Algebraic Manipulation of Scientific Datasets

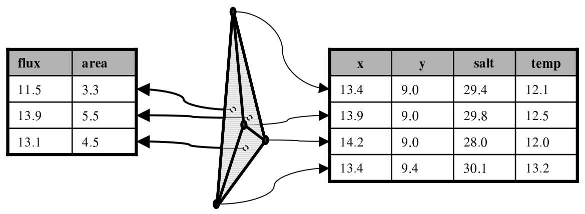
Bill Howe
OGI School of Science & Engineering at
Oregon Health & Science University
Beaverton, Oregon
bill@cse.ogi.edu

David Maier
OGI School of Science & Engineering at
Oregon Health & Science University
Beaverton, Oregon
maier@cse.ogi.edu

Abstract

We investigate algebraic processing strategies for large numeric datasets equipped with a possibly irregular *grid* structure. Such datasets arise, for example, in computational simulations, observation networks, medical imaging, and 2-D and 3-D rendering. Existing approaches for manipulating these datasets are incomplete: The performance of SQL queries for manipulating large numeric datasets is not competitive with specialized tools. Database extensions for processing multidimensional discrete data can only model regular, rectilinear grids. Visualization software libraries are designed to process gridded datasets efficiently, but no algebra has been developed to simplify their use and afford optimization. Further, these libraries are data dependent – physical changes to data representation or organization break user programs. In this paper, we present an algebra of *gridfields* for manipulating both regular and irregular gridded datasets, algebraic optimization techniques, and an implementation backed by experimental results. We compare our techniques to those of spatial databases and visualization software libraries, using real examples from an Environmental Observation and Forecasting System. We find that our approach can express optimized plans inaccessible to other techniques, resulting in improved performance with reduced programming effort.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004



flux	area
11.5	3.3
13.9	5.5
13.1	4.5

x	y	salt	temp
13.4	9.0	29.4	12.1
13.9	9.0	29.8	12.5
14.2	9.0	28.0	12.0
13.4	9.4	30.1	13.2

Figure 1: Datasets bound to the nodes and polygons of a 2-D grid.

1 Introduction

Many scientific datasets can be characterized by the topological structure, or *grid*, over which they are defined. For example, a timeseries 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.

These datasets consist of data tuples bound to the *cells* of a grid. A grid may possess cells of many dimensions; data can be associated with the nodes (0-cells), edges (1-cells), polygons (2-cells), and so on. Figure 1 shows a 2-D irregular (non-rectilinear) grid with two datasets bound to it. Geometric coordinates x and y are associated with the nodes of the grid, as are salinity and temperature values. Area and flux values are associated with each polygon. The grid structure consists of topological information only – generic cells, and incidence and adjacency relationships between cells that are invariant with respect to a particular geometric embedding. A geometric embedding in this example is captured by associating coordinate pairs with the nodes. As these datasets are manipulated and transformed, both the grid and the associated data must be updated in tandem; new grid-aware operators are required. Such operators must handle both regular grids encoded as multidimensional arrays and irregular grids that explicitly enumerate their cells. Since these datasets tend to be large, efficiency is paramount.

Gridded datasets are especially common in scientific and engineering domains. The context for our interest in gridded data is CORIE [1], an Environmental

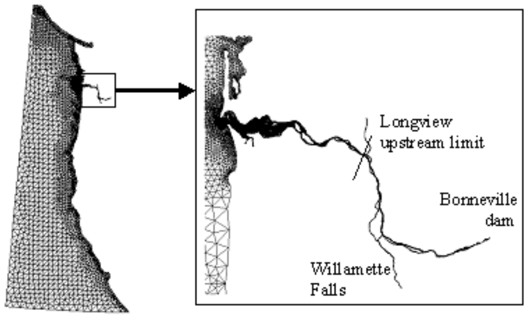


Figure 2: The CORIE grid, extending from the Baja peninsula to Alaska.

Observation and Forecasting System designed to support scientific and industrial interests in the Columbia River estuary. The CORIE system both measures and simulates the physical properties of the estuary, generating 5GB of data and thousands of *data products* for each simulation run, including visualizations, aggregated results and derived datasets. The data products are consumed for many purposes, including salmon habitability studies and environmental impact assessments. Figure 2 shows the CORIE domain. The horizontal irregular grid extends from the Baja peninsula up to Alaska to capture the large-scale influences of the Columbia River. The Columbia River estuary and the ocean waters around the mouth of the river (inset) have a very high density of grid elements, to also capture local hydrodynamic processes. Using a vertical grid to discretize the depth of the river along with this large horizontal grid, a 3-D grid can be generated. Time represents a fourth dimension.

Traditional Approaches. Database languages for processing multidimensional arrays have been proposed [2, 13], but multidimensional arrays cannot directly model irregular grids, such as those used in the CORIE system. A facility to manipulate both regular (rectilinear) grids and irregular (non-rectilinear) grids is missing. Additionally, representing different datasets bound to the nodes, edges, and faces of the same grid is difficult with multidimensional arrays. Raster GIS are similarly unable to model irregular grids precisely.

Relational databases extended with spatial types can model irregular grids, but have several weaknesses. Explicit foreign keys and redundant geometric coordinates[1] can more than double database size. With 5-20GB generated each day, even relatively inexpensive disk space is at a premium. Transfer times into and out of the database are excessive. Using the bulk load facility of Postgres [22], loading one timestep of one variable (about 800,000 floats) takes over one minute. With six primary variables and 96 timesteps per day, the load time approaches the time to generate the data in the first place on a similar platform. Retrieving data from the database involves copying tuples to fast, memory-resident structures such as arrays. When retrieving numeric datasets from a relational database, tuples are usually converted to arrays at the client, incurring an "impedance mismatch" penalty. The scale of scientific datasets makes the performance issues associated with impedance mismatch more pronounced [23]. In Section 3.5, we review modeling challenges stemming from storing gridded datasets in relational databases.

[1]Coordinates of a node are repeated everywhere the node is referenced.

Visualization libraries such as the Visualization Toolkit (VTK) [19] provide efficient grid processing, but the routines are highly data dependent and therefore quite brittle. The library functions also exhibit complex semantics, making algebraic properties difficult to derive if they exist. We discuss these issues in more detail in Section 3.5.

Our Approach. These issues led us to seek a technology that 1) efficiently generates relevant data products, 2) reduces programming effort to design and implement new data products by allowing manipulation of grid structures directly, 3) integrates neatly with client tools, especially rendering tools for visualization, and 4) manages topology considerations for both regular and irregular grids transparently.

Our approach has been to devise and implement an algebra specially suited for manipulating gridded datasets, extending previous work [9]. Our algebra consists of grids, *gridfields*, and operators over these structures. A gridfield represents the association of a dataset with a grid. Several gridfields may share the same grid; indeed this eventuality allows algebraic identities important for optimization (see Section 6). Our data model distinguishes topological information from geometric information, handling geometry as ordinary data attributes. The separation of topology and geometry allows multiple geometric embeddings to be handled simultaneously, unlike other data models proposed, e.g., for scientific visualization [5, 8, 15]. Some of our operators are analogous to those of relational algebra, but extended to correctly handle the grid structure. Other operators are specific to gridfields.

Contributions. We extend previous efforts at devising scientific data models [3, 5, 8, 9, 18] by developing algebraic optimizations at both the logical and physical levels. We contribute a data model and implementation that satisfies the goals above. Specifically:

1. The data model captures regular and irregular grids uniformly.

2. The operators manipulate grid structures directly, avoiding the complexity associated with encoding grids as assemblies of arrays.

3. The design is well-aligned with client visualization and analysis tools.

4. Our operators admit algebraic identities and consequent optimization techniques unique to gridfields.

5. We have tested our data model and implementation on real applications; we present results from the CORIE simulation system.

In this paper, we discuss the gridfield model, then describe data representation, operator implementation, and algebraic optimization of gridfield *recipes*, a form of query plan. Results are validated via experimental comparisons with existing approaches.

2 Related Work

The database community has given multidimensional discrete data (MDD) significant attention over the past decade. Query languages based on multidimensional arrays [12, 13, 25] have been developed, but arrays are not the correct abstraction for general grid manipulations. Multidimensional arrays capture only rectilinear grids. If, as in the CORIE system, cells in a particular grid may be triangles, quadrilaterals, or a mix of cell types, the grid structure is awkward to encode using arrays. The interpretation of an assembly of arrays as an irregular grid is left to the application, undermining data independence. Further, we encounter multiple datasets bound to the same grid, but perhaps to cells of different dimension. Using arrays, the relationship between these datasets is lost; each must use its own distinct "spatial domain" [2]. Finally, the topology suggested by these grids is always implicit, making it difficult to separate geometry from topology. This capability is required when attempting to support two geometric embeddings of the same grid simultaneously, e.g., into different coordinate systems.

Arrays are appropriate at the physical level as an internal data structure, and efficient storage and retrieval techniques are relevant and complementary. Database systems supporting multidimensional numeric arrays [6, 25] have contributed results in this area.

Several higher-level data models for scientific data have been proposed that capture both regular and irregular grids, and some separate topology from geometry [3, 5, 8]. However, algebraic manipulation of grid structures is not supported and experimental results are not reported.

Others have demonstrated that relational databases do not scale up to handle large scientific datasets [16, 21]. One proposed solution is to treat scientific datasets as external data sources, and access them using the SQL standard for management of external data (SQL-MED) [14]. Papiani et al. [17] report some success applying the standard to manage turbulence simulations.

Designers of spatial database systems are becoming aware that topological "connection" information can be as important as geometry for modeling and query processing. ESRI's ArcGIS version 8.3 [7] includes topology information modeled as integrity rules. Users can express the rule that every polygon representing a building must be explicitly connected to a line segment representing a road. ESRI's product also supports raster data manipulation using a Map Algebra, but irregular grids are difficult to model precisely as raster data. Laser-Scan has produced a topology-enabled GIS extension for Oracle called Radius [24]. They allow nodes to be snapped together to express topological relationships independently of geometric embeddings. However, there is no notion of a manipulable gridded dataset, and therefore, our Goals 2 and 3 are not met.

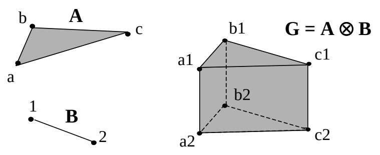


Figure 3: The cross product of two simple grids.

3 The Gridfield Algebra

Grids are constructed from sets of k-dimensional *cells*. We refer to a cell of dimension k as a k-cell, following the topology literature [3]. Intuitively, a 0-cell is a point, a 1-cell is a line segment, a 2-cell is a polygon, and so on. These geometric interpretations of cells guide intuition, but a grid does not explicitly indicate its cells' geometry.

Nodes and Cells. We will refer to a 0-cell as a *node*. A node is named, but is otherwise featureless. A k-cell c is a sequence of nodes $(c_0, c_1, \ldots, c_n)$, where $k < n$. For example, a 1-cell must refer to at least two nodes, but can refer to more. Let $N(c)$ be the nodes of a cell c viewed as a set. We say a cell c is *incident* to a cell d if $N(c) \subseteq N(d)$. The dimension k of a k-cell c is written $\mathsf{dim}(c)$.

Node sets and the incidence relationship are sufficient to encode some topological relationships. Two cells are *adjacent* if they share nodes but neither is incident to the other. Two cells are *connected* if they appear in the transitive closure of the adjacency relationship. A topological *distance* measure can be defined by counting the number of cells traversed through the adjacency relationship to reach another cell. Note that containment and overlap are geometric relationships, since they depend on a particular geometric embedding.

Namespaces. Nodes are referenced with respect to a *namespace*. For example, nodes can be named by their physical position within an array. Let L be a set of labels and C be a set of nodes. A namespace is a 1-1 function $h : C \to L$. Cell equality is only defined with respect to a particular namespace. Cells in different namespaces are assumed to be unequal.

3.1 Grids

A grid is a sequence of sets of cells, $[G_0, G_1, G_2, \ldots, G_d]$, where each set G_i contains cells of dimension i. A non-empty grid must have a non-empty set of 0-cells (nodes). The dimension of a grid G is the greatest i such that G_i is non-empty. A grid's dimension is written $\mathsf{dim}(G)$. In Figure 1, the grid has four 0-cells, six 1-cells, and three 2-cells, and it therefore has dimension 2. Note that a d-dimensional grid G must have a non-empty component G_d, but may have an empty component G_i for $0 < i < d$.

This definition is very general; a grid may be a collection of unconnected polygons for GIS data, a set of scattered points for values of a random variable, or a well-connected graph modeling the truss structure of a bridge. The grids in our application are used to discretize the Columbia River estuary, for solving the 3-D transport equations via a finite-element method.

We can define set-like operations on grids with respect to a namespace to test cell equality. The intersection of two grids G and F is the component-wise intersection of the sets G_i and F_i. That is, $G \cap F = [G_0 \cap F_0, G_1 \cap F_1, \ldots]$. Union and difference can be defined similarly.

Grids must be *well-formed*; no cell in G_i may reference a node not in G_0, for $0 < i \leq \mathsf{dim}(G)$. Operations on grids must preserve well-formedness. If nodes are removed from a grid, then cells that reference those nodes must also be removed.

Cross Product. The cross product of two grids generates a higher-dimensional grid based on cross products of their constituent sets. The *node product* of two 0-cells a and b is written ab. The result is a 0-cell x in a new namespace. The *cell product* of a cell $c = (c_1, c_2, \ldots, c_n)$ and a cell $d = (d_1, d_2, \ldots, d_m)$, written $c \times d$, is a cell e with $\mathsf{dim}(e) = \mathsf{dim}(c) + \mathsf{dim}(d)$ such that $e = (c_1d_1, c_1d_2, \ldots, c_1d_m, c_2d_1, c_2d_2, \ldots, c_2d_m, \ldots, c_nd_1, c_nd_2, \ldots, c_nd_m)$.

Figure 3 shows an example of the cross product. The cross product of grids A and B contains six 0-cells, nine 1-cells, five 2-cells, and one 3-cell. The 3-cell is the interior of the prism, the 2-cells are the three rectangular faces and the two triangular bases, the 1-cells are the edges, and the 0-cells are the vertices.

We capture all these cases using the set-theoretic cross product of the components of the grids A and B. For example, the 3-cell prism in G is generated by sweeping the triangle of A through a third dimension defined by the line segment of B. This construction can be expressed as the cross product of the 2-cells of grid A (A_2) and the 1-cells of grid B (B_1). The rectangular faces are generated by sweeping the 1-cells of A through the space defined by the 1-cell of B. Again, the construction is expressed as the cross product of A_1 and B_1. More precisely, the cells of G are given by

$$\begin{aligned}
G_0 &= A_0 \times B_0 \\
G_1 &= (A_1 \times B_0) \cup (A_0 \times B_1) \\
G_2 &= (A_2 \times B_0) \cup (A_1 \times B_1) \\
G_3 &= A_2 \times B_1
\end{aligned}$$

Evaluating these expressions, we obtain

$$\begin{aligned}
G_0 &= \{a1, b1, c1, a2, b2, c2\} \\
G_1 &= \{(a1, b1), (b1, c1), (c1, a1), (a2, b2), \\
&\quad (b2, c2), (c2, a2), (a1, a2), (b1, b2), (c1, c2)\} \\
G_2 &= \{(a1, b1, c1), (a2, b2, c2), \\
&\quad (a1, b1, b2, a2), (b1, c1, c2, b2), (c1, a1, a2, c2)\} \\
G_3 &= \{(a1, a2, b1, b2, c1, c2)\}
\end{aligned}$$

In general, let $A = [A_0, A_1, \ldots, A_a]$ and $B = [B_0, B_1, \ldots, B_b]$ be grids. The *cross product* of A and B, written $A \otimes B$, is a grid $[G_0, G_1, \ldots, G_d]$ such that $G_k = \bigcup_{j=0}^{k} A_j \times B_{k-j}$ for $0 \leq k \leq a + b$.

We have used the cross product operator frequently in expressing the data products of the CORIE system. The 3-D CORIE grid is the cross product of a 2-D horizontal grid and a 1-D vertical grid. The time dimension can be incorporated with another cross product. Note that simpler rectilinear grids can be modeled as the cross product of two 1-D grids. By commuting other operations through the cross product, we can reduce its complexity or remove it altogether. Tools that do not provide an explicit cross product operator do not have access to these optimizations, as we shall see.

3.2 Gridfields

When data is bound to a grid, the grid becomes a gridfield. Formally, a *gridfield* $\mathbf{G}$ is a triple (G, k, f), where G is a grid, k is a non-negative integer, and f is a function $G_k \to \tau$ for some type τ. The integer k is called the *rank* and can be extracted from a gridfield $\mathbf{G}$ by writing $\mathsf{rank}(\mathbf{G})$. The *type* of a gridfield is the return type τ of its function component f, written $\mathbf{G} : \tau$. We will generally use only primitive numeric types and tuples of primitive numeric types as return types.

Earlier we used a trussed bridge as an example of a grid. Gridfields defined over such a grid might return the net force at each node, or the linear force along each truss. Gridfields capture both cases naturally by binding data to 0-cells or 1-cells, respectively. Images can be viewed naturally as a gridfield defined over 2-cells of a rectilinear grid. We can also model unstructured sets as a gridfield over a grid consisting solely of 0-cells.

To support multiple geometric embeddings, geometric information is modeled as ordinary data values bound to the cells of a grid. A simple example is a

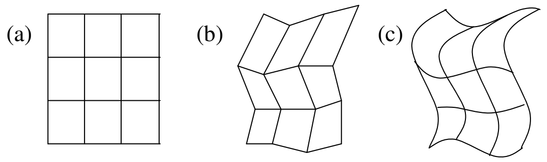


Figure 4: Three different geometric realizations of the same topological grid.

2-D grid with a gridfield binding (x, y) pairs to the nodes, which embeds the grid in Euclidean space. Additional coordinate systems can be captured through additional attributes. Many models [3, 8, 19] distinguish geometric attributes from other data, consequently requiring two versions of common operations: one for geometric attributes and one for ordinary attributes. Non-standard geometries that are not anticipated by the system designer are left unsupported. For example, the curvilinear grid shown in Figure 4 requires interpolation functions to be associated with each k-cell to specify how the cell curves in a geometric space. Our model can express such an embedding. Further, our model captures the topological equivalence between all three grids in Figure 4. Systems commonly use geometry as the identifying feature of a grid, thereby obscuring this equivalence.

3.3 Operators

The operators for manipulating gridfields must correctly handle both the grid and the bound data values. Some operators we define are analogous to relational operators, but grid-enabled. For example, our restrict operator filters a gridfield by removing cells whose bound data values do not satisfy a predicate. However, restrict also ensures the output grid is well-formed, and that cells of all dimensions are passed along. Other operators are novel, such as aggregate. The aggregate operator maps data from one grid onto another and then aggregates the values.

Bind. The bind operator constructs a gridfield from a grid G, an integer k, and a function $f : G_k \to \tau$. Bind allows us to perform operations on grids prior to associating data. We can therefore construct a topologically regular grid via cross product, but then bind irregular geometry functions to it, as in Figure 4b. The bind operator is rather simple at the logical layer, but at the physical layer, the bind operator is important for correct and efficient processing (see Section 6).

Restrict. The restrict operator behaves like a relational select, except that the output must be defined on a well-formed grid. If $\mathsf{rank}(\mathbf{G}) = 0$, then cells that reference deleted nodes must themselves be deleted. Note that if $\mathsf{rank}(\mathbf{G}) = k > 0$, then only k-cells need be removed; the grid is guaranteed to be well-formed. Formally, let $\mathbf{A} = (A, k, f)$ be a gridfield, with $f : A_k \to \tau$. Let p be a predicate over data values of type τ. Then $restrict(p, \mathbf{A})$ is a gridfield (G, k, f). For the case $k > 0$, $G = [G_0, G_1, \ldots, G_n]$, where $G_k = \{c \mid c \in A_k \ , \ p \circ f(c) = \text{true}\}$ and $G_i = A_i$ for all $i \neq k$ and $i \leq \mathsf{dim}(A)$. The predicate p is used to filter out some cells of dimension k, but all other cells are included in G. For the case $k = 0$, G_k is defined as before but we must remove any cells that reference deleted nodes. Thus, $G_i = \{c \mid c \in A_i \ , \ \forall v \in N(c).p \circ f(c) = \text{true}\}$

Merge. The merge operator computes the intersection of two grids and retains data values defined over this intersection. If the input gridfields are of different ranks, then the data values of the second argument are discarded. In this case, merge is not commutative. Formally, let $\mathbf{A} = (A, i, f)$ and $\mathbf{B} = (B, j, g)$ be gridfields. Then $merge(\mathbf{A}, \mathbf{B})$ produces a gridfield $\mathbf{G} = (A \cap B, k, h)$. For the case $i = j$, $h(e) = \langle f(e), g(e) \rangle$. For the case $i \neq j$, $h(e) = f(e)$.

Cross Product. The cross product operator for gridfields builds on the cross product operator on grids. Let $\mathbf{A} = (A, i, f)$ and $\mathbf{B} = (B, j, g)$ be gridfields. The cross product of $\mathbf{A}$ and $\mathbf{B}$, written $\mathbf{A} \otimes \mathbf{B}$, is a gridfield $\mathbf{G} = (A \otimes B, i + j, h)$, where $h(c) = \langle g(c), f(c) \rangle$.

This definition can result in a gridfield with a partial function if there are multiple ways to form cells of intermediate dimension in the cross product. To avoid this complication in the current implementation, we force the function h to be total by requiring that either $\mathsf{rank}(\mathbf{A}) = \mathsf{rank}(\mathbf{B}) = 0$, or that $\mathsf{rank}(\mathbf{A}) = \mathsf{dim}(A)$ and $\mathsf{rank}(\mathbf{B}) = \mathsf{dim}(B)$.

Aggregate The aggregate operator maps a *source* gridfield's cells onto a *target* gridfield's cells, and then aggregates the data values bound to the mapped cells. The behavior of aggregate is controlled by two functions, an *assignment* function and an *aggregation* function. The assignment function associates each cell in the target grid with a set of cells in the source grid. To perform the assignment, the function may use topological information only (e.g., a "neighbors" function that identifies incident cells), or it may use the attributes of the two gridfields (e.g., an "overlaps" function that uses geometry data).

To illustrate a simple use of aggregate, consider a timeseries of temperature values for a particular point in the river. We discretize the time dimension using a 1-D source grid S, as shown in Figure 5a. One use of the aggregate operator is to perform a "chunking" operation to coarsen the resolution of the grid. The assignment function maps each node in the target grid T to a set of n nodes, the chunk, in the source grid S (Figure 5b). The aggregation function can then, say, average the n nodes to obtain single value (Figure 5c).

We could also pass a "window" function as the assignment function to perform a smoothing operation. The target grid and the source grid are the same in that case. For target node i, the window function assigns source nodes $[i-k, i-k+1, \ldots, i, i+1, \ldots, i+k]$.

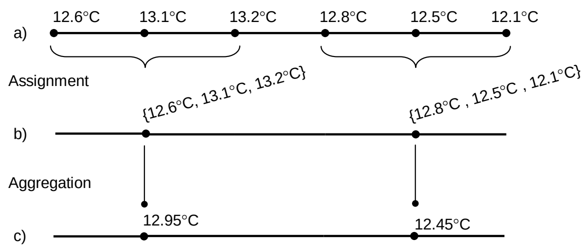


Figure 5: (a) A 1-D gridfield returning temperatures. (b) Assignment to the target grid T. (c) Aggregation using arithmetic mean.

The aggregation function could be anything, but for smoothing, an arithmetic or weighted mean seems appropriate. We have used a 1-D example for illustration, but multidimensional window and chunking functions are common.

Formally, let $\mathbf{T} = (T, k, f)$ and $\mathbf{S} = (S, j, g)$ be gridfields, where $f : T_k \to \alpha$ and $g : S_j \to \beta$. Let m be a function $m : T_k \to \mathcal{P}(S_j)$. Let $a : \mathcal{P}(\beta) \to \gamma$ be a function for some type γ. Then $aggregate(\mathbf{T}, m, a, \mathbf{S})$ produces a gridfield $\mathbf{G} = (T, k, h)$ where $h(c) = a(\{g(e) \mid e \in m(c)\})$.

3.4 Benefits

We summarize the benefits of our data model:

- Grids are first-class and of arbitrary dimension.
- Grids can be shared between datasets.
- Geometry is modeled as data, exposing topological equivalences between geometric interpretations; e.g., different coordinate systems.
- Data can be associated with cells of any dimension, avoiding ambiguities arising from associating, for example, cell areas with nodes.
- The data model captures irregular grids directly, but the cross product operator expresses the regularity of rectilinear grids.
- The aggregate operator is extensible, allowing application-specific assignment and aggregation functions.
- The operators obey algebraic identities enabling optimization (see Section 6).
- Client programs can process grids without intricate array manipulations.

3.5 Detailed Example

Many of the CORIE datasets are defined over a 3-D grid constructed as the cross product of a 2-D irregular grid and a 1-D grid. The 2-D grid H describes the domain parallel to the earth's surface, a *horizontal* orientation. The 1-D grid V extends in a *vertical* direction perpendicular to the earth's surface. These grids are illustrated in Figures 2 and 6, respectively.

Although the simulation code operates over the grid formed from the cross product of H and V, the output datasets are produced on a reduced grid. To see why, consider Figure 6. The shaded region illustrates the bathymetry of the river. The horizontal grid is defined to cover the entire surface of the water. Below the surface, some nodes in the full 3-D cross product grid are positioned underground! The simulation code outputs only valid, "wet," data values to conserve disk space. Therefore, we must define this "wet" grid to obtain an adequate description of the topology of the data. The bathymetry data can be modeled as a gridfield over the horizontal grid H, associating a depth with each node. To filter out nodes in the product grid G that are deeper than the river bottom, we need to compare the node's depth (bound to V) with the bottom depth (bound to H). In the following, we will refer to a rank 0 gridfield $\mathbf{H}$ constructed from the 2-D horizontal grid H and attributes x, y, b. The attribute b captures the river's bathymetry at a particular location. We will also refer to a rank 0 gridfield $\mathbf{V}$ constructed from the 1-D vertical grid V and an attribute z.

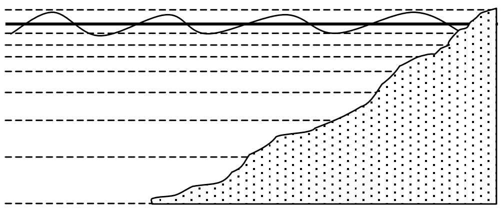


Figure 6: The vertical grid and the river's bathymetry in the CORIE domain.

The task is to construct the grid over which the simulation outputs are defined, bind a dataset to it, and visualize the results. The recipe for this task is shown in Figure 7. Each gray oval is an operator in our algebra. The unfilled oval at the right represents a client task: render the grid as an image. The recipe begins at left with the $\mathbf{H}$ and $\mathbf{V}$ gridfields. The cross product operator produces a different, 3-D gridfield. After using restrict to filter out the river bottom, we have our "wetgrid" (at the point labelled in Figure 7). After binding a salinity dataset to the wetgrid, we restrict the grid to a user-supplied region. The term "region" is shorthand for a bounding-box condition involving x, y, and z.

Using a Relational Database. Our initial attempt to manage the CORIE datasets was to load them into a relational table and manipulate them using SQL. The first task is to devise a schema that captures both the grid and the data. One method is to store each logical gridfield as a separate relation: one attribute stores the cells to which the data is bound, while the other attributes store the bound data. A problem with this approach is that each scalar dataset bound to a grid is modeled as an attribute of a relation. New datasets are generated daily. To capture each new dataset, we can either extend the existing table with an additional attribute or add the new dataset

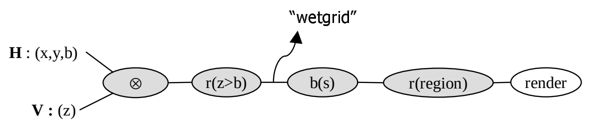


Figure 7: A recipe for visualizing a 3-D CORIE dataset.

as a separate table. Either way, we are changing the database schema daily, making robust queries difficult to write.

A better strategy for modeling grids and gridfields using an RDBMS is to allow any number of datasets to be bound to the same grid. The relation grid stores metadata about the grid. Each grid is associated with a number of cells of varying dimension, stored in the relation kcell. Tuples in the values relation are bound to cells using foreign keys, perhaps integers. Now no schema changes are required to insert new datasets, but binding a particular dataset to its grid involves a join between the kcells relation and the values relation using the ordinal. Including the bound cell's definition itself in the value relation seems to avoid the join. However, working with multiple bound datasets simultaneously requires a self-join on the cell column for each dataset. Computing joins on these complex columns is more expensive than computing joins on an integer column.

To associate cells with data values, we already must have computed the appropriate grid. However, it is valuable at times to store grids intensionally; that is, *decomposed*. For example, a frequently used CORIE grid is the cross product of the horizontal grid H and the vertical grid V. Although a relational approach allows us to express the cross product as a query, we cannot declare that the tuples in a physical table have a foreign key to a query result. An alternative is to use a 2-part foreign key, where the first part references a cell in the grid H, and the second part references a cell in the grid V. Now the space required is higher, and datasets bound to cross-product grids are accessed differently from other datasets. Precomputing and storing an intensional grid consumes space and obscures the relationship between the composed grid and its base grids.

Using Visualization Software. Another approach, which sacrifices data management capabilities for a richer toolset, is to use a visualization library specifically designed to work with gridded datasets. Such libraries are usually oriented toward working with a single dataset at a time, and therefore provide little support for reasoning about the relationships between datasets. Unfortunately, recognizing and exploiting relationships between datasets is a great source of optimization opportunities, as we show later. Further, the programmer is under a significant burden in making use of the library, as each tool has complicated and nuanced semantics.

Software libraries provide functions (or objects) for each specific task. The programmer is often asked to choose between two similar functions that differ only in the type of data on which they operate or the particular algorithm they implement. For example, in the Visualization Toolkit [19], to extract a subset of a grid, there are a variety of functions to choose from. The operation `vtkExtractUnstructuredGrid` accepts internal ids of points and cells, or a function over the geometry of the points. The operation `vtkExtractGrid` works only on structured grids and accepts i, j, and k index ranges that define a structured subgrid. The operation `vtkExtractGeometry` works on a wider range of datasets, but accepts only geometric functions rather than topological ids. A more efficient version is available for polygonal data, `vtkExtractPolyDataGeometry`. Another operator, `vtkThreshold` filters grids based on non-geometric attributes.

The physical concerns of representations and algorithms are intermingled with semantic concerns such as which data is used to filter the grid. All of the operations above can be implemented using the restrict operator, possibly with the aggregate operator to evaluate complex geometric functions. The distinction between filtering geometric data and other bound data is removed in our model.

As we gained experience with VTK and another visualization library [10], we found that simple concepts we used to describe our data products often did not have counterparts in these libraries. Below we list some specific concepts we found weak or missing.

- Cross Product Grids.
- Shared Grids.
- Combinatorial algorithms. Berti observes that combinatorial algorithms for grid manipulation are superior to geometric algorithms [3].
- Aggregation. Both libraries we reviewed implement particular instances of aggregation, but do not provide a general aggregation abstraction.
- Time. We found it useful to reason about time similarly to other dimensions.
- Irregular Grids. Manipulating regular grids is easier than manipulating unstructured grids. Since CORIE involves both kinds of grids, we sought a unifying model.

4 Gridfield Representations

A goal of this work is to support and exploit multiple representations of gridfields, for two reasons: First, supporting a variety of representations can promote interoperability with existing systems. Second, no one representation is efficient for all recipes.

We have identified four major patterns of gridfield representation used in practice. The *tabular* representation forms $\langle cell, value \rangle$ tuples, making it easy to

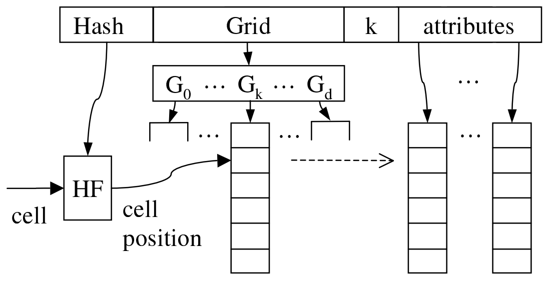


Figure 8: Internal representation of a gridfield in our current implementation.

pipeline data from one operator to another, but difficult to separate grid from data.

The *parallel* representation uses a separate array for each attribute, all aligned positionally with another array for the cells to which the attributes are bound. With this representation, binding in new attributes and projecting out unneeded ones are trivial operations.

The *decomposed* representation stores gridfields intensionally, requiring the client program to assemble the gridfield as needed. Cross product grids are often decomposed in order to save space.

The *nested* representation involves gridfield attributes that can themselves be gridfields. A 4-D timespace gridfield may be a timeseries (the outer grid) and a 3-D spatial grid (the inner, nested grid).

Our Representation. Figure 8 illustrates the representation we use in our current implementation. The gridfield at top stores an integer rank k, and pointers to the grid and each attribute. An attribute is an array of data values, and a grid is a sequence of arrays of cells. Cells of dimension k are aligned positionally with the attributes; we use the parallel representation described above. We began with the parallel representation because it exhibits good performance characteristics (see Section 8) and is used frequently in client software [10, 19] and standard file formats (cf. [11]).

A hash index (H in Figure 8) maps cell definitions to their ordinal position. This index allows the function semantics prescribed by the model to be evaluated in constant time on average. A cell definition is mapped to an array index, which is then used to lookup a data value in each of the attribute arrays. The hash function used maps each cell to its first node, exploiting the fact that seldom do more than 4 or 5 cells touch any one node. Our tests show that this hash function generates very few collisions while offering fast evaluation time.

Another index (not shown in Figure 8) speeds up navigation of the incidence relationship. Each node in the grid is mapped to the cells to which it is incident. The aggregate operator frequently uses this index.

5 Operator Implementation

Our operators are implemented in C++, with in-memory indices implemented using the Standard Template Library (STL) [20]. Physical recipes are currently constructed by hand, though we are designing a declarative query language as an interface to the physical operators.

The parallel representation improves performance in some cases. Binding a new attribute to a grid is inexpensive, as is projecting out attributes. We need not iterate of the arrays; we can simply make a copy of the gridfield header structure (see Figure 8) containing pointers to the information we want.

The merge operator might compute the intersection of two grids during evaluation, and is therefore potentially expensive. However, if the two argument gridfields are defined over the same grid, merge can be evaluated in constant time. Since gridfields may share grids via pointers, checking for grid equivalence is essentially free.

The aggregate operator admits specialized implementations for syntactic convenience and to exploit efficient algorithms. The *apply* specialization uses identical source and target grids, but applies an arithmetic expression to the data values. The *project* specialization also uses identical source and target grids, but simply removes attributes from each logical tuple. The *affix* operator changes the rank of a gridfield by transferring the data values to cells of a different dimension and averaging. The *unify* operator aggregates all of the values in a grid, binding the result to the *unit* grid consisting of a single node.

Cross product is usually the most expensive operator in the algebra. In the next section, we investigate algebraic rewrites to reduce its cost or remove it altogether. We can also improve its implementation in some cases. The cross product of a grid with nodes, edges, and polygons and a grid with nodes and edges produces nodes, edges, polygons, and polyhedra. However, for visualization purposes, we may only need the polyhedra and the nodes; cells of intermediate dimensions need not be computed. However, to use such a "prune" implementation, we must be able to determine which dimension cells will be consumed downstream.

Another implementation of the cross product operator (applicable to Figure 7) exploits the fact that it is followed immediately by a restrict. In relational algebra, a join is semantically equivalent to a cross product followed by a restrict. We can create an analogous "join" operator that evaluates the restrict as the cross product is computed, computing fewer cells overall.

6 Optimization

Having described our data representation and operator implementation, we now present optimization techniques enabled by our algebra for improving the per-

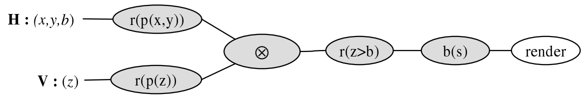


Figure 9: An optimized recipe for visualizing a 3-D CORIE dataset.

formance of recipes. Our examples are actual CORIE data products, though the techniques we use generalize to any domain involving irregular grids, cross product grids, or selected sub-regions.

Forward Binding. The recipe in Figure 7 computes a 3-D salinity gridfield, then restricts the result to a user-specified region. The logical model allows us to freely commute the restrict operator with the bind, and then with the cross product [9], to significantly reduce the size of the intermediate results. However, the physical implementation materializes functions as arrays, so special handling is required. The array we wish to bind can only be correctly interpreted using the ordinal positions of the wetgrid. If we push the restrict earlier, we will produce a grid smaller than the wetgrid, and the bound array will be misaligned. To solve this problem, we can pre-compute the ordinal positions of cells in the wetgrid and record these values in an attribute. This attribute can then be passed to the bind operator and used as offsets into the array on disk.

Our goal is to push the restrict on "region" before the cross product, but there are two obstacles. One is the cross product itself, and the other is another restrict involving attributes b and z from $\mathbf{H}$ and $\mathbf{V}$, respectively[2]. A cell's ordinal position in a cross product grid can be derived from the ordinal positions of the cells used to construct it. However, the grid we want is not just a cross product of two grids, but the restriction of a cross product. Therefore, the ordinals of the cells of the wetgrid are dependent on the condition used to filter out the "dry" cells.

In the general case, the 1000th cell in the grid prior to a restrict could be the 1st cell or the 1000th cell in the restricted grid. However, we know a *physical property* of the gridfield $\mathbf{V}$: It is sorted on the attribute z. We can therefore compute the positions of the wetgrid's cells without actually materializing the grid itself.

Recall the attribute b of the gridfield $\mathbf{H}$ stores bathymetry information for the river. Specifically, b is an index into the gridfield $\mathbf{V}$. Since $\mathbf{V}$ is sorted on z, we can use b to determine the number of cells in each vertical column of water. With these cell counts, we can compute an offset into the array to be bound to the wetgrid.

The result of these transformations is the optimized recipe shown in Figure 9. The potentially highly selective restricts on x, y and z are evaluated prior to the cross product.

[2]This restrict compares attributes from both $\mathbf{H}$ and $\mathbf{V}$ and does therefore not commute with the cross product.

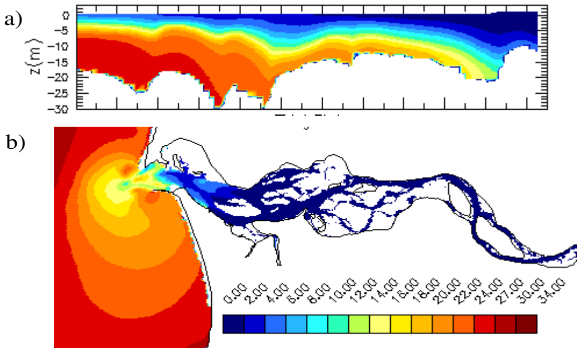


Figure 10: (a) A vertical slice data product. (b) A horizontal slice data product.

Lowering Dimensionality Two common 2-D CORIE data products are horizontal and vertical "slices." Examples of these data products for the salinity variable are shown in Figure 10. One way to express the horizontal slice data product is to use the same recipe as in Figure 7, but restrict the z dimension to a single node. As before, we could push the restricts through the cross product. This time, though, we observe that restricting $\mathbf{V}$ to a single node produces the unit grid. The unit grid is the identity for the cross product operator, up to namespace isomorphism. We can therefore remove the cross product operator altogether.

This optimization is unavailable to systems that cannot reason about grids algebraically. We have not only produced a faster recipe, but we have also naturally expressed a critical correctness criteria: The output grid is 2-D. Although the wetgrid is constructed from prism-shaped cells, this data product is defined over triangles. (We have assumed that the depth at which a slice is to be taken corresponds to one of the depths in the vertical grid V. We could relax this assumption by using an aggregate operator equipped with an interpolation function.)

Computing a vertical slice is more difficult. The horizontal grid H has an irregular topology consisting of triangles. To take a vertical slice, we must still project the 3-D grid down to two dimensions, but the target is a new grid not appearing elsewhere in the recipe. Consider a user who wants to view a vertical profile of the salinity intrusion along a deep channel near the mouth of the estuary. To specify "along a deep channel" to the system, the user selects a sequence of points in the xy plane, as shown in Figure 11a. We can connect these points to form a 1-D grid, P. A cross product with the vertical grid gives us a 2-D slice, $P \otimes V$ (Figure 11b).

Using VTK, we must manually construct the grid $P \otimes V$ producing points in 3-D space. For each point, we must search in the 3-D wetgrid for the cell that contains the point, then perform a 3-D interpolation

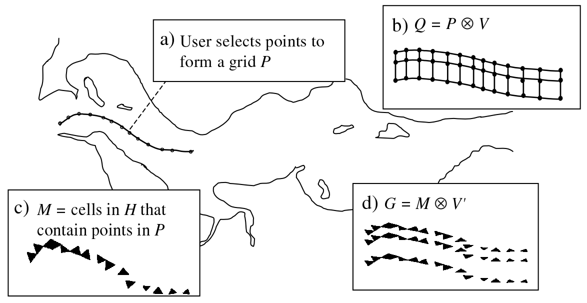

a) User selects points to form a grid P
b) $Q = P \otimes V$
c) M = cells in H that contain points in P
d) $G = M \otimes V'$

Figure 11: Four intermediate steps in an efficient "vertical slice" recipe.

of salinity values.

With the gridfield algebra, we can do the work in two dimensions for considerable savings. Each point in P can be positioned in a triangle in the horizontal grid H. We can restrict H to only those cells that contain one or more points in P using the aggregate operator followed by a restrict, producing a grid M (Figure 11c).

Since the grid M is a restriction of the grid H, we can use forward binding (as we did previously) to construct a 3-D grid (not shown in Figure 11) and bind the appropriate salinity values to it. We can now perform the same search-and-interpolate operation required by VTK, but using a much smaller 3-D grid.

An additional optimization is possible. Instead of considering V as a 1-D grid, we prune the 1-cells, leaving only the nodes. Call this grid V'. The grid $M \otimes V'$ consists of "stacks" of 2-D triangles (Figure 11d), rather than a connected set of 3-D prisms. Interpolation using triangles is much cheaper than interpolation using prisms, further reducing the cost.

By working primarily in two dimensions, we were able to produce a less expensive recipe. Lowering the dimension of the intermediate results saves time since a) higher dimensional gridfields tend to have more cells, and b) algorithms for manipulating 3-D cells are more expensive than their 2-D counterparts.

Merging Related Grids The *plume* is the region of water beyond the mouth of the river with a salt content below a given threshold. The recipe to compute the plume extends the recipe to bind salinity to the wetgrid in Section 6. We encode the definition of the plume as conditions passed to the restrict operator.

Consider a recipe to find the portion of the plume above a certain temperature. Assume temperature data has been bound, separately, to another instance of the wetgrid, and we now need to merge this data with the salinity gridfield.

We need to evaluate two restrict operators (r) and one merge operator (m) to obtain the correct gridfield. Three versions of the relevant fragment of this new recipe are shown in Figure 12. Figure 12a shows the two restricts evaluated after the merge. Previously, we improved performance by evaluating restricts early, as

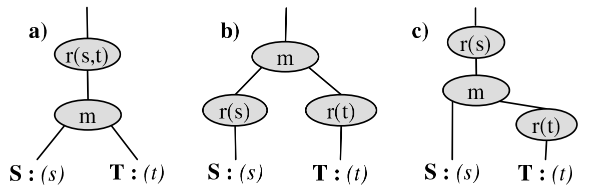


Figure 12: Three equivalent sub-recipes. (a) Restrict operations are combined into one. (b) The restrict operators are pushed through a merge, resulting in a *less* efficient plan. (c) Evaluating one restrict at a time might require less memory.

in Figure 12b. In this case the merge operator must compute the intersection of two grids – an $O(nm)$ algorithm, where n and m are the number of cells in **S** and **T**, respectively. But observe that in Figure 12a, both arguments are defined over the same grid. This knowledge allows the merge to be evaluated trivially. The recipe in Figure 12c may also be a good choice. Since the grid T is known to be a subset of the grid S, we can still evaluate the merge in constant time. We can also evict the attribute t from memory right after we have evaluated the first restrict, possibly lowering the memory footprint of the overall recipe.

7 Experimental Results

We performed experiments to 1) validate our design choices in the physical implementation and 2) to determine whether algebraic optimization techniques could improve performance over more traditional solutions.

The experiments were run on a dual 2.4 GHz processor with 4GB of RAM. This machine is nearly identical to one node of the cluster on which the CORIE simulations are executed. Each experiment involved 5 trials, and three experiments were done at different times. The samples produced a variance of less than 1% of the mean, demonstrating stability.

The CORIE horizontal grid consists of 29,602 nodes and 55,081 2-cells. The vertical grid has 62 nodes and 61 1-cells. The wetgrid has 829,852 nodes, and therefore each timestep of each dataset has 829,852 values.

Our first experiment compared physical implementations of the cross product operator. The "prune" implementation avoids extra work by computing only the nodes and polyhedra of the 3-D cross product. The "join" implementation composes the cross product and subsequent restrict. Figure 13 shows results for the original cross product implementation ("no opt"), the "prune" implementation ("prune") and with both improvements ("both"). If C_w and C_r are the cardinalities of the wetgrid and the result gridfield, respectively, then the selectivity (x-axis) is $1 - \frac{C_r}{C_w}$.

Times reflect overall execution time of the 3-D scalar data product described in Section 3.5, highlighting the cross product operator's significant cost relative to the other operators in the recipe. The graph shows that avoiding cell materialization does indeed

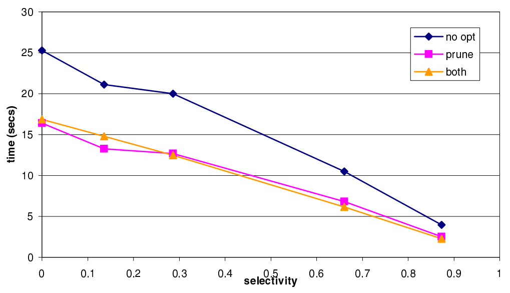


Figure 13: Comparing implementations of the cross product operator.

improve performance. On average, the prune implementation results in 35% faster times than computing the full cross product. The join implementation does not provide a consistent improvement. With the standard cross product implementation, we can predict precisely the space requirements of the output. With the join implementation, we must estimate the selectivity of the join condition and dynamically resize arrays when we are wrong. Although the join implementation produces no unnecessary cells, the extra complexity of memory management washes out the performance gain.

The second experiment compares our algebraically optimized recipe in Figure 9 with the unoptimized recipe in Figure 7, as well as with two more traditional approaches. First, we used a relational database extended with spatial data types to represent the cells. Second, we used VTK along with custom code that handles those operations inexpressible in VTK.

The relational approach uses SQL to join data with cells and select the "wet" values. Our test DBMS was Postgres [22], configured appropriately for the large main memory of our experiment platform.

The times for the relational approach are artificially low, as we did not include the time to extract the results to the client. Instead, the results were simply loaded into a temporary table on the server. We felt that the diversity of potential client interfaces muddles the results, and a query-only experiment represents a conservative lower bound. For our own approach, we did include the time required to convert our gridfield representation into a form suitable for rendering by a third party library, but not the rendering time itself.

The implementation in VTK required a custom reader for our file formats. Restrictions were implemented using the `VTKThreshold` object. The cross product and bind operators were implemented in a custom reader since these tools were not available in VTK. Unlike our general operators, we were free to design the reader for specific tasks: reading in a CORIE dataset, computing the wetgrid, and building a VTK object. This focused goal afforded a very efficient design. Indeed, the reader was not the bottleneck despite representing the majority of work.

Figure 14 shows test results for various size regions,

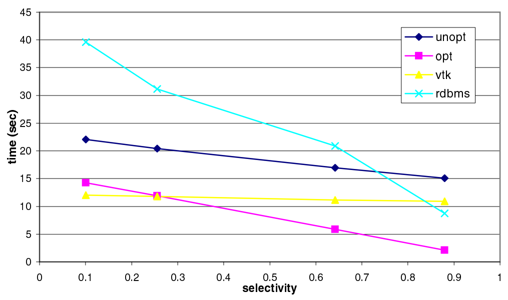


Figure 14: Optimized and unoptimized recipes compared with two traditional approaches.

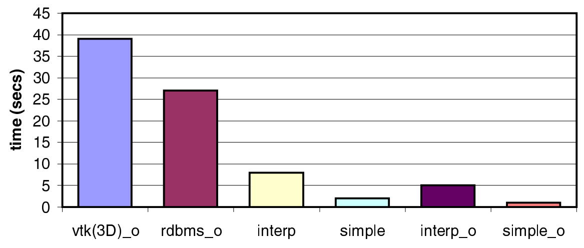


Figure 15: Experimental results for the vertical slice data product.

which translate to various selectivities of the full wetgrid. Observe that our unoptimized recipe is slower than the VTK implementation, even though they implement similar recipes. The specialized reader, implementing the cross product, restrict, and bind operations constitutes only about 15% of the total execution time. In our program, these operations constitute about 30% of the total. The specialized reader is indeed more efficient than the generic operators.

The optimized recipe performs better in all but the lowest selectivities. The advantage of reducing dataset size as early as possible is apparent here just as it is in relational processing. Note that VTK's times are effectively the same for all selectivities, as would be expected given the recipe of Figure 7a. Regardless of the region being displayed, the entire 3-D grid is generated and iterated through. The relational approach is far behind in all but the highest selectivities. Although the optimizer produces query plans that behave like our optimized recipe, the overhead of processing gridded datasets using joins dwarfs the effect.

The third experiment compares the optimized vertical slice recipe against a VTK program and an SQL query (Figure 15). The bar labelled "interp" uses interpolation as described above. The bar labelled "simple" approximates interpolation and improves performance by taking the value of a random node in the cell. The bars labelled with the 'o' suffix make use of a semantic optimization: We restrict the grid to the relevant region before searching for cells that contain points. Note that even our recipes that do *not* exploit this optimization outperform the optimized VTK program and the optimized SQL query.

8 Future Work and Conclusions

Our primary goal is a data server that can accept gridfield recipes expressed in a declarative query language and produce gridded dataset answers in a flexible yet efficient manner. As a first step, we have derived an algebra that captures procedural recipes. We are building prototype applications that will generate recipes in this algebra.

We are modeling the taxonomy of gridfield representations more precisely, so as to include representations in our space of optimization techniques. Nested gridfields seem especially flexible, as they are key to modeling and processing multi-resolution grids. Nested gridfields also provide a mechanism by which we may segment a large grid for parallel processing or for secondary storage management.

The recipes we have used in this paper have involved only a few gridfields. In reality, there are terabytes of gridded datasets one might wish to manipulate. Finding and retrieving these gridded datasets requires a form of catalog, for which relational or object-relational databases are quite appropriate. Technology such as IBM's Datalinks [4] for managing files external to the database may be useful.

We are studying additional grid properties and deriving versions of the operators to preserve them. For example, notions of grid *quality* are used by grid generation packages.

We have presented an implementation of an algebra for manipulating scientific datasets and shown that this approach offers benefits in both expression and performance. In summary, our contributions are:

- A design and implementation of a gridfield algebra.
- Algebraic optimization techniques for improving performance of gridfield recipes.
- Application to real data products.
- Experimental evidence that such processing strategies result in superior performance.

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] P. Baumann. A database array algebra for spatio-temporal data and beyond. In *Next Generation Information Technologies and Systems*, pages 76–93, 1999.

[3] G. Berti. *Generic software components for Scientific Computing*. PhD thesis, Faculty of mathematics, computer science, and natural science, BTU Cottbus, Germany, 2000.

[4] S. Bhattacharya, C. Mohan, K. W. Brannon, I. Narang, H.-I. Hsiao, and M. Subramanian. Coordinating backup/recovery and data consistency between database and file systems. In *SIGMOD*, pages 500–511, 2002.

[5] D. M. Butler and S. Bryson. Vector-bundle classes form powerful tool for scientific visualization. *Computers in Physics*, 6(6):576–584, 1992.

[6] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J.-B. Yu. Client-Server Paradise. In *VLDB*, pages 558–569, Santiago, Chile, 1994.

[7] ESRI Corporation. ArcGIS: Working with geodatabase topology. Technical report, ESRI, 2003.

[8] R. B. Haber, B. Lucas, and C. N. A data model for scientific visualization with provisions for regular and irregular grids. In *IEEE Visualization*, 1991.

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

[10] IBM Corporation. *IBM Visualization Data Explorer User Guide*, 4th edition, 1993.

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

[12] L. Libkin, R. Machlin, and L. Wong. A query language for multidimensional arrays: design, implementation, and optimization techniques. In *SIGMOD*, pages 228–239, 1996.

[13] A. P. Marathe and K. Salem. A language for manipulating arrays. In *VLDB*, pages 46–55, 1997.

[14] J. Melton, J.-E. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, and K. Zeidenstein. SQL and management of external data. *SIGMOD Record*, 30(1):70–77, 2001.

[15] P. Moran. Field model: An object-oriented data model for fields. Technical report, NASA Ames Research Center, 2001.

[16] R. Musick and T. Critchlow. Practical lessons in supporting large-scale computational science. *SIGMOD Record*, 28(4):49–57, 1999.

[17] M. Papiani, J. Wason, and D. A. Nicole. An architecture for management of large, distributed, scientific data using SQL/MED and XML. In *EDBT*, pages 447–461, 2000.

[18] P. J. Rhodes, R. D. Bergeron, and T. M. Sparr. Database support for multisource multiresolution scientific data. In *SOFSEM*, pages 94 – 114, 2002.

[19] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *IEEE Visualization*, pages 93–100, 1996.

[20] A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, 1994.

[21] E. Stolte and G. Alonso. Efficient exploration of large scientific databases. In *VLDB*, pages 622–633, 2002.

[22] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of postgres. *TKDE*, 2(1):125–142, 1990.

[23] A. Thakar, P. Kunszt, A. Szalay, and J. Gray. The sdss science archive: Object vs relational implementations of a multi-tb astronomical database. *Computers in Science and Engineering*, 2002.

[24] P. Watson. Topology and ORDBMS technology. Technical report, Laser-Scan, 2002.

[25] N. Widmann and P. Baumann. Efficient execution of operations in a dbms for multidimensional arrays. In *SSDBM*, pages 155–165, 1998.