and datablock objects, we concentrate here on the datablock case because of the much greater memory requirements involved.

Early work on the datasource layer relied on methods that returned new datablocks, and this approach was found to be unacceptably slow. Returning a new datablock for each query puts considerable strain on the memory allocation and garbage collection mechanisms, adversely affecting performance. The problem becomes most acute with composite datasources, since each query is satisfied by further method calls to component datasources, each creating a new datablock.

To address this problem, we developed query methods that take a reference to a datablock object as an argument. This allows the user to easily reuse a datablock for successive queries without repeatedly allocating and discarding memory. It also allows composite datasources to pass the argument datablock to each component so that the appropriate data values can be written to it.

4.11 Multisource Performance

Performance evaluation studies [JIANG02] show that the implementation provides very good performance. In particular, we have shown that our multisource support features, attribute and block join, incur very minimal overhead compared to the cost of accessing data that has been combined into a single file prior to run time.

4.12 Datasource Metadata

The Granite system has an XML and SQL based mechanism for persistently representing the information required to construct a datasource or collection of datasources. A detailed discussion can be found in [Mitchell02], but we present a brief overview here.

The cornerstone of this mechanism is the File Descriptor Language (FDL) file. FDL files are XML based files that describe the contents and layout of a data file. For example, the file name, its n-dimensional shape, the attributes and their ordering, and the endianness of the data are all recorded in the FDL file. We provide a utility method in the DataSource class that allows the user to create a physical datasource from an FDL file.

Granite has a user extendable type mechanism, described in greater detail in appendix A. Like the FDL file, the Type Definition Language (TDL) file is written in XML, but describes user defined types. For example, a user might define a type called *probability* which has an underlying *storage type* of float, but adds the requirement that all values fall within the range [0,1].

The Datasource Descriptor Language (DDL) file is also written in XML and describes a collection of datasources. Typically, this is used to define a tree consisting of one or more composite datasources (such as an AttributeJoinDS) that in turn refer to some number of physical data sources. By preserving this information on disk, Granite users can assemble the environment needed in order to support their research, and then recall that environment the next time it is needed.

For similar reasons, we are also able to store datasource metadata in a relational database like mySQL or Oracle. The database stores the same kind of information found in an FDL or DDL file, and adds the concept of a *WorkSpace*, which stores all the datasources, types, and lattices that define a Granite user's working environment. Users can also import data objects from other WorkSpaces as well as make objects available to other users.

CHAPTER 5

ITERATION AWARE PREFETCHING FOR LARGE MULTIDIMENSIONAL SCIENTIFIC DATASETS

5 Introduction

Multidimensional data presents special challenges when designing efficient access methods because elements that are nearby in the data space may not be nearby in the underlying data file. This chapter begins with a discussion of an application called *Slicer* that allows interactive exploration of the 39GB *Visible Woman* dataset from the National Institutes of Health [NIHVH]. The problems presented by this large file serves as motivation for the *spatial prefetching* technique described in this chapter.

5.1 Problem

The *Slicer* application presents the user with an animated display showing progressive two dimensional *slice planes* of a three dimensional volume. The *slice axis* is orthogonal to the slice plane and defines the direction of progression through the dataset. Figure 5.1 shows the three possible slice axes, which must be aligned with the principal axes. The user is able to select the slice axis and the subvolume to be visualized, similar in spirit to the *volume roaming* described in [Bhanarimka02]. Slicer was used with the Visible Woman, a dataset with dimensions 5186 x 1216 x 2048 with RGB byte values for each location, giving a total size of 39GB.

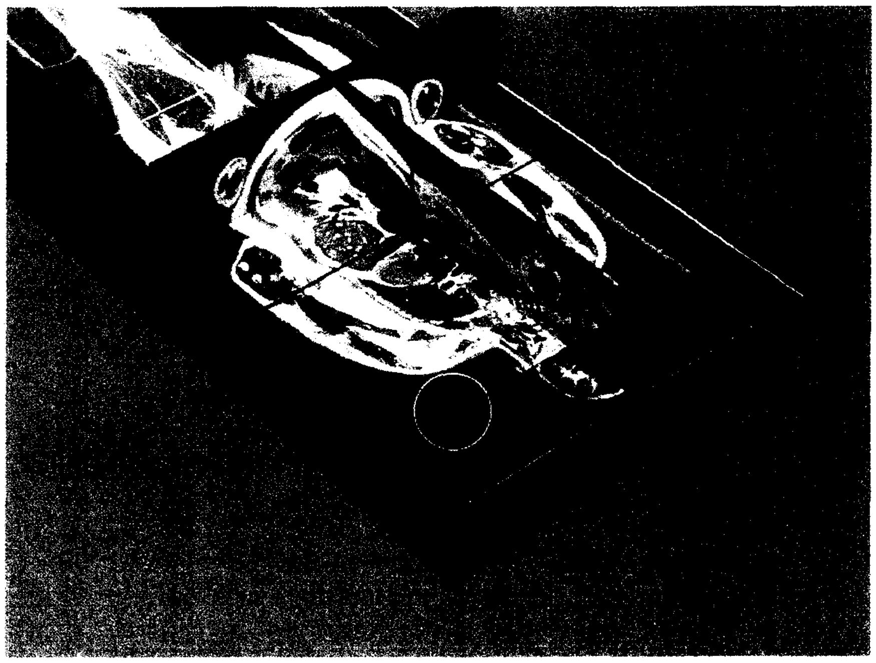


Figure 5.1. The *Slicer* application can view the Visible Woman dataset from the three principal directions by setting the *slice axis* equal to axis 0, 1, or 2.

When the user chooses to view the volume through slice axis 0, the filesystem cache performs quite well, since this view produces accesses that are contiguous in the one dimensional file space. The filesystem performs less well with slice axis 1, and is almost violently unsuited for the access pattern resulting from a slice axis 2 view.

Figure 5.2 shows a closeup of the circled corner in Figure 5.1. The numbers in the figure indicate the one dimensional file offset of the labeled element. The red region is the set of elements contained in the first slice plane for slice axis 2. If we load only the elements in this slice plane, each element requires a separate read since none of them are neighbors in the one dimensional file space, as can be seen by examining the offsets. In

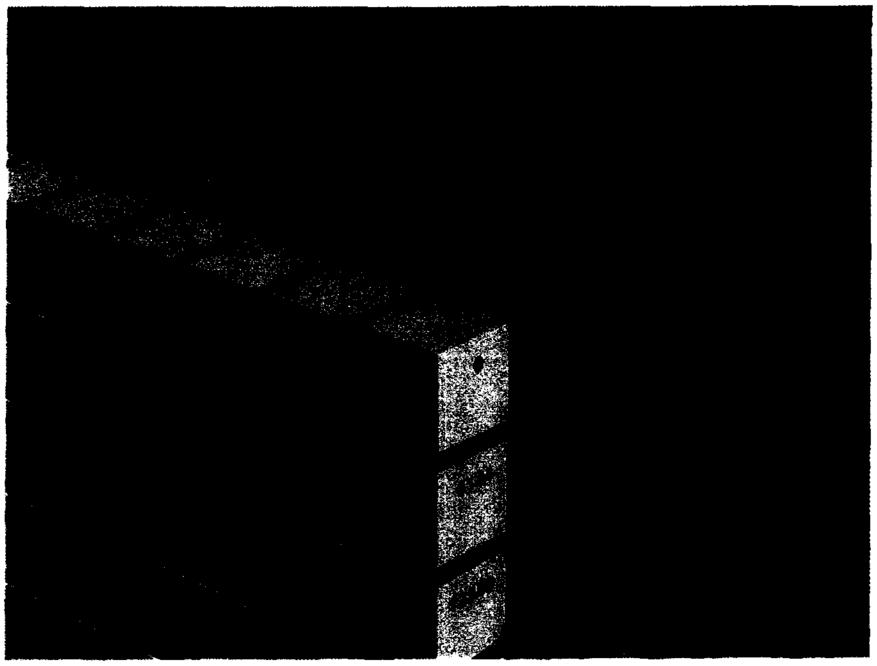


Figure 5.2. A closeup of the circled corner of figure 1. Numbers indicate offsets in the one dimensional file space. None of the elements in the red slice plane are contiguous, and are all greater than 4K apart from each other.

fact, even the elements that are closest to each other are about 6K apart, which is larger than the 4K page size typical on many systems. This means that if we render a 1024x1024 slice plane along slice axis 2, we must load 1024 x 1024 pages of 4K each, for a total of 4GB. Since very few commodity systems have this much memory available, none of the pages loaded for the first slice plane will be resident when the second slice plane is rendered. Those reads will have to be repeated, which leads to a severe degradation in performance.

Filesystems also prefetch pages following an explicitly accessed page in the hope that the prefetched pages will be accessed next and reads to disk will be reduced. In this example, this just makes the situation worse since Slicer is not proceeding through the file space in the way the filesystem expects. Prefetching just increases the number of inappropriate pages loaded, which makes it even less likely that *Slicer* will benefit from resident pages when it loads the next slice plane.

To address this problem, we use knowledge of the future access pattern to load the data for many planes at once into a three dimensional array. Contiguous sequences of elements are loaded in a single *read()* call. This method has several beneficial effects. First, it reads more data from each filesystem page, thereby reducing the number of redundant reads made to disk. Second, it reduces the number of *read()* calls made to the operating system. Third, since the array can be filled in any order, we choose to fill it in a way that most closely matches the ordering of the data in the file. This allows *Slicer* to sometimes take advantage of the filesystem prefetching that is otherwise a liability.

5.2 Caching and Prefetching Background

The filesystem cache is does not offer adequate support for *Slicer*, especially for slice axis 1 or 2. The caching and prefetching schemes present in most operating systems do not take into account the natural spatial relationships in the data, so they tend to cache, discard, or prefetch the wrong information.

Over the last fifteen years there has been a thousand-fold increase in processor speed, along with even larger gains in memory and disk capacity. During the same period, the size of scientific data sets increased even into the terabyte range. However, the average seek time of hard disk drives has improved only modestly over the same period [Coughlin, Chang01]. The work described here is motivated by the need to hide or avoid paying the now comparatively high latency or *stalling* costs associated with modern disk drive media. Using our system, a researcher can take advantage of fast I/O performance without spending time on the minutiae of efficient file access.

To implement this abstraction while still maintaining efficiency, the researcher must be able to define the application's data access pattern. We are developing a toolkit of *iterators* that succinctly describe the access pattern and also perform the iteration through the data space. This access pattern may be purely spatial, or may relate to the locales of interesting data values. For spatial access patterns, we can then generate a cache that provides a useful speedup to the application.

To the best of our knowledge, our cache design is unique in that its blocks have an n-dimensional shape, as opposed to the 1 dimensional pages of file system caches and similar methods. N-dimensional cache blocks can be given a shape which is tuned to a particular iteration and to the storage organization of the data. We choose a shape which minimizes the total number of disk accesses while reading data which is sure to be visited in the near future by the iteration. We call this method *spatial prefetching*, an example of *iteration aware prefetching*

Unlike other methods for achieving efficient I/O performance [Sarawagi94, More00], our approach does not require any reorganization of the data. That is, we can work with the original data file, rather than making a copy with a different storage organization. Much of the research in *informed prefetching* [Albers98, Cao96, Forney02, Patterson95] has not directly addressed the special problems of multidimensional access, or taken advantage of the extremely regular access patterns common in scientific computation.

The datasource layer handles multidimensional data in which sample points are arranged in a regular and rectilinear fashion throughout the domain. As with many other scientific databases, the design of the Granite system assumes that *update* operations will be infrequent or entirely absent, so the work described here is aimed toward a read-only data environment.

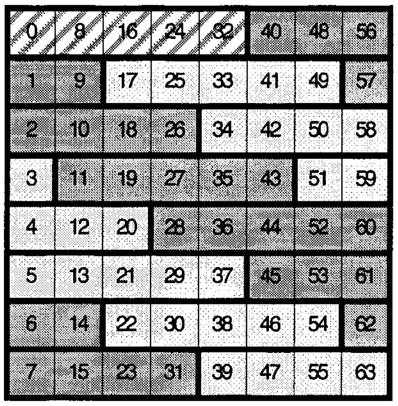


Figure 5.3. Elements nearby in the numbered iteration sequence are not contained in the same page.

5.3 Advantages of the Granite Approach To I/O

Chunking [Sarawagi94] is the most effective existing general-purpose technique for improving access to multidimensional arrays stored as files. The major drawback to chunking, however, is that the data must be reorganized using some default chunk size and it is very possible that the application program may choose to access the data in an order that is not particularly compatible with the chunking that was done. The approach adopted by the Granite system works with the original data, and requires no such reorganization.

Systems that access the data in pages suffer from not taking into account the multidimensional nature of the data. In particular, elements that are nearby in n-dimensional space may be far apart in the one dimensional file space. Since paging is essentially a one dimensional method, it may be inefficient for an n-dimensional access pattern.

Figure 5.3 shows an example of a column-by-column iteration through a 2D dataset split into pages of 5 elements each. At step 0 of the iteration, the striped page in the upper left of the diagram is loaded into memory. However, the second element in this page is not visited until the iteration has reached step 8. Worse, the last element in this page is not visited until step 32. This means that if we are to use all the data read in the first page, we must keep this page in memory until much later in the iteration. The same argument holds for all the other pages that are loaded as the iteration proceeds down the first column. In a real system, the size of the dataset and the pages themselves prohibits all these pages being kept simultaneously in memory. Pages must be discarded before all the data has been used, and then reloaded at a future time. The problem is a result of the one dimensional nature of paging, but a similar argument can be made for chunking when the dataset organization is poorly suited to an unexpected access pattern. The work described in this paper addresses these issues by creating cache blocks that are n-dimensional and shaped according to the iteration.

Many of the caching and prefetching methods meant for the file system level must work with little or no explicit information about access pattern. Such algorithms risk prefetching the wrong data, or having to make room in a cache by discarding blocks that will eventually need to be reloaded. However, the approach described in this paper takes advantage of nearly complete information about the access pattern given by our iterators. We don't have to guess which data to prefetch, and we don't discard needed data before it is used. Because of this, the various caches we have developed require at most two cache blocks to be maintained in memory at a time, which can extend the reach of an application to much larger datasets than would otherwise be possible.

5.3 Iterators

Since our system aims to improve I/O performance based on the actual access pattern, we use iterators to represent access patterns, as well as to perform the actual iteration through the datasource index space. Iterators have a value that changes with each invocation of the iterator's *next()* method. This value might denote a single location in the index space, or perhaps an entire region. In either case, the iterator value can be used directly in both *datum* and *subblock* queries.

The pattern of iteration is determined when the iterator is constructed. An axis ordering is used to help represent the behavior of iterators that proceed through the index space in rectilinear fashion. In this context, the innermost axis of the iteration is called the *run axis*. While the datasource is conceptually composed of rods, the space being traversed by a rectilinear iterator is conceptually composed of *runs*.

The *iteration space* is the space traversed by the iterator. It may be the entire index space of a datasource, or some subset of that space. We also represent the starting point and the *stride* through the iteration space in cases where the iterator skips over some locations. Along with the axis ordering, all this information is useful and available when the system creates a prefetching cache tuned to the iteration.

5.4 Iterator Aware Prefetching

A lot of the literature in caching and prefetching aims to identify when to load new blocks from disk, and choosing blocks to be discarded. Because we have near complete information about the access pattern from the iterator, these problems are vastly simplified in our system. We call our approach *Iterator Aware Prefetching*.

Most caching and prefetching methods view files as one dimensional entities, but this view of the data is not adequate for scientific applications involving multidimensional datasets because it misses the neighborhood relationships inherent in the data. The problem becomes even more acute as the dimensionality of the dataset increases. To address this issue we have designed a *multidimensional cache* that preserves the iterator's spatial data view. The iteration space is conceptually partitioned into an n-dimensional array of n-dimensional cache blocks. Data is read from disk one block at a time, and is retained in memory to quickly satisfy user queries.

Our system currently implements two different kinds of iterator aware prefetching. *Threaded prefetching* uses a separate I/O thread to fetch the next cache block while the current one is being processed. Unlike other systems using I/O threads, we don't have to speculate which block should be read next, because that information is contained in the iterator. Currently, we have only implemented and tested threaded prefetching for a single disk, so we can achieve at most the doubling of performance that occurs when the I/O time perfectly matches the computation time for each block. Even the current approach can be very effective in avoiding stalling costs, although with a larger number of disks, we may be able to reach even greater improvements in performance, by performing several disk accesses concurrently.

5.5 Spatial Prefetching

The second kind of prefetching implemented in the Granite system increases performance by adjusting the shape of the cache blocks to minimize the number of separate reads made to disk. We refer to this method as *spatial prefetching.*

5.5.1 Well Formed Cache Blocks

Typically, when a cache needs to load data from disk to satisfy a request, it loads a larger set of data in the neighborhood of the original request. Hopefully, the nearby data can be used to satisfy future requests without returning to the disk. If the pattern of future accesses is already known, however, we can choose a cache block shape that guarantees that all the needed contents will be used before being discarded. We say such a cache block is *well formed* with respect to the iteration. A more formal definition follows:

Definition D1:

Consider an axis ordering $A = \{ a_{n-1}, a_{n-2}, a_{n-3}, \ldots a_0\}$, a rectilinear index space region R of shape B and a rectilinear iterator I that performs the iteration described by A. We say B is *well formed* with respect to ordering A if for all regions R of shape B, once iterator I leaves R, it does not revisit R.

If we can construct a cache containing blocks that are well formed with respect to a given iterator, we can be assured that no cache block will need to be read more than once, and that once the iterator is done with a cache block, we can discard it. Most iterations only require a single cache block to be used at one time. Overlapping block iterators require at least two, as does threaded prefetching.

Algorithm A1:

Input:
Iterator Axis Ordering $A_n = \{ a_n, a_{n-1}, a_{n-2}, \ldots a_0\}$,
Iteration region extents $S_n = \{s_0, s_1, s_2, \ldots s_n\}$,
available memory M

Output:
A set of cache block dimensions $B = \{ b_0, b_1, b_2, \ldots b_n\}$ that represent a cache block shape that is well formed with respect to the iterator ordering.

```
$B = \{1, 1, 1, \ldots 1\}$
$SB$ = size of $B$ in bytes
$M = M - SB$

begin
    for $i = 0$ to $n$
        $axis = a_i$;
        if ( $SB \cdot (s_{axis} - 1) <= M$ ) then
            $b_{axis} = s_{axis}$
            $M = M - SB \cdot (s_{axis} - 1)$
            $SB = SB \cdot s_{axis}$
        else
            $b_{axis} = M / SB + 1$
            leave
        end
    end
end
```

Algorithm A1 generates a well formed cache block shape for a datum iterator that visits single elements in the index space. It must be given the iterator axis ordering, the space over which the iterator travels, and the amount of memory that is available for constructing a cache block.

The algorithm works by marching through the iterator's axis ordering from innermost to outermost axis, setting the corresponding dimension of the cache block shape to equal the extent of the iteration region along that axis. Below is a proof that algorithm A1 produces a well formed cache block shape for a datum iterator.

Proof P1:

Claim: Algorithm A1 produces a well formed shape B for the given iterator, iteration space, and available memory.

Base Case:
A shape with a single element is well formed with respect to $A_0 = \{a_0\}$.

Assumption:
Algorithm A1 produces a shape that is well formed with respect to ordering $A_k = \{ a_k, a_{k-1}, a_{k-2}, \ldots a_0\}$.

Induction Step:
From the iterator ordering, we know that after completing axes a_0 through a_k, the iterator will next increment axis a_{k+1}, and then repeat the iteration described by A_k, doing so until the edge of the iteration space is reached on axis a_{k+1}. From the assumption, we already have a shape that is well formed for A_k, so we only have to concatenate some number of these block shapes along axis a_{k+1} to produce a new shape which is well formed for $A_{k+1} = \{ a_{k+1}, a_k, a_{k-1}, , \ldots a_0\}$. The iterator will visit all elements in a region of this shape and not revisit any portion of it, so the new shape is well formed.

The algorithm and proof can be easily modified to account for block iterators rather than datum iterators. Since block iterators represent a sequence of block accesses, we can set the initial dimensions of the cache block shape to match a single iterator block. The algorithm then proceeds as before. The proof still holds for this case if we consider an element to be a block instead of a single position in the index space. The block version of the algorithm can also be used to handle the case where an iterator has gaps or overlap between visited elements.

5.5.2 Practicality

Whether the shape of a cache block is well formed is related only to a particular iteration. It is possible that a well formed cache block will not enhance performance with a certain dataset because of the way the data lies on disk. In order to guard against this possibility, we must check to see if a cache block shape is *practical* with respect to the storage model. We currently only consider the rod storage model, and our definition of practicality concerns the extent of the cache block shape along the rod axis.

Definition D2:

A cache block shape is *practical* with respect to a rod storage model if it has extent greater than r elements along the rod axis, where the value of r is determined by cache overhead and the performance characteristics of the I/O subsystem, and must be at least 2.

This definition is motivated by the fact that in order to get any gain in performance, we must reduce the number of reads made to disk. It follows that we must therefore make each read longer than would be performed without the cache. The extent of the cache block shape along the rod axis determines the length of these reads, so this value must be sufficiently long to provide a performance gain, even in the face of cache overhead.

5.6 Examples

Three potential cache block shapes are shown in figure 5.2. The numbered sequence indicates a column-by-column iteration over a datasource stored in row-by-row fashion. The shaded shape in the upper left is not well formed, and would never be produced by our algorithm. This cache block shape is poorly suited to a single block cache because step 4 of the iteration will cause the block to be discarded, only to be reloaded at step 8.

Algorithm A1 would extend the block shape all the way down to the bottom of the space before attempting to extend it in the horizontal direction.

The middle shaded shape in figure 5.4 does not have this problem, since it extends over the full length of the vertical axis. However, this block is not practical and cannot reduce the number of read operations. Since the rod axis is the horizontal axis, to fill this cache block would require eight separate reads, which is the same number we would require with no cache at all.

The shaded shape on the right is much better, since it can be filled with 8 reads of length 3. The striped region represents a single rod subset for this block. Depending on the characteristics of the platform, this shape may produce a useful increase in performance.

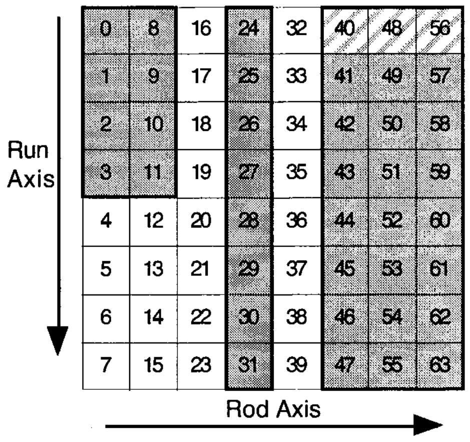


Figure 5.4. For a {1,0} iteration over a {0,1} datasource, the shape on the right is the only one which is both well formed and practical.

5.7 File Formats

When the rod storage model is used on top of the native file format, the rods consist of a series of datums stored sequentially on disk. We refer to this file format as "native" because it requires no preprocessing—the file is handled "as is". In this situation, using a well formed cache block also guarantees that no data is read from disk more than once. This is because the cache block is defined in terms of the same units (datums) as the file format.

The rod storage model can also be used on top of chunked files. In this case, the rods consist of a series of contiguous chunks that can be loaded with a single read operation. Here, the file format is defined in terms of units different from what was used to define the cache block. Because of this, data may be read more than once, even with well formed cache block shapes.

There are three possible approaches to this problem. First, we could implement an access method that allows data to be read from within a chunk without reading the entire chunk. This could save memory by allowing smaller cache blocks, but would greatly increase the number of reads necessary.

Secondly, we could allow chunks to be read more than once, making the assumption that useful speedups will still be obtained due to a total reduction in the number of reads.

A third approach is to only allow the creation of cache block shapes that will not result in any chunk being read more than once. In our current work, we have chosen this third avenue. Insuring this condition is straightforward. We only allow cache blocks with dimensions corresponding to an integral number of chunks, and for single block caches, we require the rod axis dimension to span the iteration space.

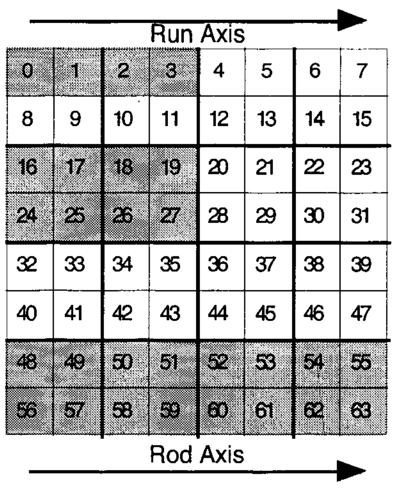


Figure 5.5. For a {1,0} iteration over a {1,0} chunked datasource, we use cache blocks shaped like the lower shaded region.

The heavy gridlines in figure 5.5 represent chunk boundaries, while the three shaded regions show potential cache shapes. The top cache shape would be well suited to the native file format, but will cause some avoidable reads in the chunked file format because it violates our first restriction. In particular, to fill that cache block, we must read the two chunks in the top left (containing elements 0,1,8,9 and 2,3,10,11 of the iteration) but then only store half of the data read, requiring it be reread when the iteration reaches the second row.

To see why the second restriction is necessary, consider the middle block shape in figure 5.5. When the iteration reaches step 16, the entire block is loaded and stored, but if only one cache block is retained, this block will be discarded at step 20, only to be reloaded at step 24. Of course, this situation can be addressed with multiple cache blocks, but this will increase the number of reads made to disk. When possible, it is desirable to use a single, larger cache block, as shown in the bottom shape of figure 5.5.

5.8 Example Code

Figure 5.4 shows a small example of a datum iteration using the Granite system. We first create the datasource from an xml file that describes such properties as dimensionality, size along each axis, and the number of attributes at each location in the index space. Next, we define an axis ordering and iterator that will traverse the datasource. We are now able to create a cache which is tuned to the iteration we wish to use. Finally, we create a datum object for retrieving data values, and perform the iteration.

```
// Create datasource
Datasource
      ds = Datasource.createDS("8gig.xml");

// Create ordering for iterator
AxisOrdering
    iterOrdering = new AxisOrdering(
                        new int[]{0, 1, 2}
                 );
// Create an iterator that traverses the entire
// datasource
ISIterator
      iter=new ISIterator(ds.getBounds(),
                        iterOrdering
                     );
// Create a spatial prefetching cache for the
// given datasource and iterator
CacheDataSource
      cds = CacheMaker.createCDS(ds, iter, freeMem);

// Create a datum to receive data values.
Datum d = new Datum(ds.getNumAttributes());

// Traverse the entire datasource index space,
// accessing the data through the cache.
for( iter.init(); iter.valid(); iter.next() )
{
      cds.datum(d,iter);
    // Process datum
}
```

Figure 5.4. Example code for a datum iteration over a cache.

This code is very flexible, and requires very minimal changes in order to work with different datasources and iterator orderings. To make the code above work on another file of entirely different size and shape, we only need to change the name of the *xml* file given in the first line of code. The iteration order is just as easily changed, and an appropriate cache will be created without further thought from the programmer.

This flexibility is especially attractive in situations where a user wants to process a large file using several different traversals. With spatial prefetching, it is a simple matter to create caches that are tuned to each iteration. With preprocessing methods, some compromise must be made when deciding the chunked format, unless the user is willing to make a separate file for each iteration.

5.9 Results

We have run our tests on a variety of machines and found that machines with fast I/O show smaller performance improvement simply because the I/O is a smaller portion of the total execution time.

We present results from the machine with the fastest I/O available to us. This is a single processor Pentium 4 machine with a 2.4GHz CPU and 2GB of RAM running the Linux operating system, version 2.4. The disk on this machine is a fast SCSI disk with a 3.8ms average read latency. Though we show here very substantial gains in performance, we got even greater gains on the other platforms.

Linux has a very effective filesystem cache that loads and stores 4k blocks of data from disk. Of course, if some or all of a file is already in this cache, stalling costs will be greatly reduced or eliminated. The filesystem also prefetches blocks stored following a requested block. Such prefetching is based upon a one dimensional view of the file, and can perform poorly with multidimensional datasets.

Since the file system cache is persistent across task execution, it is possible for a task to request an I/O block for the first time, but still get a cache hit if another task had previously read that block. Although this is a good thing in general, it is problematic for our testing environment. In order to give valid and consistent performance statistics, we need each test to be independent of what happened previously. We developed a small program that effectively "empties" the cache by filling it with blocks from a dummy file that is not used in the tests. In addition to guaranteeing a consistent environment by always starting with an empty cache, this approach portrays a more realistic behavior that a researcher might expect when dealing with very large datasets.

In the following sections, we present results for both datum and block iteration over a three dimensional 8GB dataset. On our test machine, running the *cp* command with this dataset takes approximately 400 seconds. The dataset has dimensions 1024x1024x2048, where each datum is a single floating point value. Tests were run on both native and chunked file formats. In all cases, the files had a storage ordering of {0,1,2}.

5.9.1 Datum Iteration over Native Files

Our datum iteration tests ran code very similar to the example in section 5.8. Table 5.1 compares the execution times for a traversal using no cache with a traversal using a 128MB cache. Three different iterator orderings are presented. In all cases, the cache provides a very substantial improvement in performance. Notice that the {0,1,2} ordering shows somewhat less improvement than the other orderings. This is because the filesystem prefetches blocks in the same order that the iterator requests them. Filesystem prefetching is much less effective for the other orderings, so our spatial prefetching offers more improvement. In fact, the non-cache test for {2,1,0} ordering did not complete within twelve hours. We determined that the test was making forward progress in a linear fashion, but very slowly, due to the awkward nature of this access pattern. A very simple C program that mimicked the access pattern for this test but performed no type conversion or copying of data took over 37 hours to run, so we are confident that disk access is causing the excessive runtime. We estimate the completion time for the Java implementation to be about 100 hours.

Ordering	Time without Cache (seconds)	Time with Cache (seconds)	Speedup
{0, 1, 2}	5518	777	7.1
{1, 2, 0}	10041	1204	8.3
{2, 1, 0}	360000(est)	9286	38.8 (est)

Table 5.1. Execution times for datum iteration.

Ordering	Time without Cache (seconds)	Time with Cache (seconds)	Speedup
{0, 1, 2}	556	227	2.4
{1, 2, 0}	3518	280	12.5
{2, 1, 0}	10408	2284	4.6

Table 5.2. Execution times for a 64^3 block iteration.

5.9.2 Block Iteration over Native Files

Block iteration involves loading successive n-dimensional subsets of the data from disk. The rod storage model by itself facilitates this form of access, since it breaks blocks down into sets of rods. However, spatial prefetching is still able to provide a useful performance increase by reading data for many blocks at one time. Table 5.2 shows the execution times for a 64^3 block traversal over the same dataset used in the previous section, but with 512MB allocated for the cache. Once again, the {0,1,2} case shows the least speedup since the basic iteration order follows the file storage order.

5.9.3 Datum Iteration over Chunked Files

Chunking is a common method for speeding access to spatial data, so it is important to compare spatial prefetching alone with the performance of chunked file access. An important assumption of our work is that the user access pattern is not known until runtime. Although chunking is often done with a particular access pattern in mind, a generic chunked format divides the file into chunks equal to the filesystem page size. This method provides a substantial performance improvement for most access patterns without being tailored specifically to a particular one. We therefore chose to compare spatial prefetching with this form of chunking.

Chunking generally requires some kind of cache in order to be effective with datum access, so we implemented a simple LRU cache that holds a collection of chunks. We compared the performance of our spatial prefetching cache working on top of a chunked file against the performance of this LRU cache. In our tests, the memory used for both caches is always 512MB.

Ordering	Time with LRU Cache (seconds)	Time with Spatial Prefetching Cache (seconds)	Speedup
{0, 1, 2}	3835	3513	1.09
{1, 2, 0}	5798	3934	1.47
{2, 1, 0}	5929	4730	1.25

Table 5.3. Execution times for datum iteration over chunked files.

Table 3 shows the execution times for both caches. Comparing LRU performance with the cacheless datum iteration described in section 5.9.1, it is clear that chunking is a very effective technique. However, by applying spatial prefetching on top of chunking, we produce some small but useful performance gains, especially in the last two orderings listed in the table. On machines with larger disk latency, speedup is substantial even in the first case.

Of even greater interest is the fact that the performance of spatial prefetching over a native file presented in section 5.9.1 is very competitive with the performance of the LRU cache over a chunked file. Although we don't do as well in the {2,1,0} ordering, our spatial prefetching is far superior to the chunked file performance for the other two cases. For convenience, we present this comparison in table 5.4. That such performance can be achieved without preprocessing or duplicating the file makes spatial prefetching a particularly attractive technique.

Ordering	Spatial Prefetching Time (seconds)	Chunked File Time (seconds)
{0, 1, 2}	777	3618
{1, 2, 0}	1204	5385
{2, 1, 0}	9286	5532

Table 5.4. Datum iteration over spatial prefetching on native files compared with chunking.

5.9.4 Block Traversal over Chunked Files

Our fourth group of tests compared the performance of our spatial prefetching cache over a chunked file with the LRU cache on the same file. Table 5.5 shows that spatial prefetching over chunked files provides much more meaningful speedup for block access than for datum access. Since datum access involves many more cache lookup operations, it is likely that in this case, cache overhead erodes gains in I/O efficiency.

Ordering	Time with LRU Cache (seconds)	Time with Spatial Prefetching Cache (seconds)	Speedup
{0, 1, 2}	1800	366	4.9
{1, 2, 0}	1784	352	5.0
{2, 1, 0}	1900	1048	1.8

Table 5.5. Execution times for a 64^3 block iteration over chunked files.

5.10 Volume Slicing

We use a simple visualization application to demonstrate the effectiveness of our out-of-core data access system. Our application, called *Slicer,* presents the user with an animated display showing progressive two dimensional *slice planes* of a three dimensional volume. The *slice axis* is orthogonal to the slice plane, and defines the direction of progression through the dataset. The user is able to select the slice axis and the subvolume to be visualized, similar in spirit to the *volume roaming* described in [Bhaniramka02]. The 39GB Visible Woman dataset from the National Institute of Health was used in all tests described here.

5.10.1 Slicer Implementation

Slicer was implemented in Java 1.4.2 using the *jogl* OpenGL library. Each slice of the volume is rendered by issuing a subblock query to the datasource layer, and then sending the resulting data directly to OpenGL as a texture. OpenGL then applies the texture to a rectangular shape on screen. There is essentially no processing being done on the data itself, except that which is directly related to the I/O. *Slicer* was run on the same Pentium 4 machine used for the previous tests. The filesystem cache was once again cleared between runs.

Even with an empty file system cache, file system prefetching is still active. This effect is most obvious when the iteration pattern matches the file storage pattern. In this case, the file system prefetches the same blocks that our cache strategy identifies for prefetching, so we achieve only modest improvement (if any). On the other hand, for other iteration patterns we have no way of measuring the negative effect of unwarranted file system prefetching.

Slicer includes an optional governor mechanism to provide a maximum frame rate for the visualization. This is common with programs that use hardware rendering. The governor evens out any inconsistencies in the frame generation and frame rendering processes and generally provides smoother, more consistent visualizations when used with threaded prefetching. In addition to governor frame rate, *Slicer* provides user control over the type of cache, cache memory size and the slicing axis.

Because the *Slicer* application is I/O intensive and requires very little computation for the rendering, the performance overhead imposed by Java is not a significant factor in the total run time. This makes it an effective demonstration of the I/O performance improvements that our prefetching method can provide.

5.10.2 Slicer Results

The Visible Woman dataset has dimensions 5186 x 1216 x 2048 with RGB byte values for each location, giving a total size of 39GB. We compared performance with no cache, with spatial prefetching, and with threaded prefetching. For each case, we tried all three principal view directions (slice axes). An overview of the dataset and sample images from each direction can be seen in Figures 5.7 and 5.8 at the end of this chapter. Figure 5.7 shows a 4096x1024 overview slice, while figure 5.8 shows several closeup views along all three slice axes. Only views along axis 0 are "natural", in that they correspond to photographs of body slices. All other views are synthesized by *Slicer*.

Table 5.6 shows the maximum frame rates for each of the cases. For these tests, the frame rate governor was turned off, and a stable average frame rate recorded. *Slicer* is able to "wrap around" when it finishes an iteration, but we only recorded frame rates from the first pass, to minimize the effect of the file system cache.

	Slice Axis			Slice dimensions and Cache Size
	0	1	2	
No Cache	15.9	1.8	1.6	256 x 256 32 Slices
Spatial	14.3	11.9	12.0	
Threaded	20.3	10.0	10.2	
No Cache	10.6	0.89	0.04	512 x 512 128 Slices
Spatial	11.2	10.3	2.4	
Threaded	12.4	8.8	2.2	

Table 5.6. Frames per second for the plain datasource, spatial prefetching, and threaded prefetching caches.

For the first set of tests, the slice had dimensions 256x256, with the remaining dimension set to the extent of the entire data volume. Caches were given enough memory to store 32 slices. The second set of tests displayed slices of dimensions 512x512, with memory for 128 slices given to the multidimensional caches. For axis 0, the performance without our caching is quite good, since file system prefetching is very effective for this access pattern. It is even slightly better than plain spatial prefetching, since it avoids cache overhead costs. However, with the addition of the threaded prefetching we are able to show a small but noticeable improvement.

For the other two orderings, the file system cache is unable to match the performance of either of our two multidimensional caches, which are 5 to 7 times greater than the file system cache alone.

When the slice plane includes the rod axis, as with slice axes 0 and 1, each slice can be read as a series of long rod reads. However, for slice axis 2, each datum in a slice is in a separate rod, which dramatically increases the number of reads. However, our multidimensional caches are able to extend the rod length along the slice axis, resulting in the performance improvements shown. The threaded cache performs slightly worse here, because it has two cache blocks of half the size of the cache doing spatial prefetching alone. For axis 2, this means that the rods in the threaded cache are half the size of those used with spatial prefetching which entails twice as many disk reads. Using a monitoring tool to view CPU load, we noticed that CPU load rises to 100% during disk activity for the axis 2 tests, but not for the other directions. This heavy load is likely due to the processing required for each read to disk. This makes it much more difficult for our threaded cache to show an advantage over plain spatial prefetching in this situation, since the application is essentially CPU bound. However, this problem can be overcome if enough memory is available. Using a 512 slice cache with axis 2, we got frame rates of 3.6 for spatial prefetching alone and 4.0 with threaded prefetching.

With the frame rate governor turned off, there is a pause whenever a cache block is exhausted and a new block has to be fetched from disk. The pause is lessened but not entirely avoided with the threaded slice cache, since the rendering process is able to run through a block much faster than the next block can be loaded.

The viewer of an animation is distracted by stops and starts in the motion. With threaded prefetching, setting the frame rate governor to the average frame rate results in smooth animation. This slows the rate at which the renderer runs through a block, so that the next block is ready when it is needed. This situation simulates expected behavior when the Granite system is used with a more heavyweight renderer, such as a splatting based volume renderer [Westover90]. In the optimal situation, a renderer that takes as much time to exhaust a cache block as it takes to load the block will show twice the performance with the threaded cache compared to spatial prefetching alone. Since the next block is ready just when it is needed, performance should be similar to the case where enough RAM is available to hold the entire dataset, even with very large datasets like the Visible Woman.

5.11 CDF Performance Comparison

The Common Data Format (CDF) system has been used to store and access scientific data for many years [CDF]. Despite the name, CDF is not only a format, but also a library for accessing multidimensional scientific data. Like the Granite system, it is designed to help researchers concentrate on their science by handling the details of efficient access to scientific data.

We have compared the performance of Granite against CDF for several different dataset and query sizes. Tests consisted of iterated queries over some or all of a three dimensional dataset, similar in nature to those described in section 5.9. Three different iteration orderings were used. In almost all cases, Granite substantially outperforms CDF, especially when spatial prefetching is used. Although CDF does outperform Granite for {0,1,2} traversals with small datasets, Granite shows better performance for large datasets, which is the case most important for today's scientific researcher. Granite's advantages are particularly apparent with {1,2,0} and {2,1,0} traversals over large datasets, in which performance is 10-100 times faster than CDF. For a more detailed description of these tests and the issues involved, see [ELLIS04].

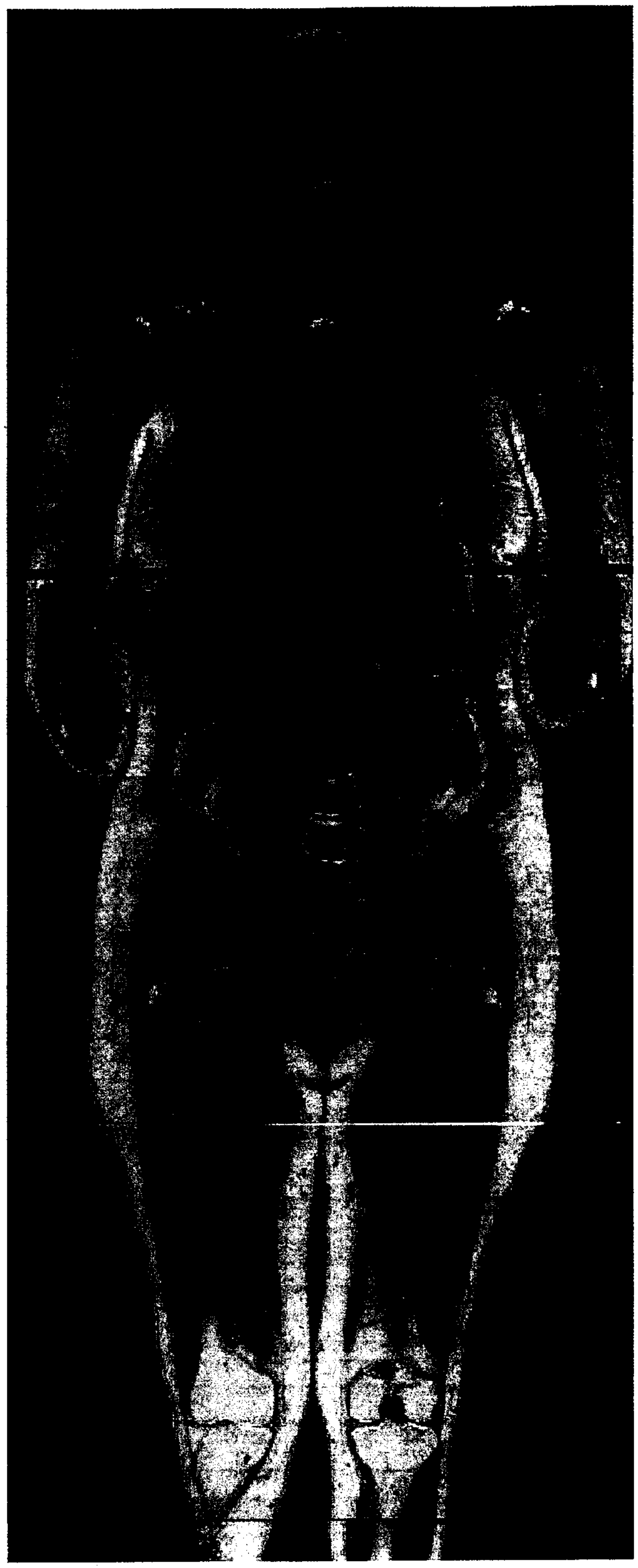
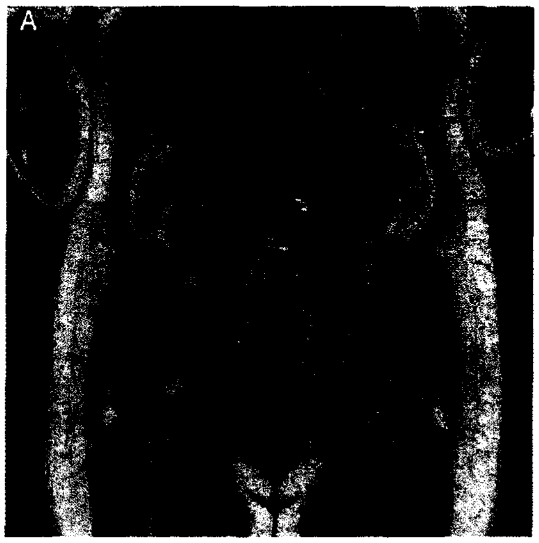
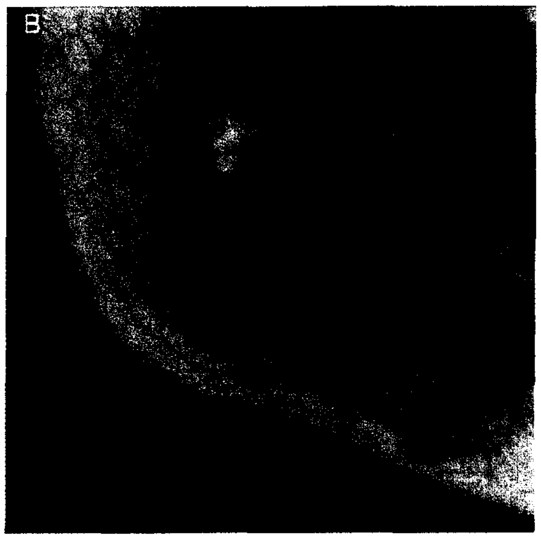


Figure 5.7. An overview slice of the Visible Woman Dataset viewed along axis 1.

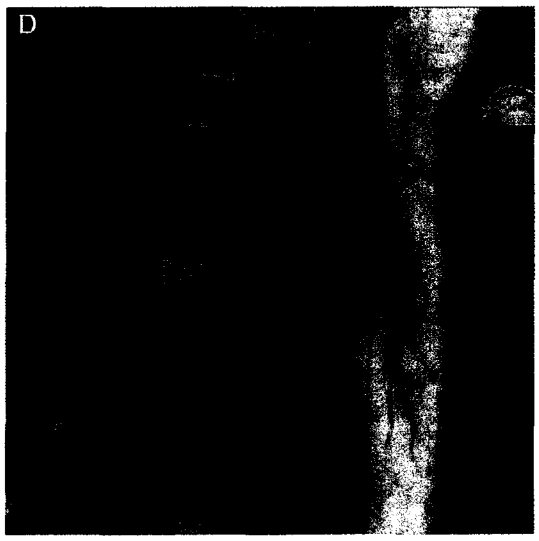


a) a close up view of the waist viewed through axis 1.



b) A view of the right hip joint viewed through axis 0.

c) The hip joint region viewed through axis 1.



d) The hip joint region viewed through axis 2.

Figure 5.8. Several example images taken from the Visible Woman dataset. All images were produced using a 512x512 slice size. Only 4b is a "natural" image, the other views are synthesized by *Slicer*.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Contributions and Conclusions

This document describes a collection of important contributions to the field of scientific databases. The formal Granite scientific data model provides a novel, comprehensive, and conceptual view of a wide range of very complex scientific data. That model served as the basis for the implementation of the Granite scientific database system which has validated the practicality and feasibility of the model.

The Granite model is unique in that it defines dataset geometry and topology as separate conceptual components of a scientific dataset. We provide a novel classification of geometries and topologies that has important practical implications for a scientific database implementation. Unlike the systems commonly in use today, the Granite model also offers integrated support for multiresolution and adaptive resolution data as well as both point and cell based data views.

The datasource portion of the Granite model offers several further contributions. In addition to providing the user with a convenient conceptual view of rectilinear data, it also offers support for multisource data. Data from various files or network sources can be combined using an *attribute join* or *block join*, thus providing an alternative view of the data without physically copying or moving the data. The *rod storage model* is an abstraction for file storage that has proven an effective platform upon which to develop efficient access to storage.

The *Granite System* is our implementation of the Granite model, and is not only a working system that provides useful and novel functionality, but also serves to validate the effectiveness and feasibility of the model. The system supports both unstructured trimesh datasets and n-dimensional rectilinear datasets. With the help of the datasource layer, the Granite system also handles adaptive resolution for rectilinear cell and point based data.

Our *spatial prefetching* technique is built upon the rod storage model, and demonstrates very significant improvement in access to scientific datasets. Together with a set of convenient iterators, it not only speeds access to datasets, it also allows machines to access data that is far too large to fit in main memory. These improvements, which apply to both chunked and native data files, bring the extremely large datasets now being generated in many scientific fields into the realm of tractability for researchers using conventional equipment.

Our implementation of the Datasource layer shows remarkable performance in several common situations, and demonstrates the effectiveness of our ideas. Datasource support for multisource data allows the scientist to work with separate datasets as a single entity.

Adaptive resolution support can greatly reduce required storage space and I/O costs by reducing the resolution of regions that are not of interest to the experimenter. We also allow both point and cell based views of rectilinear datasets.

Datasource performance has been verified with both artificial data and with real world data such as the 39GB *Visible Woman* dataset, providing effective *out-of-core* access to data that is far too large to fit in main memory. We have demonstrated the datasource layer's speed advantages, especially when spatial and threaded prefetching are used. Using the Datasource layer also allows an experimenter to access data in the way that is most convenient for doing the science, secure in the knowledge that disk access is being performed efficiently.

We have validated many of the ideas presented in our model of scientific data with our implementation of the Lattice layer . The Lattice supports rectilinear data via the Datasource layer, and adds support for unstructured data represented as meshes of triangles. It allows the user to access the data through a geometry, providing approximated data values for locations that are not sample points. It also allows the user to iterate directly over the topology, retrieving successive cells for use in rendering or further analysis.

6.2 Future Work

The Granite system provides many exciting opportunities for future work. Many of the lessons learned in the Datasource layer can be applied to the Lattice layer, enhancing performance for unstructured data. In particular, the partitions already employed by the Lattice topology can be made to fit the rod storage model. This should allow us to apply spatial prefetching to unstructured data, including tetrahedral cell data when support for that format is completed.

We are currently implementing a *remote datasource*, a new datasource that allows transmission of data over a network. Since spatial prefetching achieves its performance improvements through reducing latency costs, we believe this technique will be particularly fruitful for distributed data, due to the high latencies encountered in a networked environment. In fact, the addition of the remote datasource, coupled with spatial prefetching, should allow us to make contributions in the emerging field of *grid computing*.

Another promising area is the expansion of our toolkit of iterators. We currently have the rectilinear iterators described in this document, and a *RayCastingIterator* for rendering volumetric data. In the near future, we would like to add a slice iterator in which slices need not only be aligned to the primary axes, but can be defined for any orientation.

Taken together, these various projects will expand the scope of Granite considerably, adding support for more types of data and access patterns, as well as distributed computing. Granite has already enabled us to contribute to the field of scientific databases, and it promises to be a solid platform on which to base future research.

APPENDIX A

GRANITE FOUNDATIONS

The Granite System uses a collection of supporting classes that are used throughout the system to perform certain basic tasks. Many of these classes are defined in the *common package,* a Java package containing classes that are used by both the Lattice and DataSource layers of Granite. Some classes in the common package are parent classes refined by child classes inside the Lattice and DataSource packages, while others are used directly.

A.1 Encapsulation and Performance

Since Granite is meant to be used with large datasets, maximizing performance is an important goal. Unfortunately, we must sometimes make small compromises in design in order to achieve this performance. For example, several classes in Granite have constructors that may take arrays as arguments. It is much faster to use an argument array directly in the object, rather than copying its contents to a separate internal array. Unfortunately, using the argument directly breaks encapsulation, since it means that the environment outside of the object has a reference to an internal data member.

Another example involves the checking of arguments for correctness. Checking may include looking for "out of bounds" conditions, and that objects destined to receive data have adequate space. Since Granite methods are typically implemented using other Granite methods, it is unacceptably expensive to perform such checking with every method call.

As a compromise, we draw a distinction between public and package access methods. Public methods are visible to the Granite user, while package access methods are only visible to the package programmer. That is, they can only be called from inside of their own package. This allows us to check arguments for correctness only when the public method is called. The public method then performs its task using only package access methods, which perform no checking. Similarly, objects that use references to external arrays and other argument types can only be called from within their package. This design results in a reduced level of safety within the Granite core, but this problem can be largely addressed with the *assert* construct, recently introduced in Java 1.4.

A.2 Representing Locations in Multidimensional Space

Both the Lattice and DataSource layers use a conceptual data model in which data populates an n-dimensional space. However, the Lattice space is continuous, and is typically represented using Java's *float* primitive type. The DataSource space is discrete, and is typically represented using Java's *int* type. In either case, we require objects that specify a single location in this n-dimensional space. The common package contains a *SpaceID* class that serves as a parent class for both the DataSource *IndexSpaceID* and Lattice *Point* classes. As of Java 1.4.2, there is nothing resembling the template mechanism of C++, so the difference in primitive types means that relatively few methods can be specified in this base class. However, the upcoming Java 1.5 specification includes *generics* which allow the declaration of parameterized types. With this addition, the code which is essentially duplicated in the Point and IndexSpaceID classes can be moved down into the common package, yielding a design which is both cleaner and easier to maintain.

Currently, the SpaceID class contains methods for setting axis values, returning the dimensionality of the space, and cloning the object. Methods for setting axis values and performing arithmetic operations on them must be specified in the Point and IndexSpaceID classes, defined in the Lattice and DataSource packages. In the DataSource package, the IndexSpaceID class adds abstract methods for getting coordinate values, and several implemented methods for simple arithmetic operations like addition, negation, and comparison. There is also an *assign()* method that is available only from inside the DataSource package. This method can take a reference to an array of integers as an argument, and use the array directly in the IndexSpaceID, avoiding the cost of copying into a separate array. In the Lattice package, the Point class has functionality essentially identical to IndexSpaceId, but the axis values are represented using an array of floats instead of ints.

A.3 Representing Regions in Multidimensional Space

In addition to single locations in the index space, Granite also needs an efficient representation of hypercubic subregions of the index space. We call such regions *bounds*. The Bounds class in the common package serves as a parent class for the *GBounds* class in the Lattice layer and the *ISBounds* class in the DataSource layer. As with SpaceID, very few methods can be specified in this base class because of type conflicts. ISBounds is conceptually a pair of IndexSpaceIDs denoting the lower and upper corners of the rectangular region. The class is actually implemented using two arrays of integers, which increases performance by reducing internal method calls. The GBounds class in the

Lattice package is similar to ISBounds, except that it uses an array of floats to represent the corners of the region.

The ISBounds and GBounds classes have a large number of methods handling operations such as assignment, volume, intersection, scaling, translation , splitting, slicing, and projection. They also have boolean predicates for equality and containment, and more esoteric concepts. For example, the Granite user can ask if a shape is a "slice" of an ISBounds object. This predicate returns true if the shape is the same as the ISBounds in all dimensions except one. In that remaining dimension, the shape must have smaller extent than the ISBounds.

Lastly, *stencils* are yet another representation of n-dimensional space. As described in our model, stencils are a way of denoting a disjoint set of regions within a multidimensional space that are of interest to the user. Granite currently has a simple stencil implementation in the DataSource package that works for regular rectilinear data. This implementation consists of a list of ISBounds denoting regions of interest. The stencil class allows ISBounds objects to be added to the list as new regions of interest are discovered. Later, the list can be iterated over, returning each region in turn.

A.4 Iterators

Iterators play a crucial role in the Granite system. They are not only an important part of the user interface, but they are used extensively in the Granite core. The two kinds of iterators used most extensively in the DataSource implementation are *ISIterator* and *ISBoundsIterator*. The value of an ISIterator object is always an IndexSpaceID denoting a single location in the index space, while an ISBoundsIterator has an ISBounds value that denotes a rectilinear subregion of the index space. In either case, it is possible to specify an iteration that contains gaps between the iterator elements, and for ISBoundsIterator, the bounds produced may overlap. Currently, both forms of iterator always proceed from lowest index to highest index for any dimension. Variations such as a "zig-zag" iteration or backwards iteration are perfectly possible, but have not yet been implemented.

In the Lattice package, *GIterator* and *GBoundsIterator* correspond closely to ISIterator and ISBoundsIterator, yet work in the continuous geometry space. GBoundsIterator presents some special implementation problems caused by floating point error. Figure A.1 demonstrates one possible consequence of floating point error in a naive implementation.

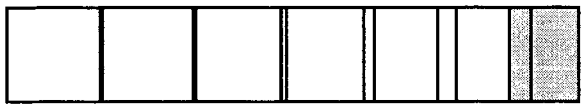


Figure A.1. A GBoundsIterator problem caused by floating point error. The last shaded square should not actually be in the iteration.

In this figure, there should only be six squares spanning the upper row of the iteration, but because of floating point error, each square is slightly to the left of its proper position. This error accumulates as the iteration proceeds. By the time the end of the row is reached, an extra (shaded) square is necessary to span the space. For some applications, this can be very undesirable behavior.

The solution to this problem is to implement the iterator so that error does not accumulate. In addition, there are two kinds of BoundsIterators in the Lattice package. *GBoundsGapIterator* is meant for iterations with gaps between the blocks. In this implementation, an integer index is computed, and then multiplied by the appropriate dimensions to produce the correct bounds value. Since the iterator value does not depend on a previous floating point value, accumulation of error is no longer a problem. The plain *GBoundsIterator* handles iterations with no gaps between bounds. Here, the implementation is similar, except that the upper axis value for a bounds is simply assigned to the lower axis value for the next bounds. This ensures that there is no possibility of a gap between bounds when none is desired, and still avoids the problem of accumulated error.

A.5 Types

Representation of data types is an important part of working with diverse kinds of scientific data. The Granite system supports the 10 primitive types defined in the Java language, as well as a *record* type that allows the definition of compound types, analogous to a *struct* in C. Users can give names to types for their own convenience, and can specify a range of allowable values.

Perhaps the most important class in the Granite type system is the *RecordDescriptor* class. This class is used in two different ways. First, it is used to describe the structure of a compound type, denoting the names and types of fields. Second, it is used to describe the field structure of a Datum or DataBlock object. Each field described by the RecordDescriptor is represented by an *AttributeDescriptor*, which contains a field name and type. An important part of the type specification is the *storage type* required. That is, even when a type is user defined, there must be some underlying Java primitive type that is used to represent the information both in memory and on disk.

Using its collection of AttributeDescriptors, a RecordDescriptor object can support a large number of methods, many of them extremely important to Granite system performance. For example, the methods *getStorageTypes()* and *getByteOffsets()* return arrays representing the storage type and locations for fields inside a datum, and are used when data is read from disk. These methods increase performance by taking advantage of information precomputed in the RecordDescriptor constructor.

When handling user defined types, both the RecordDescriptor and AttributeDescriptor classes require the assistance of the *TypeTable* class. This class stores the definitions of all user defined types. When a user type is inserted, the TypeTable object checks to see if the type name is already in use. If so, it verifies that the old and new type definitions are structurally equivalent, issuing an error if they are not. Once a type has been inserted into the table, it can be looked up by name, or by a type code, which is essentially an index into the table. The TypeTable class is part of a larger scheme which allows a Granite user to maintain a *WorkSpace* representing a persistent metadata environment that can be saved to disk or to a relational database such as *MySQL* or *PostGres*.

Two remaining classes help to specify which fields are required to satisfy a query made to a DataSource or DataBlock object. The *FieldIDMapper* class represents a mapping of fields between two datums. Fields are specified using simple integers indicating their order in the datum. Internally, the mapping is represented using two parallel arrays of integers.

FieldIDMapper allows queries in which only a subset of the available fields are retrieved, and then mapped to an arbitrary location in the datum receiving the values. Of course, with subblock queries, the datum is purely conceptual, and this mapping is applied to an entire DataBlock.

The *RecordSpec* class is used in very much the same way as a FieldIDMapper, but is somewhat more convenient, though less expressive. This class also expresses a mapping between two datums, but here the mapping is expressed using only a single list of integers, denoting the fields that should be extracted. These fields are then placed in the receiving datum in the order in which they appear in the RecordSpec.

A.6 The Datum Class

Objects of the Datum class are used to represent values at a single location in the index space. A Datum object is implemented as an array of some primitive type such as *short*, *int*, or *float*, where each element of the array represents a *field* of the datum. In the common case where the fields of the datum are all of the same type, this implementation is very efficient. If the fields are conceptually of different types, a collection of access methods allow values to be cast to the proper conceptual type. This is effective in many instances, but may cause trouble when a conceptual field type cannot be represented with complete accuracy by the Datum array. For example, the *ShortDatum* class internally represents the fields as an array of the Java *short* type, but also provides a *getFloat()* access method that returns a field as a *float*. Clearly, only a small subset of the values representable by *float* are properly representable by *short*.

Although not yet implemented, a proposed *ByteDatum* class would allow any primitive type to be extracted from an array of raw bytes. This approach solves the problem outlined above, but introduces new costs associated with the extraction. With data of uniform type, type conversion is done in bulk when the data is read from disk, which greatly improves performance. With *ByteDatum*, conversion is performed when data is accessed, and on vastly smaller units. For this reason, we have so far concentrated our research on the more common uniform case.

A.7 The DataBlock Class

The *DataBlock* classes are used to represent a collection of data values corresponding to some rectilinear region of index space. DataBlocks store data in arrays of some primitive type, and allow the user to access data either by retrieving it using a Datum object, or by returning a reference to the storage arrays themselves. The first method is conceptually easier, but the second method is generally much faster. There are two important kinds of DataBlock. A *BasicBlock* contains a single array of a primitive type. This alone is enough to handle datasets that consist of one type, even if there are multiple fields to the data. The *CompositeBlock* is used when a dataset consists of multiple primitive types. A CompositeBlock actually contains references to two or more BasicBlocks, one for each unique primitive type in the dataset. When asked for data, the CompositeBlock is responsible for determining which of its component BasicBlocks are relevant, and then translating the query appropriately for each component.

DataBlock queries fall into two types. The *datum query* takes a IndexSpaceID as an argument, and is satisfied by the return of the single datum found at the corresponding location in the DataBlock's index space. The Datum itself is usually passed by reference and the proper values filled in, but there is also a form of the datum query that will return a new datum. The pass by reference form is preferred, since it can be used repeatedly without additional load due to memory allocation and garbage collection.

The *subblock query* takes an ISBounds as an argument, and is satisfied by the return of a DataBlock filled with the data found at the specified region of the index space. As with the datum query, it is better to use the form which takes a DataBlock as an argument rather than the form which returns a newly constructed data block. For both subblock and datum queries, it is also possible to specify that a subset of the available datum fields should be returned, using either the RecordSpec or FieldIDMapper classes described in section A.5.

APPENDIX B

SCIENTIFIC DATABASES

B.1 Scientific Databases

It is the job of the scientific investigator to develop hypotheses that explain the natural world. An important part of a scientist's work is to collect data either from the real world or from simulation, and compare this data with values predicted by the hypothesis. Since it is important not to contaminate the collected data in any way, scientific datasets are not usually modified once they have been loaded into the database system. Data may be viewed in different ways, but the values themselves are not changed, although new derived datasets are often created. In contrast, an important part of traditional databases is the *update* operation, which changes existing values.

Defining scientific data, and therefore scientific databases, is not straightforward. It is perhaps wisest to say that whether a given dataset is "scientific" depends upon how it is being used. For example, information on student grades maintained by a university registrar is not scientific data. However, if that same information is used as part of a sociological study of the effect of family income on academic performance, then it may be. Perhaps we can say that whenever data is being used to support or refute a hypothesis, the data is being used scientifically. This can't be the whole story, though, since a scientist may not always have a particular hypothesis in mind when examining a dataset. Forming a hypothesis may be the very reason for the examination. However, something does distinguish the ways a registrar and sociologist use data. The registrar already knows all relevant relationships between items in the database. In contrast, the point of the sociologist's inquiry is to find new relationships within the data. Therefore, perhaps we can say that data is scientific when not all the relevant relationships within the data are known.

This view of scientific data resonates well with the opinions of other researchers. For example, Pfaltz *et al.* [PFALTZ98] list three features of scientific data, in addition to large size:

1. In the scientific database both the entities and the relationships between them are more complex than those found in traditional databases.
2. Scientific databases are not usually transaction oriented, since observations are almost never updated.
3. Retrieval of data is often "volumetric".

To point one, we would add that the relationships are not only complex, they are often initially unknown. The database should assist the researcher in their discovery. As new relationships are discovered they are stored in the database as *metadata*, the topic of the next few sections. Pfaltz's third point is addressed beginning with section 2.4, where we discuss the notion of *dimensional* data.

B.2 Metadata

While users of traditional databases are interested in updating and adding to existing data, the scientist is often more interested in adding metadata to the system. Defining metadata is not a simple task, and the meaning of the term tends to vary from field to field. Very generally, metadata is information *about* data. Cathro [CATHRO97] comes from the field of library science, and is particularly concerned with online retrieval of information. He claims that "an element of metadata describes an information resource, or helps provide access to an information resource." This definition is clearly geared toward locating material on the World Wide Web, or perhaps an online library catalog. However, he points out that metadata can also be considered data in its own right. He gives the example of a film review, which on one level is a description of an information resource (the film), and on another level is a resource in itself with an author and perhaps even a copyright. Even though Cathro is writing about library science, he points out the central problem with any definition of metadata: whether a piece of information is data or metadata is not a property of the information itself, rather, it depends upon how we are using or viewing that information at the time.

B.2.1 Kinds of Metadata

Within the metadata category, there are still further distinctions to be made. *Structural or syntactic metadata* describes the types and layout of information in a database, whereas *semantic metadata* describes meaning and relationships within the data [BERG93,KAO93]. Since all structural metadata is known even before the database is populated with data, we say it can be known *a priori*. Database designers refer to structural metadata as a *schema*. In traditional databases, the semantic metadata is also known *a priori*. However, the same cannot be said of a scientific database. Indeed, as part of the process of hypothesis justification, the relationships between different elements of the database must be discovered.

Depending on their application, different researchers have slightly different ideas about what constitutes metadata and how a system should use it. This section reviews several researchers thoughts on the metadata in different areas. We've chosen to discuss metadata issues in data mining, scientific data analysis, and Geographical Information Systems (GIS) since these areas all have some relevance to scientific databases.

B..2.1.1 Metadata for Database Mining

Cleary *et al.* [CLEARY96] discuss their ideas on metadata within the context of database mining. They divide metadata into three main groups. *Data type information, relational metadata,* and *statistical metadata.*

Data type information indicates whether an attribute is real, integer, string, data, etc. For continuous data (represented with a real), the data type information must also indicate whether the type contains a zero point, is linear, and any other information that defines the type. Cleary claims that continuous data types are always ordinal and numeric. Such factors determine what kinds of operations can be performed on the type, and what metadata can be collected for data of that type. For example, if a type does not have a zero point, the absolute value operation is meaningless. If a type is not linear, it is difficult to meaningfully compute averages and standard deviations. Relational operators like *less than* or *greater than* can only be used on ordinal data.

As an example of continuous data type information, consider an attribute of type *radians*. Such an attribute has a zero point, but is circular in structure, rather than linear. It could be argued that describing a value as being in radians is metadata, and describing radians as being circular is meta-metadata. Common usage just lumps everything except the data value into the metadata category, however.

Discrete data types are even more complicated to describe because we can not assume very much about them. Such data may or may not be ordered, linear, or have a zero point. It may be numeric, alphabetic, or enumerated. Cleary points out that a discrete type allows us to group data according to that type. For example, since age (in years) is discrete, we could group vegetarians by age. Grouping according to a continuous attribute is not likely to be useful, since very few entities have exactly the same value for that attribute.

Cleary warns that enumerated data types are very easy to mishandle, especially in an automated system. Enumerated data is represented with integers, but is not really ordinal or numeric. For example, he encodes three colors as {red=1,blue=2,green=3}. It would be inappropriate to say that red+ blue=green, or that red<blue. The underlying reason is that the information is *categorical,* i.e., it identifies a category. The integer codes are used only for internal representation, and are not meant to be treated as numeric data.

The second type of metadata Cleary considers is *relational metadata*, which specifies a relationship between two or more attributes. These relationships are divided into three kinds: the meaning, causal, and functional relationships.

A *meaning* relationship between two attributes x and y indicates that the relationship only makes sense when applied to both x and y. Cleary uses the example of *Milk Production*, an attribute which measures how much milk a cow produces. This attribute has a meaning relationship with *cow-identifier, herd-identifier,* and *farmer-identifier,* and no other attributes.

A *causal* relationship indicates that some x causes y. Such relationships are especially important for Cleary, since he is concerned with automated rule generation. Such a relationship could also be important for a scientific database.

A *functional* relationship exists between two attributes if one attribute determines the other. For example, in an employee database, *id_number* implies *name*, since if we know an employee's identification number, we know his or her name. It is important for automated systems to be aware of functional relationships so that they do not waste time generating relationships that are already known, or are redundant.

The third kind of metadata that Cleary describes, *statistical metadata* is used to help "massage" data for analysis. For example, information that is used to identify and remove outliers from a dataset is statistical metadata. Also, it is common to *classify* data according to some attribute. A typical example would be the division of homeowners into *low-income, middle-income,* and *high-income* classes. Many systems require the attribute used for this classification to be discrete. If the classification attribute is actually continuous, it must be discretized using statistical distribution and standard deviation information to put values into different "bins". If the statistical information can be stored as metadata, the process of discretization can be accelerated.

B..2.1.2 Metadata for Scientific Data Analysis

Kapetanios *et al.* [KAPET95] describe the use of metadata in a system meant to analyze scientific data. They are particularly interested in using scientific databases to support the scientific experimental process, and have developed a taxonomy of scientific metadata. Like other researchers, they lament the difficulty of rigorously defining metadata, but quote from Tsichritzis [TSICH77]:

> It's important to realize the distinction between data and information. Data are facts

collected from observations or measurements. Information is the meaningful interpretation and correlation of data that allows one to make decisions.

Kapetanios *et al.* conclude that metadata therefore lies somewhere between data and information. More specifically, they consider metadata to be "data or information that is used to provide information going beyond data or to address information related to source data."

Some metadata is already known at the time the data is gathered. For example, Cleary's data type information can be known *a priori*, before an experiment is conducted. Other metadata is actually derived from an analysis of the data. This kind of metadata is particularly important in scientific applications, since it represents new knowledge. Kapetanios divides metadata for scientific applications into nine groups. The first four are *a priori* definable and are listed below:

- **Measurements and observations**— refer to instances of observed data;
- **Transformation processes**— describe processes that transform data in some way;
- **Generation histories**— describes how data was derived, specifying the original data and the transformations used; and
- **Background knowledge**— a set of beliefs or knowledge about the scientific environment, other than those under study.

The next two are not *a priori* definable, and must be generated during exploration; they represent new scientific knowledge.

- **Experimental Laws**— despite the name, these are the relationships between observed variables inferred from the experimental data; and
- **Anomalies**— these are experimental laws that appear to contradict a theory or hypothesis.

Lastly, the following three are partly definable *a priori*. That is, unlike the previous two, it's possible to partly or even fully specify them before the experiment begins, depending upon the particular circumstances. However, they may be updated or modified after experimental results are obtained.

- **Taxonomies**— used to organize concepts into a hierarchy or some other partial ordering. This information is used to guide the extraction of new knowledge.
- **Theories/hypotheses**— A scientist's hypothesis regarding the phenomenon under study. This differs from experimental laws because a hypothesis may refer to entities or concepts not contained in the database.
- **Experiment Model**— The conditions that each run of an experiment is conducted under. Weather, temperature or humidity are all examples of this kind of data.

Kapetanios *et al.* see metadata as performing three functions: data management, access, and analysis. Since a scientific database should have strong support for data access and analysis functions, Kapetanios advocates the use of a "metadata database"— a database designed specifically for the storage and use of metadata. Such a database should provide management facilities appropriate for the metadata (knowledge structures) and knowledge representation formalisms. It must aid data access through support of metadata queries using extracted knowledge to help find relevant datasets. Data analysis is a crucial part of using extracted knowledge in this way.

This metadata database is organized as a network of metadata servers and processors. Each database server holds one or more of the metadata types listed above. Any pair of servers is connected through at least one knowledge processor. Correspondingly, the knowledge processors provide a conduit for metadata between servers, and may also modify metadata in certain cases.

The Measurements and Observations Server (MOS) is used to store a time-series representation of data. The authors feel that an ordinary Relational DataBase Management System (RDBMS) is sufficient for this fairly straightforward task, so the server represents knowledge as relations.

The other servers are not quite so simple, and must use a more complicated underlying database. The EXPER and PETRI servers both use an Object Oriented DataBase (OODB) to represent their information. The EXPER server stores transformation processes and taxonomies, both of which would be difficult in an RDBMS. Taxonomies are used to categorize data, and must also be updated in response to new knowledge. PETRI stores generation histories as an *extended Predicate-Transition Network*, a method related to Petri nets. In this network data objects are the nodes and transformation process objects form the transitions or arcs.

The EXTERN server stores information which is external to the experiment itself, but is still relevant to it. In particular, it stores background knowledge and the experimental model information. The PHEN (phenomenon) server stores experimental law metadata, along with anomalies and hypotheses. The information on this server is crucial to the scientific process, since it is the researcher's goal to justify a hypothesis with the relationships established by the experiment (experimental laws). Anomalies are relationships that appear to contradict a hypothesis, so they play an important role in confirming or disproving an hypothesis or theory. Kapetanios *et al.* have chosen to use a *semantic network* to represent the complex interactions between experimental law, anomalies and hypotheses. Semantic networks are commonly used in natural language processing, and have an expressive richness that is lacking in more straightforward logic based methods.

The authors define this justification process as a narrative that connects a particular hypothesis to experimental laws by a chain of appropriate inferences. They also connect experimental laws to observations from which high level data have been derived. If an anomaly appears to contradict a hypothesis, it must be explained through external information like the experimental model or background information, or the hypothesis itself must be discarded in favor of one that is not disproved by the anomaly.

B..2.1.3 Spatial Data and Its Metadata

Bicking *et al.* [BICK96] present a model for dynamically integrating spatial data with its metadata. They are particularly concerned with geographical information, which they refer to as *geodata*. Geographical information systems (GIS) is an area where the management of spatial data and metadata is of paramount importance, so it is valuable to examine ideas in the field

The model is geared toward preparing information about geodata for interactive use on the Internet. Toward this end, their model attempts to integrate information about geodata into a spatial data model and dynamically manage both data and metadata with the same database. Their approach allows browsing and searching with either textual or spatial information. Text searching is valuable for locating a site with relevant data, while spatial browsing is useful for refining the area of interest (AOI), and for giving an easy and intuitive indication of how relevant the geodata really is for the user's purposes.

The centerpiece of the authors' design is the *metadata catalog*, which describes the collections of geodata held by an organization. They use the *Open Geodata Interoperability Specification (OGIS) Services Specification Model*, a standard developed by the GIS community. In addition to recording the geographical location that a dataset refers to, the catalog stores summary information like the name and size of a dataset, its scale, and the projection used. The catalog also contains indices, which facilitate searching by mapping a user view to the catalog content. For example, such an index could be a table of contents (TOC), subject/author index, keyword index, or a combination. The authors decided on a keyword index, spatial index and a TOC. Their TOC presents information hierarchically in several ways. The authors give three examples:

- A subject oriented view with broad top-level categories such as transportation, hydrography, vegetation, etc., which are further divided into subcategories. The geodata content is organized by this hierarchy of subjects.

- A structure oriented view where a database is viewed as a container for datasets. Datasets are viewed as a collection of features along with their attributes and properties.

- An Organization oriented view that reflects the hierarchical structure of an organization or company. That is, the various departments of a company would have relevant data attached to them.

Accessing data through the catalog can be done both spatially and textually, with feedback between the two methods. For example, the TOC might be searched using keywords to get a list of matching map URLs. The user then selects a particular map, and is able to narrow the AOI by selecting a region with the mouse. As a region is selected, the TOC data is updated to display the metadata for that region. It is here that the authors' view of spatial metadata becomes apparent. For Bicking *et al.* the distinction between spatial and ordinary metadata is that spatial metadata refers to a point or region on the earth's surface. It is interesting to note that this information is displayed textually, rather than spatially.

B.3 Discovering New Relationships: Data Mining and Knowledge Discovery

In recent years, the need to extract knowledge automatically from very large databases has grown increasingly acute. In response, the closely related fields of *knowledge discovery in databases* (KDD) and *data mining* have developed processes and algorithms that attempt to intelligently extract interesting and useful information, i.e. *knowledge*, from vast amounts of raw data. Such techniques are used in various application domains, ranging from department stores to catalogs of stellar objects. KDD and data mining are closely related to scientific databases since they are concerned with analyzing raw data to extract new knowledge. The principle difference between them is that scientific databases are geared toward justifying an hypothesis, which is not necessarily true for KDD. For example, Wal-Mart has one of the world's largest databases of customer transactions, with over 20 million transactions being handled per day [BABC94]. Wal-Mart just wants to know to whom they should mail their next advertising circular; they aren't trying to prove an hypothesis. On the other hand the SKICAT system, a catalog of stars and galaxies, is used by astronomers who presumably are testing new theories and hypotheses [FDW96]. Yet, both systems rely heavily on the techniques found in KDD. Fayyad *et al.* [FAYYA96] give an overview of the fields of data mining and KDD. The next several subsections summarize this overview.

There are a number of other fields related to or overlapping with KDD. *Machine learning* and *pattern recognition* also attempt to extract patterns from data, but with much less human interaction than KDD. *Machine discovery* is closer to scientific databases since it attempts to discover empirical laws or relationships from experimental observations [SHRAG90]. *Data warehousing* refers to a technique used in MIS in which records of customer transactions are collected and processed for online access. *On-line Analytical Processing (OLAP)* is often used in conjunction with data warehouses to provide multidimensional summaries of transaction data.

B.3.1 KDD vs. Data Mining

There is potential for confusion about the distinction between KDD and data mining. Fayyad et al., claim that KDD is the *process* of discovering useful knowledge within data, while data mining is simply the application of algorithms for extracting patterns from data. Data mining is a class of methods used by the KDD process. KDD requires that the patterns found during data mining be "valid, novel, potentially useful, and ultimately understandable." Fayyad *et al.* define these several terms in detail, leading toward a definition of *interestingness:*

- *Data:* a set of facts F.

- *Pattern:* An expression E in some language L describing facts in a subset F_E of F. E is called a pattern if it is simpler than the enumeration of all facts in F_E.

- *Validity:* The certainty that a pattern is valid when applied to new data. Certainty is defined as a function C(E,F) that maps a pattern E in dataset F to a fully or partially ordered measurement space called M_C.

• *Novelty:* Refers to whether a pattern represents new information. For example, if a pattern is just a rephrasing of existing patterns, it is not novel. The authors assume that novelty can be represented as a function N(E,F) that returns either a boolean or perhaps a continuous value.

• *Utility*: If a pattern is useful, then it can be acted upon in some way. Utility is measured by a function U(E,F) that maps a pattern E in dataset F to a fully or partially ordered measurement space called M_U.

• *Understandability:* Patterns should be understandable by humans. The authors point out that this property is difficult to measure. (Presumably, it varies according to the human.) However, the authors suggest that the *simplicity* of a pattern is an indication of its understandability. Accordingly, they propose a simplicity function S(E,F) that maps a pattern E in dataset F to a fully or partially ordered measurement space called M_S.

The very important concept of *interestingness* is defined by the authors to be a combination of validity, novelty, utility, and simplicity. Some KDD systems use a value i=I(E,F,C,N,U,S) as a metric of a pattern's value. Other systems implicitly define interestingness by ranking the discovered patterns in some order. In either case, the notion of interestingness ultimately requires human judgment since several of its constituent functions cannot be objectively defined. Despite its subjective nature, interestingness is important because it plays a prime role in the definition of *knowledge* proposed by Fayyad, *et al.*:

Knowledge: A pattern EL is called knowledge if for some user-specified threshold $i \in M_I$, $I(E,F,C,N,U,S) > i$.

In light of these new concepts, Fayyad *et al.* offer the following definition:

> *KDD Process* is the process of using data mining methods (algorithms) to extract (identify) what is deemed knowledge according to the specifications of measures and thresholds, using the database F along with any required preprocessing, subsampling, and transformations of F.

They also give a list of the basic steps involved in this process, emphasizing the interactive nature of KDD when compared to other more AI-oriented techniques like machine learning:

1. Developing a pool of expert knowledge and end-user goals.
2. Choosing the data for which KDD is to be performed.
3. Data cleaning and preprocessing: e.g. handling noise and missing data.
4. Data reduction and projection: reducing the number of attributes to the minimum necessary to meet the end-user goals.
5. Choosing the data mining task: deciding whether the end-user goals can be met by classification, regression, clustering, etc.
6. Choosing the data mining algorithms: selecting one or more methods to be used to implement the task chosen in step 5.
7. Data Mining: searching for patterns or rules within the data. Performing steps 1-6 well can very positively affect the success of this step.
8. Pattern interpretation. The user examines the results of the preceding steps, and may decide to repeat them if necessary.
9. Consolidating discovered knowledge: incorporating new knowledge into the database. This includes accounting for conflicts with previously acquired knowledge.

Note that the KDD process may contain loops between any two of these steps, and may involve several iterations of any subset of this list. Most KDD research has been concerned with step 7, data mining, but Fayyad *et al.* are firm in their conviction that all nine steps must be carefully addressed in order for KDD to succeed in practice.

B.3.2 Data Mining

Data mining involves fitting models to, or determining patterns from observed data. A *pattern* is an instantiation of a model. In other words, a model can be viewed as a sort of template for a model. For example, the expression y=3x+5 might be a pattern fitting the model y=Ax+B. Fayyad et al. give a definition of data mining:

> *Data Mining* is a step in the KDD process consisting of particular data mining algorithms that, under some acceptable computational efficiency limitations, produces a particular enumeration of patterns E_j over F.

There are two kinds of models commonly used. A statistical model allows for some nondeterminism in the data, i.e. it allows a little "slack". So, for the model y=Ax+B, a statistical model might say that B is a random variable, with stated mean and standard deviation. In contrast, the logical approach to model specification allows no such uncertainty. However, notice that in either case, the language L that a model is expressed in may contain relational operators like < and >, allowing greater flexibility in fitting models to data. The flexibility of the statistical approach should be very helpful in dealing with error. For example, error introduced by measuring instruments or a data representation could be modeled as a random variable so that an appropriate pattern can still be found.

B.3.2.1 Goals of Data Mining

The primary goals of data mining are to describe the existing data and to predict the behavior or characteristics of future data of the same kind. Description entails finding patterns within the data that are human understandable. For example, in a bank loan dataset, a clearly understandable pattern might be: "If an individual's income is less than $20,000, then they will default on the loan." The goal of prediction can be met by discovering patterns with a high degree of certainty, as measured by the function c=C(E,F). If existing data matches the pattern with few exceptions, it is more likely that future data will also behave in the same manner.

The authors provide a list of the tasks used to meet the primary goals provided above:

- *Classification* is the process of assigning categories to features or trends within the data. Identification of interesting features within the data is a form of classification.

- *Regression* is the development of a function that approximates the mathematical relationship between two numerical attributes. For example, regression could be used to determine the relationship between the infrared reflectivity of a forest from satellite images to the percentage of deciduous trees.

- *Clustering* attempts to discern groupings within the data. For example, in a financial database, we might notice that various groups of stocks tend to behave similarly. We might divide securities into three groups, depending on which group they most closely resemble. Notice that clustering is not the same as classification, where categories are usually defined by the investigator. Clustering attempts to extract categories from the data itself.

- *Summarization* is the process of finding a compact representation for data. This may include simple statistics like mean and standard deviation, or may employ more complex methods like regression, described above. Summarization often plays an important role in the visualization and interactive exploration of a dataset.

- *Dependency Modeling* is the process of modeling dependencies between variables. The model may consist of a graph G=(V,E) in which each node represents a variable, and each edge represents a dependency. The edges may be weighted to represent the strength of the dependency.

• *Change and Deviation Detection* looks for significant changes in the data from previous values, or for data that falls outside of some normal range.

We should point out that several of these processes are important in scientific databases. Classification is closely related to feature identification, an important tool in GIS and other scientific systems. Clustering and regression both have clear scientific applications, and summarization is one of the goals of multiresolution data sets. Of course, finding relationships through processes like dependency modeling is an important part of exploratory scientific data analysis.

B.3.2.2 Data Mining Algorithms

Fayyad *et al.* give three components for any data mining algorithm: *model representation, model evaluation*, and *search*. The authors do not claim that this division is perfect, but rather offer it as a convenient way to understand the basic components of data mining algorithms.

A model is represented in some language L used to describe potential patterns. If this language is too limited, no amount of training data or processing will produce an accurate model for the data. For example, consider a model consisting only of rules like "if A.x>n then Q", where Q is some claim about the data. Such rules can only model patterns that consist of a threshold value along a dimension, in this case the x axis. If the dividing line between Q and not Q were $y=x$, this model would be unable to express this relationship.

The danger in making a model language too expressive and powerful is that the training data will be *overfitted.* This means that the model parameters will be too specifically tailored to the training example, so that new data fits the model poorly. If the model isn't too expressive, it will always be a somewhat loose fit making this problem much less severe.

Model evaluation measures how well a pattern, consisting of a model and its parameters, meets the requirements of the KDD process. Since validity is a metric of how well a pattern will match (and therefore predict) future data, it is an important evaluation criterion. The descriptive power of the model must also be evaluated, using a combination of certainty, novelty, utility, and understandability, among others.

Search methods can be broken into two levels. *Model search* looks for the model that best fits the data. Once a model has been chosen, *parameter search* looks for the model parameter values that provide the best fit for the data. Essentially, the model search process iterates over models and then invokes the parameter search process for each model. Since the space of possible models and parameters is infinite, exhaustive search is not possible, so various heuristics must be used.

B.3.2.3 Data Mining Methods

Perhaps the simplest data mining method uses *decision trees and rules* with single variable splits. Each rule is of the form "if A.x>n then Q", where x is an attribute of the data, and Q is some statement about the data. Such rules divide the data domain into

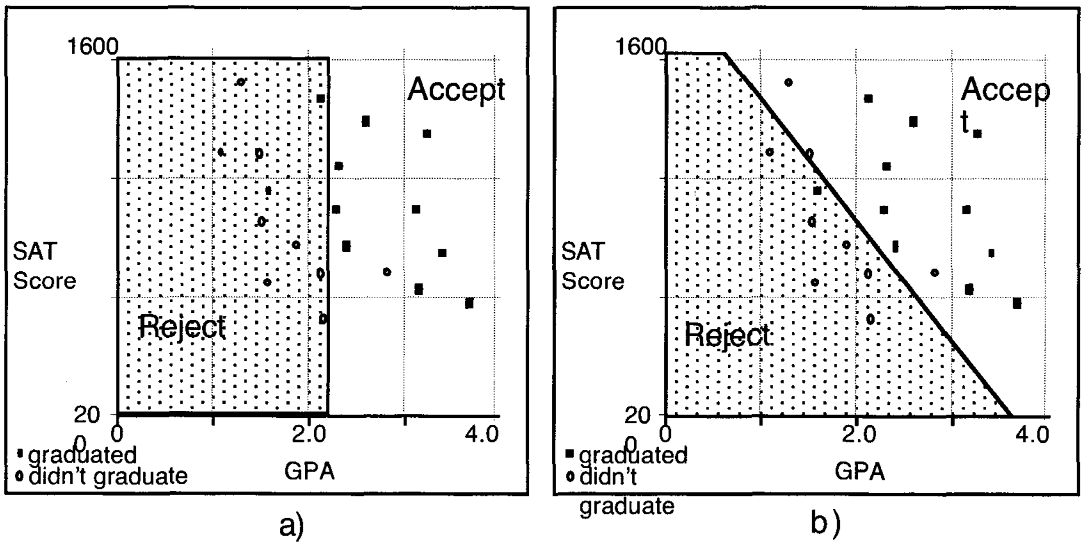


Figure B.1. Two rule based approaches

two parts using a plane that is parallel to an axis. This method is easily understood by humans, but is rather limited in power, as we saw before. An example of this approach is shown in figure B.1.a. A rule for accepting or rejecting students applying to a university is based on whether students with similar SAT scores and GPAs were able to graduate. Extending the model, as in figure B.1.b, to allow planes of arbitrary orientation increases the expressive power at the expense of understandability.

There is a family of *nonlinear* methods which attempt to match the data using linear and nonlinear combinations of a set of basis functions. This allows distinctions to be made which do not fall along straight lines. For example, a classification of the data into two or more groups might be done using a spline or other polynomial which describes an elaborate curve through the data domain. Also in this family are *feedforward neural network methods*, which use neural networks to choose the parameters of the model, which could be the coefficients for a spline. An example is shown in figure B.2.

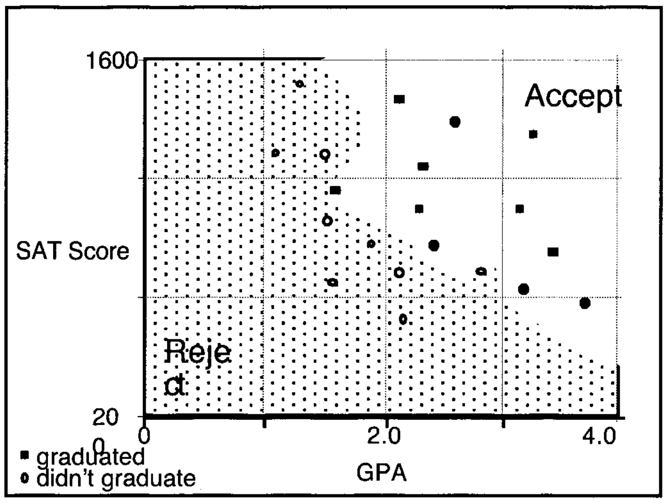


Figure B.2 A spline based approach

Example based methods are fairly simple in concept. The idea is to use existing data points to help classify and predict the properties of new data. That is, the properties of a new datapoint are taken to be the same as the properties of its nearest neighbor in the existing dataset. Finding the nearest neighbor requires the existence of a *distance measure*, which is not always easy especially with nominal or categorical data.

Probabilistic Graphical Dependency Models use a graph structure to represent the probability of a dependency between any two variates. The method arose out of AI work with expert systems in which experts set the probabilities according to their knowledge of the field. KDD researchers have focused on extracting values for these probabilities directly from the data during the model search process. Although this work is still experimental, its graphical structure should allow for clear visualization and understandability.

Relational learning models use first-order logic instead of the propositional logic of decision trees. Since first-order logic (e.g., Horn Clauses) is more expressive, relational learning models are able to succeed in situations where decision trees fail. For example, we have seen that a relation like $y=x$ can cause trouble for decision trees, but it is easily handled by relational learning models. On the other hand, such models incur a considerable cost during search, and it can be difficult for humans to specify effective models. Shen *et al.* [SHEN96] describe a system that automatically develops models (they use the term *metapatterns*) without requiring the user to specify whether particular data items are positive or negative examples of the pattern.

B.4 Scientific Data

At the beginning of section two, we claim that in order to determine whether data is scientific, we must examine the way in which the data is used. Specifically, if the data is used to develop and test a hypothesis, we say the data is scientific data. Although this definition makes some useful implications about what kinds of operations a scientific database should support, we still require a more precise model of scientific data. The remainder of this subsection presents a useful model of scientific data, and then continue by offering some useful classifications.

B.4.1 Data As a Function

A scientific database should be able to represent or model scientific data gathered either from the real world, or from simulation. In other words, a set of scientific data is a collection of sample values that represents some "natural" phenomenon [HIBB94]. Hibbard and Kao [HIBB95] point out that when the phenomenon is measured in a continuous value space, the computer can never represent a data value without some error. One solution to this problem is to model the sample points as sets of points consisting of all real values within the error bound of the sample value.

We refer to the function (phenomenon) being sampled as a function ϕ defined over a domain D. Note that D can be of arbitrary shape, although it is often a polytope [CIGNO97]. The *dataset* consists of a sampling of ϕ taken at a finite set of points $\Delta \in D$. A *mesh* consisting of the points of Δ along with connecting edges generally spans the domain D. After Cignoni *et al.* [CIGNO97], we refer to this mesh as Γ. Cignoni also postulates a function f which interpolates values of ϕ for domain points not in Δ. Characterizing f as an interpolating function may be too strict, since there may be useful *approximating* functions that do not interpolate. In any case, the mesh assists the approximating function, since edges of the mesh connect points, and also form regions within which values can be approximated or interpolated.

Notice that in order for this model to be useful, the domain D must be defined over 1 or more *dimensions*. For example, in 3D Cartesian space these dimensions would correspond to the x, y, and z axes.

B.4.2 Dimensional Data

As with metadata, opinions on what constitutes a *dimension*, and therefore *dimensional data*, varies from field to field. The OLAP community has a fairly specialized view. The OLAP Glossary offers the following definition [OLAP]:

> A dimension is a structural attribute of a *cube* that is a list of members, all of which are of a similar type in the user's perception of the data. For example, all months, quarters, years, etc., make up a time dimension; likewise all cities, regions, countries, etc., make up a geography dimension. ...Dimensions offer a very concise, intuitive way of organizing and selecting data for retrieval, exploration and analysis.

The same source defines a cube as being synonymous with a *multidimensional array*, which they essentially define as "a group of data cells arranged by the dimensions of the data." Notice that such a cube could actually have many more than three dimensions. That is, it could be a hypercube of arbitrary dimension, where each dimension is (presumably) orthogonal to the others.

The feature that really distinguishes the OLAP idea of dimension from other definitions is the way they divide a dimension into a hierarchy, as they demonstrate above with time and geography. Also, there is considerable freedom in the kinds of information that can be used for a dimension. Categorical or nominal attributes are often used as dimensions in OLAP. In contrast, the dimensions of a scientific database require ordinal information at the very least; most systems assume the dimensions are metric and continuous. To be precise, if an attribute is metric its *value space* must have a distance measure that meets the following conditions [KAO97]:

Let d be a distance function on X (the value space) and $\forall$ p,q,r $\in$X:

1. $d(p,p)=0$
2. $d(p,q) \leq d(p,r)+d(r,q)$
3. $d(p,q)=d(q,p)$
4. $d(p,q)=0 \Rightarrow p=q$

We focus our research on data that can be meaningfully represented in a continuous k-dimensional data space. Practically speaking, if one or more independent attributes of the data can be mapped to the set of real numbers, then the data is dimensional for our purposes. Note that attributes that are essentially integers can still be represented in $\Re$. Using a continuous representation allows interpolation even with integral attributes. In the event that a researcher wishes to use non-metric data as a dimension in a scientific database, Kao [KAO97] has developed techniques for imposing a metric on data that would otherwise be considered categorical or nominal.

It is not necessary for all attributes to be dimensional. If we view the data as a function, some of the dimensional attributes define the domain of this function, and the remaining attributes define the range. Our function therefore maps any point in the domain defined by the dimensions to a particular range value. Choosing which attributes should be used as dimensions is up to the researcher using the system, and can be an important part of the data exploration process. We call each possible combination of dimensions a *view* of the data, a notion similar to the "view" found in traditional databases. So, for data with *m* attributes, *k* of which are dimensional, there are 2^k-1 possible views. Each view affords a different way of looking at the same data.

A natural example of dimensional data is *spatial data* such as satellite images and fluid flow datasets. Here, the data represents an actual space. However, it is possible for a dataset to be dimensional without being spatial. For example, data from a Greenland ice core sample might contain readings for calcium, nitrogen, and carbon concentrations at different times in the Earth's history. We can represent this data dimensionally, but as it does not correspond to a real space, it is not spatial. However, it may be very beneficial to visualize the data as if it were spatial, since humans find this representation familiar and easy to grasp. For this reason, we often use the word "spatial" in this document, even when referring to data which does not represent a physical space. Furthermore, it may be convenient to treat a set of attributes as if they are dimensional attributes even though they may not satisfy all the conditions for dimensional data. In particular, we often don't know exactly which attributes are independent of each other, but we might want to assume they are independent for exploration purposes with the goal of either validating or disproving that assumption.

An example

In a relational database, data is stored in tables, which are essentially lists of tuples. Each tuple may consist of one or more attributes or fields. For example, each tuple of a table for employees might contain values for the fields *name, id number, salary, total sales,* and *travel expenses.* We can view this table as a kind of function that relates the different field values. For this reason, tables like the one shown in figure B.3 are sometimes called relations.

Name	ID number	Salary	Total Sales	Travel Expenses
Bill	123	30000	200000	2000
Bob	567	50000	157000	4000
Jane	876	60000	259000	3000

Figure B.3

It is possible to take this notion of treating data as a function even further by designating axes as in a Cartesian plot. For example, if *total sales* and *salary* are chosen as axes, each tuple would correspond to a point in the plane defined by the axes.

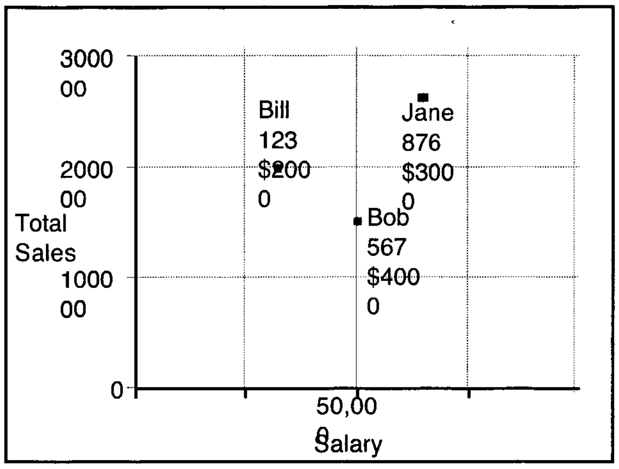


Figure B.4. A dimensional view of Figure B.3

These points should each have attributes *name, id number,* and *travel expenses*, as seen in figure B.4. We can say that the data in figure B.4 has been represented *dimensionally,* where *salary* and *total sales* are the dimensions. Of course, we are not restricted to just two dimensions. We could make a three dimensional graph by using *travel expenses* as the third dimension. Alternatively, *total sales* could take the role a dimension instead of *salary*, yielding a different two dimensional graph. However, it would not be sensible to use *name* or *id number* as dimensions, since these are nominal attributes. That is, these attributes just serve as names, and their value sets do not have orderings associated with them. In contrast, the other attributes are metric, since they have an associated distance measure and fulfill the conditions listed previously. Here, the distance measure is just the difference in the dollar amounts. Since their value sets have an implicit ordering, these attributes are also ordinal.

B.4.3 Regular and Irregular Data

In a two dimensional *regular* dataset, the points lie at regular intervals within the dimensional space, defining rectangles (2D) or hexahedrons (3D) of equal size and shape. This kind of data can be easily represented with a straightforward array, so many algorithms for the manipulation of regular data exist. If we allow the spacing of steps along the axes to vary, we have a *perimeter lattice*[SCVT]. In addition to the array holding the data, perimeter lattices require an array for each dimension holding the steps along the axis.

In addition to the placement of points in physical or geometric space, it is also important to know the *topology* of the data [KAO97]. The topology is usually described as a graph containing nodes and edges, for which nodes represent data points and edges between them represent an adjacency relationship. Some authors use the term *mesh* or *grid* to describe this graph [CIGNO97,SPER90].

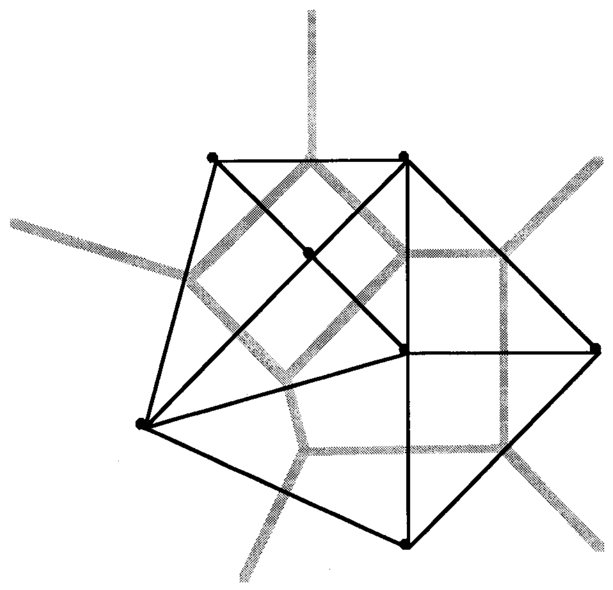
Sometimes the arrangement of sample points in physical space is not regular, but *curvilinear*. For example, a fluid flow simulation of air velocities over the top surface of an airplane wing might produce samples that lie in concentric curves echoing the shape of the wing. However a regular grid can be *lifted* (mapped) to the physical space to provide a dataset that is regular in computational space [CIGNO97].

On the other hand, *irregular datasets* consist of data points that are not distributed in a regular fashion. Cignoni [CIGNO97], claims that the term *irregular* applies only to data that is not regular and has a mesh that is known in advance. He uses the term *scattered dataset* to refer to data that has no mesh, so that one must be constructed from the data. In either case, an array based representation is very unlikely to be effective. However, a mesh of triangles (*trimesh*) can be constructed that covers the dataset using a process called *Delaunay Triangulation.* This method relies on two other concepts, *Dirichlet Tessellation* and *Voronoi Diagrams* [LATTU95]. In 1850, Dirichlet devised a way to divide the plane populated with a set of points $P=\{p_1...p_k\}$ into regions such that each region R_i contains only points that are closest to p_i. These regions, known as *Voronoi Regions*,are convex polygons covering the plane. If we take every pair of points that lie in adjacent regions and connect them with an edge, the resulting graph is the Delaunay triangulation: a mesh of triangles spanning the entire set of points. Figure B.5 shows the relationship between Voronoi regions and the Delaunay triangulation. Notice that the same technique can be extended into three dimensions by using planes instead of lines to define the Voronoi regions (polyhedra), yielding a Delaunay *tetrahedrization* in which tetrahedra instead of triangles span the dataset. This method is very useful for irregular volumetric datasets. Indeed, there is no theoretical reason why the technique couldn't be extended to an arbitrary number of dimensions.

B.4.4 Point and Region Based Data

Another important distinction is between point and region based data. In region based data, each data value represents a measurable subset of the domain. For example, if a regular dataset represents 1 square mile, and contains 100 data values, each value should represent .01 square miles. In contrast the values of point based data have no extent; they represent an infinitesimal point of the domain.

An important kind of region based data is *cell based* data. For two-dimensional cell data, each cell is made up of four points, forming the corners of a rectangle. Any value



▨ Voronoi Regions
—— Delaunay Triangulation

Figure B.5

within the rectangle can be interpolated from the corners. Naturally, this technique can be extended to an arbitrary number of dimensions.

A similar interpolation method is used with Delaunay triangulations. Recall that a dataset's mesh makes interpolation easier when finding a value for a point that is not a sample point. Such a point must lie within some triangle of the mesh, so a value can be interpolated from the three vertices of the triangle. If a region based representation for irregular data is desired, the Voronoi regions are convenient since each region contains the points that are closest to a particular data point. It follows that this region contains the set of points that are best represented by the single data point value.

B.4.5 Neighborhood Operations

A major advantage of storing data in an array is that the data is arranged in a spatially coherent manner, meaning that values that are conceptually nearby in the dataset are also nearby (in some sense) in the representation of that dataset. Once a datapoint *p* has been found, it should be inexpensive to find points that are near *p*. With arrays this is clearly the case, since adjusting an index leads to a neighboring value. The same cannot be said of a common relational database system or even a triangulated mesh, unless special data structures are used.

Ester *et al.* [ESTE97] describe a graph based method for representing the neighborhood relationship. If two data items are neighbors, the graph has an edge between two nodes corresponding to the items. This representation allows them to support operations like *get_ngraphs(item,relation)* which returns the graph of all items in the neighborhood of *item,* as determined by one of the following neighborhood relations:

- **Topological relations** such as *meet, overlap, covers, contains, inside, equal*
- **Metric relations** such as *distance<d*
- **Direction relations** like *north, south, east, west*

The neighborhood of an item is not restricted to immediate neighbors. The authors define a *neighborhood path* to be a path in which each edge satisfies a specific neighborhood relation. A neighborhood graph may contain many such neighborhood paths. Once a neighborhood graph has been computed, the *get_neighborhood(ngraph, item, predicate)* returns the set of all items directly connect to the item argument by some edge satisfying the predicate. Other operations include *create_nPaths*, which essentially computes a spanning tree for a neighborhood graph, and *extend*, which extends such a tree by a specified length. The authors claim that these operations comprise a set of operations that are basic to any spatial data mining application. They go on to explain the use of these operations in such tasks as discovering spatial trends and clusters, and classification of spatial data.

Appendix C

MULTIRESOLUTION

C.1 MR Methods for Regular Data

A common implementation of non-adaptive MR uses an array of points to represent the data set function S. Since L_0 is the original data, we expect that L_1 represents the same information at a coarser resolution, i.e., with fewer data points. A simple way to do this is to have each point in L_i represent 2^{∂} points of L_{i-1}, where ∂ is the dimensionality of the dataset. So, for a one dimensional dataset, the first point of L_1 should represent points 0 and 1 of L_0, the second point should represent points 2 and 3, and so on. So, L_1 is half the size of L_0, and L_2 half the size of L_1, etc. This approach can be extended to any number of dimensions. For example, in three dimensions, each point of L_1 represents eight points of L_0, which is the familiar octree data structure used commonly in computer graphics.

C.1.2 An Octree Method

Chamberlain [CHAM96] describes just such a method for use in a computer graphics rendering system. He is concerned with the efficient rendering of a scene consisting of some number of polygons. He notes that in many systems, considerable effort is wasted by painstakingly rendering polygons that only occupy a single pixel in the final image. His solution is to divide the scene space using an octree hierarchy of *color cubes*, which are essentially just cubic areas of the scene space with a special property: the faces of the cubes have a color and opacity matching the overall color and opacity of the polygons inside when seen from that direction. These color and opacity values are precomputed so they are readily available at rendering time. This preprocessing is done in bottom-up fashion, so that the color and opacity for a non-leaf color cube can be computed by a composition of color and opacity for its eight child cubes. For leaves, the color and opacity values must be directly computed from the polygons themselves.

For a given viewpoint, there is a set of color cubes that are so far down in the hierarchy (and therefore so small) that the space they contain maps to a single pixel in the image. Instead of wasting time rendering the polygons contained in these regions, Chamberlain uses the color cube values for a much faster rendering. In fact, the overall time complexity of the algorithm for n polygons is $O(\log n)$ compared to $O(n)$ for more traditional methods. The amount of data required to render a scene is also $O(\log n)$.

Chamberlain's paper is a nice example of a simple MR representation, and the advantages of using a coarse representation of data when appropriate to save time and space. Lounsbery *et al.* [LOUNS97] describe a more complex technique for computing approximations for distant objects that depends on wavelets.

C.1.3 Combining Data Values

Another important issue is how to combine two or more points into a single point for the next level. The method used depends upon the application. In the simplest case, where each point has only one attribute, we might just average points together to get a single value. However, it might be desirable for a point in L_i to keep track of attributes like minimum, maximum, and standard deviation for all the points it represents in L_{i-1}. Li *et al.* [LI98] store a probability density function for each point in their MR representation.

C.2 Wavelets

Wavelets are a very popular class of multiresolution representation. A wavelet representation of data includes two parts: the summary and the detail. As the names imply, the summary is an approximation of the original data, while the detail can provide a more refined representation when combined with the summary. Perhaps the simplest wavelet is the Haar wavelet, a system of compactly supported orthonormal functions [STOLL96]. Before discussing the Haar basis, we should examine a simple example of the Haar wavelet applied to a one dimensional dataset. Consider this series:

{6 8 11 7 4 2 3 7}

To compute the summary for this dataset, we simply average each distinct pair of the series. More formally, we compute:

$$\frac{c_{2i} + c_{2i+1}}{2}, \text{for i=0...} \frac{m}{2}, \text{where m=8, the size of the series}$$

Similarly, for the detail, we compute the difference of pairs of values:

$$\frac{c_{2i} - c_{2i+1}}{2}, \text{for i=0...} \frac{m}{2}$$

Our result is the summary and detail coefficients:

Summary={7 9 3 5}, Detail={-1 2 1 -2}

Notice that if we add the first detail coefficient to the first summary coefficient, we get back the first element of the original dataset. That is, 7+(-1)=6. Similarly, if we subtract the first detail coefficient from the first summary coefficient, we get back the second element of the original data. We can retrieve the third and fourth elements of the original data using the second summary and detail coefficients in the same fashion. Clearly, the entire original dataset can be recovered from the summary and detail, with no loss of information.

To complete the multiresolution representation, we repeat the process above, using the summary coefficients in place of the original data. This yields another summary and detail set, each with two elements. This new summary set can be collapsed further into a single summary value, and a single detail coefficient. The complete process is shown below:

Original data={6 8 11 7 4 2 3 7}

Summary={7 9 3 5}, Detail={-1 2 1 -2}

Summary={8 4}, Detail={-1 -1}

Summary={6}, Detail={2}

The detail coefficients can be used to reconstruct the complete MR representation, including the original data. That is, the original data can be reconstructed with no loss of information from the last summary coefficient, and all the detail coefficients listed in bottom-up order. In the example above, this would be:

{6 2 -1 -1 -1 2 1 -2}

This sequence is known as the *wavelet decomposition* of the original data. The wavelet decomposition can be used to store or transmit the entire MR representation without incurring costs above the storage or transmission costs of the original data. On the other hand, rendering times for wavelet representations are generally higher than times for octree-like representations [CIGNO97].

The other important property of wavelets is that they can be used to construct a variety of MR and AR representations. In particular, the detail component of a wavelet can be used as an error measure for the accuracy of the summary component at each level [WONG95]. This is equivalent to using only the summary component at one level to reconstruct the data at the next finer level. Other techniques discard low magnitude coefficients of both the summary and detail coefficients, replacing them with zero upon reconstruction [STOLL96]. Finally, it is possible to make local decisions about whether to go to the next finer resolution of the wavelet representation, yielding an AR representation.

Simhadri *et al.* [SIMHAD98] use the MR properties of wavelets to detect edges and motion in cloud formations. Cloud formations are difficult for traditional rigid-body motion techniques, because there is movement in different directions at different spatial scales. For example, a cold front might be moving east to west at a large scale, but if the edge of the front is examined, may will be shearing and turbulent movement at this boundary in various directions. Using satellite images represented as regular arrays of pixels, the authors use a wavelet representation to perform edge detection at different resolutions. This allows them to capture both the large scale east-west motion of the example cold front, as well as much smaller, finer movements.

C.3 MR Methods for Irregular Data

Heckbert [HECK97] offers a taxonomy of techniques for surface simplification algorithms. Since surfaces are often represented using an irregular set of points, much of this work can be applied to irregular scientific data. If a simplification algorithm is applied repeatedly, we can generate an MR representation of the data.

C.3.1 Classifying Methods

The first area to examine when classifying such algorithms is the characteristics of the problem they are meant to solve. The most important such characteristic is the nature of the input. Input data can vary by topology and geometry, and also in the number and kind of attributes of the data points. For example, the input curve or surface might be described by a mathematical function or a set of points. A set of points may be either regularly or irregularly distributed. The points themselves may have a variety of attributes, especially in scientific applications. We are particularly interested in exploring the surface simplification domain to find methods applicable to irregular scientific data.

An issue of importance is whether the output of a simplification method contains only original data points or is allowed to construct interpolated values. Either method may be appropriate in a scientific setting, but the scientist should be provided with a way to distinguish manufactured values from original values.

Another point of classification is the characteristics of the algorithms themselves. For example, *refinement* algorithms start with a coarse representation of the surface and add points to build an increasingly accurate representation of the data. *Decimation* methods start with the original data and successively remove data values to construct a coarser and more compact representation. Note that with surface simplification, only a single representation of the domain may be required. However, most decimation or refinement algorithms should be easily adaptable to MR by saving the results of intermediate steps in the levels of an MR representation. In the context of MR, decimation methods can be thought of as the *bottom-up* approach, and refinement methods as *top-down*.

One more characteristic of algorithms is the tradeoff with regard to speed versus quality. Algorithms that produce representations with maximal quality or minimal size tend to be slow. Faster algorithms must sacrifice either quality or size, or both.

As discussed in section 2.4.2, irregular data is often represented using a mesh of triangles, or trimesh, and this mesh can be generated for an arbitrary set of points using Delaunay triangulation. A great deal of research has been conducted on the multiresolution representation of trimesh surfaces because of their application in both computer graphics and GIS. In both fields, trimeshes are often used to represent terrains with mountainous or hilly features.

C.3.2 2D Irregular data

De Berg and Dobrindt [DEBE98] have developed a multiresolution method for terrains that is specifically geared towards computer graphics. Their motivation is similar to Chamberlain's (sec. C.1.2), in that they also want to eliminate the unnecessary rendering of detailed polygons in the distance. Chamberlain's method fully renders polygons that subtend more than one pixel, while de Berg and Dobrindt's algorithm allows resolution to slowly decrease as the terrain recedes into the distance. An important feature of the algorithm is that it seamlessly blends these different resolutions into a single mesh. De Berg and Dobrindt's method could therefore be classified as a kind of adaptive resolution representation.

The algorithm begins by describing a simpler tree-based MR method for trimeshes, in which each triangle of a coarse level is broken into three or more triangles in the next level. That is, given a triangle *t* of level L_{i+1}, we construct corresponding triangles in L_i by adding one or more data points to the interior of *t*, and retriangulating the interior with the new points. The process is repeated until all the data points have been added, giving the original data, L_0. Notice that this subdivision process results in a tree relationship in which each non-leaf triangle is the parent of some number of child triangles, and each nonroot triangle has exactly one parent. The problem with this approach is that it results in very slender triangles as the triangles are subdivided because edges are never removed as new points and edges are added (Figure C.1). The authors point out that such triangles may cause "robustness and aliasing problems."

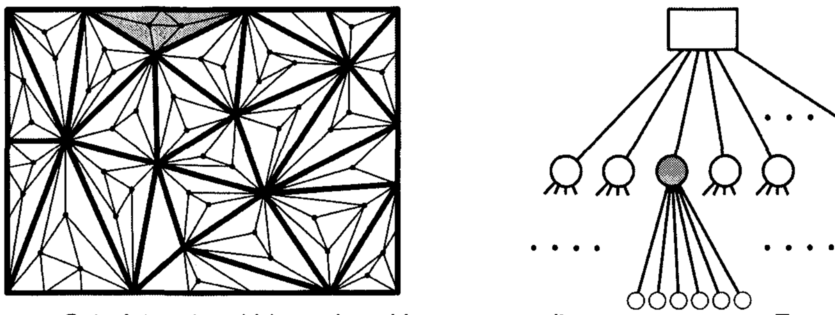


Figure C.1. A two level hierarchy with corresponding tree structure. From [DEBE98]

It is an important property of Delaunay triangulations that the minimum angle of any triangle is maximized. In other words, the Delaunay method avoids problematic slender triangles. De Berg and Dobrindt describe another hierarchy in which L_i is formed by adding points from L_{i+1} and then doing a global Delaunay triangulation on the remaining points. This method does indeed avoid slender triangles, but has a drawback for the intended application. Namely, there is no clean relationship between triangles in L_i and L_{i+1}. A triangle in L_i may have more than one parent in L_{i+1}, and there may be no triangle or set of triangles in L_i that occupies the same region as a triangle in L_{i+1}. This means that triangles from different levels won't fit together, and cannot be used to create a seamless representation of the terrain with different resolutions. However for scientific data, this technique should still be useful.

The authors build their MR representation in a bottom-up fashion rather than top-down construction of the tree method. The original data (L_0) is first triangulated using the Delaunay method. In forming each level, they remove a set of points from the previous level and then retriangulate in the area of each removed point using Delaunay. It is important to note that doing this local retriangulation globally preserves the Delaunay property. That is, each level of the MR representation is a Delaunay triangulation, which minimizes the occurrence of slender triangles. In order for this to work, some constraints must be applied to the set of points I_i removed from each L_i. The set I_i is defined to be a *maximal independent subset* of the vertices of L_i. That is, no pair of vertices in I_i is adjacent in L_i. In addition, the vertices of I_i must have degree *d* no larger than 12 (this value was determined by experimentation). The authors also allow the user to specify a set of fixed points that are considered so important that their removal is never allowed. This importance might be due to error introduced when the point is removed. For example, if removing a point makes a mountain disappear, it may well be better to leave it. Alternatively, points could be considered so interesting that the user wishes to keep these points regardless of the resolution. Note that this makes the algorithm dependent on the data via the (perhaps hypothetical) *interestingness* function, so that de Berg and Dobrindt's algorithm can be classified as adaptive resolution.

The authors refer to the union of all triangles that contain a point $p \in I_i$, as a polygon. They show that the edges of this polygon belong to the Delaunay triangulation of L_{i+1}. So, once the point p is removed from the interior of this polygon, the polygon vertices are retriangulated to form a collection of triangles that make up the polygon for L_{i+1}. Notice that the polygons in the two levels cover exactly the same area of the domain. This is the property that allows a smooth blending of different levels to form a seamless representation of the surface with different resolutions.

The algorithm continues forming new levels in bottom-up fashion through the process described until the number of points in a level is some constant multiple *c* of the number of fixed points. The value of *c* depends on *d*, the maximum degree of removable vertices. For d=12, the authors found that c=2 worked well. Notice that the number of levels therefore depends on the size of the original data, which makes the algorithm true MR rather than LoD, according to the definitions of Cignoni *et al.*

C.3.3 3D Irregular data

In addition to providing valuable background information about visualization of scientific data, Cignoni *et al.* present two methods for representing irregular volume (3D) datasets [CIGNO97]. The first method is a refinement (top-down) technique, and extends work first presented in [CIGNO94]. It uses Delaunay tetrahedrization, the three dimensional extension of Delaunay triangulation in which the domain is covered with a mesh of tetrahedra. The domain is assumed to be a convex polyhedron. Any such polyhedron has a tetrahedrization that uses only the vertices of the polyhedron, without requiring the addition of extra vertices. In the case of non-convex polyhedra, such a tetrahedrization may not exist. Furthermore, deciding on its existence is an NP-complete problem. Cignoni's algorithm begins with a tetrahedrization of the domain using only domain vertices, so he rules out the problematic case of non-convex domains.

Problems also occur when a mesh defined in a rectilinear computational domain is lifted to a curvilinear physical domain. If a mesh is constructed in the rectilinear space, it will be *projected* into the curvilinear physical space. The projection process affects only the vertices of the mesh, and not the edges. Edges therefore remain straight lines after projection onto the physical domain even though they should really be curved to avoid error. Such error is called *warp* and cannot be eliminated entirely, but can be reduced by minimizing the length of edges. Another source of error is the interpolation that is done between mesh vertices. This interpolation is usually linear, but the function being approximated often isn't. Cignoni *et al.* use δ and ε to represent warp and error, respectively, and combine these into a single threshold pair $\mu=(\delta,\varepsilon)$. They compute μ values both for individual points and also for entire meshes. Finding the maximum μ value for any point is a useful characterization of the mesh.

The overall structure of Cignoni's refinement algorithm is fairly straightforward. They start with a tetrahedrization of the domain, as described above, and add points to this mesh one at a time until the mesh satisfies a bound for μ. This process involves three major steps:

1. Check to make sure that the mesh does not already satisfy the desired μ. This process is accelerated by using a list for each tetrahedron that keeps track of all data points inside it. The warp and error for each of these points is then used to find the maximum μ for the mesh. If the desired μ is satisfied, then stop.
2. Find the vertex not already in the mesh with maximum μ. Along with the lists mentioned in step 1, a priority queue is maintained, organized according to μ value.
3. Update the mesh by inserting the vertex found in step 2 using the Delaunay method. Go back to step 1.

With curvilinear datasets, there is yet another hazard. A mesh defined in the computational space might become *inconsistent* when the mesh is lifted to the physical space. In particular, the bending action of the lifting process may cause one or more tetrahedra to essentially turn inside out. The solution is to use a larger number of tetrahedra (with shorter edges) in that region. So, if an inconsistent tetrahedron is detected, a point contained in that tetrahedron is assigned an infinite warp value. This ensures that the point is chosen in step 2 above, and the offending tetrahedron replaced.

Cignoni *et al.* offer a variation of the above algorithm that divides the domain into some number of separate blocks that are each processed separately. The motivation is apparently to accommodate a parallel or distributed implementation of the refinement algorithm. In such cases it is important that the boundary faces of adjoining blocks be triangulated in the same way. They are able to show that this is indeed the case, and that the blocks would fit together when reassembled. However, they point out that the resulting tetrahedrization is not the same as one produced without splitting the domain.

Cignoni *et al.* also describe a decimation (bottom-up) method that is better suited for nonconvex datasets, since it begins with the original mesh and successively deletes vertices until the μ value for the mesh becomes too large. Once again, we can characterize the algorithm with three steps:

1. Check to see that the mesh satisfies the maximum acceptable μ. If not, then stop.

2. Select a vertex to be removed from the mesh.

3. Update the mesh by removing the vertex found in step 2. Go back to step 1.

The first step is simpler here than in the refinement algorithm, since any change in error or warping is local to the vertex last removed. Therefore, this local μ value can replace the global μ value for the mesh if it is larger. Unfortunately, step 2 is much more complicated. The algorithm must choose the vertex that increases the mesh μ value the least. It would be prohibitively expensive to simulate deletion of all mesh vertices, so heuristics must be used. The heuristic finds the edge v,w with minimum $\Delta\nabla_{v,w}$, the change in field gradient from point v to point w. A small change in field gradient implies low curvature in the function that the dataset represents, so linear interpolation along edge v,w has minimum error. Therefore, the vertex with smallest $\Delta\nabla_{v,w}$ is a candidate for removal. Warp for a vertex v is estimated by examining the distance between v and the plane containing the points adjacent to v. A large distance suggests a large warp value.

Once a suitable vertex has been chosen, it must be determined whether it can be safely removed from the mesh. Because tetrahedrizations of nonconvex polyhedra do not always exist, removal may not be possible. Since deciding this issue is an NP-complete problem, heuristics must once again be used. The heuristic attempts to remove a vertex v by collapsing an edge incident on v. That is, for some neighbor of w of v, the edges of v are connected to w. If this process results in an inconsistent tetrahedron, the deletion is not performed and the mesh remains unchanged. The error and warp of v is set to infinity so this vertex is not chosen in future iterations.

Appendix D

ARCHITECTURE AND SYSTEM ISSUES

D.1 Architecture and System Issues

There are a number of issues relevant to a multiresolution scientific database that require further examination. A comprehensive treatment of error is a necessary part of any system dealing with scientific data. This includes both error introduced by reducing resolution, and also error inherent in the data collection method. In addition, we must examine methods for accessing both multidimensional and multiresolution data structures, as well as ways to support search in a multiresolution environment. Lastly, we look at the issues involved in distributed and parallel computing using MR data.

D.2 Error

All scientific data contains some amount of error from the moment it is generated. If data is gathered from the real world, the instruments used cannot give perfect results. Values from simulation also have some uncertainty associated with them. After the initial data gathering phase, operations on the data may introduce further error. It is therefore important for a model of scientific data to account for error in any piece of data.

There are various ways to represent uncertainty in a data value. *Absolute error* is a value that does not vary with the magnitude of the measurement. For example, if an instrument has an absolute error of ±5 Volts, and we measure a value of 100 Volts, the true value lies somewhere between 95 and 105 volts. In contrast, *Relative Error* varies with the measurement's magnitude, and is usually expressed as a percentage. If we measure 1000 Volts on a second instrument with relative error of 5%, the true value lies somewhere between 950 and 1050 volts. Note that if 1000 Volts is measured on our first instrument, the true value would lie between 995 and 1005 volts.

Another important issue is the difference between accuracy and precision. Accuracy refers to how close a measurement is to the true value. Precision refers to how widely distributed a set of measurements are. So, if a set of measurements are closely grouped they are precise, but they could still be inaccurate if they are grouped around a value that is far from the true value. Knowing that a dataset is precise but not accurate has ramifications for the kind of conclusions that can be drawn from it. If a curve is plotted from this data, the shape of the curve would be correct, but it would be displaced by some amount from the proper position. For a discussion of precision and accuracy in the GIS community, see [FOOTE95].

Relative and absolute error are only two of many ways of representing error. For example, error could be described using a probability distribution function. Or, the precision and accuracy of the data set could be explicitly recorded as a pair of numbers. Ultimately a domain expert must decide what representation is appropriate and necessary.

In addition to the error representation, some kind of *error metric* must be chosen. Heckbert describes two very common metrics [HECK97]. L_2 error between two vectors **u** and **v** of length n is defined as:

$$\|\mathbf{u}-\mathbf{v}\|_2 = \left[\sum_{i=1}^{n}(u_i - v_i)^2\right]^{1/2}$$

Squared Error is defined to be the square of the L_2 error, and *root mean square (RMS)* error is the L_2 error divided by $\sqrt{n}$. L_∞ error, which is also called *maximum error*, is defined as:

$$\|\mathbf{u}-\mathbf{v}\|_\infty = \max_{i=1}^{n}|u_i - v_i|$$

Our model of scientific data requires *localized error,* meaning error is assigned to every point within the domain of the data. Ideally, each domain point would have an exact error value associated with it, but this is not always the case. However, it should always be possible to at least estimate the error of a given point. Instruments and simulations typically specify an error tolerance, so it should be easy to associate an error with data values gathered from these sources. For data values that are interpolated, perhaps only an interpolated estimate can be made. Cignoni *et al.* describe a way to generate localized error estimates for irregular data using a scan line technique [CIGNO98]. They evaluate surface values using a regular grid of sample points, and compare the values given by a simplified surface with those given by the original surface.

Our other requirement is that it should be possible to accumulate error. If data begins with error E_1, and an operation is performed that introduces additional error E_2, then the error E_r of the resulting data should reflect both E_1 and E_2. That is, $E_r=E_1 \oplus E_2$, where $\oplus$ is an error accumulation operator. This operator may be as simple as addition, but it need not have all the arithmetic properties of an addition operator.

D.3 MR Access

In addition to the spatial access methods outlined in the previous section, MR access methods must also provide a way to select the desired resolution of the query result. The most obvious way to select resolution is to ask for a particular level from an MR hierarchy. However, this is not an efficient method for all kinds of MR representations. For MR methods like those described in [CIGNO97], it is much more efficient to specify an error bound that must be met by the query result.

D.3.1 Error Based Access

To facilitate error-based access, Cignoni, *et al.* introduce the notion of an *historical sequence* [CIGNO97]. Each tetrahedron in the dataset is tagged with "birth" and "death" accuracies μ_b and μ_d. A tetrahedron is said to be *μ-alive* if $\mu_b \leq \mu \leq \mu_d$. The points and tetrahedra making up the dataset are stored in two separate files sorted by birth accuracy in non-decreasing order. To satisfy a request for a given accuracy, the tetrahedron file is sequentially scanned for all tetrahedra that are μ-alive. This search terminates once a tetrahedron with μ_b better than μ is found. Once the tetrahedra are retrieved, the vertices are obtained by scanning the list of points up to and including the index of the "greatest" point referenced by the tetrahedra set. This works because in the refinement algorithm described in section 3.4.2, points are never removed once they are introduced. Therefore, the set of vertices required by the tetrahedra set is a prefix of the sequence of vertices in the point file. Notice that if a query asks only for a subset of the domain using one of the MD query methods outlined in the previous section, modifications have to be made to the algorithms discussed here.

D.3.2 The FED Method

Resnick, Ward, and Rundensteiner have developed a method for specifying queries on dimensional data [RESN98]. Their work does not specify a data structure, but rather provides a user-level query model. Under this model, a user query consists of three parts:

- **Focus ($\mathcal{F}$)** describes the point in the domain that is the center of attention. $\mathcal{F}$ is specified as a single point or *k*-vector, where *k* is the number of dimensions in the dataset.

- **Extent ($\mathcal{E}$)** specifies the bounding hyperbox for the region of interest. $\mathcal{E}$ is usually specified as two *k*-vectors delimiting opposite corners of the bounding hyperbox.

- **Density ($\mathcal{D}$)** specifies the amount and distribution of data relative to the focus. The authors initially specify density using a single *k*-vector of values on the range 0…1, where 1 means "all data" and 0 means "no data" for the corresponding dimension. Values greater than one might request interpolation.

Since density and resolution are closely related, the specification of density makes this model potentially useful for MR access. For rectilinear data, we can specify different behavior in each dimension. For example, the k-vector $\langle 1.0, 0.5, 0.25\rangle$ specifies that along the x axis, all data should be displayed, but only every other row on the y axis, and every fourth plane on the z axis. The authors have other ideas, however. They mention that a *variable density* might be specified using a polynomial function such that the density is greatest at the focus, but falls off sharply toward the edges. This could easily be accommodated by selecting from different levels of an MR hierarchy. Detailed data at the focus is taken from lower levels, while the periphery provides context with points from higher levels.

D.4 Search with MR

We have already seen that a distinguishing feature of scientific data is that we don't know what patterns and relationships are contained in the data beforehand. Therefore, the techniques described in section 4.2.2 are not adequate, since they only help to retrieve known data. We need methods that aid the investigator in discovering new knowledge. As described in section 2.3, pattern detection techniques are important tools for knowledge discovery, so we should look for ways to apply them to multiresolution data.

Seale *et al.* [SEAL98] point out that performing object recognition on a compressed image data stream saves the work of transforming the data into pixels, and also reduces the volume of data that must be examined. Their work used the JPEG compression format, which is not really a multiresolution representation. However, their work hinges on the fact that certain key compression steps in the JPEG standard still retain spatial correspondence with the original image. This property is also a major feature of wavelets and multiresolution methods in general.

D.4.1 MR Feature Extraction Methods

Notice that both k-d trees and quadtree based techniques described in section 4.2.2 involve a hierarchy to help locate points in space. Since any multiresolution representation implies a hierarchy, we should be able to use this hierarchy to help locate features in MR data. The following two methods take advantage of the MR hierarchy to perform recognition and feature extraction.

Juffs *et al.* [JUFFS98] have developed a distance measure for images that is well suited for use with MR data, especially the Haar wavelet. Their measure is called a *Gray Block Distance* or GBD. Consider two images I and I' such that their average gray levels are *g* and *g'* respectively. A gray level of zero signifies black, while a level of one corresponds to white.

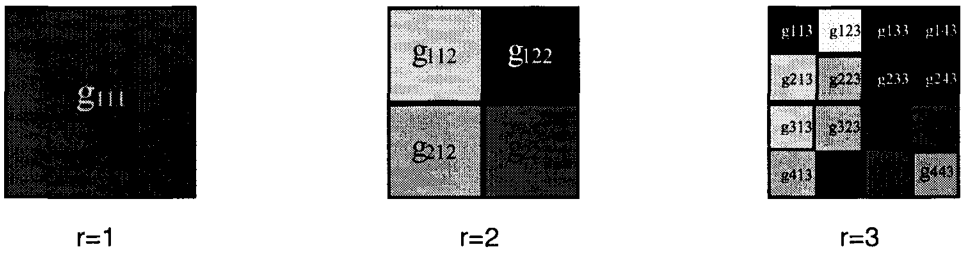


Figure D.1. Gray blocks for resolutions 1 through 3. Subscripts are in i,j,r order. From [JUFFS98].

In order to compute the GBD for a pair of images, the gray levels are compared at different resolutions. In figure D.1, it can be seen that the gray level for resolution one is the gray level for the entire image. For resolution two, four gray levels are computed—one for each quarter of the image. This sequence may continue for an arbitrary number of levels. So, given that we have gray levels for all required resolutions, we can compute the average difference in grayness for any level r:

$$\frac{1}{2^{2r-2}}\sum_{j=1}^{j=2^{r-1}}\sum_{i=1}^{i=2^{r-1}}\left|g_{ij}-g'_{ij}\right|$$

If the maximum difference between any two gray levels is one, then the maximum average difference for any level r is also one. However, in the full GBD given below, the average difference for each resolution is given a weight $\frac{1}{2^r}$:

$$GBD(I,I') = \sum_{r=1}^{\infty} \frac{1}{2^r} \frac{1}{2^{2r-2}} \sum_{j=1}^{j=2^{r-1}} \sum_{i=1}^{i=2^{r-1}} \left| g_{ijr} - g'_{ijr} \right|$$
$$= \sum_{r=1}^{\infty} \frac{1}{2^{3r-2}} \sum_{j=1}^{j=2^{r-1}} \sum_{i=1}^{i=2^{r-1}} \left| g_{ijr} - g'_{ijr} \right|$$

This means that as resolution increases, the differences between images become less and less significant. The authors claim that this mimics the human visual system in that we de-emphasize difference in detail in favor of overall similarity. In addition, this decreasing weight guarantees that the complete GBD is a number defined on the range 0...1.

The authors point out that there is a close relationship between this distance metric and the Haar wavelet. In particular, we should be able to use the summary coefficients from the Haar decomposition in place of the *g* values. This suggests a potentially valuable search method for wavelet data. First, the wavelet decomposition of a template must be computed. This template must be an example of the kind of target we are searching for. Next, a low resolution representation of the template is compared against the corresponding resolution of the data. The locations with the smallest average difference are recorded, and the next level of the template is compared against the next data level in these locations. The process may continue until we have found target regions in the data that match the template closely enough for the experimenter's needs. The chief advantage of this technique is that it avoids an expensive examination of the original data by taking advantage of the MR hierarchy. Areas that are found to be a poor match at the lowest resolutions are not examined in greater detail. This should provide an efficient way to locate features of interest for patterns that can be represented using a simple template.

Simhadri *et al.*[SIMHAD98] have developed an algorithm for feature extraction that is especially useful for motion detection in oceanographic images. They point out that detecting motion of ocean currents is fundamentally different from solid body motion detection because the motion occurs at different scales. Although a current may have a general direction at a large scale, the edges of the current may be moving in an orthogonal direction, with many swirls and eddies. Because of the multi-scaled nature of ocean features, the authors developed a multiresolution approach based on wavelets.

Their algorithm consists of the following steps:

1. Apply a wavelet transform to the image.

2. Truncate all summary coefficients below a user-defined threshold to zero.

3. Reconstruct the image at some resolution *j*.

4. Apply an edge detection algorithm using another user-defined threshold *T*

5. If the result is not satisfactory, then descend to the next level *j+1*, reduce the threshold *T,* and return to step 4.

The edge detection method used in step 4 is extremely simple. It involves applying a 3×3 window to every possible position in the image, and finding the difference between the maximum and minimum values contained in the window. If this difference is less than the threshold *T*, then the central pixel is replaced with a zero. Otherwise, the window is moved to the next location.

Notice that the threshold *T* is reduced when the algorithm descends to a finer resolution. The authors point out that the contrast of an edge is inversely related to the resolution of the image. If we have more pixels with which to represent a transition, then the difference between nearby pixels will be less. Notice also that the number of detected edges increases with increased resolution, also due to the larger number of pixels. When enough edges have been detected, the algorithm halts.

This process of edge detection is repeated on images taken at different times, so that motion can be inferred from the changes in the edges. The authors note that their technique is better than traditional edge detection techniques at handling faint edges and small details. Traditional methods have more difficulty distinguishing between less prominent features and noise. Here, the multiresolution technique has clear benefits.

D.5 Parallel and Distributed Computing with MR

The large size of scientific datasets suggests that there are benefits to applying more than one processor to the problems found in scientific databases. Certainly, many researchers have looked at how best to divide datasets among several processors. Parallel database systems are often implemented on a cluster of workstations connected by a high speed network. A distinct but related idea is *distributed computing*. Distributed computing is conducted over a network, but unlike parallel systems a distributed system does not necessarily divide the dataset among processors. For example, it may instead divide different stages of a process among several machines.

D.5.1 Distributed Computing

Charles Hansen and Stephen Tenbrink [HANS93] explain that imaging and visualization were major motivations for developing a new network protocol at the Los Alamos National Laboratory (LANL). Scientists at LANL run very processor intensive simulations, and need a convenient way to view and steer their progress. The authors report that the scientists were very reluctant to walk down the hall to a special laboratory, and instead wanted the convenience of working in their own offices. Since the office machines were inadequate for running the simulation itself, the visualization task was separated from the simulation.

The idea of distributing tasks among several machines can be taken much further, especially with large scientific databases. For example, the data itself may be stored on one set of machines, while the processing is done on another, and the visualization and user interaction is done on a researcher's personal workstation. Such schemes require that data be transmitted from machine to machine over a network.

Methods for transmitting MR data over a network often use *progressive transmission*. That is, a coarse representation of the data is transmitted first, followed by more detailed information that allows the data to be refined progressively. Notice that the wavelet decomposition, described in section 3.3, works naturally with progressive transmission. Wavelet coefficients are transmitted in order of magnitude, sending the largest first [STOLL96]. The representation on the receiving end is refined as the more detailed information arrives. This means that the researcher may view the data as it is updated, rather than wait until the entire representation has been transmitted. They may also decide that some intermediate degree of refinement is adequate for their purposes, and decline to download all of the data. Progressive transmission can be used with other multiresolution techniques besides wavelets. Descriptions of such methods can be found in [CIGNO97, HOPPE96]. Cignoni *et al.* couple progressive transmission with the decimation method described in section 3.4.2. Recall that the decimation method begins with the original mesh and selectively deletes vertices until the representation becomes sufficiently coarse. Progressive transmission reverses this process by sending the coarse representation to the remote machine first, and then sending the vertices in an order opposite from their deletion order. That is, the vertex that was deleted last is transmitted first. In order for this method to work, the vertices must be reinserted into the tetrahedral mesh. This requires an operation that is the inverse of edge collapse, namely *vertex splitting*. To perform vertex splitting, the transmitting machine must indicate which previously transmitted vertex the new vertex was collapsed from. With this information the collapsed edge can be reconstructed.

D.5.2 Parallel Computing

Parallel computing involves breaking up a single task among multiple processors. These processors may be part of a single machine, or may be in separate machines connected through a network. The principle advantage of parallel computing is performance.

If we use eight processors to attack a problem, we might (optimistically) expect the time to reach a solution to be one eighth the time required for one processor. That is, we expect the *speedup* to be 8.0, where speedup is defined as [LEIGH92]:

$$speedup = \frac{time\ \ for\ \ one\ \ processor}{time\ \ for\ \ n\ \ processors}$$

Furthermore, the *efficiency* is defined to be

$$Efficiency = \frac{100 \bullet speedup}{n}$$

In practice, an efficiency of 100% is very difficult to achieve for reasons outlined below. Still, reducing computation time is the principle reason to use multiple processors for exploring scientific data.

Load balancing plays an important part in optimizing parallel computation. Since the completion time for a parallel program is the time within which the last processor finishes its work, it is desirable to distribute workload evenly among the processors. Ahrens and Hansen [AHRE95] point out that load balancing must be done carefully. In some cases, the costs of distributing the load outweigh the benefits of balanced computation. In such cases, load balancing actually degrades performance.

Load balancing methods can be divided into two classes—*static* and *dynamic*. Static load balancing is done before the beginning of computation, and is not performed while the program is running. Lin and Li [LIN95] point out that many problems have unpredictable behavior, making it difficult for static load balancing to yield good performance. It is not known where the bulk of the workload lies until after computation has begun. To address this problem, dynamic load balancing is performed repeatedly while computation is in progress. Both static and dynamic load balancing can be done either locally or through global control. Load balancing through global control tends to yield better distribution because decisions can be made based on a large amount of accurate information. If load balancing is performed locally, each processor gathers information from processors in its immediate neighborhood. This may degrade the quality of the distribution somewhat, but also requires less communication than the global approach.

Nakano *et al.*[NAKANO97] have developed a dynamic load balancing technique that works with multiresolution physical chemistry data. Their system updates the distribution of data after every 60 iterations of their program. Specifically, they notice when an atom has moved from one processor's physical space to another, and subsequently adjust both the data and boundaries to compensate. In contrast, Pfaltz *et al.* [PFALTZ98] distribute data in their ADAMS scientific database by *oid* (object id). The ADAMS system emphasizes queries on large, usually dimensional datasets using boolean conditions and set operations. Their system uses 64 bit oids to uniquely identify each data object in the database. Data objects are distributed among $n=2^d$ processors using the d least significant bits of the oid. Although their system is static, they claim good performance for the kinds of operations their system supports, especially with larger datasets.

One difficulty with the ADAMS approach is that it seems to discard geometric locality when dealing with dimensional or spatial data. If a researcher wants to conduct an operation that applies to some neighborhood of points, then the ADAMS approach to load balancing results in excessive communication. Points that are geometrically in the same neighborhood are scattered over several processors. On the other hand, Nakano's work divides the geometric space over the processors so that each processor gets a roughly cubic contiguous chuck of the space. This means that many operations can be conducted locally on each processor with no need for external data. To handle other cases where data from other processors is needed, Nakano simply duplicates the data. Each cube sends the coordinates of atoms near its boundaries to the six neighboring processors.

APPENDIX E

UNCERTAINTY VISUALIZATION METHODS IN ISOSURFACE VOLUME RENDERING

E.1 Introduction

We describe two techniques for rendering isosurfaces in multiresolution volume data so that the *uncertainty* (error) in the data is shown in the visualization. In general the visualization of uncertainty in data is extremely difficult, but the nature of isosurface rendering makes it amenable to an effective solution. In addition to showing the error in the data used to generate the isosurface, we can also show the value of an additional data variate on the isosurface .

E.1.1 Visualizing uncertainty

With the exception of geographic information systems (see, for example, Hunter et al. [Hunt93]), there has not been much research into identifying and visualizing the uncertainty in data. Recently, however, researchers in other fields have begun to address this issue. For example, Lodha and Pang have experimented with visualizing uncertainty in vector fields [Pang94, Lodh96a, Lodh96b, Witt96, Shen98] and Cignoni et al. [Cigno98] have developed a tool, *Metro*, for visualizing mesh surface approximation error. In addition to the very difficult problem of identifying and maintaining the error itself, it is also very difficult to present that error to the user in an effective and meaningful form.

Incorporating uncertainty into any visualization requires rendering at least one more variate (actually we need to add one new variate for each variate that has its own error measure). Although many innovative multivariate visualization techniques have been developed in recent years and some have proven useful in some situations, this is an extremely difficult problem which is exacerbated by the enormous size of modern scientific data.

In principal, we would like to incorporate the uncertainty information into the data visualization. When the error information is *locally defined* (i.e., it has about the same resolution as the data), this approach usually results in some form of degradation in the display of the data itself. Wittenbrink et al. [Witt96] call these *overloading* techniques (as opposed to their glyph-based technique which they call *verity* visualization). Cedilnik and Rheingans [Cedi00] use annotations on a visualization in order to reduce the distraction caused by the error visualization. In order to be most effective, it is important that the user have the ability to turn the uncertainty visualization on and off interactively. With uncertainty visualization disabled, the user is likely to have the best chance of understanding the fundamental nature of the data. After enabling the uncertainty visualization, the user can now get an understanding of the error in the data.

E.1.2 Multiresolution data

One major reason for the relatively low emphasis placed on uncertainty visualization in the past is that uncertainty information is seldom available except in very abstract forms. With the growing interest in the generation of coarse resolution approximations to a large dataset, this particular limitation can often be overcome. Creating multiresolution data certainly introduces additional error into the data, but it is often relatively easy to measure this new error and it is usually significantly greater than the error in the original data. Consequently, we can expect to be able to create and access error information about coarse resolutions of a multiresolution data hierarchy. Furthermore, it is particularly important to incorporate error into the visualization of data that is only a coarse approximation to the "real" data. A scientist needs to know what portions of a coarse resolution visualization have relatively low error (and therefore are an authentic representation of the data in that area) and which have a relatively high error. The representation of the low error regions is likely to be reasonably authentic, but the scientist is likely to want to visualize areas of high error at a higher resolution.

E.1.3 Isosurface rendering

Isosurface volume rendering is a very good candidate for adding uncertainty visualization. Rendering an isosurface within a volume of univariate data is a very effective technique for many applications. Since all the data being visualized has the same data value, the particular value does not need to be incorporated into the visualization.

Conventional isosurface rendering assigns a constant color to the vertices of the triangles that define the isosurface, and uses standard lighting models and Gouraud interpolation to give a sense of the shape of the isosurface. Consequently the color parameter is actually available for visualizing the uncertainty. It's important to realize that this approach allows us to visualize the error of the data used to generate the isosurface rather than the error between the low resolution isosurface and its corresponding high resolution isosurface.

E.1.4 Research overview

In this paper we describe some experiments with incorporating an uncertainty variate into isosurface rendering. Our visualization tool is part of a broader research effort to develop a formal model and a support environment for dealing with large multiresolution and adaptive resolution data sets [Spar94, Rhodes01]. A fundamental aspect of this model is the incorporation of local error measures into the data representation.

Although our uncertainty visualization does not depend on any particular technique for generating the volume data, we start by describing our wavelet-based multiresolution volume data which does incorporate a meaningful error component. We conclude with some specific examples of the visual results of the approach.

E.2 Multiresolution volume data

Our motivation for developing a tool for incorporating uncertainty into isosurface rendering arose from our interest in using multiresolution data representation for large scientific data sets [Wong95, Wong00]. We are particularly interested in generating coarse approximations to a large dataset that are more tractable in terms of size but still retain sufficient authenticity to be useful. For this approach to be viable, it is critical that we provide an estimate of the error that is introduced into the coarse data *on a local basis*. In principle, every data point in each level of a multiresolution data hierarchy includes both data and an error measure associated with that data – its uncertainty. In other words, we want to identify the regions of the data where the coarse representation is not an authentic representation of the original data.

E.3 Isosurface rendering with error

E.3.1 Overview

We have extended the standard Marching Cubes algorithm to incorporate a measure of the error of the data. Volume data points contain both data values and the error associated with each data point. During the Marching Cubes algorithm, we compute an error associated with each triangle vertex by interpolating between the error values of the associated cube vertex error values. We use the error value for each triangle vertex to modify the appearance of that vertex.

E.3.2 Uncertainty rendering using color

The vertices of the triangles that define the isosurface are assigned a color based on hue, saturation and brightness and the triangles are then rendered using an external light source and Gouraud shading. For basic isosurface rendering (without uncertainty enabled), all triangle vertices have the same color. The user may choose to map the uncertainty to any of the three color parameters (hue, saturation and brightness), while leaving the other two parameters fixed. In addition, the user can interactively select what constant values should be used for the other two color parameters. Since hue is specified as an angle between 0 and 360, it is clearly not desirable that the full range be used – if it were, the largest and smallest error values would have the same hue. Consequently, we allow the user to select the range of hues that should be used for the uncertainty.

Although we allow users to assign the uncertainty rendering to any of the three color parameters (hue, saturation, brightness), we recognize that the only reasonable mapping for this particular problem is to map the error to the hue. The brightness component is needed in order to effectively represent the shape of the isosurface and it is well-known that we are far more sensitive to hue changes than saturation changes. In general, the perceptual issues associated with color usage are orthogonal to the goals (and scope) of this paper.

E.3.3 Uncertainty rendering using texture

We have developed a second error visualization method that uses texture to show regions of the isosurface with high uncertainty. Textures and texture hardware have been used by various researchers as an aid to data visualization [Boada01, Cigno98, Cabral94, Guan94, LaMar99]. These approaches either use texture hardware to accelerate visualization, or rely heavily on the color component of the textures for their visual effects. Our approach, on the other hand, does not use hue as part of the texture, so that it is available for visualizing another variate on the isosurface.

Our implementation uses a second *texture surface* which envelops the original isosurface, but is slightly offset from it. A stipple texture is mapped to this surface, and the opacity is varied according to the uncertainty of the data. That is, the texture will be most visible in areas with high uncertainty, but absent or faint where uncertainty is low.

Figure E.10 shows the interaction between color and texture visualization. The topmost row simply shows a set of typical hues. The second row shows a texture imposed over a green surface. The texture becomes increasingly visible as the tiles progress to the right. Notice that the underlying green can still be seen, even in the rightmost tile.

In the third and fourth rows, the hue of each tile varies as in the first row, but now we have imposed the texture as well. For the third row, the texture becomes increasingly visible as we progress to the right, but the opposite occurs in the final row. In either case, both the texture and underlying hue are suitable for visualizing distinct variates. For example, with fluid flow data, we might use the pressure variate to compute the isosurface, map the error of the pressure to the texture surface and render the temperature variate to the surface hue.

E.4 Experimental results

Our isosurface software is implemented in Java and is built on the *VisAD* system [VisAd, Hibb92] which uses Java3D for rendering. Figures E.1 through E.9 were rendered directly in this system. The remaining figures were rendered in a separate program using gl4java [Gl4java], since we needed a lower-level API to implement the texture based error visualization.

For these tests we used a CAT scan of a cadaver head provided via *ftp* courtesy of North Carolina Memorial Hospital and Siemens Medical Systems, Inc., Iselin, NJ. The original data is 113x256x256. For the convenience of the wavelet transform, we appended 15 slices of zeros to get a 128x256x256 dataset. We then applied a 2D Haar wavelet to each slice and three successive 3D Haar wavelets to get a 4 level hierarchy. Figure E.1 shows an isosurface rendering of the 128^3 dataset for the isovalue 0.185 (a skin value). The next two coarser resolutions of the skin isosurface are shown in Figure E.2 (64^3) and Figure E.3 (32^3). It is clear that the surface shown in Figure E.2 is coarser than that shown in Figure E.1, but the overall impressions of the two surfaces are very similar. Figure E.3, however, shows a substantial loss of accuracy of the surface.

E.4.1 Uncertainty mapped to hue

Figure E.4 shows the skin value isosurface of the 128^3 resolution data with constant saturation and brightness and uncertainty mapped to the range of hue from 144 degrees (green) *down to* 0 degrees (red). In other words, green represents low uncertainty and red high uncertainty. The error associated with the 128^3 dataset is very low and this is reflected in the visualization. At normal scale no high error areas are visible although at very high magnification it is possible to see some very light pink areas around the mouth region. Figure E.5 shows the uncertainty visualization of the 64^3 dataset using the same visualization parameters as Figure E.4. Here more error is readily discernible as reddish areas around the mouth, eyes, forehead and other places. Figure E.6 shows the 32^3 resolution data with the same visualization parameters. As we would expect, there is obviously increased error in many areas of the visualization.

It is not clear what range of error might be expected for different kinds of input data and so it is also unlikely that there is a single ideal mapping of error to color. Figures E.7 and E.8 show the 64^3 and 32^3 data sets with a narrower hue range (108 to 0) intended to accentuate the error while maintaining red as the color of the highest error.

The last two figures show our texture based error visualization method. Figure E.11 shows the skull data with error mapped to texture transparency. Regions of high error can be seen above the ear and proceeding left towards the forehead. Figure E.12 demonstrates the use of texture visualization of error while hue is mapped to another variate. For this example, we generated a synthetic variate based on polygon normals to demonstrate the technique. The texture visualized error can be clearly seen even though it interferes minimally with the accurate visualization of the synthetic variate.

E.5 Conclusions and future research

Isosurface rendering of multiresolution data is an ideal candidate for including uncertainty visualization. The incorporation of the uncertainty into the visualization using color is relatively easy and provides effective feedback about where the visualization is unreliable without detracting significantly from the data visualization, especially in areas of low uncertainty. Texture based visualization of uncertainty has the additional benefit of making the surface color available for the visualization of another variate. In addition to the MR isosurface renderer we have shown here, we have incorporated this technique into a system for creating and rendering adaptive resolution volumes [Laramee02]. Figure E.9 shows a rendering from that system. We intend to incorporate uncertainty into more complex visualization techniques, such as direct volume rendering (DVR) and flow visualization.

Figures

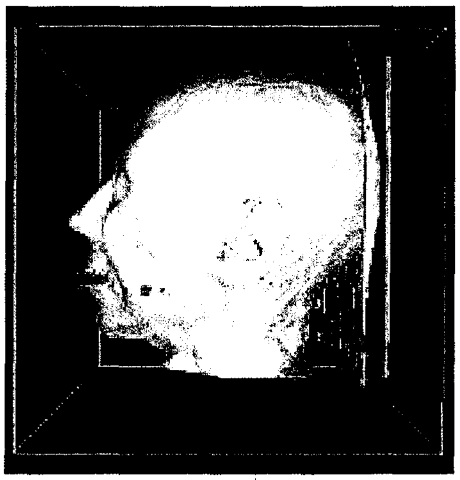


Figure E.1. 128^3 data; skin isovalue (0.185);uncertainty disabled

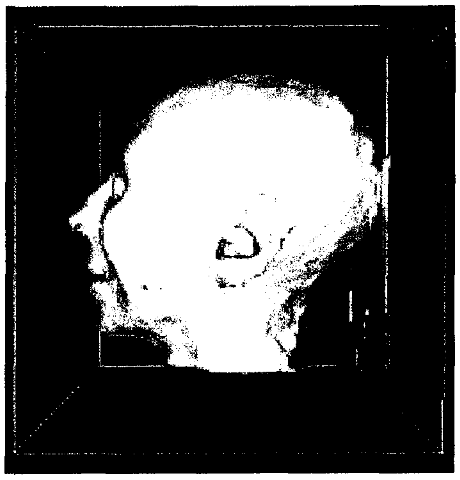


Figure E.2. 64^3 data; skin isovalue (0.185); uncertainty disabled

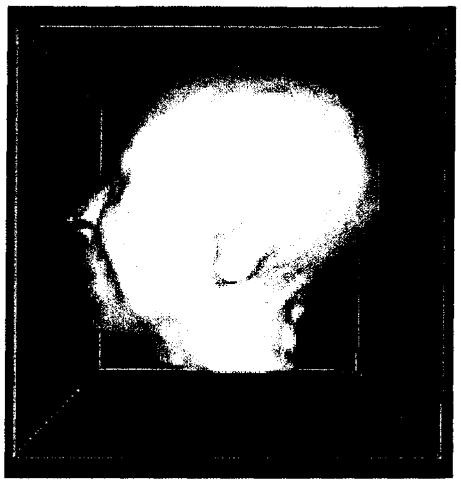


Figure E.3. 32^3 data; skin isovalue (0.185); uncertainty disabled

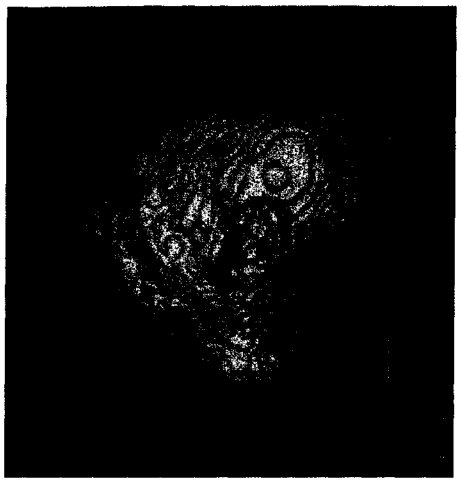


Figure E.4. 128^3 data; skin isovalue (0.185); uncertainty mapped to hue with range (144,0)

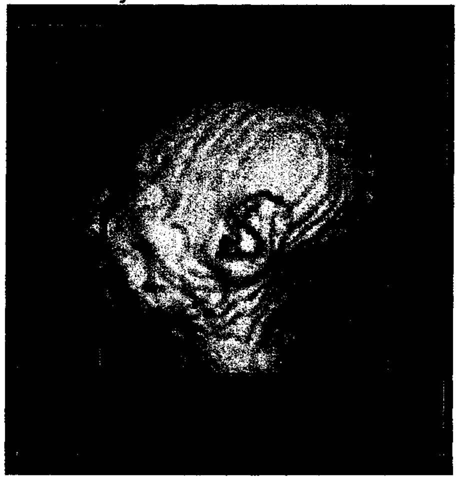


Figure E.5. 64^3 data; skin isovalue (0.185); uncertainty mapped to hue with range (144,0)

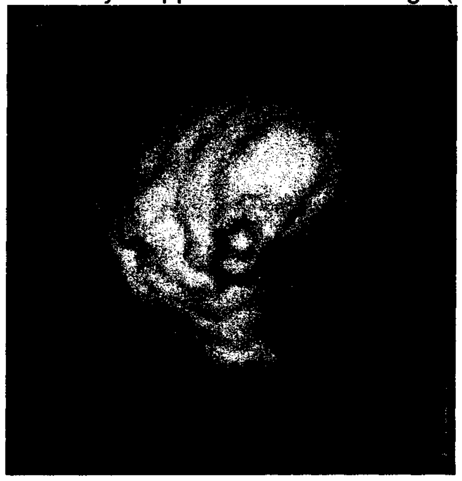


Figure E.6. 32^3 data; skin isovalue (0.185); uncertainty mapped to hue with range (144,0)

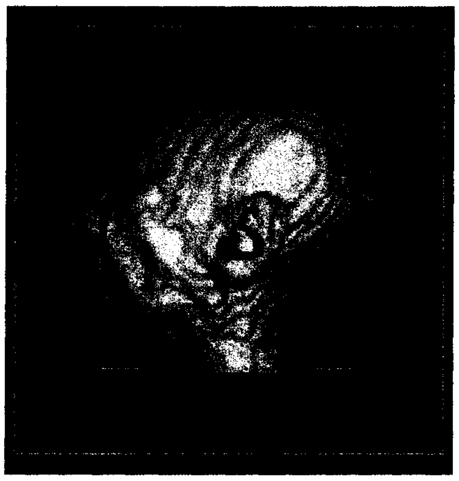


Figure E.7. 64^3 data; skin isovalue (0.185); uncertainty mapped to hue with range (108,0)

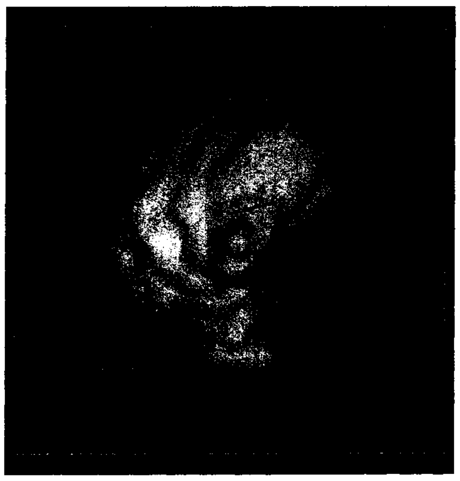


Figure E.8. 32^3 data; skin isovalue (0.185); uncertainty mapped to hue with range (108,0)

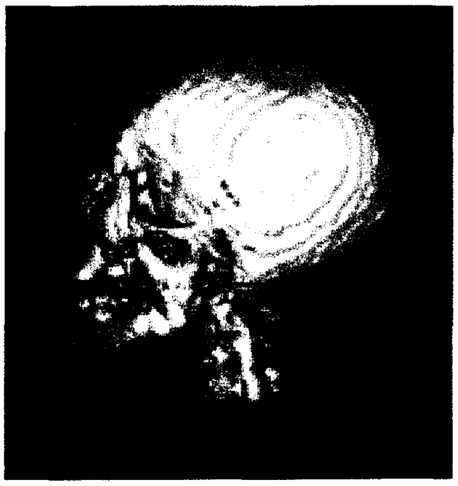


Figure E.9. 5% AR Data; bone isovalue (0.378); uncertainty mapped to hue with range (144,0)

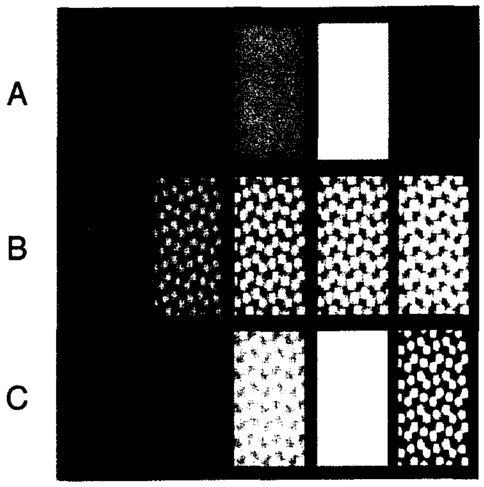


Figure E.10.
a) Varying hue only..
b)Texture ofincreasing opacity over constant hue.
c) Texture of increasing opacity over varying hue

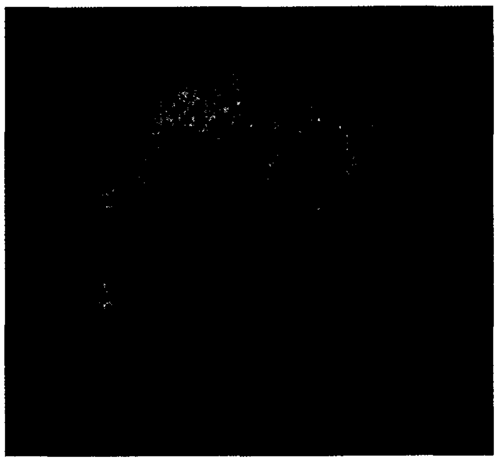


Figure E.11. Error mapped to texture opacity over a constant hue.

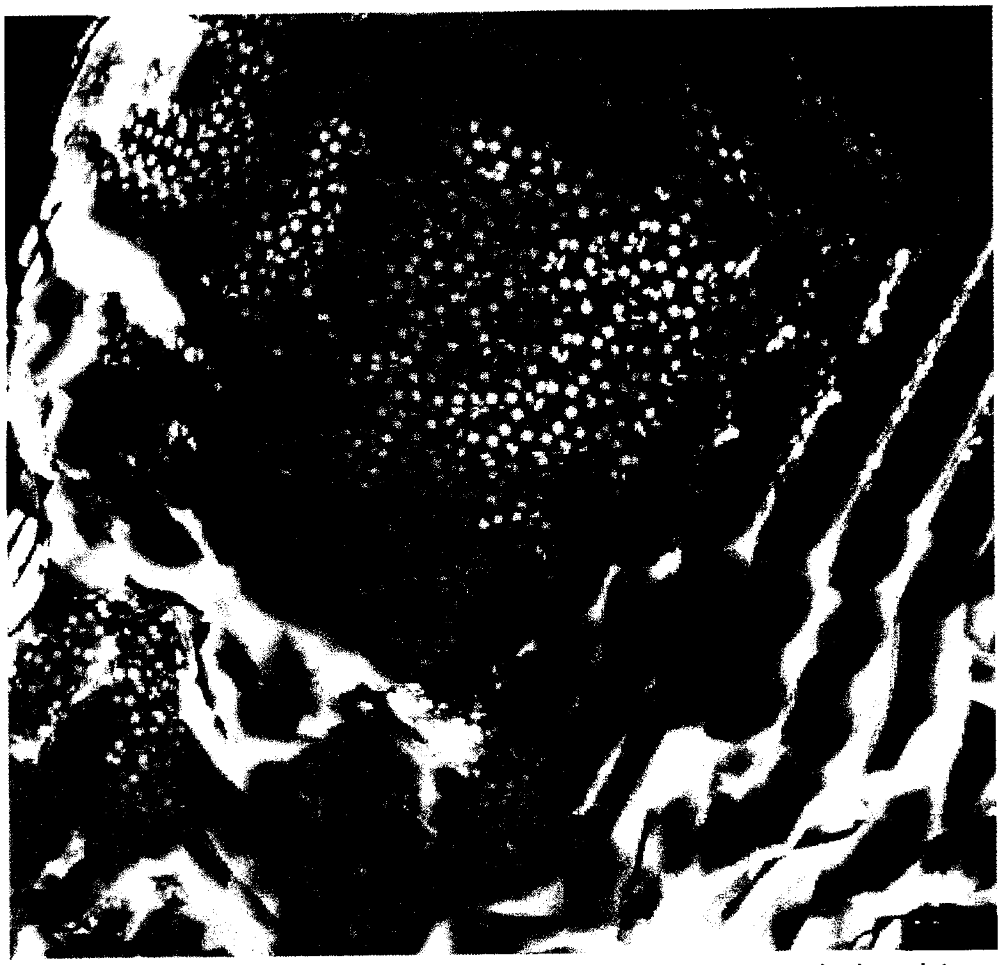


Figure E.12. Error mapped to opacity over a hue mapped to a synthetic variate

BIBLIOGRAPHY

[AHRE95] James P. Ahrens, Charles D. Hansen, "Cost-Effective Data-Parallel Load Balancing", *Technical Report TR-95-04-02*, University of Washington, Seattle, 1995

[Albers98] S. Albers, N. Garg and S. Leonardi, Minimizing Stall Time in Single and Parallel Disk Systems, *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pp. 454-462, 1998

[Amer02] A. Amer, D. Long, and R. Burns. Group-Based Management of Distributed File Caches. In *Proceedings of the 17th International Conference on Distributed Computing Systems,* 2002

[BABC94] C. Babcock, "Parallel Processing Mines Retail Data", *Computer World*, 1994

[Bhaniramka02] P. Bhaniramka, Y. Demange, OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data Sets, *Proc. Volume Visualization and Graphics Symposium 2002.*

[BERG89] R. Daniel Bergeron, Georges G. Grinstein, A Reference Model for the Visualization of Multi-Dimensional Data, *Eurographics '89*, Elsevier Science Publishers, North Holland, 1989

[BERG93] R. Daniel Bergeron, William Cody, William Hibbard, David T. Kao, Kristina D. Miceli, Lloyd A. Treinish, Sandra Walther, "Database Issues for Data Visualization: Developing a Data Model", *Database Issues for Data Visualization, Proceedings of the IEEE Visualization '93 Workshop (LNCS 871),* Springer, Berlin, 1993

[Bhanarimka02]. P. Bhaniramka, Y. Demange, OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data Sets, *Proc. Volume Visualization and Graphics Symposium 2002.*

[BICK96] Barbara Bicking, Russell East, "Towards Dynamically Integrating Spatial Data And Its Metadata", *First IEEE Metadata Conference*, IEEE, Los Alamitos, CA, 1996

[Boada01]Boada, I., I. Navazo, and R. Scopigno, "Multiresolution Volume Visualization with a texture-based octree", *The Visual Computer (2001),* 17:185-197, Springer-Verlag 2001

[Cabral94] Cabral, B., N. Cam, and J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware", *Proceedings of the 1994 Symposium on Volume Visualization*, pp. 91-98

[Cao96] P. Cao and E. Felten, Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling, *ACM Transactions on Computer Systems, vol. 14, No. 4,* 1996

[CATHRO97] Warwick Cathro, "Metadata: An Overview", *Standards Australia Seminar*, National Library of Australia, 1997

[Cedi00] Cedilnik, A. and P. Rheingans, "Procedure Annotation of Uncertain Information", *Proceedings of IEEE Visualization '2000*, October 2000, pp. 77-83.

[CHAM96] Bradford Chamberlain, Tony DeRose, Dani Lischinski, David Salesin, John Snyder, "Fast Rendering of Complex Environments Using a Spatial Hierarchy", *Proceedings of Graphics Interface '96*, Toronto, 1996

[CChang00] Chialin Chang, Tahsin Kurc, Alan Sussman, Joel Saltz, Optimizing Retrieval and Processing of Multi-dimensional Scientific Datasets, In *Proceedings of the Third Merged IPPS/SPDP Symposiums*. IEEE Computer Society Press, May 2000

[CChangADR] Chialin Chang, Tahsin Kurc, Alan Sussman, Joel Saltz, *Active Data Repository Software User Manual*, http://www.cs.umd.edu/ projects/hpsl/ResearchAreas/ADR-dist/ADR.htm

[CDF] The CDF Webpage: http://nssdc.gsfc.nasa.gov/cdf/cdf_home.html

[Chang01] F. Chang, Using Speculative Execution to Automatically Hide I/O Latency, *Ph. D. Dissertation*, Carnegie Mellon University, 2001

[CIGNO94] Paolo Cignoni, Leila De Floriani, Claudio Montani, Enrico Puppo, and Roberto Scopigno, "Multiresolution Modeling and Visualization of Volume Data Based on Simplicial Complexes", *Proceedings ACM Symposium on Volume Visualization'94*, ACM, Washington, DC, 1994

[CIGNO97] Paolo Cignoni, Claudio Montani, Enrico Puppo, Roberto Scopigno, "Multiresolution Representation and Visualization of Volume Data", *IEEE Transactions on Visualization and Computer Graphics*, Volume 3, No. 4, IEEE, Los Alamitos, CA, 1997

[CIGNO98] P. Cignoni, C. Rocchini and R. Scopigno, "Metro: Measuring Error on Simplified Surfaces", *Computer Graphics Forum*, Vol. 17, No. 2, Blackwell Publishers, Oxford, UK, 1998

[CLEARY96] John Cleary, Geoffrey Holmes, Sally Jo Cunningham , and Ian H. Witten, "MetaData for Database Mining", *http://www.computer.org/conferen/meta96/holmes/DataBaseMining.html*, IEEE, Los Alamitos, CA, 1996

[Coughlin] Coughlin, Thomas, High Density Hard Disk Drive Trends in the USA, tech report at http://www.tomcoughlin.com /techpapers.htm

[DEBE98] Mark de Berg, Katrin T. G. Dobrindt, "On Levels of Detail in Terrains", *Graphical Models and Image Processing* 60:1--12, Academic Press, San Diego, CA, 1998

[Ellis04] Ellis, Lorna, CDF and Granite: A Comparison, Technical Report, Department of Computer Science, University of New Hampshire, 2004

[ESTE97] Martin Ester, Hans-Peter Kriegel, Jörg Sander, "Spatial Data Mining: A Database Approach", *Advances in Spatial Databases, Proceedings of 5th Int'l Symposium SSD LNCS 1262*, Springer , Berlin, 1997

[FAYYA96] Usama M. Fayyad, Gregory Pietetsky-Shapiro, Padhraic Smyth, "From Data Mining to Knowledge Discovery: An Overview (Chapter 1)", *Advances in Knowledge Discovery and Data Mining* (book), AAAI Press/MIT Press, Menlo Park, 1996

[FDW96] Usama M. Fayyad, S. George Djorgovski, and Nicholas Weir, "Automating the Analysis and Cataloging of Sky Surveys", *Advances in Knowledge Discovery and Data Mining* (book), AAAI Press/MIT Press, Menlo Park, 1996

[FOOTE95] Kenneth E. Foote, Donald J. Huebner, "Error, Accuracy, and Precision", *http://wwwhost.cc.utexas.edu/ftp/pub/grg/gcraft/notes/error/error.html*, University of Texas, Austin, 1995

[Forney02] Brian C. Forney, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Storage-Aware Caching: Revisiting Caching for Heterogeneous Storage Systems, *First USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey, CA, USA, January 2002.

[GAEDE98] Volker Gaede, Oliver Günther, "Multidimensional Access Methods", *ACM Computing Surveys*, Vol. 30, No. 2, ACM, New York, 1998

[Gl4java] gl4java home page, http://www.jausoft.com/gl4java.html

[Griffioen94] J. Griffioen and R. Appleton, Reducing File System Latency Using A Predictive Approach, *University of Kentucky Technical Report #CS247-94*

[Guan94] Guan, S., R. Lipes, *"Innovative Volume Rendering Using 3D Texture Mapping"*, Proceedings Medical Imaging, vol. 2164, 1994, pp. 382-392

[HANS93] Charles Hansen, Stephen Tenbrink, "The Impact of Gigabit Network Research on Scientific Visualization", *The Visual Computer*, Vol. 9, No. 6, 1993

[HART94] Ami Harten, "Multiresolution Representation and Numerical Algorithms: A Brief Review", *NASA Contractor Report 194949*, ICASE, Hampton, VA, 1994

[HECK97] Paul S. Heckbert, Michael Garland, "Survey of Polygonal Surface Simplification Algorithms", *Technical Report at http://www.cs.cmu.edu/~ph*, Carnegie Mellon University, Pittsburgh, 1997

[Hibb92] Hibbard, W.L., C.R. Dyer, and B.E.Paul, "Display of Scientific Data Structures for Algorithm Visualization", *Proc. of IEEE Visualization '92*, October 1992, IEEE Computer Society Press

[HIBB94] W.L. Hibbard, C.R. Dyer, and B.E.Paul, "A Lattice Model for Data Display", *Proceedings of IEEE Visualization '94*, IEEE, Washington, DC, 1994

[HIBB95] W.L. Hibbard, D.T. Kao, and A. Wierse, "Database Issues for Data Visualization: Scientific Data Modeling", Database Issues for Data Visualization, *Proc. IEEE Visualization '95 Workshop*, LNCS 1183, Springer, Berlin, 1995

[Highley02] T. Highley, P. Reynolds and V. Vellanki, Absolute Cost-Benefit Analysis for Predictive File Prefetching, *University of Virginia Technical Report #CS200211*

[Highley03] T. Highley and P. Reynolds, Marginal Cost-Benefit Analysis for Predictive File Prefetching, *Proceedings of the 41st Anuual ACM Southeast Conference (ACMSE 2003)*, Savannah, GA

[HOPPE96] H. Hoppe, "Progressive Meshes", *ACM Computer Graphics Proceedings, Annual Conference Series, (SIGGRAPH '96),* IEEE, Los Alamitos, CA, 1996

[Hunt93] Hunter, G. and M. Goodchild, "Managing Uncertainty in Spatial Databases: Putting Theory into Practice", *URISA J.*, v. 5, no. 2 (1993), pp. 55-62.

[Jiang02] Jiang, Feng, Implementation and Performance Evaluation of Block-Oriented Adaptive Resolution for Multisource Scientific Datasets, *Master's Thesis, Department of Computer Science,* University of New Hamphsire, 2002

[JUFFS98] Philip Juffs, Edwin Beggs, Farzin Deravi, "A Multiresolution Distance Measure for Images", *IEEE Signal Processing Letters*, Volume 5, No. 6, IEEE Computer Society Press, Los Alamitos, CA, 1998

[KAO93] David T. Kao, R. Daniel Bergeron, Ted M. Sparr, "An Extended Schema Model for Scientific Data", *Database Issues for Data Visualization, Proceedings of the IEEE Visualization '93 Workshop (LNCS 871)*, Springer, Berlin, 1993

[KAO97] David T. Kao, *A Metric-Based Scientific Data Model for Knowledge Discovery*, Ph.D. Thesis, University of New Hampshire, Durham, 1997

[KAPET95] Epaminondas Kapetanios, Ralf Kramer, "A Knowledge-Based System Approach for Scientific Data Analysis and the Notion of Metadata", *Proceedings of the 14th IEEE Symposium on Mass Storage Systems*, IEEE Computer Society Press, Los Alamitos, CA, 1995

[LaMar99] La Mar, Eric., Berndt. Hamann, Kenneth. Joy, "Multiresolution Techniques for Interactive Texture-Based Volume Visualization", Proceedings IEEE Visualization, 1999, pp. 355-362

[Laramee02] R.S. Laramee and R. D.Bergeron. "An IsoSurface Continuity Algorithm for Super Adaptive Resolution Data", *Advances in Modelling, Animation, and Rendering: Computer Graphics International (CGI) 2002 Conference Proc.*, John Vince and Rae Earnshaw editors, pages 215-237, 3-5 July 2002, Bradford, UK

[LATTU95] Roberto Lattuada, Jonathan Raper, "Applications of 3D Delaunay Triangulation Algorithms in Geoscientific Modelling", *Presented at GISRUK '95 (http://www.iah.bbsrc.ac.uk/gis/gisruk95.ps)*, 1995

[LEIGH92] F. Thomson Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, CA, 1992

[LI98] Zuotao Li, Sean Wang, Menas Kafatos, Ruixin Yang, "A Pyramid Data Model for Supporting Content-based Browsing and Knowledge Discovery", *Proceedings of Tenth International Conference on Scientific and Statistical Databases*, IEEE Computer Society Press, Los Alamitos, CA, 1998

[LIN95] Chengjiang Lin, Sanli Li, "Strategy and Simulation of Adaptive RID for Distributed Dynamic Load Balancing in Parallel Systems", *Chinese Journal of Computers*, Vol.18, No.10, 1995

[Lodh96a] Lodha, S.K., C.M. Wilson and R.E. Sheehan, "LISTEN: Sounding Uncertainty Visualization", *Proc. IEEE Visualization '96*, 1996, pp. 189-195.

[Lodh96b] Lodha, S.K., A. Pang, R.E. Sheehan and C.M. Wittenbrink, "UFLOW: Visualizing Uncertainty in Fluid Flow", *Proc. IEEE Visualization '96*, IEEE Computer Society Press, 1996, pp. 249-254.

[LOUNS97] Michael Lounsbery, Tony DeRose, Joe Warren, "Multiresolution Analysis for Surfaces of Arbitrary Topological Type", *ACM Transactions on Graphics*, Volume 16, No. 1, ACM, New York, 1997

[Madhyastha96] Madhyastha, T. M., Elford, C. L., and Reed, D. A, Optimizing Input/Output Using Adaptive File System Policies, In *Fifth NASA Goddard Conference on Mass Storage Systems and Technologies*, September 1996

[Madhyastha97] Madhyastha, T. M., and Reed, D. A, Input/Output Access Pattern Classification Using Hidden Markov Models, In *Workshop on Input/Output in Parallel and Distributed Systems*, November 1997, pp. 57-67.

[Mitchell02] Design and Prototype of a Metadata Model for Multsource Scientific Datasources, *Master's Thesis, Department of Computer Science,* University of New Hamphsire, 2002

[More00] Sachin More, Alok Choudhary, Tertiary Storage Organization for Large Multidimensional Datasets, *8th NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies and 17th IEEE Symposium on Mass Storage Systems*, 2000

[Nadeau] David R. Nadeau, An Architecture for Large Multi-Dimensional Data Management, *Scalable Visualization Tools White Paper*, http://vistools.npaci.edu/

[NAKANO97] Aiichiro Nakano, Timothy Campbell, "An adaptive curvilinear-coordinate approach to dynamic load balancing of parallel multiresolution molecular dynamics", *Parallel Computing*, Vol. 23, pp. 1461-1478, Elsevier, North Holland, 1997

[NIHVH] Website at http://www.nlm.nih.gov/research/visible/visible_human.html

[OLAP] "OLAP Glossary", *http://www.olapcouncil.org/research/glossaryly.htm*, The OLAP Council

[Pang94] Pang, A., J. Furman and W. Nuss, "Data Quality Issues in Visualization", *SPIE Vol. 2178: Visual Data Exploration and Analysis*, Feb. 1994

[Patterson95] Patterson, R.H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J., Informed Prefetching and Caching. In *Proceedings of the 15th* ACM Symposium on Operating Systems Principles, December 1995, PP. 79-95.

[PFALTZ98] John L. Pfaltz, Russell F. Haddleton, James C. French, "Scalable, Parallel, Scientific Databases", *Proceedings 10th International Conference on Scientific and Statistical Database Management*, IEEE, Los Alamitos, CA, 1998

[RESN98] Richard J. Resnick, Matthew O. Ward, and Elke A. Rundensteiner, "FED—A Framework for Iterative Data Selection in Exploratory Visualization", *Proceedings of Tenth International Conference on Scientific and Statistical Databases*, IEEE Computer Society Press, Los Alamitos, CA, 1998

[Sarawagi94] S. Sarawagi, M. Stonebraker, Efficient Organizations of Large Multidimensional Arrays, *Proceedings of the Tenth International Conference on Data Engineering*, February 1994

[SCVT] Introduction to Scientfic Visualization Tools", *http://scv.bu.edu/SCV/Tutorials/SciVis/input_data.html*, Boston University Scientific Computing and Visualization Group, Boston, 1998

[SEAL98] W. Brent Seales, Cheng J. Yuan, Wei Hu, Matthew D. Cutts, "Object Recognition in Compressed Imagery", *Image and Vision Computing* 16(1998), Elsevier, North-Holland, 1998

[SHEN96] Wei-Min Shen, Bing Leng, "A Metapattern-Based Automated Discovery Loop for Integrated Data Mining—Unsupervised Learning of Relational Patterns", *IEEE Transactions on Knowledge and Data Engineering,* Volume 8, No. 6, IEEE Computer Society Press, Los Alamitos, CA, 1996

[Shen98] Shen, Q. and A.Pang, "Data Level Comparison of Wind Tunnel and Computational Fluid Dynamics Data", *Proc. IEEE Visualization '98*, IEEE Computer Society Press, 1998, pp. 415-418.

[SHRAG90] J. Shrager, P. Langley (eds.), *Computational Models of Scientific Discovery and Theory Formation,* Morgan Kaufmann, San Francisco, 1990

[SIMHAD98] Kiran K. Simhadri, S.S. Iyengar, Ronald J. Holyer, Matthew Lybanon, John M. Zachary, "Wavelet-Based Feature Extraction from Oceanographic Images", *IEEE Transactions on Geoscience and Remote Sensing,* Voume 36, No. 3, IEEE Computer Society Press, Los Alamitos, CA, 1998

[Spar94] Sparr, T. M., R. Daniel Bergeron, L. D. Meeker, "A Visualization-Based Model for a Scientific Database System", *Focus on Scientific Visualization*, Hagen, Muller and Nielson (eds.), Springer, 1994.

[SPER90] D. Speray, S. Kennon, "Volume Probes: Interactive Data Exploration on Arbitrary Grids", *Computer Graphics*, Vol. 24, No. 5, ACM, 1990

[STOLL96] Eric J. Stollnitz, Tony D. DeRose, David H. Salesin, *Wavelets for Computer Graphics: Theory and Applications*, Morgan Kaufmann Publishers, Inc. , San Francisco, CA, 1996

[TSICH77] D.C. Tsichritzis, F.H. Lochovsky, *Data Base Management Systems*, Academic Press, 1977

[Vellanki99] V. Vellanki and A. Chervenak, A Cost-Benefit Scheme for High Performance Predictive Prefetching, *Proceedings of Supercomputing '99*, November 1999

[VisAD] VisAD home page, http://www.ssec.wisc.edu/~billh/visad.html.

[Westover90] K. Westover, Footprint Evaluation for Volume Rendering, *Computer Graphics, vol. 24*, 1990, pp. 367-376

[Witt96] Wittenbrink, C.M., A.T. Pang, and S.K. Lodha, “Glyphs for visualizing uncertainty in vector fields”, *IEEE Trans. Vis. Comp. Gr.*, v. 2, no. 3 (Sep 1996), pp. 266-279.

[WONG95] Pak Chung Wong, R. Daniel Bergeron, "Authenticity Analysis of Wavelet Approximations in Visualization", *Proceedings of IEEE Visualization ‘95*, IEEE Computer Society Press, Los Alamitos, CA, 1995

[Wong00] Wong, P.C. and R.D. Bergeron, “Performance Evaluation of Multiresolution Isosurface Rendering”, *Proc. Dagstuhl '97 Scientific Visualization*, IEEE Computer Society Press, pp. 332-341, 2000.

[Ye04] Ye, Li, Representation of Adaptive Resolution Rectilinear Cell Data, *Master’s Thesis, Department of Computer Science,* University of New Hamphsire, 2004