Database Support for Multisource Multiresolution Scientific Data

Philip J. Rhodes, R. Daniel Bergeron, and Ted M. Sparr

University of New Hampshire, Durham NH 03824, USA
{rhodes, rdb, tms}@cs.unh.edu
http://www.cs.unh.edu/~{rhodes, rdb, tms}

Abstract. We extend database technology to provide more meaningful support for exploration of scientific data. We have developed a new data model that incorporates spatial semantics with localized error and are implementing a prototype database system based on the model. Our data model and system focus on support for retrieval and visualization of gridded scientific data at multiple resolutions. While these semantics may not apply naturally to every scientific application, they are common to many. This paper summarizes the data model and describes the key functionality of our prototype system.

1 Introduction

Database support has not extended into scientific applications (broadly construed to mean having a large quantitative component). Much scientific data includes complex semantics in its structure that existing data models do not represent effectively. Thus, scientific applications have to supply most of the semantics while conventional database systems add overhead but relatively little value compared to file systems. Our work focuses on representing spatial semantics and using it to support multiresolution representations of the data.

Increasingly, scientific datasets are too large to be completely examined in detail, even by automated filtering techniques. Multiresolution support enables scientists to examine less detailed, more abstract summary views, then extract the finer, more concrete details for the most interesting areas. These large datasets can be more effectively explored for insights if they can be represented in multiple resolutions with local error information available for the coarser representations.

We envision pre-computed data archives that represent scientific data with generic spatial semantics. The base layer is the original data; the upper layers have increasingly more compact, coarser representations, each with localized error information. A scientist downloads a coarse layer for initial examination, typically the smallest layer containing acceptable error. The examination identifies interesting areas for more detailed study. The scientist then downloads data in these areas at the appropriate detail (i.e., meeting authenticity requirements)

[1] This work is supported by the National Science Foundation under grants IIS-0082577 and IIS-9871859

from the multiresolution hierarchy in the archive. For convenience, the scientist might move portions of frequently accessed coarser layers to a local server.

Our model includes three principal components — *lattices, data sources,* and *multiresolution data hierarchies.* The *lattice* presents the scientist with a view of a gridded dataset at uniform or varying resolution. The structure may be a regular rectilinear grid, a more general rectilinear grid, a curvilinear grid, or an unstructured grid [13]. The *data source* maps lattice data onto an n-dimensional array-shaped computational space and then to one or more 1-dimensional linear data streams. A *multiresolution hierarchy* is a stack of lattices that view the same data with different resolutions. We elaborate these concepts in sections 2 and 3, which summarize a previous paper [11].

The principal focus of this paper is to describe the design and implementation of a Java-based system that supports the formal data model. We present the key components of both the data source and lattice implementations. We also present some examples of how a scientist might define and access a composite multisource data set using a variety of common topologies and geometries.

2 Lattice Model

We consider that scientific data represents some phenomenon [4] that is a function over a domain [8]. The Cartesian product of the value ranges of the attributes defines the value space of the function on the domain. Scientific data has large size, complex entities and relationships, and volumetric semantics [10].

2.1 Dimensional and Spatial Data

Much scientific data can be meaningfully represented in an n-dimensional space [1, 5] where attribute values are defined on continuous value ranges. Such an attribute is termed *dimensional* and a data set with at least some dimensional attributes is a dimensional data set. The term *spatial data* applies to dimensional data from domains that are actual physical spaces. However, much dimensional but non-spatial data can still be treated as if it were spatial. This approach is useful because people find this representation natural.

Spatial data is often represented as points and or cells defined on a regular or irregular grid. The choice of grid impacts the nature of the representation chosen for the data and the specification of algorithms for analyzing it.

2.2 Lattice

A domain can be thought of as a hyper-volume from which data are collected. Our *lattice model* presents a view of a gridded dataset that represents data sampled from a domain. The view may have uniform or varying resolution. The terminology used in the literature to describe various grid systems is not standardized. We define a consistent framework for describing grids that encompasses most reported grid structures [7, 8].

2.3 Geometry, Topology, and Neighborhoods

The lattice separately represents the underlying space in which the grid is defined, which we call the *geometry*, and the point and cell relationships implied by the grid, which we call the *topology*. The geometry refers to the dimensional domain and the locations of data within it; it describes the "shape" of the grid. The topology defines how the points of the grid are connected. The topology is a graph with data points or cells as nodes and edges defining adjacency and neighborhood relationships. This approach enables database support for applications to process data either geometrically or topologically.

We can represent all the conventional commonly used grids [13] including regular rectilinear grids, more general rectilinear grids, unstructured grids, and curvilinear grids. We also support lattices whose resolution varies over the domain. Such lattices include localized error information that encodes the accuracy of the representation so users can make informed decisions about resolution. Additionally, we combine our approaches to unstructured and structured grid functionality in one representation to handle data that is structured in some dimensions and unstructured in others similar to the notion of fiber bundles [3].

2.4 Multiresolution Data

A *multiresolution hierarchy* (MR) is a stack of lattices viewing the same hypervolume 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 based on 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*.

2.5 Adaptive Resolution

An adaptive resolution (AR) 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.

3 Data Source Model

Each lattice maps its geometry and topology specifications onto an n-dimensional array, which forms the *computational space* of the data set. A *data source* represents this array-structured view of the data set.

For some grids, especially regular grids, the topology and/or geometry do not need an explicit representation because they derive easily from the indexes of the array that stores the data points. Other more complex geometries and topologies may have a separate representation from their computational spaces. Typically, processing the data using this indexed computational space is much more efficient than using the data in its lattice form. Moreover, our approach introduces little overhead when the lattice grid structure maps directly to the data source array organization.

A data source also maps the n-dimensional computation space to the data as it appears in files (or URLs). We define several kinds of data sources.

- A *physical data source* corresponds to one file/URL and maps an array shaped grid structure onto a one-dimensional data stream.
- An *attribute-join data source* integrates representations where different data attributes are stored in separate parallel files of congruent shape.
- A *blocked data source* integrates representations where the domain is spatially partitioned into contiguous non-overlapping components.
- A *variable resolution data source* supports views of data at different uniform resolutions.

In our data source model each lattice has a *root data source* that defines an indexable n-dimensional space. The root may be a physical data source when the lattice view of the data matches the file storage organization. The root may also be defined by composite data sources, recursively defining a hierarchy with physical data sources at the leaves. Each interior level of the hierarchy knows how to map between its index space and the index spaces of its components.

4 Data Source Implementation

In this section we provide insight into our data source implementation. We begin with a summary of some overriding design criteria, which were developed based on extensive performance evaluations of a preliminary implementation. We then describe the primary components of our current implementation that more effectively incorporates these guidelines. We focus on the principal classes, *DataSource*, *Datum*, and *DataBlock* along with some of their children classes.

4.1 Design guidelines

Since *DataSource* objects are the principal interface to the actual scientific data, it is particularly important that their implementation be as efficient as possible. Our implementation design was driven by several key guidelines:

- avoid object creation since it is rather expensive;
- minimize the number of physical reads to a file;
- use *lazy* evaluation to reduce the space used for data at run-time;
- minimize the number of times data values are copied;
- make access to individual data values as efficient as possible for the application analysis and graphics routines.

A key consequence of the guidelines is that the *DataSource* object does not directly store the data values that it *appears* to contain. In other words, a *DataSource* object represents an extremely large data file, but the data values in the file are not extracted until the application requests them. (As discussed in section 4.4, pre-fetching and caching violate this principle in the interest of better performance.) Application code can access individual data points (via *datum* methods and *Datum* objects) or blocks of data (via *subblock* methods and *DataBlock* objects). Upon execution of a *datum* or *subblock* method call, appropriate *read* requests are formed, the data is read from its file and stored in a *Datum* or *DataBlock* object that is returned to the caller.

4.2 Datums and Attribute Access

An object of type *Datum* is located at each valid indexable position of a *DataSource* and stores the values of all the attributes of the *DataSource* at that position. For example, assume that a *DataSource*, *ds*, is a 2D data source with the attributes *carbon, nitrogen,* and *hydrogen*, defined as three *float* values.

The user's application can access the attribute values of a particular position in *ds* with code such as the following:

```
IndexSpaceId position = new IndexSpaceId( i, j );
Datum        values   = ds.getDatum( position );
float        carbon   = values.getFloat( "carbon" );
float        nitrogen = values.getFloat( "nitrogen" );
float        hydrogen = values.getFloat( "hydrogen" );
```

Accessing individual fields of a *Datum* object one at a time provides a flexible implementation model, but it imposes significant overhead with very large data sets. More efficient access can be achieved using block access.

4.3 *DataBlock*s

Rather than retrieving one *Datum* object at a time from the *DataSource*, application code can request retrieval of arbitrarily sized blocks of data via *subblock* method calls. This technique provides significantly more efficient access

by extracting many data values at the same time. In addition, the extracted data can be formatted in a way to provide more efficient access to the data by application-level code. Data extracted from a *DataSource* via a *subblock* method is encapsulated in a *DataBlock* object. *DataBlock* is an abstract class whose derived classes are designed to provide efficient support for particular data formats as described in the following.

Point vs. Attribute Ordering. Our conceptual data model for both lattices and data sources treats a sample point as if it contains a *Datum* object that encapsulates all the attribute values associated with the sample point. We call this approach to organizing the data *point order* since the data values are grouped by the sample points. For real data it is often convenient to organize the data in *attribute order*, in which all of the values for a particular attribute are grouped together. Given a data source with n sample points and k attributes, a point-ordering of this data might consist of an n-element array of groups of data values containing k attributes, whereas an attribute-ordering might consist of k arrays of n elements where each array contains all the data values for a particular attribute. Even though our data model is based on point ordering, our underlying implementation allows the data to be stored either in point order or attribute order. The application code can specify a desired data format for the data in a *DataBlock* by instantiating appropriate children of the *DataBlock* class. It is also possible to allow the system to create *DataBlock* objects that match the organization of the data as it is stored on disk.

CompactBlock. A *CompactBlock* object stores all its data in a single Java array whose type can be *float, int, double,* or *byte*. Given this constraint, a *CompactBlock* object can store the following formats:

- point order data if all attributes are of the same type;
- attribute order data of all the same type where all the values for each attribute are stored together in the array;
- point order data of different types but stored as a *byte* array – this option requires conversion of each attribute from *byte* to its correct type every time the data value is accessed;
- attribute order data of different types, stored by attribute in a *byte* array – this requires the same conversion at every access to each data value.

In addition to the data array, a *CompactBlock* contains information for each attribute including its data *type*, an *offset* into the data array indicating the position of the first instance of the data attribute, and a *stride* field that defines the number of entries to skip to get to the next value for the attribute. Fig. 1 shows the internal structure of a *CompactBlock* for a data block with 3 data points and 3 *float* attributes called *C, N,* and *H*. Fig. 1(a) uses point order, while Fig. 1(b) uses attribute order. In both formats, the i-th instance of any attribute is located at `dataArray[ offset + i*stride ]`.

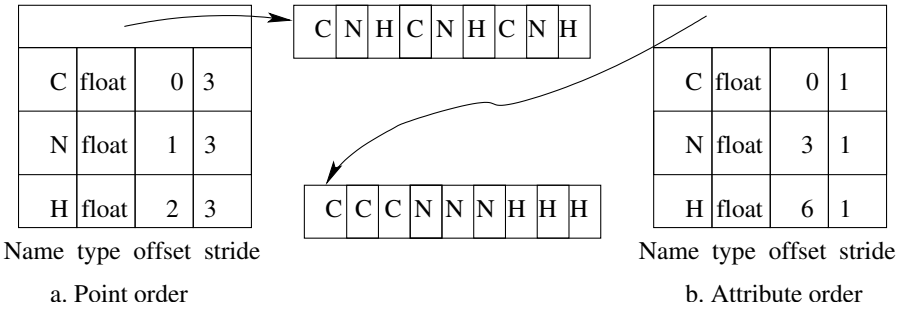

Fig. 1. *CompactBlock* internal structure

This structure also provides a very efficient framework for storing data attributes that apply to an entire block. For example, suppose that a data block represents a series of data samples all of which have the same *time* value. The *time* can be stored as the first data value in the data array and included in the attribute table with an *offset* of 0 and a *stride* of 0. Thus, the *time* value is stored only once, but it appears as if it is stored once for each data point.

CompositeBlock. Although *CompactBlock* provides efficient storage for several useful data formats, its scope is somewhat limited. However, the combination of two or more *CompactBlock*s can be used to represent a very large range of data formats efficiently. The *CompositeBlock* class encapsulates multiple *CompactBlock*s into a single conceptual data block. Particularly useful formats include attribute ordering where each attribute is in a separate array and mixed type blocks where all the attributes of a specific type are grouped into one of the *CompactBlock* components. There is a small amount of additional overhead needed to pass requests through the *CompositeBlock* to its component *CompactBlock*s, but this overhead is usually more than offset by the added efficiency for accessing the data arrays.

Direct Data Access. A particularly important advantage of the *CompactBlock*/*CompositeBlock* implementation is that the actual data values can be made accessible as simple Java *double, float* or *int* arrays. This allows application analysis and/or graphics functions to access the raw data without going through multiple levels of objects. The *DataBlock* class hierarchy also provides methods that allow user code to convert between different *DataBlock* formats. This is especially valuable when the application uses other analysis or graphics packages that expect large sets of data values as simple arrays.

4.4 *PhysicalDataSource*

PhysicalDataSource objects provide the direct interface to a data file or a network source for the data. A *PhysicalDataSource* object is instantiated with a reference

to a specific data file and is responsible for knowing the exact format of the data, including dimensionality, bounds, attribute types and ordering, byte ordering, etc. This information can be stored and retrieved from a relational database interface that is part of the system, or it can be defined by an XML-based file descriptor that can be read by a system utility class, *FDLReader*.

4.5 *CacheDataSource*

One of the principal mechanisms for reducing I/O is the *CacheDataSource*, a child of the *DataSource* abstract class. An instance of *CacheDataSource* can be associated with a *DataSource* of any type and supports both *read-ahead* and a *cache* of recently read *Datum*s for its associated *DataSource*. Any *datum* read request to a *CacheDataSource* is transformed to a *subblock* request of the associated *DataSource*. After the subblock is returned, the *CacheDataSource* saves the subblock in its cache memory then extracts the requested data and returns it in a *Datum* object containing the data. For subsequent *datum* method invocations, the *CacheDataSource* first checks if the desired data already has been read and is stored in its cache. If so, it returns the data from the cache; if not, it reads a new subblock, caches it, and then returns the desired data values.

Performance analysis has shown that this approach can yield a significant performance improvement when the application program performs a sequential read through the data source, but it is not effective with random access [6]. In general, the improvement is not very sensitive to typical parameters, such as cache data size and the exact nature of the sequential access patterns.

4.6 *AttributeJoinDataSource*

The *AttributeJoinDataSource (AJDS)* allows a user to treat data from multiple files each containing a single attribute as if the data were stored as a single multi-attribute set of data. A simple attribute join takes multiple data sources with compatible index spaces and creates a new data source that has all the attributes of all the component data sources. Parameter options let the user select attribute subsets and rename and reorder the attributes in the new data source. From the application code perspective, the new data source appears the same as if the multiple attributes are stored in a single file.

Performance analysis of the *AJDS* implementation [9] shows very little if any additional overhead caused by the introduction of the composite data source. Access to n separate data sources to get one attribute each is comparable in cost to accessing a single n-attribute *AttributeJoinDataSource* with n components of one attribute each.

4.7 *BlockedDataSource*

Multi-source data sets can also be created from individual files based on spatial attributes. For example, there may be a set of data files that represent satellite

images covering a particular range of latitude and longitude. It might be convenient for an application to create a data source that combines 2 or more such files. A *BlockedDataSource* supports the creation of a data source composed of other data sources that are placed into non-overlapping portions of the index space of the composite. In its simplest form the components of a *BlockedDataSource* are defined with their own index space ranges that do not overlap. For example, the index ranges could be defined to coincide with the latitudes and longitudes represented by the data. Options to the *BlockedDataSource* constructor, however, can be used to map the index space ranges of the components to arbitrary positions in the composite's index space.

Performance analysis of the *BlockedDataSource* implementation [6] shows that the introduction of the composite blocked data source adds little or no overhead if the composite data source is blocked in a regular fashion. In this case, access to n separate data sources to get a datum or a subblock is comparable to computing the position of the datum or subblock from a set of separate data sources and then accessing the data from the correct data source. A *BlockedDataSource* with blocks of irregular size and shape does introduce additional overhead for the search for the correct block. In both cases the *BlockedDataSource* is far more convenient for the application, since it does all the mapping which otherwise would be required by user-level code.

4.8 *DataSource* Trees

Users can create data sources from multiple levels of attribute join and blocking operations and combinations of attribute join and blocking operations.

Performance analysis on data sources defined by multiple levels of attribute join operations indicate very little overhead caused by the nested definition [9]. Complex trees that combine multiple levels of attribute joins and blocking do create additional mapping overhead, but similar functionality implemented directly in user code would require equivalent processing [6, 9].

4.9 Variable Resolution Data Sources

Normally, data sources are defined with a *fixed* resolution that is uniform throughout the index space of the data source. A *VariableResolutionDataSource* (*VRDS*) allows a data source to be accessed at different resolutions. A *VRDS* is a *wrapper* class for another data source object, called the *subject*. The *VRDS* has a *base resolution* that matches the resolution of the subject data source. Initially, the resolution of the *VRDS* is the same as its base resolution, but user code can make the resolution finer or coarser with a simple mode setting method invocation.

When the *VRDS* resolution is finer than its subject data source, the *VRDS* extracts the coarse data from its subject and uses that data to generate a higher resolution representation of it. The higher resolution representation can be created by either *replicating* the existing points, or by *interpolating* within them.

When the *VRDS* resolution is coarser than its subject, the *VRDS* extracts the higher resolution representation of its subject and converts it to a coarser

form. This can be achieved by either *selection* of a subset of the subject's data, or by (weighted) *averaging* of the finer resolution data.

Performance evaluation of the *VRDS* implementation was generally very good [6]. The cost of both *datum* and *subblock* access depends primarily on the amount of data requested from the *VRDS*. Thus, accessing a 4096x4096 data source through a *VRDS* as if it were 256x256 is only a bit more expensive than accessing a 256x256 conventional data source.

4.10 Adaptive Resolution Data at the Data Source Level

The *VRDS* class provides a mechanism for varying the resolution of a data source, but the entire data source still has the same uniform resolution at any given time. This can be extremely convenient, but it does not take advantage of situations where we don't need the same resolution throughout a data source. Although most of our adaptive resolution functionality is intended to be associated with lattices, we have implemented some basic functionality at the data source level by means of the *BlockedDataSource* and *ARBlock* classes.

Using spatial joins, a user can easily create a composite *BlockedDataSource* whose components do *not* have the same resolutions. User code can then access the data in this composite as if it were of uniform resolution just as is done with a *VRDS*. The user also has the option, however, of extracting uniform resolution subblocks one at a time from the composite. The subblocks are of type *ARBlock* which provides access to the original components at their original resolution. User code must know about the different resolution levels and how to combine neighboring results to achieve a complete solution.

5 Lattice Implementation

Our prototype implementation of the lattice model is capable of handling both regular data and some kinds of unstructured data. The *Lattice* class has three important data members: *Geometry*, *Topology*, and *DataSource*. The *Geometry* and *Topology* classes have children that are specialized for handling different kinds of data grids. By choosing the appropriate *Geometry* and *Topology*, the *Lattice* is able to handle different data types in much the same way. The *Lattice* also contains a *root data source* that serves as a portal to the underlying file(s).

The remainder of this section describes the principal lattice functionality, followed by a discussion of geometry and topology implementation issues and concludes with several lattice examples using different varieties of scientific data.

5.1 Lattice Functionality

The lattice must support several different methods for accessing the data. Perhaps the most obvious is to return a datum corresponding to a geometric point, p in the lattice domain. In this case, the geometry and topology must work together to efficiently map a geometric location to an index suitable for use with the root data source.

The data source contains only sample points, but the lattice must also be able to approximate values for any domain location. *Cells* can be very useful for deciding which sample points to use in the approximation. A cell is a collection of sample points connected by edges forming a perimeter. All the sample points in a cell lie on this perimeter. We consider only non-overlapping cell structures that completely cover the lattice domain. When the lattice is asked to compute an approximate value for some point p in the domain, it first finds the cell which encloses p, and then uses the sample points of this cell to compute an approximate value.

We can also ask the lattice to return a set of datums which lie within either a topological or geometric neighborhood. Legitimate queries include "retrieve all sample points within radius r of the point p", and "retrieve the k sample points nearest the point p". The first query requires knowledge of the geometry, while the second may or may not. For some geometries, e.g. regular rectilinear geometry, the second query requires geometry only to map p onto the index space of the data source. The nearest k points can then be determined directly from the index space without knowing their geometric positions. Another query can ask, "retrieve a c by c block of datums spanning the rectangle R". This query usually requires approximation unless c^2 sample points happen to align exactly with the region R.

Iteration is another important means of interacting with data. Instead of asking for a single datum or subblock, iteration specifies a sequence of data to be returned. To use iteration, we must first specify the range of the iteration and the nature of the items returned. Examples include "return the k nearest neighbors of a point p one after the other", "iterate over points or cells in a geometric region R", and "return sets of points corresponding to subregions of R". This latter example is particularly useful with computations like convolution, in which an operation is applied to whole blocks of data at a time.

Although iteration over geometric space is convenient and intuitive, it is more common for efficiency reasons to iterate over a *computational space*. This can be accomplished by mapping a geometric region to the corresponding (or a containing) computational space. This is particularly useful when the computational space is much simpler than the inherent geometric space of the data, or when the researcher does not need data values at non-sample points. Topological iteration can also provide more advanced functionality, as when the user wishes to retrieve topological neighbors of p in a breadth-first ordering, so that sample points within k edges of p are returned as a single set.

In summary, the lattice must support several different kinds of data access:

- *Datum* retrieval from sample points
- *Datum* retrieval from arbitrary domain locations
- Cell retrieval
- Topological neighborhood operations
- Geometric neighborhood operations
- Iteration over topology
- Iteration over geometry

5.2 Geometry Implementation

The *Geometry* object contains information necessary to map between a sample point location and its index in the underlying data source. For regular rectilinear grids, the geometry is largely implicit. In this case the step sizes within the grid are constant in each direction. Hence, there is a very simple mapping between points and their indexes, based only on the position of the origin of the grid and the step sizes in each dimension (section 5.5). A more general rectilinear grid (the perimeter grid) needs additional grid spacing information. In this case, there is an array of step sizes needed in each dimension (section 5.6). For curvilinear and unstructured grids, the geometry is usually represented by storing explicit position coordinates for each sample point (sections 5.7 and 5.9).

An important responsibility of the Geometry class is its ability to *partition* itself into regular rectilinear non-overlapping regions. The number and sizes of the partitions can be explicitly specified by the user, or can be determined by the system. Unless forbidden by the user, the partitioning can be dynamically altered by the system to adjust to changing conditions. This partitioning is not particularly useful for topologies that are already regular and rectilinear, but it is a key feature for providing efficient processing of curvilinear, semi-structured and unstructured grids as described below.

5.3 Topology and Cell Implementation

The primary responsibility of *Topology* is to represent the neighborhood relations between sample points. That is, for a sample point s, there are a set of points that are considered neighbors of s. In general, this information forms a graph structure with points at the vertices and edges connecting immediate neighbors. As with geometry, the topology implementation is simple for regular data because the straightforward mapping between geometry and the data source index space means that the topology needs no explicit representation; it is implicit in the grid. For example, a sample point in a rectilinear grid commonly has four neighbors in 2-D (*north, south, east, west*) and additional *front* and *back* neighbors in 3-D. For a request for the north neighbor for a point p with data source index *i,j*, the topology knows to return the value at index *i,j+1*. For unstructured data, the neighborhood relationship graph must be represented explicitly.

When the topology can be used as a basis for partitioning the domain into covering non-overlap regions, we say that the data set has a *cell* structure. The cell structure can improve search efficiency and provide a basis for approximating data values at arbitrary points in the domain.

For example, 2-D (3-D) unstructured data is often represented using a mesh of triangular (tetrahedral) cells. The vertices are stored in a vertex array and cells are defined as a set of vertex indexes stored in a cell array. For unstructured data, there is no simple mapping between the geometric region of a cell and its index, just as is the case with the sample points. For such data, it is difficult to locate a cell that corresponds to a geometric location. As the size of the data grows, exhaustive search quickly becomes impractical. By using the geometry's

partitioning, we can dramatically reduce the set of cells that must be examined. As in spatial partitioning for ray-tracing [2], each cell is distributed to its appropriate partition(s). We can then map a geometric query point p to the geometric partition it occupies and search only those cells that overlap that partition.

5.4 Rectilinear Grids

In this section, we discuss data for which the topology is a rectilinear grid, but with various geometries. For all these different data types, we need a mapping between the geometric and topological spaces. For some data, this mapping is very straightforward, while for others it is much more complicated. In any case, the key to working with such data efficiently is to take advantage of the rectilinear nature of the topology.

5.5 Regular Geometry

Fig. 2 shows data that has a rectilinear topology and a regular geometry. The geometry extends from (0.0, 0.0) to (40.0, 24.0) and the topology from (0,0) to (8,8). Suppose the lattice is asked for the data values corresponding to a geometric location p, as with the following code:

```
Datum d = lattice.datum( p );
```

In this example, there are two possibilities. If p does not correspond to the location of a sample point, then the lattice must compute an approximate value. To do so, it must first find the cell that surrounds p. This situation can be seen in the upper right of Fig. 2. For rectilinear data, we don't need to store cells explicitly, but can instead retrieve the cell vertices with the help of the mapping between geometry and topology. Since the cells are all identical with regular data, our mapping is simply:

$$i = \left\lfloor \frac{p_x}{\Delta x} \right\rfloor, j = \left\lfloor \frac{p_y}{\Delta y} \right\rfloor. \tag{1}$$

where i and j are the topological coordinates of the lower left corner of a cell, and Δx and Δy are the geometric size of a cell in the x and y dimensions (5.0 and 3.0). To get the indices of the other vertices in the cell, we simply increment i and j appropriately. Once all four indices are found, they can be given to the root data source, which returns the corresponding sample points. The lattice can now compute an approximate value for p from these sample points.

The other possibility is that the query point p is equal to (or within some epsilon of) the location of a sample point. We see this situation in the lower left of Fig. 2. Here, no approximation is necessary, so we can retrieve the datum from the data source directly.

If a topological neighborhood such as the one shown at the center of Fig. 2 is desired, we can either use our simple mapping to map a geometric region to the topological index space, or access the topology directly. In either case, we issue a *subblock()* query to the root data source. The code looks something like:

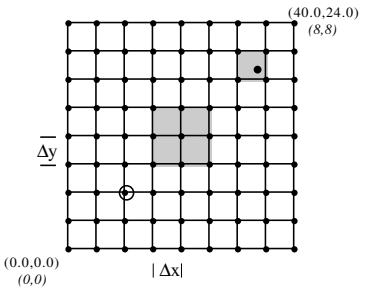

Fig. 2. Regular rectilinear data with geometric and topological coordinates

```
rootDS     = lattice.getRootDS();
topology   = lattice.getTopology();
geomBounds = new GBounds( new Point(15.0, 9.0),
                          new Point(25.0, 15.0) );
topoBounds = topology.map( geomBounds );
theBlock   = rootDS.subblock( topoBounds );
```

Notice that the first couple of lines of code get the root data source and topology from the lattice. Although the lattice provides a very convenient geometric view of the data, it may be desirable to manipulate the sample points directly through the topology. In this code example, we use the topology to map a geometric region to a topological region, and then give this region to the root data source, which will return a block of sample points.

Topological iteration is similarly straightforward. We give the `topoBounds` computed above to an *ISIterator* object that iterates over the sample points. Note that the *ISIterator* class is a child of the *IndexSpaceId* class which is the type required by the *datum* method.

```
ISIterator iter=new ISIterator(topoBounds);
for( iter.init(); iter.valid(); iter.next()){
   System.out.println( rootDS.datum( iter ));
}
```

5.6 Perimeter Grids

Fig. 3 shows a *perimeter grid*, also known as a *perimeter lattice* [12]. In this case, we must store an array of values for each axis called *perimeter* arrays (also shown in the figure). The i-th value in a perimeter array gives the geometric location of the i-th row or column of elements along an axis. Using this information, we can produce a mapping between geometry and topology. For example, consider the point at (20.0, 13.5) in Fig. 3. For the x axis, we use the array $A[]$ along the top of the figure, and note the index i at which the x value of 20.0 is less than or equal to $A[i+1]$. We perform a similar operation for the y axis using $B[]$ to discover an index j for which the y value of 13.5 is less than or equal

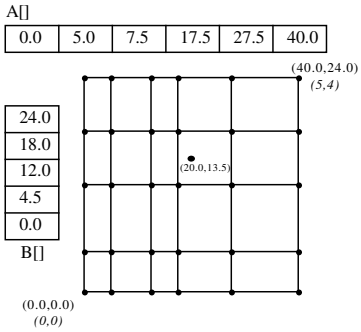

Fig. 3. A Perimeter Grid and its attendant perimeter arrays

to $B[j+1]$. It's easy to see that values $i = 3, j = 4$ satisfy our requirements and give us the topological coordinates of the containing cell's lower left corner. As with the regular case, we can now easily compute the topological indices of the remaining three corners of the cell and retrieve four data values from the root data source, using them to compute an approximate value for our sample point. In fact, assuming that the topology has been given the necessary mapping information, the sample code shown in the last section can be used unchanged.

5.7 Curvilinear Grids

A *curvilinear* grid has rectilinear topology, but a geometry that is curved in a space and can even have higher dimensionality than the topology dimension. Often such grids are viewed as a regular rectilinear computational space that has been *warped* in a geometric space, like the surface of an airplane wing, as shown in the upper left of Fig. 4. If this warping transformation is simple enough, it may be possible to analytically construct the mappings between the geometric and topological spaces. The implementation would then be similar to the examples we have already seen.

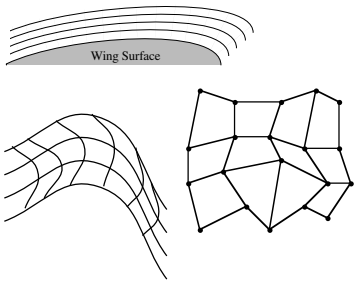

Fig. 4. Some more difficult varieties of rectilinear data.

Of course, such analytical mappings are not always possible. For such data, it is necessary to explicitly store the geometric location of each sample point. The root data source must therefore be consulted whenever we wish to map from one space to another. Still, from the above examples, we know that once the mapping from geometry to topology has been performed for one point, it is easy to navigate the topology as required to find geometric or topological point and cell neighbors. Regardless of the geometric distribution of the points, we are able to take advantage of the rectilinear nature of the topology.

5.8 Adaptive Resolution with Rectilinear Data

We can support adaptive resolution (AR) data using two different approaches. If the adaptive resolution is based on the topology of the lattice, we can use the adaptive resolution features of the data source implementation as described in sections 4.9 and 4.10. If the adaptive resolution is based on geometric regions, the rectilinear structure of the topology cannot be maintained, so this case must be handled as unstructured data (see section 5.10).

5.9 Non-Rectilinear Grids

Unstructured and semi-structured grids add another level of complexity over and above even the most unruly rectilinear grids since the vertices belonging to a cell can no longer be computed from the topology alone. Not only must we store the geometric position of each point explicitly in the data source, we must also store a list of cell definitions that specifies the vertices comprising each cell.

***Datum* Queries.** In sections 5.2 and 5.3 we noted that exhaustive search of the cell list is out of the question for data of even moderate size, and that partitioning the geometry can be very helpful in limiting the search to a small number of cells. In conjunction with Fig. 5, a simplified implementation of a

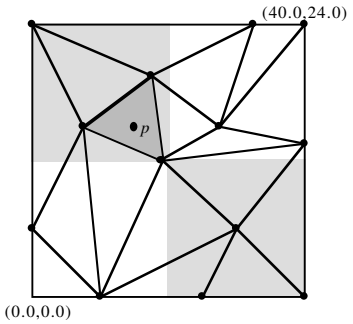

Fig. 5. An Unstructured Grid with partitions.

datum query of the form `lattice.datum(p)` illustrates how partitioning is used to accelerate data access and as a bridge between the topology and geometry:

```
Datum datum( Point p ){
   IndexSpaceID partitionId = geometry.mapToPartition( p );
   Cell c = topology.findEnclosingCell( partitionId, p );
   return c.approximate( p );
}
```

Fig. 5 shows four partitions dividing the space into four equal quadrants. The first line of the example code asks the geometry to map the point p to one of these partitions. This can be done in a manner essentially similar to the mappings for regular data described in section 5.5. It should be apparent that the result of the mapping selects the upper left partition. Next, we ask the topology to find the cell that contains the query point. The result corresponds to the darkened triangle in the figure. Notice that the topology is able to use `partitionId` to access the partition that was previously identified by the geometry. Finally, the cell computes an approximated *Datum* for p and the result is returned.

Topological Iteration. As with rectilinear data, geometric iteration can be performed through repeated *datum()* queries on the lattice. However, topological iteration is sometimes more desirable because of efficiency, and because the researcher wants to deal with actual sample points only. Nevertheless, it is still possible to give a geometric specification of the extent of a topological iteration. For example, Fig. 6 shows the retrieval of all sample points within a radius r

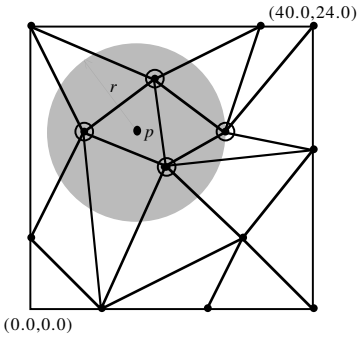

Fig. 6. A Topological Iteration with Geometric Extent

of the geometric location p. The same methods outlined above can find the cell surrounding p. The cell's list of neighbors identifies nearby cells to visit using a standard breadth first or depth first search. Any cell vertex meeting the geometric constraint is included in the iteration. The iteration terminates when

the set of visited cells covers the geometric extent. Of course, an iteration could be specified using only topology, such as "access all the topological neighbors of this point or cell". In either example, it is important to note that topological navigation need not consider the geometric partitions.

5.10 Further Applications for Partitions

The link between the topology's cells and the geometry's partitions also plays an important role in our multiresolution and adaptive resolution data representations. In this section we present ideas for wider employment of partitioning throughout our lattice implementation.

Semi-Structured Grids. Consider a dataset with a periodic topology, as shown in Fig. 7. Since the topology has the same local pattern repeated over the domain, we can extend the partitioning method used for unstructured data to help represent periodic topologies efficiently. We can represent one copy of the pattern in a data structure called a *supercell* [11]. Conceptually, the supercell is

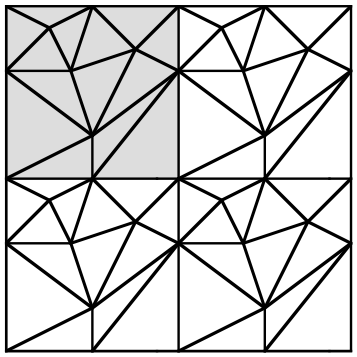

Fig. 7. A Periodic Topology

repeated over the domain, forming the complete geometry and topology without incurring the storage costs of a large number of cells. Each supercell has a *position index* (2D in this example) and each point has a *point index* indicating its position inside the supercell. *Datum* access is slightly more complicated, since we no longer have cell vertices that correspond to a single datum. However, we can easily form a data source index for a sample point from the supercell's position index and the point's point index. Looking at Fig. 8 we see that the position index (1,1) for the circled point is combined with the point index (4) to form a data source index (1,1,4), with which we can retrieve a datum.

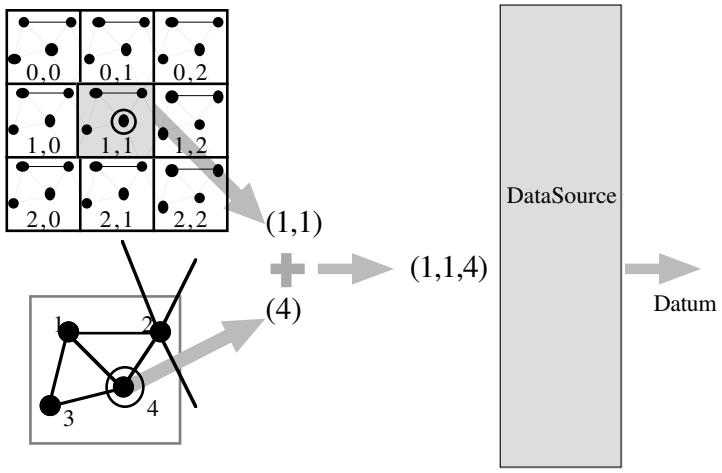

Fig. 8. Using supercells to represent periodic topologies.

Adaptive Resolution with Unstructured Data. The adaptive resolution features of data sources is not sufficient to handle unstructured data, so we must provide lattice level support for this case. As described in section 5.9, we can use the geometry partitions to locate the topological cells. Now, however, the partitions hold cell meshes of different resolutions. Typically these meshes will have been harvested from different levels of an MR hierarchy.

Multiresolution Hierarchies. Multiresolution data can be implemented as a hierarchy of lattices. Partitions are helpful here because they provide a convenient specification of regions for *support* and *influence*. Each level of the hierarchy (lattice) has its own geometric partitioning which gets increasingly coarse along with the coarseness of the lattice. The geometry and topology used in MR hierarchies can refer to partitions above and below them in the hierarchy to denote the set of points that they support or influence. Since partitions can be specified with a simple index, this is a compact way of specifying a region in the domain.

6 Conclusions and Future Research

We have developed a comprehensive data model for multiresolution multisource scientific data and have implemented a prototype system that provides database support for the model. This paper describes the principal features of the data model and the principal components of the implementation. The implemented features of the data model provide application code with a clean *natural* straightforward view of user data even if it is constructed from many disparate components. Preliminary evaluation of our implementation indicates that the features provided by the model can be provided with little or no additional overhead.

Our current implementation only provides limited support for adaptive resolution data and does not yet handle multiresolution hierarchies in a comprehensive way. Although the data model includes features desirable for *distributed* data and computation, these features are also not yet implemented.

Acknowledgements

We thank the National Science Foundation which has supported this research under grants IIS-0082577 and IIS-9871859. We also thank the other members of our research team for their contributions to the design and implementation of the system: Feng Jiang, Wenhui Li, Gang Lu, Denise Mitchell, and Xuan Tang.

References

1. Cignoni, P., C. Montani, E. Puppo, R. Scopigno, "Multiresolution Representation and Visualization of Volume Data", *IEEE Trans. on Visualization and Computer Graphics*, Volume 3, No. 4, IEEE, Los Alamitos, CA, 1997.
2. Glassner, A.S., "Space Subdivision for Fast Ray Tracing", *Comp. Gr. Appl*, Vol. 4, No. 10, Oct. 1984, 15-22.
3. Haber, R.B., B. Lucas, N. Collins, "A Data Model for Scientific Visualization with Provisions for Regular and Irregular Grids", *Proc. IEEE Visualization '91*, San Diego, CA, 1991.
4. Hibbard, W.L., C.R. Dyer, and B.E. Paul, "A Lattice Model for Data Display", *Proc. IEEE Visualization '94*, IEEE, Washington, DC, 1994.
5. Hibbard, W.L., D.T. Kao, and Andreas Wierse, "Database Issues for Data Visualization: Scientific Data Modeling", *Database Issues for Data Visualization*, Proc. IEEE Visualization '95 Workshop, LNCS 1183, Springer, 1995.
6. Jiang, F., "Implementation and performance evaluation of block-oriented adaptive resolution data for multi-source scientific datasets", M.S. Thesis, Computer Science Department, University of New Hampshire, Durham, NH, 2002.
7. Kao, D.T., R. Daniel Bergeron, Ted M. Sparr, "An Extended Schema Model for Scientific Data", *Database Issues for Data Visualization*, Proc. IEEE Visualization '93 Workshop (LNCS 871), Springer, Berlin, 1993.
8. Kao, D.T., "A Metric-Based Scientific Data Model for Knowledge Discovery", Ph.D. Thesis, Computer Science Department, University of New Hampshire, l Durham, 1997
9. Li, W., "Multisource dataset support in a scientific database system", M.S. Thesis, Computer Science Department, University of New Hampshire, Durham, NH, 2002.
10. Pfaltz, J.L., R.F. Haddleton, J.C. French, "Scalable, Parallel, Scientific Databases", *Proc. 10th International Conference on Scientific and Statistical Database Management*, IEEE, Los Alamitos, CA, 1998.
11. Rhodes, P.J., R.D. Bergeron, T.M Sparr, "A Data Model for Distributed Multiresolution Multisource Scientific Data" in G. Farin, H. Hagen, and B. Hamann, eds. (2002), *Hierarchical and Geometrical Methods in Scientific Visualization* , Springer-Verlag, Heidelberg, Germany, to appear.
12. SCVT, "Introduction to Scientific Visualization Tools", *http://scv.bu.edu/SCV/Tutorials/SciVis/input_data.html*, Boston University Scientific Computing and Visualization Group, Boston, 1998.
13. Speray, D., S. Kennon, "Volume Probes: Interactive Data Exploration on Arbitrary Grids", *Computer Graphics*, Vol. 24, No. 5, ACM, 1990.