Retrofitting a Data Model to Existing Environmental Data

Bill Howe David Maier

Department of Computer Science
Portland State University
Portland, Oregon
{howe, maier}@cs.pdx.edu

Abstract

Environmental data repositories are frequently stored as a collection of packed binary files arranged in an intricate directory structure, rather than in a database. In previous work, we 1) show that environmental data is often logically equipped with a topological grid structure and 2) provide a data model and algebra of gridfields for manipulating such gridded datasets. In this paper, we show how to expose native data sources as gridfields without preprocessing, bulk-loading, or other prohibitively expensive operations. We describe native directory structures and file contents using a simple schema language based on nested, variable-length arrays. This language is capable of describing general binary file formats as well as custom formats such as those used in the CORIE Environmental Observation and Forecasting System. We provide optimization techniques for extracting arrays by 1) analyzing file structure and 2) generating specialized code. Using extracted arrays, we assemble gridfields for more sophisticated manipulation and visualization. We show results from the CORIE Environmental Observation and Forecasting System. We find that generic access methods allow logical manipulation of physical data sources via the gridfield algebra without reformatting existing data.

1 Introduction

Integration of data within institutional and regional environmental systems is hindered, in part, by the heterogeneity of data formats. For example, the Northwest Association of Ocean Observing Systems (NANOOS) [1], chartered in response to a congressional initiative, aims to federate various institutional systems to provide a more comprehensive view of the coastal ocean in the Pacific northwest. The NANOOS charter acknowledges the significant number of ocean observing systems, but warns that these systems are not integrated in that they "do not share standards or protocols." In the interest of accelerating federation efforts in the environmental sciences, we have been studying the logical and physical structure of environmental data.

Environmental simulation and observation data are frequently defined over a topological grid structure. For example, a timeseries of sensor measurements might be defined over a 1-dimensional (1-D) grid, while the solution to a partial differential equation using a finite-element method might be defined over a 3-dimensional (3-D) grid. Datasets can be bound to a grid structure, producing what we call a *gridfield*. In previous work [7, 9], we develop a data model and associated query language for manipulating gridfields.

The salient feature of the gridfield model is that the grid structure of the datasets is explicit. Traditionally, data were stored and manipulated as arrays; the logical grid structure was appeared only in the code itself. By reifying this hidden grid structure, we are better able to describe and implement a variety of manipulations using a small set of algebraic operators. Further, the data model helps separate logical and physical concerns, insulating software layers from changing physical representations.

However, in order to use gridfields to manipulate data from existing disparate sources, we must be able to read and interpret existing stored data; that is, we need appropriate access methods. Environmental datasets (indeed, most scientific datasets) are stored directly on a filesystem in packed binary files. Legacy applications can interpret these files, but new applications based on gridfields cannot.

One approach is to convert existing datasets to a special format already equipped with a gridfield interface. Indeed, database vendors frequently assume this approach: Before your data can be manipulated using the relational model, you must surrender control to the DBMS via bulk load operations. Unfortunately, the growth rate of collected scientific data is sufficiently large that sweeping conversion efforts are unlikely to succeed. Besides scalability issues, legacy analysis tools dependent on a particular format are common in scientific domains; mandatory rewrites of these tools would be unpopular.

Our initial solution was to hand-code custom access methods for each file format we encountered. Besides being time-consuming, this approach is inflexible with respect to datasets that span multiple files. To generate a gridfield, code to iterate over multiple files is layered on code to interpret each file's format. Finally, the results are used to assemble gridfield objects suitable for manipulation with the gridfield algebra. These kind of routines became common enough to look for an appropriate abstraction that could capture all of them. We presented the vision for this approach in previous work [8]. In this paper, we describe languages and tools for accessing filesystem data with arbitrary structure without resorting to mass conversion. We do not discuss the output of gridfield expressions; results are generally piped into a visualization system for interactive analysis.

The context for our interest in grids is the CORIE Environmental Observation and Forecasting System being developed at the OGI School of Science & Engineering at Oregon Health & Science University [2]. 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.

In a particular run of a simulation, 3-D spatial datasets are produced at regular intervals 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 an 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 changes in either directory structure or file format.

As we see with the CORIE system, logical datasets are not necessarily one-to-one with the files that house them. The physical organization of logical datasets is subject to operational constraints, and can sometimes cause incconvenience for application writers. One dataset may span several files due to file size limits of the OS, for example. Portions of a dataset arriving at different times may be stored in separate files, as are the checkpoint files described above. Several datasets may be stored together in one file to simplify transfer over a network or to share metadata in the filename or path. Access methods for filesystem data should support these situations.

The file or files that make up one logical dataset are not just lumped together on the filesystem, but rather arranged in a potentially intricate directory structure. This directory structure may itself contain important information. For example, the run directories in Figure 4a contain the week and the year. To construct a gridfield representing a weekly average temperature, we would like to extract the week number from the directory name itself, while averaging the temperature values extracted from file content. Access methods should not ignore directory structure information.

The boundary between file name and file content is not inherent in the logical structure of the data, and can change depending on the situation. For example, the directory tree illustrated in Figure 4a has a separate file for each day of the week (per variable). In this case, the day of the week and the week number is not stored within the file, and is therefore inaccessible to tools for reasoning only about a file's content [11, 15]. 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 allow uniform access to data stored in a file or data stored in the surrounding directory structure. Further, access methods should accommodate changes in physical organization without significant programming effort.

Since existing data comes in two forms – embedded in the directory structure and inside files – two physical access methods are required. However, adopting a single logical interface to both forms of data is desirable for conceptual economy.

Imagine we wish to visualize the average temperature near the water's surface for each week in 2004. The gridfield model allows us to perform aggregation and visualization, but first we must collect the appropriate data from the filesystem. Pseudo-code to gather the data might look like this:

```
for each run in 2004:
  for the temperature variable:
    for each timestep:
      for each horizontal surface node:
        for the 1st two vertical depths:
          add the value to the result
```

The boundary between directory-level data and file content data is not apparent in the pseudo-code, nor should it be. We want the system to accept queries in terms of the logical structure, invoking the appropriate physical access method as necessary. To provide such functionality, the system must understand that a "run" corresponds to a directory, that "temperature" and other variables are each stored in a separate file, and that each of these files contain horizontal and vertical dimensions nested within a time dimension. Further, we need the ability to identify the runs for 2004, and the "first two" depths.

To communicate the physical structure of the data repository to the system, users write *schema files* in which they declare relevant *types*. Each type is associated with either 1) a regular expression identifying a set of files, or 2) an expression describing a block of binary data. With an appropriate schema file, we can express the above pseudo-code as follows:

```
run[year=2004].temp.times.horizs.depths[0:2]
```

The result is an array built by copying the values to a sequential block of memory. This array can then be used as part of a gridfield object for further processing. The code to traverse directories, iterate over files, and interpret a file's content efficiently is provided by the system.

The flexibility of accessing directory structure data uniformly with file content data can negatively impact performance. To maintain efficiency, users can generate specialized access programs for a schema to improve performance. For binary files, programs can be further specialized by providing a representative instance for a class of related files. In this case, we can partially process the instance to generate a program tailored for answering queries over other instances of the same form. For example, although the structure of the checkpoint files can be highly variable, files for the same simulation run often have the same structure. By generating a program for one instance, we can efficiently access all related instances.

1.1 Contributions

Our contributions are the following:

- A data model for describing arbitrary binary data.
- A complementary data model for describing data embedded in directory structures and file names. Together, we refer to these two mini-models as the *Native Data Model*.

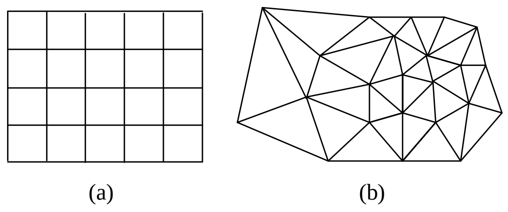

Figure 1. (a) A structured grid. (b) An unstructured grid.

- Access methods derived from the Native Data Model for extracting filesystem data.
- Optimization and code generation techniques to efficiently evaluate extraction queries over native content.
- Evidence of utility from the CORIE project.
- Experimental evidence that suitably optimized generic access methods can perform competitively with hand-coded access methods.

We will present our two-level data model in a top-down fashion. In Section 2, we review the salient features of the gridfield data model. In Section 3, we give examples of schema files for accessing binary file content as well as data encoded in the directory structure. In Sections 4 and 5, we focus on evaluation techniques and experimental results, respectively, for accessing binary content. We end by discussing related work, future work, and some conclusions.

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. Cells of dimension k are called k-cells. A grid has dimension d if it contains no higher dimensional cells. Cells are connected through an explicit or implicit *incidence relation*. For example, a triangle is a 2-cell to which three 0-cells (the vertices) and three 1-cells (the edges) are incident. Every nonempty 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. With this model, we can have geometric attributes x and y bound to the vertices of a triangle, and an area attribute a bound to the 2-cells.

The gridfield model provides an algebra with which to manipulate gridded datasets. Some operations in the algebra are reminiscent of relational operators but equipped to manage topology considerations. These include Restrict and Cross, which are like relational selection and cross product, respectively, but extended to maintain topological invariants [7]. Other operators are specific to gridfields. These include the Bind operator, which adds additional attributes to a gridfield, and the Aggregate operator, which

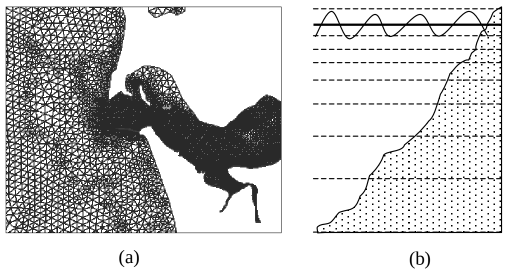

Figure 2. (a) The horizontal unstructured CORIE grid. (b) Illustration of the river's batyhmetry. The shaded region is underground.

```
1: GridField H
2: H.grid.cells[0] = implicit 10
3: H.grid.cells[d] = <array expr>
4: H.x[0] = <array expr>
5: H.y[0] = dot63.y

6: GridField V
7: V.grid.cells[0] = implicit 15
8: V.z[0] = dot63.z

9: GridField G = Restrict(b>v, Cross(H, V))
10: G.temperature(0) = dot63.temp
```

Figure 3. Examples of gridfield assembly syntax.

can map cells of one grid onto another and aggregate the attribute values appropriately.

Grids are said to be *structured* or *unstructured*; our model treats both cases uniformly. The grid in Figure 1a is 2-dimensional structured and the grid in Figure 1b is a 2-dimensional unstructured grid consisting of triangles. Structured grids have implicit topology and can be modeled naturally by multidimensional arrays. Unstructured grids require explicit topology; the connections between cells must be included as part of the representation. Structured grids are easier to represent and admit very efficient algorithms. However, unstructured grids allow more precise modeling of a complex domain such as a coastline. Fewer cells may be required with an unstructured grid, which means less work during processing.

The CORIE system uses a 2-dimensional unstructured grid to model the surface of the water around the mouth of the Columbia River Estuary (Figure 2a). This horizontal grid is repeated at each depth in a 1-dimensional structured grid, creating a 3-dimensional grid. The sloping bathymetry of the river causes many of the grid cells of this 3-dimensional grid to be positioned underground. Figure 2b illustrates the situation. Each dotted line represents a copy of the horizontal surface grid repeated at a particular depth. The shaded region represents the bottom of the river. The horizontal levels towards the bottom contain fewer valid "wet" cells than the levels near the surface. These invalid cells must be removed to correctly interpret CORIE datasets.

Given a horizontal grid H and a vertical grid V, the following expression generates the appropriate 3-dimensional gridfield for the CORIE system and associates a dataset $salt$ for further processing.

$$\mathsf{G} = \mathsf{Bind}(salt, 0, \mathsf{Restrict}(b < z, \mathsf{Cross}(H, V))) \quad (1)$$

The cross product of H and V (the Cross operator) represents the full 3-D domain of the Columbia River estuary and surrounding ocean. The Restrict operator cuts away the portion of the grid positioned underground (the shaded region in Figure 2b). The Bind operator reads in an array named $salt$ and attaches it as an attribute of the grid's 0-cells.

To use gridfields, programmers can construct them "manually" in their code, or they may write and reuse *gridfield declarations*. An example of a gridfield declaration appears in Figure 3. All parts of the gridfield can be described individually as a sequence of values and represented physically as an array. In previous work, we describe different representations of gridfields [7]. In this paper, we use the array-based representation exclusively.

The 0-cells of the grid are usually specified implicitly, using the keyword `implicit`. The declaration in Figure 3 specifies that the grid of G will have 20 nodes when assembled. Cells of higher dimensions are defined as sequences of integer references to 0-cells. A triangle will have three references, and so on.

To bind the attribute x to cells of dimension 0, we use the syntax in line 4 of Figure 3. The placeholder `<array expr>` represents an extraction query (described in Section 3). Here, we omit the query itself for clarity. 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. Other attributes are bound similarly.

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 in the gridfield algebra, as in lines 7 and 8 of Figure 3. Given two gridfields `H` and `V` defined on lines 1 and 5, we construct their *cross product* on line 7. The grid `G` will have 150 nodes and one bound attribute, `temperature`. With these declarations, we have specified the same gridfield as in Equation 1. The details of the gridfield operators can be found in a previous paper [7].

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
                  :
       /grids     /horiz.grd
                  /vert.grd
       /scripts   /do_run.pl
```

b)

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

Figure 4. Simulation results stored on an ordinary filesystem.

3 Native Data Model

In this section we discuss the lower-level data models for accessing data encoded in directory structures and data encoded in binary files.

A filesystem-based data repository is described via a collection of declarations housed in a schema file. There are two types of declarations. *FileType* declarations describe relevant directory structures and allow access to data encoded within file and directory names. *BinaryBlockType* declarations describe the layout of portions of binary files.

3.1 Data From Directory Structures

Scientists frequently store and manage their data using direct filesystem interfaces, using filenames and directory structures equipped with metadata. For example, a dataset available from the National Climate Data Center (NCDC) website is stored in a file named `meso-eta_215_20030803_1800_fff` [12]. The meaning of the date string "20030803" is apparent, but the other fields require some external information to parse. A schema file can store this external information.

Consider the filesystem in Figure 4a. 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) day of the week (1-7).

To access the data embedded in these file names, we can write a *path pattern* in the style of the Unix scanf command.

```
[wk, yr, dy] = /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 generated by evaluating the pattern against a particular filesystem context. Note that the sequence order is determined by the manner in which the directory is traversed by the system calls for a particular OS.

Schema designers declare file types by associating a type name with a path pattern. Given a filesystem context, each variable defined for a file type can be accessed as an array whose type is given by the code of that variable's wildcard placeholder. For example, we can define separate types for the run directories, the salinity data, and the temperature data as in Figure 4b.

To extract data from a filesystem that conforms to this schema, we write a path-like expression navigating through the FileTypes, where the right-most identifier is a variable name.

```
weekly_run.salt63.day
```

This expression returns an "array" of all day values extracted from salt63 files in all weekly run directories. A natural extension to this basic form is to allow XPath-like conditions.

```
weekly_run[week=04].salt63[day<4].day
```

This expression restrict the results to a particular week and particular days. To reflect the array semantics, we can also allow array-style indexing expressions. An expression $\mathsf{name}[n:m]$ returns all elements $\mathsf{name}[i]$ for $n \leq i < m$.

```
weekly_run[0:2].salt63[1:3].day
```

This expression selects the zeroth and first `weekly_runs`, and the first and second `salt63` files. The indexes refer to the ordering of the files as returned by the operating system, not the values of the day variable itself. Note that FileType declarations are not linked to each other; a schema does not prescribe a directory hierarchy. Different queries may express different sequences of FileTypes. Any, all, or none of these sequences may be valid with respect to an actual filesystem. For example, an individual scientist may have several "loose" `salt63` files stored in their home directory. Queries can then reference them directly, without having to first navigate through a run directory.

BinaryBlockTypes, described in the next section, do impose a particular structure for the content they describe. If a query attempts to navigate the binary data in an manner not supported by the schema, an error is raised.

```
BinaryBlock ragged : {
   header : 10c
   n : i
   sizes : n * {
      size : i
   }
   outer : n * {
      inner : size * {
         v:f u:f
      }
   }
}
```

(a)

```
"RAGGED_ARR"

4

1, 3, 1, 2

(0.6,0.4)
(0.2,0.9),(1,3),(1.7,2.6)
(2.1,9.4)
(3.8,8.9),(4.2,1.6)
```

(b)

```
5241 4747 4544 5f41 5252
0000 0400 0000 0100 0000
0300 0000 0100 0000 0200
0000 9a99 193f cdcc cc3e
cdcc 4c3e 6666 663f 9a99
993f cdcc 4c40 9a99 d93f
6666 2640 6666 0640 6666
1641 3333 7340 6666 0e41
6666 8640 cdcc cc3f
```

(c)

Figure 5. A schema file for extracting binary file content and two representations of a file instance. Each color of shading represents a different logical component in the schema.

3.2 Data From Binary Files

Scientists frequently use packed binary encodings of large datasets to preserve space and improve performance. In this section we describe a model of this data and it's interface with the model of the directory structure. We model the content of binary files as a sequence of *components*. Each component is either a *primitive* component with an associated name and typecode, or an *array* component, with a name, a length, and an element type. Our examples will use the primitive typecodes 'f', 'i', and '#c', representing floating point numbers, integers, and character arrays of fixed length '#'. The element type of an array is another sequence of components. In the tree resulting from these nested sequences, each leaf is a primitive component and each internal node except the root is an array component. Arrays are one-dimensional; multidimensionality is captured by nested arrays.

A file format for storing a simple ragged array is described in Figure 5, an instance of the file, in ASCII, is shown in Figure 5b, and a hexadecimal representation is shown in Figure 5c. The root component in Figure 5a is labelled as a `BinaryBlock` and given the name `ragged`. The top-level components, in order of their appearance in the file instance, are a 10-character string named `header`, an integer `n`, an array `sizes` and an array `outer`. Primitive components are written `<name> : <type>`. Array components are written `<x>*(<components>)`, where `x` is a *length expression* evaluating to an integer and `components` is a sequence of components describing structure of the array's elements. The length expression of an array can be an integer literal, a reference to a primitive component appearing earlier in the file, or an arithmetic expression involving one or more of these items. In Figure 5a, the length of the array `outer` is a reference to the component `n`. The element type of the array `outer` is another array component, `inner`, representing a second dimension.

Scanning the file instance in Figure 5c sequentially, we encounter a ten-character header "RAGGED_ARR", then the integer 4, then 4 integers 1, 3, 1, 2, and finally a longer sequence of floating point numbers (Figure 5b).

The length of the array `inner` is a reference to the integer component `size`, which itself is a sub-component of an array `sizes`. For each element of `outer`, a different size is specified by indexing into the array `sizes`. The portion of the instance in Figure 5b corresponding to the component `inner` consists of $1 + 3 + 1 + 2$ pairs of floating point values. Each pair has a value for the component `u` and the component `v`. Since the two arrays `outer` and `sizes` have the same expression for their array length (the expression '`n`'), there is no ambiguity as to which particular element of `sizes` is being referenced.

Programmers can access file data by writing path expressions navigating through the schema. Members of a sequence are accessed by name. 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. In the "return type" column, we write an array of size n with element type τ as `array[n]`(τ). Elements may be of a primitive numeric type, a tuple of named attributes, or another array.

We require that the length of the array appear before the array itself. Without this restriction, the only way to interpret the file would be to have explicit pointers to information deeper within the file. Allowing the length of an array to be defined anywhere earlier in the file is a generalization of other binary description formats that require that the length of a variable-length array be defined immediately prior to the array's elements [11, 15]. Many formats, including netCDF [10], and HDF [4], CORIE's own internal format make use of this generalization.

Unlike path expressions over directory structure data, we currently do not support expressions involving selection predicates such as `outer.inner[v=4].u`. We currently focus on *structural* navigation of binary data rather than *value-dependent* navigation.

Table 1. Example extraction queries and their types.

expression	return type
`ragged.n`	`integer`
`ragged.outer.inner[1]`	`(u:f, v:f)`
`ragged.outer.inner.u`	`array[`$(\sum_{i=0}^{i<n}$ `sizes[i]`$)$`](u:f)`
`ragged.outer[a:b].inner.u`	`array[b-a](array[*](u:f))`
`ragged.outer.inner[a:b:c].v`	`array[(b-a)/c](array[*](v:f))`

4 Evaluating Queries Over Binary Data

In this section we describe the query evaluation engine. We wrote the prototype in Python extended with the numarray library for handling large numeric arrays. C code was emitted for the code generation experiments.

After writing a schema file, users can parse and process it by calling Python functions. Given the schema file shown in Figure 5, the schema parser produces an internal representation of the schema as shown in Figure 6. This representation is called a *schema graph*. Nodes in this graph represent *readers* for the appropriate schema component, and arrows represent pointers. The dashed lines represent pointers used in array length expressions. Nodes in a sequence point to their immediate predecessor in the sequence. Nodes below the root level point to their parent array component, and each array component points to each of its children (bidirectional arrows in Figure 5).

Each primitive reader can report a value given a *context* taken from the query expression. A context corresponds to a sequence of labeled coordinates, one for each level of nested array. For example, a particular value of `v` is specified with two coordinates. The corresponding context is a sequence of (exp, i) pairs, where exp is a label for the length expression of an array component, and i is an integer such that $0 \leq i < \mathsf{eval}(s)$. For example, for the parsed schema in Figure 6, we can ask the `v` reader to produce a value for the context `[('n', 1), ('size', 1)]`, `[('n', 1), ('size', 2)]`, and so on. Bounds checking prevents invalid contexts from being evaluated.

Some readers require less context information than others. A primitive value at the beginning of a file, for example `header` in Figure 5, has no parent and no immediate predecessor, and knows its size statically. Therefore `header.size = 10` and `header.position = 0`. The reader for `n` reports its size as 4, and its position depends recursively on the position and size of `header`. The size of the array `outer` depends on the reader for its length `n`, and the size of each of its children. The position of `v` is computed by adding the *partial sizes* of each level of nested array to the local offset within each element type. The nth partial size of an array is the sum of the sizes of elements 0 through $n - 1$.

Given this simple representation, a naïve program to extract a `v` value from the instance in Figure 5b might make many recursive calls as it asks for positions and sizes of every element preceding the desired value, then repeating the process for each value requested. A better plan is to analyze the schema, looking for sizes and positions that can be computed quickly. For example, we can compute the length of the `sizes` component by reading `n` and multiplying by a constant, since its only child is a fixed-size primitive.

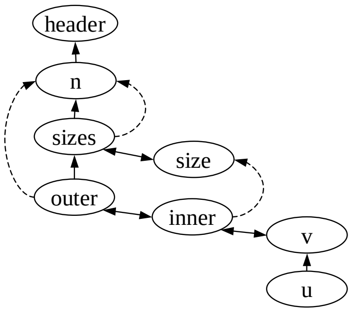

Figure 6. Internal graph representation of a schema file.

In Figure 7, we illustrate the processing steps to evaluate queries. The parser produces a schema graph as discussed. The Static Optimizer transforms the parser's output by instantiating specialized readers for particular situations. Arrays whose children are all primitives have a fixed element size can compute partial sizes quickly. A sequence of primitives at the beginning of the file have fixed positions. Arrays whose length is a primitive at the top-level can cache the value when it is read the first time, and arrays with a literal length need not ever read it at all. The Static Optimizer propagates this information throughout the schema graph to reduce the number of recursive calls.

The output of the Static Optimizer is used by two different modules. The Dynamic Optimizer uses a file instance to partially evaluate the expressions encoded in the schema graph. The dynamic optimizer was originally designed to exploit similarities between related files by preprocessing structural information (i.e., lengths of arrays) shared between them. However, the dynamic optimizer is also used, unaltered, as the first step during ordinary query evaluation. There are two modes of operation for this module: 1) Given a representative file, the system can partially eval-

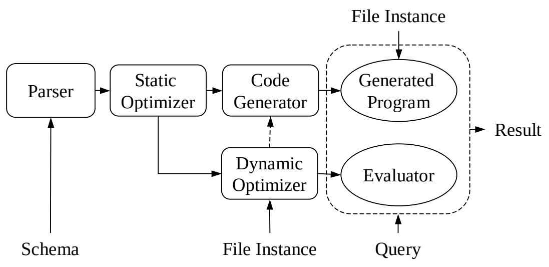

Figure 7. Schematic of the query evaluation engine for binary blocks.

(a)
```
BinaryBlock transformed : {
   header : 6c
   n : i
   sizes : n * {
      size : i
   }
   outer_inner : (2 * sum(size)) * {
      uv:f
   }
}
```

(b)

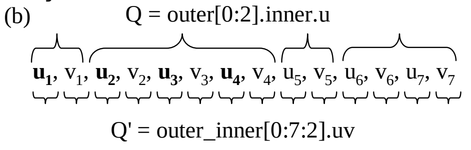
Q = outer[0:2].inner.u
$\mathbf{u_1}$, v_1, $\mathbf{u_2}$, v_2, $\mathbf{u_3}$, v_3, $\mathbf{u_4}$, v_4, u_5, v_5, u_6, v_6, u_7, v_7
Q' = outer_inner[0:7:2].uv

Figure 8. A transformation of a schema to simplify query processing. Q' is a simplified version of Q.

uate the computations encoded in the schema graph. The results can be reused for future queries on future file instances. Currently it is up to the programmer to ensure that any future file instances agree with the representative file. 2) If a query and a file arrive at the same time, the first step is again to apply dynamic optimizations and the second step is to evaluate the query. For mode 1), the programmer indicates which schema components to preprocess. For mode 2), the list of components to preprocess is discovered by analyzing the query. The Code Generator produces access methods for each primitive reader in the schema. For each primitive reader, the module traverses the optimized (and partially evaluated) schema graph as if evaluating a query for all values in all contexts for the reader. For example, for the component `v`, the Code Generator begins to evaluate the query `outer.inner.v`. However, instead of making calls into the internal representation to determine sizes and positions of various components, we emit code for computing the values at run time.

The Code Generator takes input from either the Static Optimizer or the Dynamic Optimizer. Generally, though, we use only the output from the Static Optimizer to avoid over-specializing the generated reader programs. The Code Generator can produces either Python or C code. The Python output was used primarily for testing, since the code structure is iteration-heavy and incurs a significant performance overhead in an interpreted language. The Python output and the C output are effectively isomorphic.

The query processor itself, bordered by the dashed line in Figure 7, may use either generated programs or the internal representation. Currently, the programmer is responsible for managing the generated programs and making them available to the query processor. An appropriate extension is to have the system manage the generated programs, and then automatically select an appropriate program if it is available.

The Evaluator is a function that traverses the optimized internal representation and evaluates the query directly. The most significant feature of this function is its ability to transform and simplify the schema in response to a particular query. By finding and exploiting a simplified schema that can answer the given query, we reduce the amount of work (i.e., number of recursive calls) required to evaluate the query.

In Figure 8a, we show a transformed version of the schema in Figure 5a, along with a transformed query. The modified portions of the schema are in bold. We have blended the two components `u` and `v` into a single component named `uv`. Further, the nested, ragged array has been flattened into a single, longer array. The new length is the sum of all the `size` values times two (since there are two floats, `u` and `v`).

In Figure 8b, we have a sequence of values representing the contents of the schema instance in Figure 5b. The query Q is the original, and Q' is the result of transformation. The values to be returned by both queries are the same; they appear in bold. The brackets represent the grouping structure specified by the original schema (the top brackets) and the transformed schema (the bottom brackets). The advantage of this type of transformation is that we can read a large block of data and then "slice" the resulting array to select the appropriate values. In the original structure, we were forced to loop through the internal representation of the schema, looking up the sizes of the inner dimension. The transformed schema results in fewer iterations. In Section 5, we show that this type of transformation can result in significant performance improvement for our Python-based implementation.

Although we identify and exploit simple instances of this transformation, it remains future work to describe the general form of these simplifications for arbitrarily nested variable-length arrays.

5 Experimental Results

Our experiments use data from the CORIE system [2]. Figure 9 shows the schema for the simulation result files on which we ran our experiments. Logically, these files house a timeseries of three-dimensional datasets bound to a grid. For each simulation run, one of these files is produced for each of 7-10 variables, for each day of simulated time. The sizes of these files range from 35MB for two-dimensional variables such as surface elevation or wind pressure at the surface, to 655MB for horizontal velocity vectors for each three-dimensional point. The entire data repository holds around 5000 simulation runs, with additional runs executed every day.

The header portion of each file, shown in non-bold type, gives time-independent information. The vertical grid gives `z` coordinates (the `zcor` component). The horizontal grid consists of nodes with `x` and `y` coordinates (`nodedata` component) and triangles consisting of three nodes each (the `cells` component). Other time-independent information includes the river's bathymetry (`bathymetry`), represented as an index into the vertical grid for each node in the horizontal grid. This index points to the deepest level for each horizontal node that is still above the bottom of the river. For example, there are 62 levels of depth in the vertical grid for our test case, and 29602 nodes in the horizontal grid. For each of the 29602 nodes, the component `b` gives an integer from 0 to 61 representing one of the 62 depth levels. All levels below the indicated level are underground. Data for underground points are not stored to save space on disk.

The time-varying portion of the file, highlighted in bold type, is a nested array component. There are four levels of nesting. The outermost component is `timestepdata`. For each element, there is a float and an integer representing the number of seconds since midnight and the logical timestep, respectively. Following that, there is an array component called `surfdata`. Similar to the `bathymetry` component in the header, the `surface` component gives an index into the vertical grid. This index represents the surface of the water: the highest level that still holds valid data. However, the water's surface changes over time, we need a surface array for each timestep. After the surface indices, we have the actual simulated data. For each horizontal node, we have a variable number of depth levels, depending on the bathymetry. For each depth level above the bathymetric boundary, we have a vector of length `rank`. Scalar values have a rank of one, velocity values may have a rank of two or three, and so on.

We compare the performance of four representative queries using seven different techniques. The queries are listed in Table 2 and the results are presented in Table 3. Query 1 extracts the first complete timestep of data from the file, copying it to an array in memory. Query 2 access all timesteps, but extracts only a relatively small portion of the horizontal data; 3000 nodes. Query 3 extracts the first two depth levels for every horizontal node, for every timestep. Since we do not know how many depth levels exist at each node until we read the file's header, we cannot derive the result size statically; some nodes may have as few as one depth level. Query 3 returned the largest result size in practice, at 5.6 million floating point numbers. This query also caused the largest problem for our software. Query 4 extracts the surface information for timesteps 30 through 45. This query does not directly involve any variable-length arrays and is therefore easier to compute for most techniques. This query also had the smallest result size.

```
magic : 48c, version_string : 48c
start_time : 48c, variable_nm : 48c
variable_dim : 48c, nsteps : i, timestep : f
skip : i, rank : i, idim : i,
vpos : f, zmsl : f, levels : i
zcor : levels * {
  z : f
}
nodes : i
elements : i
nodedata : nodes * {
  x : f,  y : f,  h : f,  bathymetry : i
}
cells : elements * {
  nodeids : 3 * { id :i }
}
# time-varying data
timestepdata : nsteps * {
  tstamp : f, tstep : i
  surfdata : nodes * { surf : i  }
  horizontal : nodes * {
    depths : (levels - bathymetry + 1) * {
      vector : rank * { value : f }
    }
  }
}
```

Figure 9. A BinaryBlock describing the CORIE simulation output.

The simplest technique we tested was to use the internal representation produced by the static optimizer directly (experiment (a) in Table 3). While this basic tool worked well on small files during testing and development, it struggled on the large files of the CORIE system. After 500 seconds, we stopped the experiments. The use of the Dynamic Optimizer leads to improved performance (experiment (b)). The Dynamic Optimizer prefetches information from the header during query evaluation, and is therefore able to simplify or eliminate many computations.

Experiment (c) in Table 3 uses a small class of schema and query transformations (see Section 4). These transformations act as shortcuts, allowing us to read large blocks of data from the file and then "slice" them to extract the appropriate values. For example, Q1 returns an entire timestep. The system can detect that there is no need to iterate over

Table 2. Tested queries with result sizes.

label	query	result size
Q1	timesteps[0:1].H.V.vector.v	3.3MB
Q2	timesteps.H[2000:5000].V.vector.v	16.3MB
Q3	timesteps.H.V[0:2].vector.v	22.7MB
Q4	timesteps[30:45].surfdata.surf	1.77MB

each horizontal node and each vertical level; it can simply compute the size s of a timestep from the header information, seek to the beginning, and read all s bytes. The result is dramatic improvement in performance. The effect is especially pronounced due to our choice of technology. The Python language can perform well when expensive routines are pushed down into the compiled C code underpinning the language. Reading and slicing arrays are examples of these fast operations.

Note that for Q3, we are unable to identify an appropriate transformation. Since the vertical component has a variable length, there is no simple pattern we can use to extract the data values we want. We must access much more data to navigate through the file. The hand-coded reader (experiment (g) in Table 3) is able to evaluate this query efficiently by precomputing cumulative sizes of the variable-length arrays and striding through them as necessary. It remains future work to identify the general form of this technique.

Experiment (d) executes identical code to Experiment (c). For this case, we subtract the time spent prefetching and optimizing. The rationale is that for large classes of similar files, this work need only be done once rather than for each file.

Experiments (e) and (f) use generated C programs to compute the results. The first is a program generated directly from the static optimizer's output. We performed these experiments to see whether a compiled language would outperform the Python routines even without significant optimization. The results show that traversing these large files without guidance is prohibitively expensive even for a compiled C program. The specialized generated program is created from the results of the dynamic optimizer and is specialized for a particular class of file instances. The rationale is that many of these specialized readers could be generated and stored by the system. When planning evaluation of a query, the specialized programs could be considered as fast alternative access methods. Unfortunately, these generated programs do not perform as well as expected. The reason is that hte transformation optimizations that gave good performance in experiments (c) and (d) are not incorporated in the generated code. In ongoing work we are improving the quality and performance of the generated programs. Note that the specialized generated program exhibits stability; it was able to evaluate Q3 without incurring the same magnitude of penalty as the other approaches.

Table 3. Response times in seconds by query.

Experiment	Q1	Q2	Q3	Q4
(a) static	>500	>500	>500	>500
(b) dynamic	28	138	284	8.64
(c) dynamic + transf.	0.83	1.0	86	0.85
(d) spec. + transf.	0.02	0.24	85	0.26
(e) generated, gen.	65	104	>500	3.2
(f) generated, spec.	4.7	22	32	2.6
(g) by hand	0.02	0.5	0.6	0.02

6 Related Work

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 prolific. Legacy systems are built from efficient but 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 [13] 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 retrofit a gridfield interface onto in situ data.

Batory gave a taxonomy of record-oriented file structures used by commercial databases in terms of fields and pointers [3]. Our work similarly provides a description of file structures in terms of arrays.

The Binary Format Description Language (BFD) [11] 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 [15] 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 most recent version added support for nested arrays, but only if their length is fixed.

The External Data Representation standard (XDR) [14] is a data description language focused on machine-level number representation issues. Variable-length arrays in

XDR must have homogeneous elements (i.e., their elements cannot themselves be variable-length), and their lengths must be encoded directly prior to the first element. Further, XDR obviously does not describe directory structures, a feature critical for datasets that span multiple files.

Code generation has been used by the database community to improve performance. The EXODUS and later, the Volcano optimizer generators [5, 6] processed algebraic transformation rules and produced compiled code to apply the rules efficiently while searching for a query plan. We generate access methods themselves rather than a system for choosing an access method.

7 Future Work and Conclusions

We have identified several areas for ongoing work.

Constraints. We would like to include constraints as part of the schema language for binary block types. Constraints could be used to validate files before processing. Constraints can also be used to represent optional components. Some condition must be true if the optional component exists; the condition will be false if it does not.

Alternative schemes for the same binary content. We would like to add explicit support for multiple schemas over the same binary content. We derive new schemas internally during dynamic optimization, but we cannot reason about user-specified alternative schemas. Schemas specialized to particular access plans can be supplied by the user and considered during query evaluation.

More expressive queries. We plan to add support for value-based predicates on arrays, in order to push some gridfield processing into the native data model subsystem. We are also working on "output schemas" that allow files to be restructured without programming. Given 1) an input schema and 2) an output schema with a subset of the input's declarations, we aim to allow instances of the input schema to be automatically and efficiently transformed into instances of the output schema. The output of gridfield expressions could be represented similarly.

We advocate in situ processing of large scientific data repositories. Converting Terabytes of data to support new data models is infeasible, and continuously writing access methods for changing formats is time consuming. Our results show that although file formats with high variability can be expensive to process without programmer guidance, hand-coded access methods can be replaced with generic or generated access methods. We recognize that a logical dataset can often span multiple files in practice, and that the directory structure and filename can encode part of the dataset's structure. These techniques can facilitate data sharing between research groups and institutions with heterogeneous data formats.

8 Acknowledgements

We would like to thank the CORIE science team for their input and support. This work was was supported by NSF ITR Award No. ACI-0121475

References

[1] Northwest Association of Networked Ocean Observing Systems. http://www.nanoos.org.

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

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

[4] N. C. for Supercomputing Applications (NCSA). HDF5: API specification reference manual. http://hdf.ncsa.uiuc.edu/, 2004.

[5] G. Graefe and D. J. DeWitt. The exodus optimizer generator. *SIGMOD Rec.*, 16(3):160–172, 1987.

[6] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218. IEEE Computer Society, 1993.

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

[8] B. Howe and D. Maier. Algebraic manipulation of scientific datasets. *IEEE Data Eng. Bull.*, 27(4):30–37, 2004.

[9] B. Howe, D. Maier, and A. Baptista. A language for spatial data manipulation. *Journal of Environmental Informatics*, 2(2), December 2003.

[10] H. L. Jenter and R. P. Signell. Netcdf: A public-domain-software solution to data-access problems for numerical modelers. Unidata, 1992.

[11] J. Myers and A. Chappell. Binary format description language. Technical report, Pacific Northwest National Laboratory, 2003.

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

[13] 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.

[14] R. Srinivasan. XDR: External data representation standard, RFC 1832. Technical report, Sun Microsystems, 1995.

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