Logical and Physical Data Independence for Native Scientific Data Repositories

Bill Howe
OGI School of Science & Engineering at
Oregon Health & Science University
Beaverton, Oregon
bill@cse.ogi.edu

David Maier
Department of Computer Science
Portland State University
Portland, Oregon
maier@cs.pdx.edu

Abstract

Many datasets in the physical sciences, especially the results of simulations, are defined over a topological grid structure. Applications in these domains would benefit from a principled interface to gridded datasets via a specialized data model. Traditionally, benefits of a data model are realized only after data is ensconced within a managed database environment. However, massive bulk-loading and re-loading operations in large-scale data repositories are prohibitively expensive. Instead, we superimpose a specialized data model over native data repositories stored on directly on OS filesystems rather than managed by a database system. Views in a specialized data model can be defined via references to native directory structures and file content, providing physical and logical data independence. This non-intrusive approach appears to reduce space requirements, speed development, and cooperate with legacy applications.

1 Introduction

Scientific data repositories are approaching petabyte scales. Data manipulation in the context of such large repositories benefits from a well-defined data model interface. Physical and logical data independence provided by a data model insulate applications from the organization details and provide custom interfaces without copying and restructuring data. Algebraic optimization can recover performance penalties incurred by an extra layer of software and simplify previously hand-optimized applications.

In practice, most data models are implemented as heavyweight database systems requiring that legacy data be injected into a managed environment before benefits can be realized. Loading and re-loading data into database environments during development, and even simply inserting new data after successful deployment, can become infeasible. In our experience, bulk-loading simulation results into a relational database took almost as long as the simulation itself [4].

Further, legacy applications dependent on the existing interface are entrenched within the infrastructure. Maintaining two copies of data during the transition, one managed and one native, is a possibility since updates are rare, but storage space can be quickly depleted.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

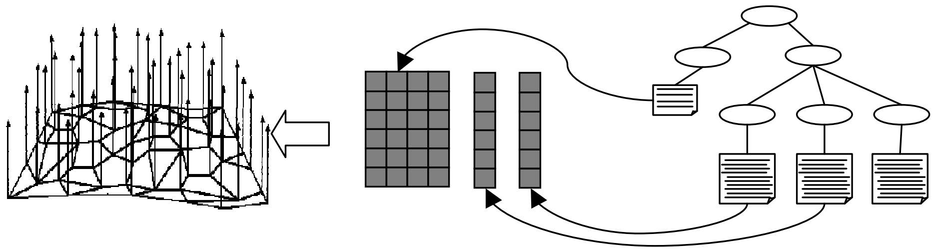

Figure 1: Mapping data from a filesystem model to an array model to a gridfield model.

Another approach, advocated in this paper, is to superimpose a data model over *in situ* data to provide principled access methods, specifically in Earth science domains. That is, given a scientific data repository stored directly on a filesystem, we aim to 1) describe it and 2) access it through a more specialized grid-based data model developed in previous work [4].

The context for our work is the CORIE Environmental Observation and Forecasting System being developed at the OGI School of Science and Engineering at Oregon Health and Science University [1]. The CORIE system is a multi-purpose platform for studying the fluid dynamics of the Columbia River estuary. Customers of CORIE's data products include commercial fisheries, environmental policy makers, and external research institutions. The CORIE repository consists of forecast and "hindcast" simulations covering time periods since 1998. Each day, forecast simulation runs add about 5GB to the data repository, while batches of hindcast runs, batches of calibration runs, and individual researchers' experiments are executed concurrently.

Simulation results are encoded in custom binary files and arranged in complex directory hierarchy. Three-dimensional spatial datasets are produced for each instant of simulated time, for each of several physical variables. These timestep datasets are distributed across several *checkpoint* files, each one usually covering a 24-hour period of simulated time. Checkpoint files have a custom binary format, and are arranged in a directory structure by week, by code version, and sometimes by purpose; e.g., calibration runs as opposed to final results. As a concrete example, a checkpoint file for the first Saturday of 2004 might have the path `hindcasts/01-2004/1_salt.63`. Every application accessing these data must understand the semantics of file and directory names, or interpret custom binary file formats, or both. The resulting situation is that much of the CORIE software is rather brittle with respect to changing data characteristics.

In previous work, we proposed a *gridfield* data model for manipulating gridded datasets [4] independently of physical organization details. The gridfield data model promotes topological structures to first-class entities, exposing equivalences and enabling optimization opportunities hidden by other tools. For example, the expression `Restrict(a>b, Cross(A, B))` computes a kind of join between two gridfields A and B, while preserving certain topological invariants.

In order to provide a gridfield interface to CORIE's significant existing data repository without destroying their native representations, we map the existing data structures (directories, files, arrays) into gridfields using *gridfield schema files*. Gridfield schema files describe how to translate source data into arrays, and thence how to assemble arrays to produce gridfields. Figure 1 illustrates the situation.

A gridfield schema file (henceforth *schema file*) holds a set of declarations. *FileType* declarations describe directory structures and allow access to data encoded within file and directory names. *BinaryBlocks* describe the layout of binary files. GridField declarations specify their constituent data through references to FileTypes and BinaryBlocks. With these declarations, we can assemble gridfields by pulling components together from multiple files, by extracting a portion of a single file, by extracting information from file names and directory names, or by an arbitrary combination of these methods. The finished schema file can then be *applied* to a *context* to determine which GridField declarations (if any) can be activated given the available data. With this mechanism, users have a common interface for both a structured data repository as well as individual files resulting from private experiments.

Effective use of information encoded in file and directory names requires more than simply invoking OS system calls and parsing strings. The logical structure of a data repository is frequently reflected in the directory structure, and filenames carry semantic information about their relationships with other files. A thorough model should allow transparent access to information in both a file's name and its content.

To illustrate, consider that the boundary between file name and file content can change depending on the situation. For example, the directory tree illustrated in Figure 3a has a separate file for each day of the week (for each variable). In this case, the day of the week is not stored within the file, and is therefore inaccessible to tools for reasoning only about a file's content [5, 7, 8]. This representation reflects the manner in which the data was generated: A checkpoint file was recorded for each day of the week to simplify recovery in case of failure. An individual researcher's ad hoc experiment might not require such caution; she might lump a week's worth of results into a single file without saving any checkpoints. In order to provide transparent access to either of these two representations, the model must capture information that spans file and directory names as well as file contents.

We acknowledge two roles involved with this framework. The *Schema Designer* writes FileType, BinaryBlock, and Gridfield declarations to a text file and publishes it to a well known location. The *Application Developer* applies the schema file to a context and manipulates available gridfields by writing expressions in a gridfield algebra. To preserve physical data independence, application developers are not expected to work directly with FileTypes or BinaryBlocks.

We do not support updates through the gridfield interface, though we do support some forms of append operations. Scientific data repositories tend to be append-only due to the need for provenance; even "bad" data tends to be kept, uncorrected, if space permits. Though support for updates to data values is not a priority, minimizing the impact of evolving representations is. By allowing arbitrary gridfield expressions to appear in a schema file, we provide a simple view facility and therefore logical data independence. Applications using the gridfield interface are insulated from changing representations. Only those portions of the schema file affected by the change must be adjusted by the schema designer.

The following list summarizes the benefits of a our approach.

- **Transparent interoperation with legacy applications.** No reorganization of the data is required for use.
- **Incremental return on investment.** As additional schema declarations are recorded, additional functionality is supported.
- **Transparent support for partially conforming data.** By not requiring all declarations to be activated in all contexts, the same schema file can be used as an interface to private data (with less structure perhaps) and public data (with more structure).
- **Generality.** External sources of data published to the web often follow a filesystem-like data model, and can therefore be accessed with the same framework as local data repositories.

2 GridField Data Model

A gridfield consists of a *grid*, and one or more *attributes*. A grid is a set of *cells*, partitioned by dimension. Each cell of dimension $d > 0$ (d-cell) is defined by a sequence of references to cells of dimension $d = 0$ (*nodes*). For example, a triangle is a sequence of three references to nodes. Every non-empty grid must have at last one 0-cell. Each attribute is bound to the cells of exactly one dimension d, such that each d-cell maps to exactly one value of the attribute.

a)

```
1: GridField G
2: G.grid.cells[0] = implicit 20
3: G.grid.cells[d] = <cell array expr>
4: G.a[d] = <array expr>
```

b)

```
1: GridField Horiz
2: Horiz.grid.cells[0] = implicit 10
3: GridField Vert
4: Vert.grid.cells[0] = implicit 15

5: GridField G = Cross(Horiz, Vert)
6: G.temperature(0) = <array expr>
```

Figure 2: Two examples of gridfield assembly syntax.

In this section we discuss how to assemble gridfields from arrays. An array named $name$ with n elements of type τ will be written `name[n]` (τ). Elements may be of a primitive numeric type, a tuple of named attributes, or another array. We will describe the source data model of arrays and tuples in more detail in the Section 4.

An example of a GridField declaration appears in Figure 2a. The nodes for the grid component of the gridfield G can be defined by an array expression returning integers (explicit node references) or by using the keyword `implicit`. The declaration in Figure 2 specifies that the grid of G will have 20 nodes when assembled. The third line in Figure 2a specifies how to construct a set of cells of a dimension $d > 0$ (e.g., line segments, triangles, tetrahedra). The placeholder `<cell array expr>` must have one of the following two forms.

$$array[N](m : int, array[m](n_1 : int, n_2 : int, ..., n_m : int))$$

where N is the number of cells (the size of the array), and each cell is a sequence of m references to a node. The second form is

$$array[N](n_1 : int, n_2 : int, ..., n_M : int))$$

where N is the number of cells and every cell has M nodes. The first form corresponds to mixed cell types found in grids used for finite-element simulation (e.g., both triangles and quadrilaterals appear in the same grid.) The second form corresponds to the uniform cell types found in graphics applications (e.g., triangles in two dimensions or tetrahedra in three).

To bind an attribute a to cells of dimension d, we use the syntax in line 4 of Figure 2a. The placeholder `<array expr>` must return an array of primitive elements (no nested sequences or arrays). Array elements are associated with cells positionally; the first cell is bound to the first array element, the second cell is bound to the second array element, and so on.

We can also remove the integer argument to the keyword `implicit`. Without this argument, the keyword indicates that the number of nodes is unconstrained, and may be derived from the number of values in an attribute bound to the 0-cells. When the argument is present, the number of nodes is constrained to be the argument's value, and binding attributes with a different cardinality results in an error.

Gridfields can also be declared through expressions of a gridfield algebra, as in Figure 2b. We first define two gridfields `Horiz` and `Vert` in lines 1 and 3, and then construct their *cross product* in line 5. The grid `G` will have 150 nodes and one bound attribute, `temperature`. The details of this and other gridfield operators can be found in a previous paper [4].

In the next section, we will describe how to extract the requisite arrays from native data formats.

3 Modeling Filesystem Data

Scientists frequently store and manage their data using direct filesystem interfaces. Limitations to this approach are evident. A file's metadata must frequently be embedded in file and directory names. For ex-

a)
```
/run  /01-2004 /1_salt.63
               /1_temp.63
               /2_salt.63
               /2_temp.63
               :
      /02-2004 /1_salt.63
               /1_temp.63
               /2_salt.63
               /2_temp.63
               :
```

b)
```
FileType weekly_run
pattern[wk,yr] = /run/%i-%i/
FileType salt63
pattern[day] = %i_salt.63
FileType temp63
pattern[day] = %i_temp.63

GridField Days
G.grid.cells[0] = implicit
G.day[0] = salt63.day
```

c)
```
FileType alldays
pattern[wk,yr,day]
   = /run/%i-%i/%i_salt.63

GridField AllDays
G.grid.cells[0] = implicit
G.week[0] = alldays.wk
G.year[0] = alldays.yr
G.day[0] = alldays.day
```

Figure 3: A gridfield schema for extracting information from a filesystem directory structure.

ample, a dataset available from the National Climate Data Center (NCDC) website is stored in a file named `meso-eta_215_20030803_1800_fff` [6]. The meaning of the date string "20030803" is apparent, but the other fields require some external information to parse. Constraints are also difficult to express. The file from the NCDC above was found in a directory named `20030803`. Presumably, all files within this directory should be tagged with this date, and those that are not represent anomalies.

The example in Figure 3b is taken from the CORIE system. The root directory `runs` contains directories with simulation results for each week of 2004 in the form `<week number>-2004`. Each week directory contains 14 files, one for each variable (salinity, temperature) and day of the week (1-7). To access the week number or day number embedded in these file names, we can write a *path pattern* in the style of the Unix scanf command.

```
[week, year, day] = /runs/%i-%i/%i_salt.63
```

The left-hand side of this expression is a tuple of variable names. The right-hand side is a pattern matched against the set of all files in some filesystem context. Wildcard placeholders are given a one-character type code (`i` for integer, `f` for float, etc.). Each variable name can be accessed as a sequence of values once the pattern is applied to a context. Note that the sequence order is imposed by the directory traversal; a potential extension to the language is to allow arrays to be sorted to enforce a different ordering.

Path patterns are lifted into FileTypes by assigning them a type name. Given a filesystem context, a FileType can be evaluated to return a sequence of tuples whose attribute names are given by the list of variable names, and whose types are given by the type codes of the wildcards. For example, we can define separate FileTypes for the run directories, the salinity data, and the temperature data as in Figure 3b.

A simple GridField declaration referencing these FileTypes appears at the bottom of Figure 3b. The gridfield `Days` consists of 0-cells only, and its cardinality is determined implicitly by the number of files that match the path pattern. The array expression for attribute `day` of the declaration references the FileType `salt63`, and then uses the dot notation to access one of its sequence variables. If the schema in Figure 3b were applied to the directory `02-2004/` of Figure 3a, the gridfield `Days` would consist of seven nodes, since there are seven files that match the pattern. Note that we may chose `temp63` to define the `Days` gridfield. The ambiguity reflects the fact that the `day` attribute is redundantly encoded in the filenames of all simulation variables.

Another set of GridField declarations appears in Figure 3c. The gridfield `AllDays` effectively "unnests" the day numbers from within the week directories, producing (week, year, day) tuples.

4 Modeling File Content

Scientists frequently use packed binary encodings of large datasets to preserve space and improve performance. We model the content of binary files using named primitive types, records (where each element is named and

a)

```
BinaryBlock image
image.content = (
  header : 10c
  h : i
  w : i
  data : h*(
    row : w*(
      r:i g:i b:i
    )
  )
)
```

b)

```
GridField Horizontal
Horizontal.grid.cells[0] = implicit image.h

GridField Vertical
Vertical.grid.cells[0] = implicit image.v

GridField Image = Cross(Horizontal, Vertical)
Image.r[0] = image.data.r
Image.r[0] = image.data.g
Image.r[0] = image.data.b
```

Figure 4: A gridfield schema file for extracting binary file content.

may have a different type) and arrays (where all elements have the same type and are referenced by position only). Both structures support arbitrary nesting. Arrays are one-dimensional; multidimensionality is captured through nesting.

A schema for a block of binary content representing an image appears in Figure 4a. The top-level component is named `image`, which contains a 10-character string named `header`, the height `h` and width `w` of the image, and a two-dimensional array of RGB values. The height and width are primitive components of the form `name : type`, where type in this case is `i` signifying an integer. The data component has a type of the form `x*(component)` where x is the name of an integer component (or is an integer literal) and $component$ is another named component in the current scope. In this case, another array component is nested within the first to represent the second dimension.

Tuple elements are accessed by name or position. An array element is accessed through conventional integer indexing. An array can also be "sliced" to produce another array as in APL or Matlab. Some examples of these expressions appear in Table 1.

This language is not particularly expressive: For example, we cannot transpose, sort, or aggregate arrays. Array expressions are meant only as a means of assembling gridfields from raw binary data; we rely on gridfield algebra expressions to perform more complex transformations.

To construct gridfields from BinaryBlocks, we use a dot notation syntax similar to that used in Section 3, as in Figure 4b. We discuss combining FileTypes and BinaryBlocks in Section 5.

5 Nested Gridfields

Each GridField declaration is evaluated against a *context*. A context is either the root context, a FileType, a BinaryType, or a concatenation of contexts. The root context is provided by the application developer during connection to the data repository. So far, we have written declarations as if they all are evaluated against this root context. To assemble more complex *nested* gridfields, we need to pass a context in as an argument to each declaration. Effectively, GridField declarations are functions mapping contexts to gridfields.

Consider the schema in Figure 5, where the GridField declarations are adorned with a context argument. As an argument, a context represents a single entity. Using the dot notation, we can concatenate contexts and access data items within contexts. For example, for the GridField declaration `Time`, `ROOT` is a single entity, the directory (or file) supplied by the application developer to which the schema is being applied. The expression `ROOT.gridfile.time`, however, will return an array of values extracted from the time field of FileType `gridfile`. The expression `ROOT.gridfile.scalardata` requests that the contents of each grid file be interpreted according to the binary block declaration `scalardata`. Since each element of this array has complex structure, we cannot construct an ordinary attribute of primitive values. However, we can construct a

Table 1: Example expressions and their types.

expression	type
`image.h`	`integer`
`image.data.row[1]`	`(R:i, G:i, B:i)`
`image.data.row.R`	`array[h*w](R:i)`
`image.data[a:b]`	`array[b−a](array[w](R:i, G:i, B:i))`
`image.data.row[a:b:c]`	`array[(b−a)/c](array[w](R:i, G:i, B:i))`

nested gridfield, where each element of a new attribute is itself a gridfield.

The syntax for a nested gridfield involves referencing the name of the GridField declaration and passing it an array expression as an argument. Each element of the array expression is considered a context in which to assemble a gridfield. Therefore, the syntax is effectively shorthand for a list comprehension expression, as denoted by the C-style comment below the declaration of GridField `Time`.

In the declaration for GridField `Geometry`, a grid with explicit triangular cells is constructed. The gridfield has x and y values bound to the nodes representing the position of each node in two-dimensional space. The gridfield `Temp` and `Velo` are both assembled from the gridfield `Geometry`. Of course, we might have decided to model these data as a single gridfield with three attributes `temp`, `u`, and `v`. By separating the temperature data from the velocity data ,we illustrate the variety of modeling choices open to the schema designer.

6 Related Work and Discussion

Scientific applications today in some ways resemble business applications circa 1977. Copious amounts of data are stored in files with intricate formats. Skepticism regarding database technology is rampant. Legacy systems are built from efficient but extremely brittle software components. To mitigate the perceived (and real) risk of adopting unproven database systems, early data models were implemented as file transformation engines.

The EXPRESS system [7] provided two languages: one for describing a file's structure, and another for transforming that structure. Transformations were used as a query facility, but also as a bulk load facility to translate legacy data into a new format. Our approach is similar, though we distinguish two data models: one for source data (directory structures and file content) and another for target data (gridfields). We have not yet considered materializing gridfields assembled using schema files. That is, we do not permanently transform source data into gridfields, but rather provide a gridfield interface to native data.

Batory gave a taxonomy of record-oriented file structures used by commercial databases in terms of fields and pointers [2]. Our work similarly provides a description of grid-oriented file structures in terms of arrays. Once precisely described, Batory showed how you could convert from one representation to another in a principled manner, or compose the transforms. Tools to convert between array-oriented representations are valuable, though we are primarily interested in lifting arrays into more semantic structures, particularly gridfields.

The Binary Format Description Language (BFD) [5] is an XML dialect that describes binary formats and allows transformation of binary data to XML data. While this tool has a niche, our interest is to support efficient and flexible access to binary data – converting binary data to XML is clearly impractical for large datasets. The BinX [8] library is also related to this proposal. Binary data file formats are described using instances of a specialized XML Schema. An API allows access to the data and automatic reformatting according to the local machine's byte order and bit order. The current proposal aims to provide more powerful reasoning capabilities than simple file access. BinX supports only sequential file access, and file structures are all assumed to fit in memory. BinX also offers no support for transforming representations; any restructuring is done "manually" in a C program after reading in an entire source file.

Schema integration techniques, particularly the concept of local-as-view [3], is also related to our approach.

```
FileType gridfile
pattern[time] = %i_*.grd

BinaryBlock griddata
content = (
  nodes : i
  elements : i
  triangles : elements*
    ( n1:i n2:i n3:i )
  geometry : nodes*
    ( x:i y:i )
  velocity : elements*
    ( u:f v:f )
  temp : elements*
    ( value:f )
)
```

```
GridField Geometry(griddata)
Geometry.cells[0] = implicit griddata.nodes
Geometry.cells[2] = griddata.triangles
Geometry.x[0] = griddata.geometry.x
Geometry.y[0] = griddata.geometry.y

GridField Temp(griddata) = Geometry(griddata)
Temp.temp[2] = griddata.temp.value

GridField Velo(griddata) = Geometry(griddata)
Velo.u[2] = griddata.velocity.u
Velo.v[2] = griddata.velocity.v

GridField Time(ROOT)
Time.cells[0] = implicit
Time.time[0] = ROOT.gridfile.time
Time.velo[0] = Velo(ROOT.gridfile.griddata)
Time.temp[0] = Temp(ROOT.gridfile.griddata)
// = [Temp(c) | c <- ROOT.gridfile.griddata]
```

Figure 5: A gridfield schema involving nested gridfields.

However, the source data being integrated (usually stored in relational databases) are more suitable for query and manipulation than our raw native repositories. Wrappers and mediators also serve to provide access to native data. However, data sources are usually modeled as black-boxes with a limited interface. Our context allows us to reason about the source data in considerable detail, though we cannot prescribe changes in its organization.

In conclusion, we advocate in situ processing of large scientific data repositories as a means of harnessing database techniques without the substantial cost of database deployment. Database technology is not prolific in the scientific community; the real and perceived cost of ownership is part of the reason. If some database techniques can be teased apart from their monolithic implementations, the database community stands a chance of contributing a tangible benefit to the scientific community.

References

[1] A. Baptista, M. Wilkin, P. Pearson, P. Turner, M. C., and P. Barrett. Coastal and estuarine forecast systems: A multi-purpose infrastructure for the columbia river. *Earth System Monitor, NOAA*, 9(3), 1999.

[2] D. S. Batory. Modeling the storage architectures of commercial database systems. *ACM Trans. Database Syst.*, 10(4):463–528, 1985.

[3] M. Friedman, A. Y. Levy, and T. D. Millstein. Navigational plans for data integration. In *AAAI/IAAI*, pages 67–73, 1999.

[4] B. Howe and D. Maier. Algebraic manipulation of scientific datasets. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB2004)*, 2004.

[5] J. Myers and A. Chappell. Binary format description language. Technical report, Pacific Northwest National Laboratory.

[6] National Climatic Data Center. NCEP AWIPS eta model data. http://nomads.ncdc.noaa.gov:9090/dods/NCDC_NOAA Port_ETA.

[7] N. C. Shu, B. C. Housel, R. W. Taylor, S. P. Ghosh, and V. Y. Lum. EXPRESS: A data EXtraction, Processing, amd REStructuring System. *ACM Transactions on Database Systems*, 2(2):134–174, 1977.

[8] M. Westhead and M. Bull. Representing scientific data on the Grid with BinX - binary XML description language. Technical report, EPCC, University of Edinburgh, 2003.