DISSERTATION APPROVAL

The abstract and dissertation of Bill Howe for the Doctor of Philosophy in Computer Science were presented on December 8, 2006, and accepted by the dissertation committee and the doctoral program.

COMMITTEE APPROVALS:

David Maier, Chair

Lois Delcambre

Leonard Shapiro

Timothy Sheard

Gerald Recktenwald
Representative of the Office of Graduate Studies

DOCTORAL PROGRAM APPROVAL:

Cynthia Brown, Director
Computer Science Ph.D. Program

ABSTRACT

An abstract of the dissertation of Bill Howe for the Doctor of Philosophy in Computer Science presented December 8, 2006.

Title: Gridfields: Model-Driven Data Transformation in the Physical Sciences

Scientists' ability to generate and store simulation results is outpacing their ability to analyze them via ad hoc programs. We observe that these programs exhibit an *algebraic structure* that can be used to facilitate reasoning and improve performance. In this dissertation, we present a formal data model that exposes this algebraic structure, then implement the model, evaluate it, and use it to express, optimize, and reason about data transformations in a variety of scientific domains.

Simulation results are defined over a logical *grid* structure that allows a continuous domain to be represented discretely in the computer. Existing approaches for manipulating these gridded datasets are incomplete. The performance of SQL queries that manipulate large numeric datasets is not competitive with that of specialized tools, and the up-front effort required to deploy a relational database makes them unpopular for dynamic scientific applications. Tools for processing multidimensional arrays can only capture regular, rectilinear grids. Visualization libraries accommodate arbitrary grids, but no algebra has been developed to simplify their

use and afford optimization. Further, these libraries are data dependent—physical changes to data characteristics break user programs.

We adopt the grid as a first-class citizen, separating topology from geometry and separating structure from data. Our model is agnostic with respect to dimension, uniformly capturing, for example, particle trajectories (1-D), sea-surface temperatures (2-D), and blood flow in the heart (3-D). Equipped with data, a grid becomes a *gridfield*. We provide operators for constructing, transforming, and aggregating gridfields that admit algebraic laws useful for optimization. We implement the model by analyzing several candidate data structures and incorporating their best features. We then show how to deploy gridfields in practice by injecting the model as middleware between heterogeneous, ad hoc file formats and a popular visualization library.

In this dissertation, we define, develop, implement, evaluate and deploy a model of gridded datasets that accommodates a variety of complex grid structures and a variety of complex data products. We evaluate the applicability and performance of the model using datasets from oceanography, seismology, and medicine and conclude that our model-driven approach offers significant advantages over the status quo.

GRIDFIELDS:
MODEL-DRIVEN DATA TRANSFORMATION
IN THE PHYSICAL SCIENCES

by

BILL HOWE

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE

Portland State University
2007

DEDICATION

For my father, Dr. Gerry Howe, whose kindness, wisdom, honor, and patience are the model for my conscious behavior, however well I implement it.

ACKNOWLEDGEMENTS

Were it not for convention, the work represented by this dissertation would be attributed to a whole host of indirect co-authors, each of whom has had an effect deserving more than just "acknowledgement." Formatting conventions being what they are, however, and not wishing for these individuals to share any blame that may accompany any credit, I have settled for using the plural first person in the body of this dissertation as a little anonymous symbol of my gratitude.

I offer loving thanks to my wife and best friend Tammi. Her apparently limitless patience and support have always intensified precisely when I least deserved it.

I am deeply grateful to David Maier, my thesis advisor, for continual support and guidance as an advisor, a colleague, and a friend. His ability to maintain a calm demeanor while slogging through early drafts of papers and this dissertation amazes me in hindsight. I also made frequent use of Dave's ability to listen carefully to an hour of wild, tenuous ideas, yet identify and articulate the five-minute kernel of significance when it (occasionally) existed. Finally, Dave deserves special thanks for retaining me as a student after I ran off with his wife's umbrella in a hail storm.

I thank the rest of my committee for not only their helpful comments and questions, but for tolerating my unusual scheduling demands: Lois Delcambe, Len Shapiro, Tim Sheard and Gerald Recktenwald.

I am grateful to Antonio Baptista and his staff for providing such fascinating data and such significant challenges in the context of the CORIE project. Dr. Baptista's advice and support were crucial in the development of these ideas. I am especially grateful to Paul Turner for his feedback as my prototypes evolved.

I thank my parents, who managed to provide me an excellent education and even instill in me a passable work ethic despite my best efforts to the contrary. They have my love, respect, and heartfelt thanks.

I was extremely fortunate to have ready access to excellent personal and professional advice as a student. I appreciate all the faculty I have had the opportunity to chat with. I am indebted to Lois Delcambre for several crucial and inspirational conversations along with the way. I thank Wu-chi Feng for contagious energy and plying me my only healthy hobby.

I owe my gratitude to the excellent database group at OGI and Portland State, past and present: Vassilis Papadimos, Kristin Tufte, Sun Murthy, Shawn Bowers, Pete Tucker, Mat Weaver, Laura Bright, Susan Price, James Terwilliger, Jin Li, and Nick Rayner. All were consistently available to offer help, feedback, and general conversation; their collective impact on the successful completion of this work cannot be underestimated.

I am in debt to Laura Bright for her excellent work developing the curriculum for our Summer course at a time when this dissertation kept me rather distracted.

I wish to also thank the students of the Systems Group: Francis Chang, who is clearly the better pool player; and Ed Kaiser and Chris Chambers, whom had I not known, I might have graduated three months earlier.

I thank the excellent staff at OGI and Portland State for cheerfully tolerating my continual requests and occasional complaints: Lorie Gookin, Dana Director, Jo Ann Binkerd, Cindy Pfaltzgraff, Renee Remillard, Kathi Lee, and Leai Rose.

I owe thanks to the the National Science Foundation, whose successful Information Technology Research program funded this work.

Finally, I extend my thanks to my friends, currently scattered around the world:

John Bowen, Curtis Atkins, Andy Verras, Gary Moss, and Chuck Petrakopolous. I look forward to returning your favors.

CONTENTS

List of Tables . . . . **ix**

List of Figures **1**

1 Introduction **1**
1.1 Computational Science 6
1.2 Source Data Organization 10
1.3 Data Products 12
1.4 Example Domain 16
1.5 Related Work 21
1.6 Contributions 28

2 Data Model **30**
2.1 Model Requirements 30
2.2 The Gridfield Model 36
2.2.1 Grids 37
2.2.2 Grid Properties 39
2.2.3 Grid Relations and Operations 42
2.2.4 Gridfields 49
2.3 Grid Computation Requirements 51
2.3.1 Grid Serialization 52
2.3.2 Direct Rendering 53
2.3.3 Smoothing 54
2.3.4 Streamlines 56
2.3.5 Feature Extraction 59
2.3.6 From Algorithms to Recipes 59
2.4 Gridfield Algebra 64
2.4.1 Bind 64
2.4.2 Restrict 64
2.4.3 Accrete 67

2.4.4 Merge 69
2.4.5 Cross Product 70
2.4.6 Regrid 71
2.4.7 Abstract Syntax and the Fixpoint Operator 79
2.4.8 Feature Summary 81
2.5 Examples and Discussion 82
2.5.1 Example: Wetgrid 83
2.5.2 Example: Plume Volume 85
2.5.3 Example: Minimum Cell Diameter 87
2.5.4 Example: Feature Extraction 88
2.5.5 Discussion 89

3 Data Structures for Gridfields 91
3.1 Characterizing Grid Representations 95
3.2 Review of Grid Data Structures 99
3.2.1 Incidence Graph 99
3.2.2 Indexed Cell Set 101
3.2.3 Edge-based Structures 106
3.2.4 Cell Tuples 109
3.2.5 Generalized Combinatorial Maps 110
3.2.6 A Relational Implementation 113
3.2.7 A Data Structure for Visualization 117
3.3 Implementing and Testing Gridfield Structures 119
3.3.1 Bound Attributes 119
3.3.2 Cells and Grids 122
3.4 Experiments 130
3.4.1 Restrict Operator over ICS, NGMap, and TIGraph 131
3.4.2 Neighborhood Operator over NGMap and TIGraph 134
3.5 Conclusions 136

4 Retrofitting Gridfields onto Existing Data 139
4.1 Introduction 140
4.2 Related Work 146
4.3 Running Example 149
4.4 Native Data Model 152
4.4.1 Data From Directory Structures 152

4.4.2 Data From Binary Files 155
4.4.3 Extraction Queries 157
4.5 Evaluating Queries Over Binary Data 158
4.5.1 Schema Transformation 162
4.6 Experimental Results 165
4.7 Conclusions 169

5 Applications and Interfaces 171
5.1 Reasoning with Gridfields 171
5.2 Programming with Gridfields 176
5.2.1 Persistent Eddies 177
5.2.2 Pushing restricts 182
5.2.3 Optimizing Slice Products 183
5.2.4 Plume Front 186
5.3 Evaluating Gridfield Expressions 189
5.3.1 Catalog Interface: Scan and Bind 194
5.3.2 Optimization in the Presence of bind 197
5.3.3 Animation and Parallelism 202
5.3.4 Conclusion: A Canonical Gridfield Application 206

6 Experimental Evaluation 209
6.1 Highly Regular Datasets 212
6.2 Partially Regular Datasets 221
6.3 Design and Population of a Relational Database 225
6.4 Irregular Datasets 232
6.5 Conclusions 235

7 Conclusions and Outlook 238
7.1 Future Work 244
7.2 Gridfields as Scientific Middleware 250
7.3 Interpreting Data Models Topologically 256

References 258

LIST OF TABLES

3.1 Operators for the Quad-Edge data structure. 108

4.1 Tested queries with result sizes. 168
4.2 Response time in seconds by query. 169

5.1 Propagation of grid, scheme, and size through the gridfield operators. 196

LIST OF FIGURES

1.1 Mesh structures used for Finite Element simulations in mechanical engineering, seismology, and medicine. 2
1.2 Simulating the behavior of a thin rod under a vertical load. 7
1.3 A sample solution to the problem in Figure 1.2. 9
1.4 Two examples of data products. 13
1.5 Datasets bound to the nodes and polygons of a 2-D grid. 14
1.6 A recipe for visualizing a 3-D CORIE dataset. 17
1.7 Two types of grid: structured and unstructured. 20

2.1 Two simple grids: uniform rectangular and variable rectangular. . . 32
2.2 A deformed structured grid modeling airflow over a blunt fin rising from a flat plate. 33
2.3 An unstructured grid modeling the airflow around a wing. 34
2.4 The possible interfaces between blocks of a multi-block mesh. 35
2.5 Three different geometric realizations of the same topological grid. . 36
2.6 The components of a grid. 38
2.7 Examples of grids allowed by the basic model. 39
2.8 Examples of grid operations. 42
2.9 The cross product of two simple grids. 47
2.10 Illustration of the "smoothing" algorithm. 56
2.11 Streamlines indicating the direction of blood flow near the surface of an aneurysm. 58
2.12 Possible semantics for restrict. 67
2.13 An example of the accrete operator. 68
2.14 An illustration of the semantics of merge. 69
2.15 An example of the regrid operator. 72
2.16 A horizontal grid (a) and a vertical grid (b) used to discretize the domain of the Columbia River Estuary. 83
2.17 A recipe for visualizing a 3-D CORIE dataset. 84

2.18 A plan for computing the volume of the *plume*. 86
2.19 Two equivalent plans for finding the minimum "diameter" of all grid cells. 88
2.20 A plan for extracting a "feature." 89

3.1 A simple grid we use to illustrate representation alternatives. 99
3.2 A Hasse diagram of the incidence graph of the grid in Figure 3.1. . 100
3.3 An example of the ICS representation. 102
3.4 The ICS representation is sensitive to physical node order. 103
3.5 The winged-edge data structure. 105
3.6 The quad-edge data structure. 107
3.7 The barycentric subdivision of the grid in Figure 3.1. 110
3.8 An illustration of an n-dimensional combinatorial map. 112
3.9 Internal representation of a gridfield for the ICS representation. . . 123
3.10 Performance of the restrict operator using three different representations. 132
3.11 Performance improvement using TIGraph for three different situations at three different selectivities. 133
3.12 The traversal of an *orbit*, a concept used with the NGMap representation. 135
3.13 The performance of the neighborhood operator over the NGMap and TIGraph representations. 136

4.1 Two grids used in the CORIE applications. 150
4.2 Examples of gridfield assembly. 151
4.3 Describing data stored in the names of files and directories. 153
4.4 Describing data stored in binary files. 154
4.5 The representation of a file schema and a particular instance. 159
4.6 Rewriting both the schema and the query to extract data efficiently. 163
4.7 A BinaryBlock describing the CORIE simulation output. 166

5.1 The CORIE grid evolved over time, requiring significant changes to the analysis pipeline. 172
5.2 The structural differences between CORIE grids are expressed succinctly using gridfields. 175
5.3 A visualization of average velocity over time interpolated to a particular depth. 177

5.4 Interpolating over the vertical grid. 178
5.5 This recipe for interpolating values at a particular depth is complicated by the movement of data from one rank to another. Recipes such as this motivated the current multi-rank gridfield data model. The accumulate operator (used near the leaves) is a specialization of the regrid operator used to assign an identifier to each cell (see Section 2.4.6). 180
5.6 In this recipe for interpolating values at a particular depth, all ranks can be processed with a non-branching sequence of operators. This recipe computes the interpolation of data at a particular depth over the σ-grid in Figure 5.1. Recipes such as this motivated the current multi-rank gridfield data model. The accumulate operator (used near the leaves) is a specialization of the regrid operator used to assign an identifier to each cell (see Section 2.4.6). 181
5.7 Optimizing a recipe for visualizing a 3-D CORIE dataset. 182
5.8 Two types of "slice" data products. 184
5.9 Four intermediate steps in an efficient "vertical slice" recipe. 185
5.10 The polygons containing user-selected points for computing a transect. 187
5.11 The plume expressed as a region of high gradient magnitude. 188
5.12 A finished data product derived from the same recipe used in Figure 5.11. 189
5.13 Operators can share or destructively update grids and attributes. . 192
5.14 An array can be bound to its "natural" grid gridfield bound. 199
5.15 A screenshot from a prototypical gridfield application, DPBuilder. . 207
5.16 The server-side architecture supporting DPBuilder. 208

6.1 The number of grids per user as a function of time. 210
6.2 Seven sample datasets, each formed as the cross product of 3 linear grids. 215
6.3 A comparison of three gridfield recipes and a specialized VTK algorithm for extracting the edges of a mesh. 216
6.4 Completion time for a "subslab" operation 217
6.5 Calculating the position of a cell in the original grid based on its position in a subslab. 219
6.6 Comparing implementations of the cross-product operator. 220
6.7 Optimization of a recipe for visualizing a 3-D CORIE dataset. . . . 221

6.8 The performance of optimized and unoptimized recipes compared with VTK and PostgreSQL implementations. 223
6.9 Experimental results for the vertical slice data product. 225
6.10 A relational schema for storing CORIE simulation results. 226
6.11 Gridfield recipe implementing aggregation over depth at a particular timestep. The accumulate operator is used to assign positions to the cells; see Section 2.4.6. 230
6.12 Gridfield recipe implementing aggregation over depth for a specific bounding box, but for all timesteps. The accumulate operator is used to assign positions to the cells; see Section 2.4.6. 231
6.13 Three equivalent sub-recipes illustrating a non-intuitive optimization technique. 233
6.14 A model of a carotid artery equipped with pressure data. 234
6.15 A region of a carotid artery associated with increased risk of dislodged plaque and stroke. 235

7.1 A hierarchical grid. 245
7.2 A rule for transforming cells. 248
7.3 Heterogeneous observation methods. 252

Chapter 1

INTRODUCTION

As engineers and physical scientists work on increasingly complex problems, simulation has emerged as a third avenue to scientific knowledge, alongside theory and experiment. Problems such as climate prediction are too complex to yield an analytic solution, and have domains too large (or too small) to be replicated in the laboratory. Simulation extends the utility of intractable mathematics and incomplete observational data, yielding a formalized approximation for reality.

- Oceanographers must use simulation due to the complexity and scale of their domain of interest. Directly measuring the physical attributes of the ocean at sufficient scales to study currents, plume dynamics resulting from freshwater jets at the mouths of rivers, and tsunami impacts is not possible. However, complex coastlines and complex bathymetry prevent derivation of analytic solutions.
- Mechanical engineers use simulation to analyze failure modes of materials under stress. Crack propagation in the polycrystal microstructure of stressed metals (Figure 1.1(a)), for example, requires modeling individual grains and simulating their interaction [HPD+05].
- Seismologists use simulation to reconstruct the development of an earthquake, solving the inverse problem: inference about the displacement of the earth from sparse data acquired observationally post hoc. This procedure

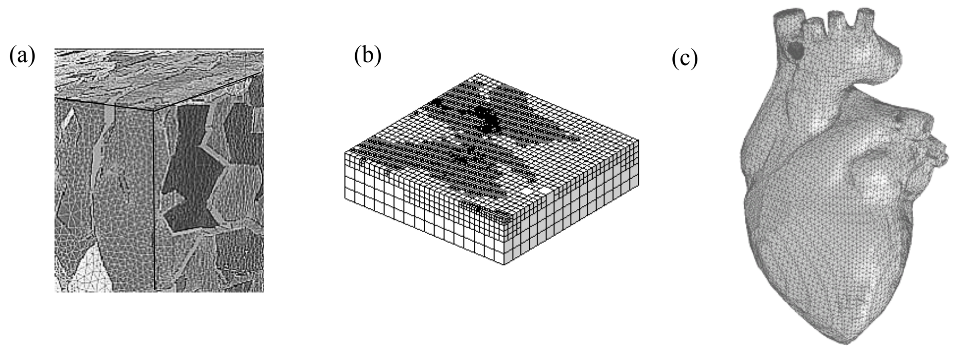


Figure 1.1: Mesh structures used for Finite Element simulations in mechanical engineering, seismology, and medicine. (a) Individual grains of a metallic polycrystal structure are modeled with a mesh to study crack propagation in aircraft wings [HPD+05]. (b) A mesh representing the earth around the San Fernando Valley developed to study the aftershocks of the 1994 earthquake in the area [ABB+03]. (c) A detailed mesh of a human heart. The inner structures are modeled in similar detail, but are not visible in this image [ZB04].

---

is a prerequisite to a shift from forensics to prior warning and prediction [ABB+03].

- Finite Element simulations in medicine allow medical researchers to study organ function in extreme detail. Cardiologists apply numerical methods for solving fluid dynamics equations to study fetal heart development [DTP+03], preconditions for stroke [Sch02], and muscle contraction dynamics [ZB04].

We will consider simulations of continuous physical domains whose behavior is described by differential equations. Discrete event simulations such as those modeling factory flow or network traffic will not be considered, nor will mathematical models that do not involve a spatial domain, such as ecological population models.

Simulation is an effective tool, but it is a hammer rather than a scalpel. Simulation provides a surrogate for reality, but does not automatically filter out irrelevant information or highlight important features. Scientists looking for specific phenomena, such as the effect of aerosol particles on cloud formation [NS03], must analyze large unrefined datasets generated by coarse-grained simulations. However, the tools used to store, manage, and analyze these datasets has not keep pace with our ability to generate them.

The explosion of data being produced by modern science has been noted by domain scientists as well as database researchers [OLNS$^+$05, GLNS$^+$05]. Improved observational technology (satellites, telescopes, particle accelerators) contribute to the deluge, but simulations represent the dominant source. For example, the BaBar project at the Stanford Linear Particle Accelerator has produced nearly a petabyte of particle-collision data; about two-thirds of these data are the results of simulations [BW].

The terms “online science” and “e-science” [Gob05] describe a shift away from hypothesis-oriented, top-down data acquisition, to proactive, bottom-up data acquisition. Traditionally, data acquisition was the dominant cost of scientific inquiry. Oceanographers, for example, would survey bathymetry with a lead weight and a string, and have plenty of time during the winter to catalog and analyze the data. “Simulations” were calculated by hand, and were therefore only formulated over very simple (and unrealistic) problem domains. Due to the cost, one had to be selective about which hypotheses would be investigated. We refer to this kind of hypothesis-driven science as “querying the world” (QW).

However, as the cost and effort of data acquisition drops with improvements in both computing hardware and observational equipment, acquisition activities need not be motivated by a particular hypothesis. Instead, data can be acquired in a bottom-up manner, with the expectation that a comprehensive data repository will lead to a variety of scientific discoveries on a potentially larger scale sometime in the future. We call this model of data science “downloading the world” (DW).

We see the shift to DW in many domains. Biologists sequence entire genomes and publish the results for public annotation and analysis [HAC$^+$05, MSK$^+$]. Proteomics databases have followed closely [BBA$^+$03]. Astronomers are systematically mapping the sky rather than competing for telescope time to test individual hypotheses [SGT$^+$02]. Earth scientists are interested in connecting ocean models to ground water models to ecological models developed for varying purposes to allow cross-scale and even cross-domain analysis [PCD$^+$03, BL00, SBM$^+$00, ND05].

In some domains, bulk observations are not yet technologically feasible. For example, although satellites can measure sea surface temperature over large areas at once, they cannot measure the temperature or any other physical quantity at any significant depth. When the technology to perform DW-style observations is not available, scientists turn to DW-style simulations, where large domains are simulated at multiple scales [BWP$^+$99, ABB$^+$03, DTP$^+$03, LFA$^+$, HG05]. The repositories generated by this kind of project are the topic of this dissertation.

Researchers (and funding agencies) place significant intrinsic value on massive data acquisition projects. Alexander Szalay, an astronomer on the Sloan Digital Sky Survey project, says that the utility of a collection of datasets scales as $\Omega(N^2)$, as each pairwise comparison of acquired datasets potentially leads to new discoveries [SGT$^+$02]. For example, temperature measurements acquired by a permanent buoy can be meaningfully compared with measurements of different variables, using different instruments, in different regions, under different conditions. Each such comparison can improve understanding of fluvian, estuarine, or ocean processes. The implication of Szalay's maxim is that simply acquiring datasets leads to quadratic returns in scientific knowledge. However, arbitrary and complete pairwise comparisons may not be feasible unless access and manipulation of the acquired datasets become convenient and efficient.

Unfortunately, the computer science community has not produced general techniques for accessing and manipulating scientific data repositories. Scientists still tend to store their data in files on an ordinary UNIX filesystem, in a variety of formats with varying APIs. The hardware architectures on which the data reside are state-of-the-art: multi-core systems, clusters of commodity machines, and Grids. However, the tools scientists use to access them are updated versions of grep [DG] and FTP [ABKL05].

Is a more sophisticated data management solution possible? Gray et al. report success using relational databases to support finite element simulations [HG05], though each solution relies heavily on application-specific code. Other research groups have discovered that UNIX tools are insufficient for their needs, and have developed specialized software for their particular applications [SA02, LCA^{+}03, BW]. However, general principles for data management in this domain have not emerged.

Computer scientists have an opportunity to develop more general models and tools useful across many scientific applications, and perhaps across many domains. In fact, such generalization is critical, as *integrative* science emerges as a primary source of discoveries. Traditional science tends to produce silos of specialization, but the cost of discovery within narrow sub-disciplines is growing (witness multi-billion dollar particle accelerators). The areas between silos are fertile, but we need general tools to bridge the gaps. For example, oceanographers are beginning to model ecological variables such as bio-mass and fish movement in addition to physical variables [MtG^{+}99]. The ability to conveniently and efficiently combine and transform data from different disciplines is crucial for scientific progress, and computer scientists must stay involved to accelerate progress.

To this end, we provide middleware for filtering, aggregating, and transforming results of simulations of continuous fields, a particularly prolific type of scientific data. The middleware is in the form of a data model and associated algebra that we refer to as the *gridfield model* (GF). We position gridfields between client-side visualization programs and large, potentially distributed data repositories.

Users and application developers write gridfield expressions to extract, transform, aggregate, and restructure datasets using both server and client resources. Client-side visualization programs may be employed to render the results of a gridfield expression and to manage user interaction features such as translation, rotation, and zoom.

1.1 COMPUTATIONAL SCIENCE

All scientific disciplines began as observational activities. As the discipline matures, patterns in the observation inventory emerge, and scientists work to capture and describe these patterns. With increasing rigor, descriptive models evolve into prescriptive, mathematical models (e.g., the periodic table suggested the existence of undiscovered elements). Historically, the models were necessarily simple, since calculations were labor-intensive. With the advent of inexpensive computers, however, complex models (and models of complex domains) have become manageable.

In the physical sciences, mathematical models describe continuous domains and continuous processes as a set of partial differential equations. A computer cannot be used to solve these problems directly—the problem must be re-expressed as a system of discrete algebraic equations. Once re-expressed and solved, the results are interpreted to improve the model and answer scientific questions. This last step—result interpretation—proceeds by creating and studying *data products*. A data product is a derived dataset used to convey a specific scientific message. For example, a data product may be a plot, an aggregate value, or a visualization. The graphics capabilities of modern computers make visualizations a common and effective form of data product; we will use visualizations as the typical case.

In this section, we describe the procedure of computational science in detail using a simple example. The procedure involves six steps:

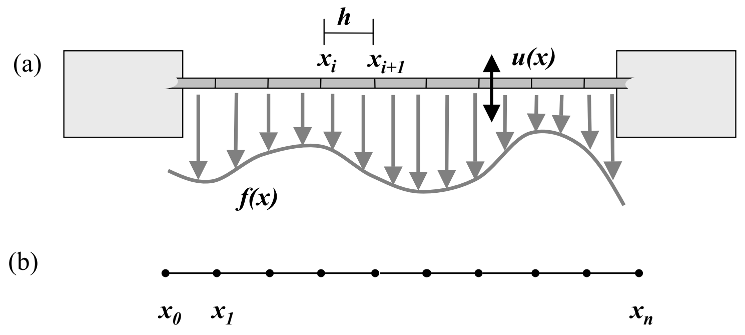


Figure 1.2: (a) A thin rod with fixed endpoints subject to a vertical load $f(x)$. (b) The logical grid describing the rod.

---

1. Select the governing equations and boundary conditions for the physical system under investigation. Only the simplest of problems can be solved analytically; practical problems require numerical methods.
2. Select a particular numerical method and formulate the discrete version of the problem. The choice must balance computational efficiency against accuracy with respect to the characteristics of the problem.
3. Discretize the domain of the problem; that is, overlay a *grid* on the continuous domain to which the solution values will be bound.
4. Use the grid and the numerical method to derive and solve a set of algebraic equations.
5. Evaluate and characterize the error and uncertainty of the solution.
6. Interpret the results via data products to improve the model and answer scientific questions. The creation of data products is the focus of the dissertation.

To illustrate the process, consider a thin rod with fixed endpoints oriented horizontally with a vertical force $f(x)$ acting along its length (Figure 1.2(a)). Under some simplifying assumptions, the deformation of the rod $u(x)$ can be expressed as

$$-E\frac{d^2u}{dx^2} + S\frac{d^4u}{dx^4} = f \qquad \text{for } 0 \le x \le 1$$

where E and S depend on the physical and geometric properties of the rod. Since the rod's endpoints are fixed, we can assume the following boundary conditions

$$u = 0, \quad \frac{du}{dx} = 0 \qquad \text{for } x \in \{0, 1\}$$

One method we can use to solve this problem is finite differences, where the derivatives are estimated as the slope calculated between neighboring points. As with most methods, finite differences requires that we divide the continuous domain of the thin rod into discrete cells whose endpoints will hold the solution. For this simple problem, we can represent the grid as a series of points $x_0, \ldots, x_N$ (Figure 1.2(b)). In this case, we use a uniform grid, so that $x_i = ih$. Using the Taylor series expansion to estimate higher order derivatives, each value in the solution u_i can be expressed relative to its neighbors as follows:

$$E\frac{1}{h^2}(-u_{i+1} + 2u_i - u_{i-1}) + S\frac{1}{h^4}(u_{i+2} - 4u_{i+1} + 6u_i - 4u_{i-1} + u_{i-2}) = f(x_i)$$

The boundary conditions provide an approximation for the set of points x_i, $i \in \{0, 1, N-1, N\}$, and an algebraic system of equations can be derived for the set of points x_i, $1 < i < N-1$:

$$Au = f, \text{ where}$$

$$A = E\frac{1}{h^2}\begin{bmatrix} 2 & -1 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 \\ \vdots & & & & \ddots & \vdots \\ 0 & 0 & 0 & & \cdots & 2 \end{bmatrix} + S\frac{1}{h^4}\begin{bmatrix} 6 & -4 & 1 & 0 & \cdots & 0 \\ -4 & 6 & -4 & 1 & \cdots & 0 \\ 1 & -4 & 6 & -4 & \cdots & 0 \\ \vdots & & & & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 6 \end{bmatrix}$$

Based on the properties of the matrix A, the modeler now selects a method of solving this system. To visualize the results of this simple problem, we can plot

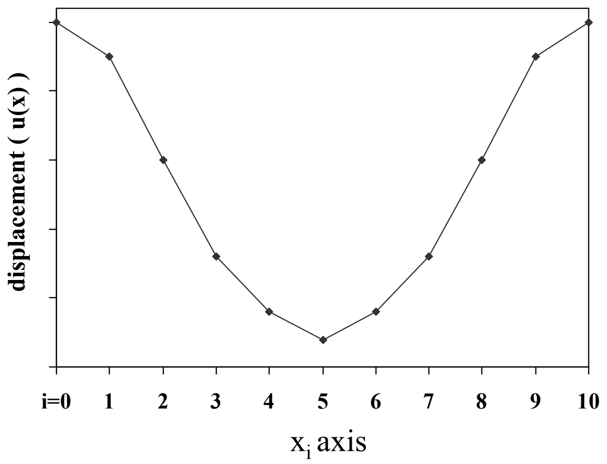


Figure 1.3: A sample solution to the problem in Figure 1.2.

the displacement against x. An illustration of the solution to a sample problem of this form appears in Figure 1.3.

In the QW model of science, we might assume that an individual scientist would carry out Steps 1) through 6) serially. The analog to this thin rod problem in the DW model of science, however, is a database of related problems involving rods with varying properties under different forms of load (point, continuous, clamped), and solved using a variety of numerical methods. Each combination of parameters represents a different experiment, and different groups continually conduct such experiments and log the results. The scientific questions become more holistic.

Of course, the scenario presented here is trivial to carry out for new parameters whenever the need arises; an analytic solution for this particular problem is known. However, if one considers three-dimensional problems, complex domains such as a coastline, simultaneous solution for several variables, a wide variety of grids and solution methods, and enormous scale, analysis of the resulting data can become the bottleneck to discovery and insight.

In Step 6) of the procedure above, computational scientists must experiment with different data products and evaluate how well they convey the behavior of the physical system and the accuracy of their simulation, perhaps with respect to particular phenomena. They therefore need tools that make it easy to design and generate data products.

In the next section, we consider how the solutions are physically organized.

1.2 SOURCE DATA ORGANIZATION

Software used to derive a data product from raw simulation results must parse, access, and manipulate those results in whatever manner they are represented. How are simulation results represented in practice? We find that the status quo unit of data is the *dataset*, defined operationally as the collection of data generated by a single experiment, observation, or simulation. The typical representation of a dataset is as one or more files organized in a directory or directory tree. Descriptive metadata may either be embedded within the files themselves or represented separately.

The advantage of the dataset model is that each datum is associated with the scientific activity that produced it. This form of provenance is required for correct interpretation and also serves to identify the dataset. For example, a series of salinity measurements may be identified as coming from "Oregon State University cruise, June 2005." If anomalies are discovered in these data, researchers can investigate the calibration of the sensors used, the telemetry procedures, or other potential sources of error.

The disadvantage of the dataset model is that the logical decomposition of data into datasets tends to dictate the physical decomposition of data into digital artifacts. Each dataset is materialized as one or more files in a particular format, obscuring the continuous nature of the domain being studied. Simultaneous access to many related datasets to expose large-scale features of the data requires careful scripting by the analyst. The relationships between datasets are entirely implicit in the programs used to manipulate them. For example, many datasets may be defined over the same grid, but be stored separately. Time spent interpreting file formats and gluing together datasets is time not spent interpreting the results and designing new data products.

The dataset model of data is a natural result of the QW model of science. Each dataset is the result of a specific world query, formulated to answer a specific world question. In the DW model of science, however, information about the world are downloaded en masse without regard to a specific scientific question. The challenge, then, is to model and manipulate data captured through DW science.

Since the dataset model does not lend itself to holistic analysis, perhaps a database makes a better choice. Indeed, the concept of storing a large collection of data for use by many users in an ad hoc fashion immediately suggests the features that database systems provide. However, commercial and open-source database software are not extensively used to store scientific data. Gray et al. report encountering the following excuses from scientists for their reluctance to use database software [GLNS$^+$05]:

- We don't see any benefit in them.
- The cost of learning the tools (data definition and data loading, and query) doesn't seem worth it.
- They do not offer good visualization or plotting tools.
- I can handle my data volumes with my programming language.
- They do not support our data types (arrays, spatial, text, etc.).
- They do not support our access patterns (spatial, temporal, etc.).
- We tried them but they were too slow.
- We tried them but once we loaded our data we could no longer manipulate the data using our standard application programs.
- They require an expensive guru (database administrator) to use.

Our experience agrees with these reports: We find that schema design, data ingest, performance tuning, and legacy application rewrites all require significant up-front effort, and, once complete, acceptable performance is not necessarily guaranteed.

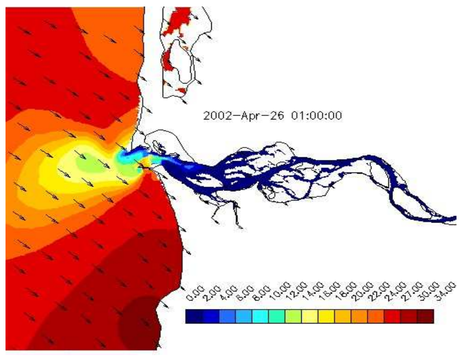
For better or for worse, then, DW science frequently relies on expansive, carefully organized filesystems to store data. To ease the burden of connecting our software to these data, we provide a language for extracting portions of files and exposing them as gridfields. We describe this language in Chapter 4.

Our mechanism for accessing native filesystem data is motivated by the need to integrate with existing repositories rather than replace them. Research projects routinely manage terabytes of information [SvPAG, MC99, TKSG02, HM04a], and discussion of how to manage a petabyte is underway [BW]. Large repositories cannot afford to translate all of their data to a new format whenever new technology becomes available. On-demand translation is feasible, but such tools tend to operate on one file at a time; combinations of multiple files or restriction to a small part of one file must be handled by the application. We describe the structure of a file and allow a program to extract just the portion it needs in the form it needs it.

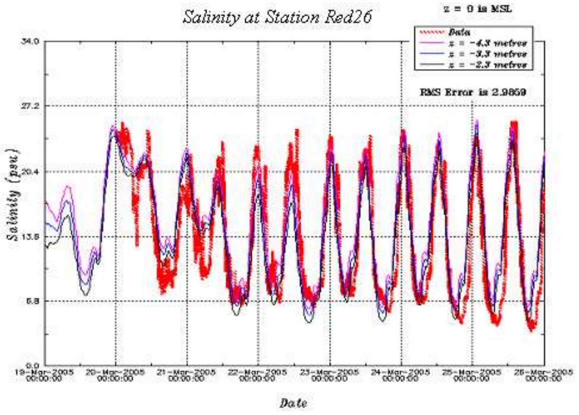
1.3 DATA PRODUCTS

We have characterized the input to the gridfield middleware; in this section we describe the output.

A data product is a visualization or derived dataset computed from the results of the simulation that makes interpretation or validation easier. For example, a data product may be a simple rendering of a scalar dataset (Figure 1.4(a)), or a comparison of simulated data with observational data (Figure 1.4(b)), or the result of a complex computation with no visual component. In practice, we define a data product to be either a visualization (in an on-disk image file format or on the screen) or a derived dataset (in an on-disk file format).



(a)

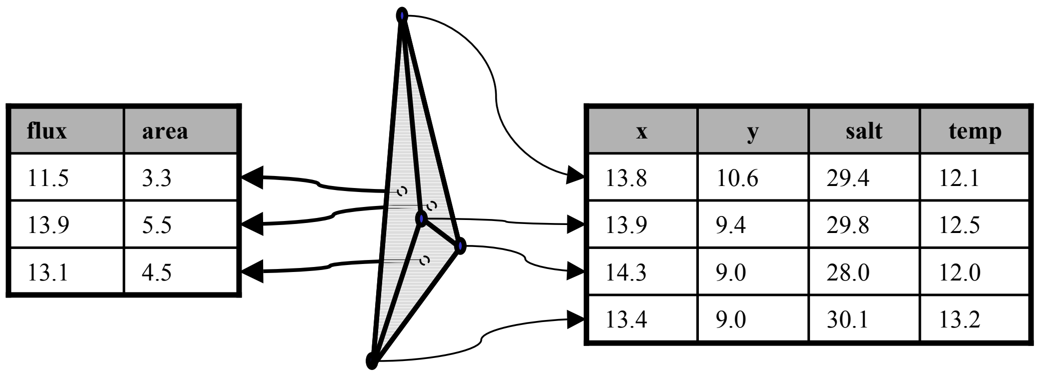


(b)

Figure 1.4: a) A 2-D plot of simulated salinity data at the surface of the Columbia River Estuary. An estuary is an area of mixing between fresh and salt water, usually driven by tides. The arrows represent the wind direction. b) A timeseries of salinity measurements compared with the model output at a particular station in the estuary.

We believe that the data product is beginning to supplant the research paper as the currency of scientific communication: a data product can convey a singular scientific message in a succinct and compelling manner, and is more accessible to non-specialists and the general public. Expression, evaluation, and interpretation of data products are therefore becoming fundamental activities for a computational scientist.

The goal of this work is to simplify the activity of creating data products for simulation results by employing a model of *gridfields* designed especially for the domain. A grid is simply a collection of cells; a *gridfield* is a grid along with data tuples bound to the cells of that grid. Data can be associated with cells of any dimension. Figure 1.5 shows a 2-D 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



flux	area
11.5	3.3
13.9	5.5
13.1	4.5

x	y	salt	temp
13.8	10.6	29.4	12.1
13.9	9.4	29.8	12.5
14.3	9.0	28.0	12.0
13.4	9.0	30.1	13.2

Figure 1.5: Datasets bound to the nodes and polygons of a 2-D grid.

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.

The types of data products one can create is limited by the tools one uses to create them. Simulation results may fit into a standard format such as netCDF [JS92], HDF [hdf04], or FITS [Wel00]. These standard formats come equipped with an API for limited manipulation. The netCDF format, for example, is a storage format for multidimensional arrays, and supplies a "projection" operation for choosing a particular variable and a "subslab" operation for slicing the array along one or more dimensions. However, not all simulation results can be expressed as a multidimensional array, so many system designers use their own proprietary formats and write their own APIs in a general purpose programming language.

Using these APIs, scientists write programs to connect visualization and analysis routines to their data. This approach is oriented toward processing an individual file to generate an individual data product. The DW model of science involves running campaigns of simulations with emphasis on coverage of larger or more refined time and space scales. During analysis, multiple scientists or research groups experiment with a variety of data products.

For example, given temperature, salinity, and velocity data over the estuary illustrated in Figure 1.4(a), an oceanographer might plot the maximum salinity intrusion against time, while an ecologist might be interested in connected regions of "salmon habitability" defined using all three variables.

A general-purpose language can express any data product, but at a tradeoff of development time and maintenance. Many scientists designing many data products can result in redundant effort. In our work with oceanographers, we encountered four different data manipulation programs (three in C and one in Fortran) with significant overlap in functionality. Duplicated code may simply suggest that conventional software engineering principles have been overlooked. However, the common routines were subtly different in their interface and purpose. For example, each program used a nested iterative structure, traversing the timesteps, then the horizontal nodes, then the vertical nodes. However, one program aggregated over depth, another summed the volume of cells that matched a particular criterion, and a third extracted values at a particular point in space. The differences were as prominent as the commonalities, making an appropriate abstraction non-obvious. Our approach is to put aside the specifics of this application for a while and derive simple, sensible operators for *generalized* grids. We are then able to recreate complex, application-specific programs as compositions of simple operators.

Since the fundamental structure used in simulation is the grid, we promote grids to first-class citizens, rather than requiring them to be translated into more traditional structures. The algebra is designed to mimic scientists' own English descriptions of data products as an improvement over programs in a general-purpose language. In this sense, we are proposing a domain-specific language [LM99] for generating data products from the results of numerical simulations in the physical sciences.

1.4 EXAMPLE DOMAIN

We have been working extensively with environmental scientists involved with the CORIE Environmental Observation and Forecasting System (EOFS) being developed at the OGI School of Science & Engineering at Oregon Health & Science University [BWP$^+$99]. The CORIE system, named for the Columbia River Estuary, is a multi-purpose platform for studying the fluid dynamics of coastal waters around the world. 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 1998 to the present. Each day, forecast simulations add thousands of data products and gigabytes of raw data to the repository, while sets of hindcast runs, batches of calibration runs, and individual researchers' experiments are executed concurrently. Our examples will primarily involve the CORIE system, though we will emphasize general principles throughout.

To orient the reader, Figure 1.6 gives an example of a gridfield *recipe* used to visualize a 3-D dataset in the CORIE domain. A recipe is an expression in the gridfield algebra; we avoid the overloaded terms "query" and "plan" so as not invoke presumptions about how these recipes are used or evaluated.

At the left of Figure 1.6, the gridfield **H** (named for its *horizontal* orientation) has attributes x and y and models the surface of the Columbia River estuary. The gridfield **V** (oriented *vertically*) has a single attribute z and models the depth of the estuary. The cross product of **H** and **V** is a gridfield with attributes x, y, and z, and models the full 3-D domain. To specify the dataset we are interested in, we use the bind operator and pass it the identifier *salt* to indicate the salinity variable. To restrict the result to a user-specified region of interest, we use the restrict operator and pass it an appropriate selection predicate (*region*). The result is a gridfield **G** with attributes x, y, z, and *salt*.

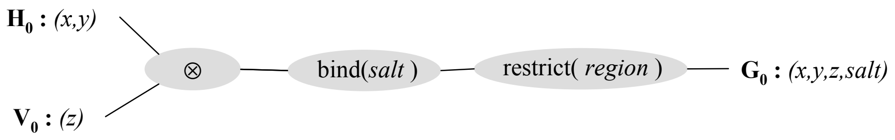


Figure 1.6: A recipe for visualizing a 3-D CORIE dataset.

We observe several challenges in working with simulation applications in the physical sciences, which we describe in the remainder of this section. These challenges are *heterogeneity*, *large scale*, *non-standard data types*, and *complex data products*

Heterogeneity. Science has traditionally rewarded specialization and reductionism, but the limitations of this approach are becoming evident. For example, oceanography is traditionally divided into several distinct subfields. Nearshore oceanographers study shallow waves, storm impacts, and beach dynamics. Coastal oceanographers study currents and tidal processes, but must also consider the effects of the ocean floor and the presence of the coast, such as, friction and upwelling, respectively. Deep ocean oceanographers study global currents and usually disregard terrestrial effects. However, some physical and biological processes cannot be understood without consideration of all of these scales simultaneously. Fish lifecycle patterns, for example, are difficult to model without cross-scale analysis [MtG+99].

To provide a holistic view of complex phenomena, there has been increasing interest in linking data and models defined at different scales and in different disciplines. Each of these oceanographic subfields involves similar computational tools (finite element analysis, visualization). However, rarely do two research institutions use the same file formats or data analysis procedures [ND05]. These differences are often artifacts of technical choices made for convenience or efficiency, rather than reflections of intrinsic incompatibilities.

A direct but infeasible solution to this problem is to require that all data be stored in a standard form, so that all data may be analyzed uniformly. However, scientists must retain the freedom to store and manage their data using specialized techniques, since their applications tend to push the limits of computing and have unique requirements. However, in the interest of data integration, it is critical that any and all commonalities between scientific applications be identified and exploited, at least at the logical level.

To address data integration challenges resulting from heterogeneity, we identify an especially prolific class of data—those datasets produced by scientific simulation—and provide operations for manipulating them at a high level of abstraction. In Chapter 4, we will describe how to use these operations with the ad hoc file formats found in practice. In Chapter 5, we will describe a programming model involving loosely-coupled services that we argue is superior to a single monolithic software package considering the diverse and changing requirements of this domain.

Large scale. Scientific applications intrinsically push the envelope of compute resources. Hardware performance is increasing rapidly, but the demands of these applications scale up in lock step. Give a computational scientist twice the compute power, and she will immediately increase the resolution of the existing simulations, or double the number of simulation runs.

Compute resources are always at a premium in these environments, making slower, general-purpose software unattractive. For example, the overhead of a relational database management system (RDBMS), designed with update-intensive transaction processing in mind, is difficult to justify when a great deal of scientific data is stored as read-only numeric arrays.

To address the large scale of scientific data processing, we tailor our architecture to this domain from the ground up. Though we will show in in Chapters 3 and 6 that existing database systems are not a good fit, we can still leverage 30 years of database research. Management of the memory hierarchy, logical and physical data independence, and data-aware algebraic optimization are valuable concepts that can be applied to many types of software, not just databases.

Non-Standard Data Types. The fundamental data type used in scientific programming is the multidimensional array. The database community has proposed several techniques for handling arrays, though few have made it into commercial systems. NetCDF, HDF, FITS and other file formats come equipped with libraries that provide access and simple query capabilities specialized for arrays. These *nascent databases* [GLNS+05] mimic some of the features of a conventional database, but their ability to model complex grids and express complex manipulations is limited.

Further, many scientific datasets cannot be expressed directly as a multidimensional array. Grids are said to be *structured* or *unstructured*; our model treats both cases uniformly. The grid in Figure 1.7(a) is a 2-D structured grid and the grid in Figure 1.7(b) is a 2-D unstructured grid consisting of triangles. Structured grids have implicit topology and can be modeled satisfactorily 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. In Chapter 2, we will describe a model of grids general enough to express unstructured grids without ignoring the regularity of structured grids. In Chapter 3, we will investigate alternative data structures for representing generalized grids. In Chapter 6, we will evaluate our implementation on structured and unstructured grids.

Complex Data Products. Computations on grid-structured datasets generally produce other grid-structured datasets. To generate these derived datasets, computations may iterate over the cells in a grid, iterate over the neighborhoods of each cell, and access values stored in related datasets. A language for manipulating grids should operate at a level of abstraction that makes these styles of

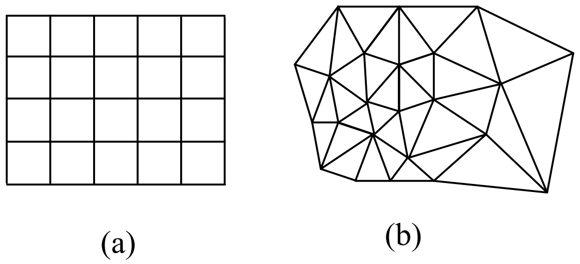


Figure 1.7: (a) A structured grid. (b) An unstructured grid.

computation natural.

For example, algorithms to estimate the gradient of a scalar field involve aggregating values from neighboring cells. However, depending on the representation, access to neighboring cells can be awkward or inefficient. For instance, if a polygonal cell is represented by a sequence of nodes, then each polygonal cell's "neighborhood" of nodes can be accessed directly. However, the neighborhood of cells around a given node is more difficult to compute in this case, requiring a search through all the cells. An alternative representation would store the mapping of nodes to neighborhoods of cells to avoid this search. In either case, the programmer should be insulated from the specifics of the physical representation when possible. The operators we will describe in Chapter 2 can be composed to implement these complex tasks.

In this dissertation, we present and evaluate a model for manipulating gridded datasets designed to address the challenges listed in this section. To demonstrate the efficacy of the model, we provide an implementation informed by an experimental evaluation of various data structures (Chapter 3), connect the implementation to legacy, ad hoc file formats found in practice (Chapter 4), show how the model and implementation may be used as a reasoning tool and programming aid, respectively (Chapter 5), and evaluate the implementation experimentally (Chapter 6). First, however, we will consider related work in this domain.

1.5 RELATED WORK

In this section we position our work amongst data models designed specifically for scientific data, adaptations of more general models to scientific data (relational, GIS, visualization), and application-specific solutions. Work related to the more technical aspects of our results will be presented in context in later chapters.

Relational databases extended with spatial types can model unstructured grids, but have several weaknesses. Explicit foreign keys and redundant geometric coordinates[1] can more than double database size. Since even moderately-sized scientific applications routinely produce several gigabytes or more every day [BW, HM04a, SA02, SKT$^+$00], disk space is at a premium despite falling prices per byte.

Transfer times into and out of the database are also excessive [HM04a]. Retrieving data from the database for manipulation by application programs 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 [TKSG02]. In Section 4.6, we review modeling challenges stemming from storing gridded datasets in relational databases.

Libraries such as the Visualization Toolkit (VTK) [SML96] provide efficient grid processing, but the routines are highly data dependent and programs composed of them are therefore rather brittle with respect to evolving requirements. The library functions also exhibit complex semantics, making algebraic properties difficult to derive if they exist.

Data models designed exclusively for scientific data have been primarily motivated by the need for visualization. Butler and Pendley applied *fiber bundle* structures found in mathematics and physics to the representation of scientific

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

data to support visualization [BP89, BB92, HLC91, Tre99]. Fiber bundles are very general topological structures capable of modeling many diverse classes of objects such as simple functions, geometric shapes such as tori, vector fields, and visualization operations.

Fiber bundles showed great promise in their generality and support for formal reasoning. However, limitations to their direct use for scientific data modeling appeared. First, fiber bundles were developed to model continuous fields (manifolds), and do not take into account the discretization necessary for representing the data in a computer [Mor01]. Second, much of the expressive power of fiber bundles is designed to accommodate complexities that most scientific gridfields do not exhibit. Most scientific gridfields are cast as "trivial" fiber bundles—simple Cartesian cross products—that do not exercise the fiber-bundle machinery. Finally, algebraic manipulation of grid structures is not supported and experimental results are not reported, so it is not clear that performance requirements can be met.

Berti's Grid Algorithms library (GrAL) [Ber00], another system designed to support simulations in the physical sciences, is a hierarchy of C++ classes leveraging generic programming techniques (C++ templates) to improve abstraction. Templates allow the specification of generic algorithms that operate with a variety of different data representations. We adopt a higher level of abstraction, identifying a few generic operations with which to express grid-oriented algorithms. However, Berti's thorough treatment of the topology theory underlying grids strongly influenced our formal development.

The database community has given multidimensional discrete data (MDD) significant attention over the past decade. OLAP systems have been extended with multi-resolution visualization capabilities [STH02], but modeling and querying unstructured grids in a relational system is difficult, as we demonstrate later. Query languages and processing techniques based on multidimensional arrays exist

[DKL$^+$94, LMW96, MS97, WB98], but arrays are not the correct abstraction for general grid manipulations.

Multidimensional arrays can 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, then the grid structure is awkward to encode using arrays. The interpretation of an assembly of arrays as an unstructured grid is left to the application, resulting in programs that are tightly-coupled to specific data characteristics. Further, multiple datasets can be bound to the same grid, but to cells of different dimensions. Using arrays, the relationship between these datasets is lost; each must use its own distinct "spatial domain" [Bau99]. For example, to encode the dataset in Figure 1.5, we would need to define two unrelated spatial domains—one for the 0-cells and one for the 2-cells. The relationship between them is not captured. 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., in different coordinate systems.

Others have demonstrated that relational databases do not scale up to handle large scientific datasets [MC99, SA02]. 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) [MMJ$^+$01]. Papiani et al. [PWN] report some success applying this standard to manage turbulence simulations, though gridded datasets are not directly modeled.

C-Store [SAB$^+$05] and MONET [BK99] both use column-oriented storage and query processing for read-mostly applications instead of conventional tuple-oriented techniques. Scientific data management applications meet the prerequisites that make this approach effective; we also adopt a column-oriented approach. However, the differences between the gridfield algebra and the relational algebra prevent us from using the technology directly.

Designers of GIS are realizing that topological "connection" information can be as important as geometry for modeling and query processing. ESRI's ArcGIS version 8.3 [ESR03] includes topology information modeled as integrity rules. For example, 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 unstructured grids are difficult to model precisely as raster data. Laser-Scan has produced a topology-enabled GIS extension for Oracle called Radius [Wat02]. 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, which has proven useful in our work. Other GIS systems tend to focus on raster data [CdSP$^+$00, WB97, Eas97] (structured grids) and operate primarily in two dimensions. Our language is agnostic with respect to dimensionality and grid representation.

The database community has investigated query-language extensions supporting efficient order-aware collection types, often citing scientific data processing as a motivation [FM95, LMW96, MS97]. However, the link between collection types such as arrays and scientific data processing is a red herring. The ubiquity of arrays in this domain is not evidence that arrays constitute an appropriate data model for scientific analysis, but rather that more specialized data models have not been developed. Further, direct use of arrays and other low-level data structures were the source of the data dependence problem [Cod90] that declarative languages solved. Incorporating them back into one's "declarative" language seems to take a backwards step.

Retreating from the heterogeneity of scientific datasets, some designers do not attempt to model the contents of datasets, treating them as opaque files that support a single operation: retrieval [FB01, FVWZ02, WG93]. Metadata attached to files allow tracking of *data provenance*, or the information about how the dataset was produced. Users of the Chimera system [FVWZ02] can register their analysis programs and input files and allow Chimera to manage execution. This facility gives rise to a notion of *virtual data* that can be derived on demand.

Our work differs in two ways. First, these systems model datasets coarsely as opaque files; we model the contents of the datasets (in the domain of computational science). Second, correct and efficient manipulations of datasets are the responsibility of the programmer in these systems; we offer a specialized language for expressing manipulations.

There also have been efforts to develop combined scientific data management and visualization systems. The Active Data Repository (ADR) [KaC$^+$01] optimizes storage and retrieval of very large multidimensional datasets, including gridded datasets. Queries over the data involve three transformations related to our operators: selecting a subregion, mapping one grid onto another, and processing the results. ADR is highly customized for particular domains. New C++ classes are developed for each application. The Aurora system [Jir94] is a database system for scientific datasets providing persistence, metadata, and limited query facilities.[2] The Aurora data model cannot capture fully unstructured grids, and operations are limited to mapping on-disk data into a "memory set" suitable for processing by a general purpose programming language. Operations supported by this mapping include restriction and type-casting, but not generalized aggregation and combination.

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

[2]The Aurora system described here should not be confused with a more recent stream-processing system by the same name.

transformation engines.

The EXPRESS system [SHT$^+$77] 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). The source model allows us to retrofit a gridfield interface onto in situ data [HM05].

Batory gave a taxonomy of record-oriented file structures used by commercial databases in terms of fields and pointers [Bat85]. Our work similarly provides a description of file structures used to store nested arrays.

The Binary Format Description Language (BFD) [MC03] 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 [WB03] library is also related to our approach. 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 the lengths of the inner arrays are known statically.

The External Data Representation standard (XDR) [Sri95] is a data-format language focused on machine-level number representation issues. Variable-length arrays in XDR must have homogeneous elements (i.e., their elements are not variable-length), and their lengths must be encoded directly prior to the first element. Further, XDR obviously does not describe directory structures, complicating access to datasets that span multiple files.

With the PADS system [FG05], users describe ad hoc file formats using C-like type declarations and validating access methods are generated automatically. Data transformation is left up to the application. We will describe the PADS system in more detail in Chapter 4.

Platforms for scientific query and analysis include AQSIM [LCA+03] and the Active Data Repository [KaC+01]. Both of these systems provide a common platform for accessing a variety of scientific data, but both require centralized storage and management. Other systems such as Chimera [FVWZ02] and Godiva [MWN+04] supervise the execution of data access programs, but rely on users to write them in the first place. We operate in a different space of requirements: We propose access methods for data that the user does not necessarily control.

The Open Data Access Protocol (OpenDAP) [ope05] is comparable to our approach. OpenDAP allows analysis software such as Ferret [HHO+96] or Matlab [HH00] to connect to data sources on the web, and provides a common model for describing data. Institutions wishing to publish their data install an OpenDAP server and make their data available in one of several file formats. Related OpenDAP software provides a language for describing custom file formats, allowing it to be published in the same manner.

Our work is distinguished in that our data model and operations are strictly more general. The OpenDAP data model is based on multidimensional arrays and can express only structured grids. We focus on *unstructured* grids and capture structured grids as a special case. Unstructured grids are more complex than regular grids, primarily since the topology of the grid, the central concept in our model, must be represented explicitly. Our language for manipulating and transforming gridded datasets is also more expressive. OpenDAP operations are limited to "subslicing" of arrays along one or more dimensions and "projection" of particular variables. In addition to these operations, we provide operators that allow gridded datasets to be transformed, aggregated, and combined in a variety of ways.

Papadomanolakis et al. proposed an index for finding an arbitrary point in a tetrahedral mesh [PAL+06]. Their technique is to assign each 3-D cell a position on a space-filling curve. These positions can then be indexed with a conventional B-Tree. A query point is positioned on the space-filling curve and a candidate cell is identified. From that point on, the space filing-curve is ignored. From the candidate cell, the topology of the mesh is traversed to find the actual cell that contains the query point. The index was implemented in the PostgreSQL database management system. We can express this indexing mechanism (logically) as a gridfield expression, demonstrating the broader scope of our work.

1.6 CONTRIBUTIONS

We contribute

1. a logical data model and logical algebra for describing and manipulating simulation results,
2. a physical data model and physical algebra in terms of gridfield components,
3. optimization techniques for converting logical gridfield expressions into efficient physical gridfield expressions,
4. a framework for providing a gridfield interface to native file-based repositories,
5. an implementation of the model including a canonical GUI interface for writing gridfield expressions,
6. empirical evaluation of the implementation against the CORIE Environmental Observation and Forecasting System,
7. evidence of applicability to other scientific and engineering domains including seismology and computational medicine.

The data model and associated operators are described in Chapter 2. The model offers several benefits over competing solutions:

- The data model supports topology directly, independently of any particular geometric interpretation.

- The data model captures regular and irregular grids uniformly.
- The operators manipulate grid structures directly, avoiding the complexity associated with encoding grids as assemblies of arrays.
- The design is well-aligned with client visualization and analysis tools.
- Our operators admit algebraic identities and consequent optimization techniques unique to gridfields.

A discussion of alternative data structures and the design and implementation of our core data structures are described in Chapter 3. We describe how the native formats for scientific data may be accessed generically and efficiently in Chapter 4. The programming model for gridfields and example of its use as a reasoning tool appear in Chapter 5. In addition, we describe a canonical GUI application for writing, executing, and visualizing the results of gridfield expressions. The application was originally designed specifically for the CORIE system, but was generalized to work with any gridfield application. We evaluate our model and implementation against popular competing approaches in Chapter 6 using datasets from various domains, with emphasis on those from the CORIE EOFS. In the last chapter, we summarize our conclusions about the efficacy of gridfields and present an outlook for future applications.

Chapter 2

DATA MODEL

The notion of a *field* is widely used to model the physical world. A field is a function whose domain is a topological space, frequently $\mathbb{R}^n$. In the Earth sciences, the domain usually represents 3-D space or 4-D spacetime.

There have been several data models based on fields proposed in the literature [BP89, Mor01, Tre99]. These efforts indicate the ubiquity of the field concept, but have not gained wide acceptance. The models focus on the underlying spatial domain, providing abstractions to hide the grid structure used to physically represent the domain in the computer. Unfortunately, computations involving fields depend intimately on the choice of grid. These models therefore offer little guidance to the algorithm designer. Our model lifts the grid up as the primary interface for computation.

What are the requirements for a grid-oriented model of continuous fields? The model must capture the types of grids numerical modelers use, as well as the kinds of computations they perform. To extract the requirements for the gridfield model, we first review grid types and grid computations found in practice.

2.1 MODEL REQUIREMENTS

There are many ways to decompose a continuous domain into discrete cells. The choice has a significant effect on computational efficiency, model accuracy, numerical robustness, and ease of implementation. In this section, we will describe the types of grids used in practice. We use the term *mesh* to indicate a a grid associated with a particular geometric realization.

A mesh divides a potentially complex geometric domain into simpler control volumes, which we refer to as *cells*. A mesh provides three kinds of information: the cells it contains, the connectivity between its cells, and the geometric embedding of those cells. Intuitively, each cell has a dimension: a point has a dimension of zero, an edge has a dimension of one, and so on. We use the term cell when we wish to ignore dimension; we will adopt the more intuitive terms point, edge, and face to indicate cells of dimensions 0, 1, or 2, respectively.

Cells (with or without dimension), connectivity, and geometry alone do not fully describe the characteristics of the region being modeled; in practice each cell additionally has associated data values representing either physical quantities (e.g., pressure, velocity, temperature) or computational control values (e.g., identifiers, pointers). We postpone discussion of these associated data values until the presentation of the formal model.

A *uniform rectangular structured mesh* divides the domain into rectangles of equal size and shape (Figure 2.1(a)). Despite the use of the term rectangular, this kind of mesh can be used to describe a domain of any number of dimensions. The rectangles of any structured mesh can be addressed by a vector of indices, and the nearby nodes, edges, and other rectangles can be addressed in constant time using simple arithmetic expressions on this vector. Since this mesh is also *uniform* rectangular, the Euclidean coordinates of a point (i, j) can be computed as $(h_x i + k_x, h_y j + k_y)$, where h_x, h_y, k_x, k_y are constants. Structured meshes allow small representations and efficient access to connectivity information, but cannot precisely describe complex domains.

A *variable rectangular structured mesh* (Figure 2.1(b)) retains the simple connectivity, but allows more control over the geometry. The Euclidean coordinates of a point (i, j) are (x_i, y_j). For $0 < i < n$ and $0 < j < m$, a representation of a variable rectangular structured mesh must, in general, explicitly store n x-coordinates

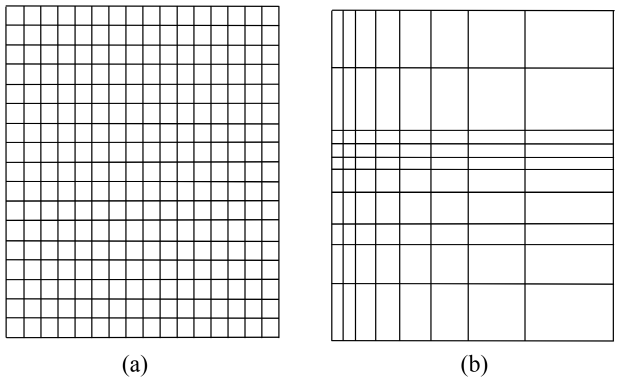


Figure 2.1: Two simple grids. (a) A uniform rectangular grid. (b) A variable rectangular grid.

and m y-coordinates or $n-1$ x-intervals and $m-1$ y-intervals. The advantage of the increased complexity is that the modeler can increase the density of cells in regions of special interest without having to increase the overall number of cells in the mesh.

A variable rectangular structured mesh is still limited to describing a rectangular domain. Structured meshes can also be *deformed* to accommodate more complex regions, as in Figure 2.2. Here, the structured mesh is fit to a curved three-dimensional geometry representing the space around a blunt fin [HB84]. The desirable properties of the connectivity information are retained, but the geometric information must, in general, be fully explicit. That is, in three dimensions, the Euclidean coordinates of a point (i, j, k) are $(x_{ijk}, y_{ijk}, z_{ijk})$. For $0 < i < n$, $0 < j < m$, and $0 < k < p$, the mesh must store nmp coordinate triples in the general case. The term "structured mesh," without qualifying adjectives, usually indicates a deformed structured mesh.

Deformation of a structured mesh leads to some difficulties. Computations of flux at the interfaces between cells now involve a tangential and normal component, which vary from cell to cell. The curvilinear boundaries of cells also complicate

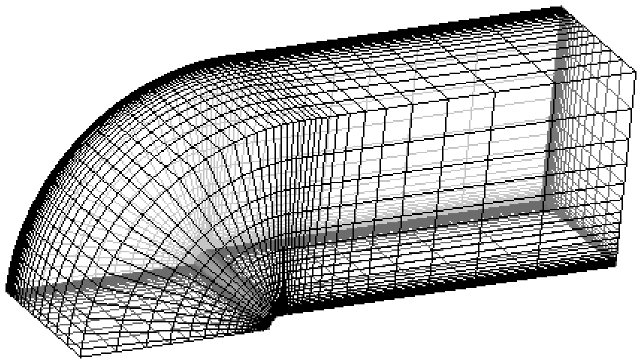


Figure 2.2: A deformed structured grid modeling airflow over a blunt fin rising from a flat plate.

interpolation at points falling within a cell [HT04a] (or even determining which cell contains a given point). Finally, deformation of a structured mesh is not sufficient to fit every conceivable domain. Discontinuities produce degenerate cells with a zero or negative area.

As an alternative to deformation, a *masked* structured mesh may be used to indicate that certain portions of the mesh are invalid. The mask is implemented with Boolean values associated with each cell. Algorithms may break iteration early to skip regions of invalid cells, or return immediately when encountering an individual invalid cell, to improve performance. The boundary of a masked mesh will be a jagged, "stair-step" approximation of a smooth domain boundary, but since the shape of the cells are still rectilinear, interpolation on unmasked cells is no more difficult than with a rectangular structured mesh.

To achieve more precision in cell density, modelers can abandon a structured mesh for an *unstructured* mesh. The unstructured mesh for a wing in Figure 2.3 [AMW98] conforms well to the wing's shape, a difficult feat for a structured mesh. An unstructured mesh has an irregular geometry, like the deformed structured mesh, but also has an irregular connectivity. The neighbors cannot be determined

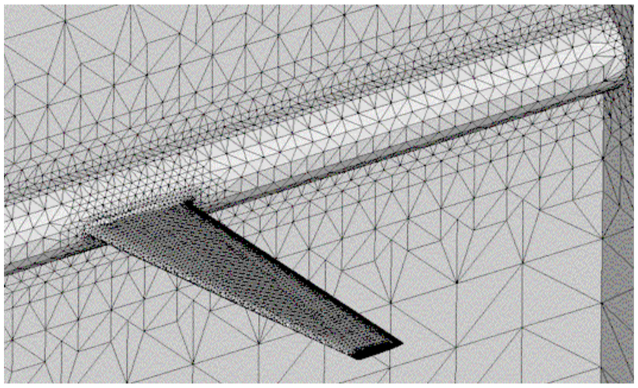


Figure 2.3: An unstructured grid modeling the airflow around a wing.

using index arithmetic. Traversing the neighborhood of a given cell involves potentially random access to a data structure on the order of the size of the grid.

A *block-structured mesh* is composed of multiple structured meshes describing the overall region. The boundary between the blocks must obey certain constraints, depending on the method used. Cells sharing a boundary may have a one-to-one relationship (Figure 2.4(a)), a many-to-one relationship (Figure 2.4(b)), a many-to-many relationship (Figure 2.4(c)), or they may overlap (Figure 2.4(d)). The CHIMERA grid generation package [SDB83] uses overlapping structured grids to improve interpolation near the boundaries, for example.[1] A *hybrid mesh* is similar to a block structured mesh, but each block may be unstructured. The DRAGON hybrid grid generator replaces the overlapping regions of a CHIMERA grid with an unstructured grid [LK94].

The requirements suggested by these mesh types include access to nodes, edges, and higher-dimensional cells, the connectivity between them, and geometric information. Topology information can be computed implicitly for structured meshes, but must be looked up for unstructured meshes. Combinations of different types

[1]The CHIMERA grid generation package should not be confused with the Chimera Virtual Data System described in Chapter 1.

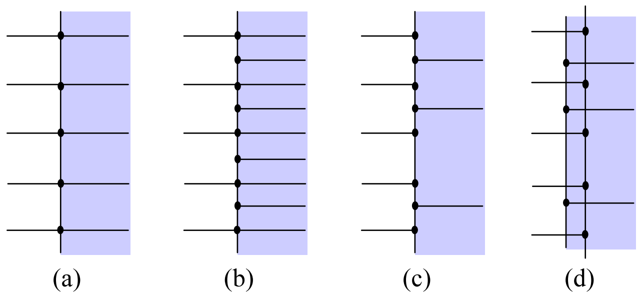


Figure 2.4: The possible interfaces between blocks of a multi-block mesh: (a) one-to-one, (b) one-to-many, (c) many-to-many, (d) overlapping.

of grids are possible. This diversity suggests that no one data structure is suitable for all applications, and indeed, there have been a great many data structures proposed for a variety of specific applications.

Before we discuss our physical representations and data structures in Chapter 3, we propose a general model of sets of cells equipped with 1) an incidence relation capturing topology, and 2) functions mapping cells to data values. By providing an explicit incidence relation, the model is oriented towards unstructured meshes. This choice is sensible, since unstructured meshes are strictly more general than structured meshes.

This simple model can capture the topology of structured and unstructured meshes, but two other requirements must be addressed: geometry information and hybrid (multi-block) meshes. We model geometry information as ordinary data values associated with the cells, as any other data might be associated with the cells (for example, the boolean values implementing a masked mesh.) The model for a multi-block mesh assumes that the blocks can be logically flattened into a single mesh during computation. For example, the incidence relation can be extended to accommodate the interfaces in Figure 2.4.

Before we discuss the requirements derived from computations over meshes, we formalize the grid concept.

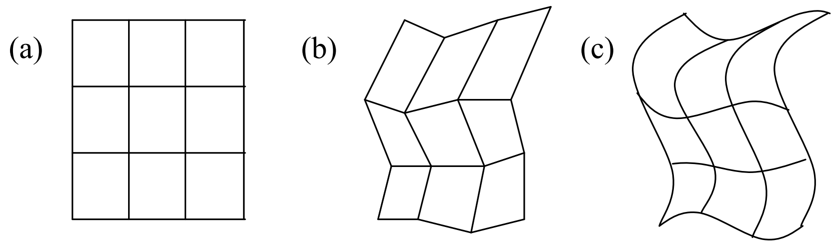


Figure 2.5: Three different geometric realizations of the same topological grid.

2.2 THE GRIDFIELD MODEL

Existing data models for gridded datasets consist of two components:

- A representation of the topology and geometry, and
- A representation of the data defined over the topology.

Most systems and practitioners use the word geometry broadly to refer to the first component; few use the word topology. It is important to realize that the two concepts are distinct. The study of topology arose from the observation that some problems depend only on connection properties (such as adjacency and containment) and not the geometric properties (such as shape or size). For example, Figure 2.5 shows three geometric realizations of the same topological grid. Topology involves only the properties of neighborhoods of points and not absolute positions in a metric space. Our data model makes the same distinction, and does not require absolute positions in space nor time. This approach reduces the complexity of the data model; geometric data require no special treatment. We view geometric data as just that-data. We claim that grid topology is the distinguishing feature of scientific data, and the only feature requiring specialized solutions. All other data can be modeled as functions over the topological elements.

In this section we describe the topology component of the model—the grid—then we add data to create a gridfield. Operators for both grids and gridfields along with examples will be described throughout.

2.2.1 Grids

Grids are constructed from sets of *cells* of various dimension connected by an incidence relationship. We refer to a cell of dimension k as a k-cell, following the topology literature [Ber00]. Intuitively, a 0-cell is a point, a 1-cell is a line segment (or poly-line), 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.

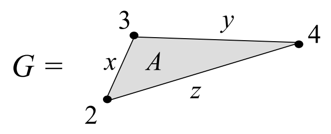
Our grid model affords interpretation in terms of well-studied concepts from topology, particularly *cellular complexes* (c.f. Fitsch and Piccinini [FP90]). However, we have made an effort to avoid strict dependence on these ideas, for two reasons. First, very little of the mathematics of topology is directly implementable in the computer without a suitable representation theory. Second, the management of data bound to topological structures requires a different set of tools than the topology field has to offer; the database community has significant experience designing such tools and their results should be integrated where possible.

Let $\mathcal{C}$ be a universe of featureless *cells*. Let dim be a function $\mathcal{C} \to \mathbb{N}$ assigning a non-negative integer *dimension* to each cell in $\mathcal{C}$. Then a grid G is a pair $(X, \preceq_G)$ where X is a finite subset of $\mathcal{C}$, and $\preceq_G$ is an *incidence relation.* An incidence relation is a partial order on X that respects dimension; i.e., lower-dimensional cells are incident to higher-dimensional cells.

We will write k-cell to indicate a general cell c such that $\mathsf{dim}\,(c) = k$. In Figure 2.6, the grid has three 0-cells, three 1-cells, and one 2-cell. The incidence relation in the figure makes each node incident to two edges, each edge incident to the triangular face, and by the transitivity of a partial order, each node is also incident to the face A.

In our illustrations, we will use draw nodes (0-cells as black points, 1-cells as line segments, and 2-cells as shaded regions. From these kinds of pictures, we can

$$\boxed{G = (X, \preceq_G)}$$

$G =$ 

$X =$ 2, 3, 4, x, y, z, A

$\preceq_G =$

2	z
4	z
z	A
3	y
4	y
y	A
2	x
3	x
x	A
0	A
1	A
2	A

Figure 2.6: The components of a grid: A set of cells, labeled here by integers (0-cells), lowercase letters (1-cells), and uppercase letters (2-cell).

---

unambiguously derive a grid by creating a cell for each drawing component, and establishing an incidence relationship between cells if they touch in the drawing and their dimensions are different.

Intuitively, the incidence relation encodes which lower-dimensional cells "touch" a higher-dimensional cell. Specifically, $x \preceq y$, read "x is incident to y," implies that either $\mathsf{dim}(x) < \mathsf{dim}(y)$ or that $x = y$, so that two distinct cells with the same dimension cannot be incident to each other. We also define $\prec$ to be the anti-reflexive restriction of $\preceq$. In Figure 2.6, for example, $2 \prec x$, but $2 \not\prec 2$. We will write $c \gtrless d$ to indicate that $c \prec d$ or $c \prec d$. For example, $A \gtrless x$ and $x \gtrless A$ in Figure 2.6. If two cells x and y of the same dimension are incident to a common cell or have an incident cell in common, then we say that x is *adjacent* to y. That is, x is adjacent to y if $\mathsf{dim}(x) = \mathsf{dim}(y)$ and there exists a cell c such that $x \gtrless c$ and $y \gtrless c$. In Figure 2.6, 2 is adjacent to 3 and 4, and x is adjacent to y and z. This definition of adjacency is captures the relationship between cells that are "one hop" apart, regardless of the dimension of the cell being "hopped."

Given a grid $G = (X, \preceq_G)$, we will write G_k to indicate the cells in X of a particular dimension k. That is, $G_k = \{x \mid x \in X, \mathsf{dim}(x) = k\}$. We define

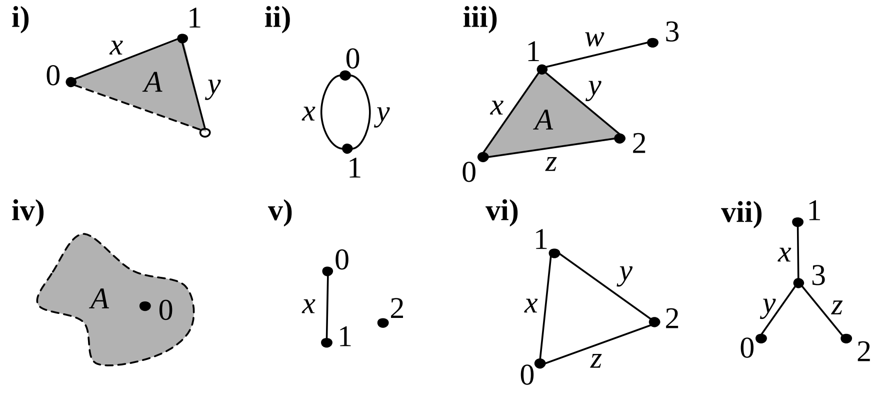


Figure 2.7: Examples of grids allowed by the basic model.

the dimension of a grid G, written $\mathsf{dim}(G)$, as $\mathsf{max}(\mathsf{dim}(x))$ over all $x \in X$. The grid in Figure 2.6 is dimension 2. Note that for a d-dimensional grid, G_d must be non-empty, but G_i may be empty for $0 \leq i < d$.

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 one of our applications are used to discretize the Columbia River estuary. The resulting grid is used in solving the 3-D transport equations via a finite-element method.

2.2.2 Grid Properties

The generality of our grid model invites specification of some pathological grids. Some of the grids in Figure 2.7 may be problematic for specific applications. To help reason about what constitutes an appropriate grid in a particular instance, we formalize several properties of grids. For the remainder of this section, we will refer to a d-dimensional grid $G = (X, \preceq)$.

A grid G is ***homogeneous*** if all k-cells, for any $k < d$, are incident to at least one d-cell. Homogeneity reflects the intuition that d-dimensional grids are often manipulated as a set of d-dimensional cells, where isolated nodes and edges are disallowed. In Figure 2.7, only iii) and v) are not homogeneous.

A grid is ***connected*** if every cell is reachable from any given cell by traversing the incidence relationships in either direction. Let the predicate $\mathsf{reachable}(x, z)$ be true if $x \gtrless z$ or there exists a cell y such that $\mathsf{reachable}(x, y)$ and $\mathsf{reachable}(y, z)$. Then G is connected if for all $x, y \in X$, $\mathsf{reachable}(x, y)$ holds. In Figure 2.7, only v) is not connected.

The grid G is ***embeddable*** if every $(d-1)$-cell x is incident to no more than two d-cells. Intuitively, this property captures the *topological dimension* of a grid. If a 1-dimensional grid cannot be embedded in a line segment, then its topological dimension is greater than 1. In Figure 2.7, only vii) is not embeddable. Note that vi) *is* embeddable, even though it represents a closed loop: It is embeddable in a 1-dimensional space, as is a circle.

The *closure grid* $\overline{c}_G$ of a cell c with respect to a grid $G = (X, \preceq_G)$ is a grid $(C, \preceq_C)$ such that C is the set of all cells in X incident to c, and $x \preceq_C y$ if and only if $x, y \in C$ and $x \preceq_G y$. The closure grid c_G is the unique subgrid of G consisting of the cells $\{d \mid d \preceq_G c\}$.

A cell is said to be ***well-supported*** in a grid according to the following rules:

- All 0-cells are well-supported.
- A 1-cell c is well-supported in G if there are exactly two distinct 0-cells in G incident to it.
- Let $C = \overline{c}_G$ for a k-cell c in G, where $k > 1$. The cell c is well-supported in G if 1) C_{k-1} is not empty, 2) every cell in C_{k-1} is well-supported, and 3) every cell in C_{k-2} is incident to exactly 2 cells in C_{k-1}.

Intuitively, the second condition states that only simple line segments are well-supported 1-cells. The third condition says that the boundary of a well-supported cell must be continuous. Every node incident to a well-supported face must be incident to two edges. Every edge incident to a well-supported volume must be incident to two faces. In Figure 2.7(i), A is not well-supported since the 0-cell 0 is incident to only one edge, x. Also, A is not well-supported since y is not well-supported. In Figure 2.7(iv), A is not well-supported since it has no 1-cells incident to it.

A grid G is said to be well-supported if all of its cells are well-supported in G.

A k-cell c is *unique* in G if $k = 0$, or if the set of cells incident to it is unique and non-empty. That is, a cell x is unique with respect to G if $\mathsf{dim}(x) = 0$, or for all $y \in X$, $\overline{x}_G - \{x\} = \overline{y}_G - \{y\}$ implies $x = y$. A grid is ***minimal*** if all its cells are unique. In Figure 2.7, ii) is not minimal since both x and y have the same set of incident cells, $\{0, 1\}$.

These properties provide a combinatorial characterization of intuitively correct grids. Topological formulations of similar properties can be found in the literature [FP90, Ber00, Lie94, Kov89]. However, the connection between the combinatorial structure manipulated in the computer and the formal mathematics is sometimes tenuous. For example, a topological grid is *manifold* if the local neighborhood of every point resembles Euclidean space, even if the global structure is more complicated. Authors do not agree in the literature on a unilateral combinatorial definition of this topological constraint. One definition due to Kovalevsky [Kov89] asserts that a pseudo-manifold is *non-branching*, *homogeneous*, and *strongly-connected*. The non-branching property requires that each $d-1$-cell be incident to no more than 2 d-cells, but allows it to be incident to 0 d-cells. Our definition of homogeneous matches Kovalevsky's. A grid G is strongly-connected if every d-cell can be reached by every other d-cell via a path of the form $c_0^d, c_1^{d-1}, c_2^d, \ldots, c_{n-1}^{d-1}, c_n^d$, where c_i^k is in G_k, meaning that, for example, there can be no 3-cells that are connected by an edge only. We use this definition in our discussions.

We chose to develop a grid model without reference to topological theory to allow greater flexibility in designing data structures. This choice allows us to reason about the data structures used in practice that do not have formal underpinnings.

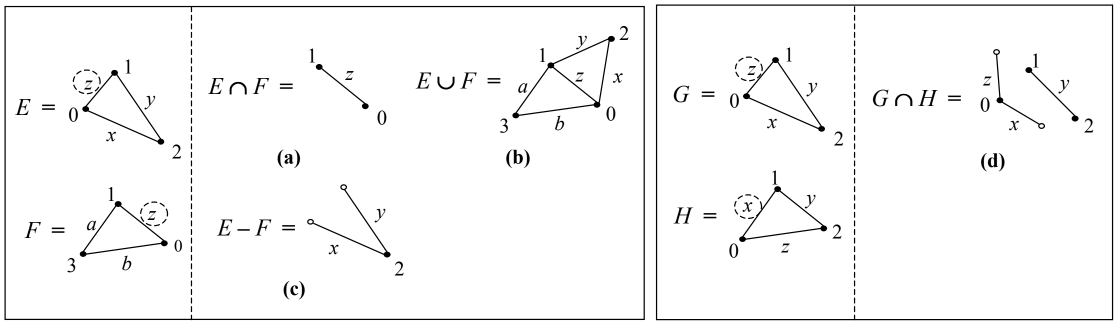


Figure 2.8: Examples of grid operations: (a) intersection, (b) union, (c) difference. G and H at the right are incompatible; they involve the same cells in conflicting incidence relationships. (d) The intersection of these incompatible grids is probably not useful.

Specifically, we can explain why these data structures offer improved performance in many cases. We will return to this discussion in Chapter 3.

2.2.3 Grid Relations and Operations

We can define set-like operations on grids, though we impose a constraint to ensure that the results are useful. First, we define the notion of subgrid.

Definition 1. Let E be a grid $(X, \preceq_E)$ and F be a grid $(Y, \preceq_F)$. The grid E is a ***subgrid*** of the grid F, written $E \subseteq F$, if and only if $X \subseteq Y$ and for all $x, y \in X$, $x \preceq_E y \Leftrightarrow x \preceq_F y$.

This definition mandates that both grids must possess exactly the same incidence relationships for the cells they share (rather than allowing $\preceq_E$ to be an arbitrary subset of $\preceq_F$). As one expects, two grids that are mutual subgrids are equivalent.

Proposition 1. *Let E and F be grids. Then $(E \subseteq F$ and $F \subseteq E) \Rightarrow E = F$.*

Proof. Let $E = (X, \preceq_E)$ and $F = (Y, \preceq_F)$ be grids such that $E \subseteq F$ and $F \subseteq E$. Then $X \subseteq Y$ and $Y \subseteq X$ by definition, and therefore $X = Y$. The incidence relations are also equivalent, since $x \preceq_E y \Leftrightarrow x \preceq_F y$. $\square$

The definition of subgrid may seem straightforward, but consider the grids G and H at the right of Figure 2.8. Both G and H use the same cells but organize them differently. The edges x and z, circled with dashed lines, have different nodes incident to them in each grid. The intersection of G and H, shown in the figure, is probably not very useful. In fact, the result of $G \cap H$ is not even a subgrid of G nor of H by the definition given. The reason is that these grids use the same cells, but with contradictory incidence relationships. The following definition makes this intuition precise:

Definition 2. Let E and F be grids. If there exists a grid G such that $E \subseteq G$ and $F \subseteq G$, then we say that E and F are **compatible** under G. When the grid G is not important to the discussion, we simply say that E and F are compatible.

Intuitively, compatibility prevents mixing grids from different "universes." For example, the grid in Figure 2.2 is unrelated to the grid in Figure 2.3, so their intersection (for example) is not well-defined. We define grid union and grid intersection with respect to compatibility.

Definition 3. Let E and F be grids. The ***grid union*** $E \cup F$ is a grid $(X \cup Y, (\preceq_E \cup \preceq_F)^+)$, where $+$ denotes transitive closure.

The union of two partial orders is not necessarily a partial order, so the naïve definition of grid union, $(X \cup Y, \preceq_E \cup \preceq_F)$, is not necessarily a grid. To generate a partial order from the union of two partial orders, two things can go wrong. First, there could be conflicts; e.g, would $x \prec_E y$ and $y \prec_F x$ for some $x, y \in X \cap Y$? The answer is no, since this condition implies that $\mathsf{dim}(x) < \mathsf{dim}(y)$ and $\mathsf{dim}(y) < \mathsf{dim}(x)$, a contradiction. Second, a partial order must be transitively closed; e.g., if $x \prec_E y$ and $y \prec_F z$, then (x, z) is not necessarily in $\preceq_E \cup \preceq_F$. For this reason, we define grid union in terms of the transitive closure of the incidence relations. An example of grid union appears in Figure 2.8(b).

Proposition 2. *Let $E = (X, \preceq_E)$ and $F = (Y, \preceq_F)$ be grids. Then $E \subseteq E \cup F$ and $F \subseteq E \cup F$ if and only if E and F are compatible.*

Proof. Consider

$$E \subseteq E \cup F$$
$$F \subseteq E \cup F$$

Therefore E and F are compatible under $E \cup F$ by definition. For the if case, let E and F be compatible under some grid G. Then $X \subseteq X \cup Y$, but we must prove that for all x, y in X, $\preceq_E$ is equivalent to $\preceq_\cup$. Let $\preceq_\cup = (\preceq_E \cup \preceq_F)^+$. Since $\preceq_E \subseteq \preceq_\cup$, we know that

$$x \preceq_E y \Rightarrow x \preceq_\cup y$$

For the converse, consider that

$$\preceq_E \quad \subset \quad \preceq_G$$
$$\preceq_F \quad \subset \quad \preceq_G$$

and therefore

$$\preceq_E \cup \preceq_F \subset \preceq_G$$

Since $\preceq_G$ is transitively closed,

$$\preceq_\cup = (\preceq_E \cup \preceq_F)^+ \ \subset \ (\preceq_G)^+ = \preceq_G$$

Let c, d be cells in X. Then

$$c \preceq_\cup d \Rightarrow c \preceq_G d \Rightarrow c \preceq_E d$$

□

If two grids are compatible under G, then their union is a (not necessarily proper) subgrid of G. Properties of G that are preserved through the subgrid relationship, then, are preserved through the union operation. Minimality, in particular, is preserved.

Definition 4. Let E and F be grids. The ***grid intersection*** $E \cap F$ is a grid $(X \cap Y, \preceq_E \cap \preceq_F)$; see Figure 2.8(a).

The intersection of two grids E and F would intuitively be a subgrid of both E and F. The following proposition relates this property to compatibility.

Proposition 3. *Let E and F be grids. Then $E \cap F \subseteq E$ and $E \cap F \subseteq F$ if and only if E and F are compatible.*

Proof. Let $E = (X, \preceq_E)$ and $F = (Y, \preceq_F)$ be grids. Then the proof for the if case proceeds fairly mechanically as follows.

$$\begin{aligned}
&\text{Let there exist a grid } G \text{ such that}\\
&E \subseteq G \text{ and } F \subseteq G\\
\Rightarrow &\forall x, y \in X,\ x \preceq_E y \Leftrightarrow x \preceq_G y\\
&\qquad \text{and } \forall x, y \in Y,\ x \preceq_F y \Leftrightarrow x \preceq_G y\\
\Rightarrow &\forall x, y \in X \cap Y,\ ((x \preceq_E y \Leftrightarrow x \preceq_G y)\\
&\qquad\qquad \text{and } (x \preceq_F y \Leftrightarrow x \preceq_G y))\\
\Leftrightarrow &\forall x, y \in X \cap Y,\ ((x \preceq_E y \Leftrightarrow x \preceq_G y)\\
&\qquad\qquad \text{and } ((x \preceq_E y \text{ and } x \preceq_F y) \Leftrightarrow x \preceq_G y))\\
\Leftrightarrow &\forall x, y \in X \cap Y,\ ((x \preceq_E y \text{ and } x \preceq_F y) \Leftrightarrow x \preceq_E y)
\end{aligned}$$

The last line, together with the fact that $X \cap Y \subseteq X$, proves that $E \cap F \subseteq E$. A similar argument shows $E \cap F \subseteq F$.

For the only if case, we assume that $E \cap F \subseteq E$ and $E \cap F \subseteq F$ and prove that $E \subseteq E \cup F$. Let c, d be cells in X. If $c \preceq_E d$, then $c \preceq_{E \cup F} d$ by the definition of $\preceq_{E \cup F}$.

If $c \preceq_{E \cup F} d$, we must show that $c \preceq_E d$. If $c \preceq_E d$, we are done. Otherwise, there are two cases to consider:

Case 1: If $c \preceq_F d$, then c, d in Y, so c, d in $X \cap Y$. Then by $E \cap F \subseteq F$,

$$c \preceq_F d \Rightarrow c \preceq_{E \cap F} d$$

and by $E \cap F \subseteq E$,

$$c \preceq_{E \cap F} d \Rightarrow c \preceq_E d$$

Case 2: If $c \not\preceq_F d$, then let k be the smallest integer such that

$$c \preceq_{E \cup F} e_1 \preceq_{E \cup F} e_2 \preceq_{E \cup F} \ldots e_k \preceq_{E \cup F} d$$

Then for all $i < k$, $e_i \notin X$, since otherwise i would be a smaller integer than k meeting the same condition. Therefore e_i must be in Y for $0 < i < k$, so $c \preceq_F d$. By Case 1, $c \preceq_E d$.

□

A naïve definition of grid difference is $E - F = (X - Y, \preceq_E - \preceq_F)$. However, the result grid with this definition could be "over-specified," where its incidence relation involves cells that are not present in the grid. An example of grid difference appears in Figure 2.8(c).

Definition 5. Let E and F be grids. The ***grid difference*** $E - F$ is a grid $(H, \preceq_E \cap (H \times H))$, where $H = X - Y$.

The expected subgrid relationship, $E - F \subseteq E$ does not require compatibility. No special relationship between E and F is necessary to produce a subgrid of E, since the incidence information of F is ignored in the definition.

Proposition 4. *Let E and F be grids. Then $E - F \subseteq E$ in all cases.*

Proof. Let $E = (X, \preceq_E)$ and $F = (Y, \preceq_F)$ be grids. Then

$$E - F = (H, \preceq_E \cap (H \times H))$$

where $H = X - Y$. The relations $\preceq_E \cap (H \times H)$ and $\preceq_E$ obviously agree on H. □

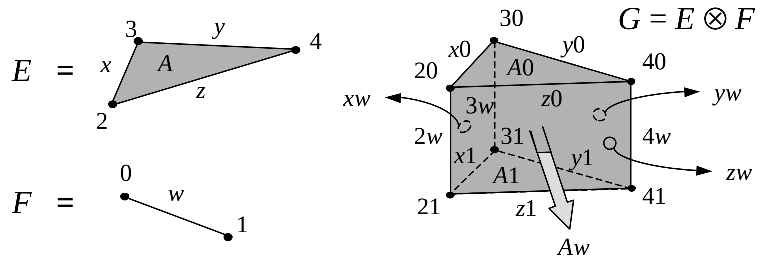


Figure 2.9: The cross product of two simple grids.

The significance of compatibility is that generally grids can be combined in meaningful ways only if there is a reference grid of which they are all subgrids. Without this constraint, two different applications may organize the same cells in different ways, resulting in *incompatible* grids. This constraint is fairly intuitive in practice. The CORIE application, for example, has made use of several distinct grids to describe the same geometric space on the Earth's surface as algorithms are refined and tested, new compute platforms are deployed, and different phenomena are studied. The cells of these grids are assumed to be disjoint. In order to compare data on these two incompatible grids, we must first map the data of one grid to the cells of another grid explicitly. Reasonable assignments will depend on geometry or other external data, rather than the topology alone. In Section 2.4, we describe the *regrid* operator that implements such assignments.

We can also define a cross-product operator, used to produce a higher-dimensional grid from two lower-dimensional grids. The definition of grid cross product involves the notion of pairs of cells. Specifically, we assume that for cells x and y in the universe $\mathcal{C}$, the pair (x, y), the *cell product*, is also a cell in $\mathcal{C}$. Further, $\mathsf{dim}((x, y)) = \mathsf{dim}(x) + \mathsf{dim}(y)$. This rule tells us that the product of two nodes is another node $(0 + 0 = 0)$, the product of an edge and another edge is (informally) a quadrilateral $(1 + 1 = 2)$, and so on. There are multiple means of generating a cell at a particular dimension. For example, the product of a node and an edge is an edge, as is the product of an edge and a node. In our discussion, we will write xy for the pair (x, y).

Figure 2.9 shows an example of the cross product operation. The cross product of grids E and F 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 nodes.

The 3-cell prism in G is just the pair Aw. Geometrically, this prism is generated by sweeping the triangle A through a third dimension defined by the line segment w. This geometric interpretation of the cross product operation is instructive, but is not formally a part of the grid definition. All we know about the prism is encoded in the explicit incidence relation between its constituent cells. For example, the figure suggests that in the result grid G, the 1-cell $x0$ is incident to the 2-cells $A0$ and xw. This relationship makes sense, since in the grid E, the 1-cell x was incident to the 2-cell A, and in the grid F, the node 0 was incident to the 1-cell w.

We can further explain the intuition behind cross product by considering one dimension at a time. The cells of G, by dimension, are given by

$$\begin{aligned}
G_0 &= E_0 \times F_0 \\
G_1 &= (E_1 \times F_0) \cup (E_0 \times F_1) \\
G_2 &= (E_2 \times F_0) \cup (E_1 \times F_1) \\
G_3 &= E_2 \times F_1
\end{aligned}$$

Evaluating these expressions, we obtain

$$\begin{aligned}
G_0 &= \{20, 30, 40, 21, 31, 41\} \\
G_1 &= \{x0, y0, z0, x1, y1, z1, 2w, 3w, 4w\} \\
G_2 &= \{A0, A1, xw, yw, zw\} \\
G_3 &= \{Aw\}
\end{aligned}$$

Definition 6. Let $E = (X, \preceq_E)$ and $F = (Y, \preceq_F)$ be grids, and let $d = \mathsf{dim}(E) + \mathsf{dim}(F)$. The ***cross*** of E and F, written $E \otimes F$, is a grid $G = (X \times Y, \preceq_G)$ where $pq \preceq_G rs$ if and only if $p \preceq_E r$ and $q \preceq_F s$. The cells of G for a particular dimension k are given by $G_k = \bigcup_{j=0}^{k} X_j \times Y_{k-j}$, for $0 \leq k \leq d$.

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. Oceanographers refer to this construction (where the depth grid is linear) as an *extruded* grid. The cross operator generalizes the concept of an extruded grid. Time 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 expense 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.

2.2.4 Gridfields

Consider a digital elevation map (DEM): a triangulation of space (the grid), where each node is assigned a height above sea level (the bound data). A DEM cannot be expressed as a grid, but it can be expressed as a gridfield.

When data are bound to a grid G, the result is a gridfield $\mathbf{G}$. The set of all gridfields is written $\mathcal{GF}$. Bound data are organized into *attributes*. Let $\mathcal{V}$ be a universe of values. A distinguished value $\perp$, assumed to have a distinguished type $\langle\perp\rangle$, is included in $\mathcal{V}$. We will sometimes use the term *rank* when speaking of a dimension at which data is bound; e.g., “the attributes at rank k.”

To define a DEM over a triangulation G, we need to create an attribute representing the elevations at each node. The values themselves are modeled as a function h from nodes to floating point numbers. To create an attribute, we equip h with a name, the grid on which it is defined, an integer dimension (0 in this case), and a return type. Since the type of the function h depends on the grid G, we cannot typecheck gridfield expressions in conventional type systems. Rather than require a dependent type system, we reify the function's domain and range.

Generally, an attribute is a triple (a, τ, f) where a is a string (the attribute name), τ is a type, and f is a function mapping cells in G_i to values of type τ for some dimension i. If c is a cell in G_i and (a, τ, f) is an attribute, we will use $a(c)$ and $f(c)$ interchangeably to indicate the value of the attribute named a associated with the cell c.

Each rank of a grid may be associated with a set of attributes. A gridfield is written $(G, g_0, g_1, \ldots, g_n)$, where each g_i is a set of attributes such that each member of g_i has the form (a, τ, f). The grid of a gridfield may be written $\mathsf{grid}(\mathbf{G})$, and the set of attributes for a gridfield may be written $\mathsf{field}(\mathbf{G})$. The *scheme at rank* i of a gridfield $(G, g_0, g_1, \ldots, g_n)$, written $\mathsf{sch}_i(\mathbf{G})$ is the set $\{(a, \tau) \mid (a, \tau, f) \in g_i\}$. This construct is used for notational convenience. At rank 0, the scheme of the DEM we are building is simply $\{(\text{“}\textit{elevation}\text{”}, \mathsf{float})\}$. We constrain this set to have unique attribute names. A *tuple* for a cell $c \in G_i$, written $g_i[c]$ is a set $\{(a, f(c)) \mid (a, \tau, f) \in g_i\}$. The value of an individual attribute named a at a cell c is written $a[c]$. Note that we adopt a *named perspective* [AHV95] for tuples rather than relying on tuple ordering.

We will not develop a formal type system for extensible records in this work; c.f. Jones et al. for one proposal implemented for Haskell [JJ99]. Informally, we will say that $g_i[c]$ for $c \in G_i$ has type $\mathsf{sch}_i(G)$, written $g_i[c] : \mathsf{sch}_i(G)$. We will say that a tuple t of type $\mathsf{sch}_i(G)$ *conforms* to another tuple type s if $\mathsf{sch}_i(G) \subset s$, ignoring attributes with type $\langle \perp \rangle$. The set of all tuples whose type conforms to a scheme s is written $[s]$. The set of all tuples is denoted $\mathcal{T}$.

Some operators will accept functions over tuples as arguments: A *predicate* on tuples is a function $[s] \rightarrow \mathsf{Bool}$, and a unary tuple function is a function $[s] \rightarrow \upsilon$, where s is the scheme of some gridfield at some rank, and υ is some value type.

An attribute function need not be total; the distinguished null value $\perp$ is assumed to be returned when such a function is applied to cells outside its domain. Further, if no data are bound to the cells of a particular dimension k, $g_k[c]$ is assumed to return an empty tuple for all cells, denoted $()$.

Consider a trussed bridge modeled as a 1-dimensional (non-embeddable) grid. A gridfield defined over such a grid might return the net force at each node and the linear force along each truss. A gridfield can capture both cases by binding data to the 0-cells and 1-cells, respectively. Images can be viewed naturally as data bound to the 2-cells of a rectilinear product grid. We can also model unstructured sets of values as a gridfield over a grid consisting solely of 0-cells.

To support multiple geometric embeddings of a grid, geometric information is modeled as ordinary data values bound to the cells of a grid. A simple example is a 2-D grid with a gridfield binding (x, y)-pairs to the nodes, which embeds the grid in 2-D Euclidean space, assuming edges are line segments and faces are polygons. Additional coordinate systems can be captured through additional attributes. Many models distinguish geometric attributes from other data [Ber00, HLC91, SML96], 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 2.2 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. Systems commonly use geometry as the identifying feature of a grid, thereby obscuring the topological equivalence among the three grids in Figure 2.5.

2.3 GRID COMPUTATION REQUIREMENTS

The gridfield model as described only defines the information that a grid must provide. We have not yet discussed how this information is accessed by computations involving gridfields.

We now analyze several examples of computations over grids and gridded datasets to see what kind of requirements they impose on the gridfield model. We provide pseudo-code algorithms for each; the operations shared among these algorithms are easier to identify syntactically than semantically. The first two examples are adapted from Berti's dissertation [Ber00], where they are used for similar expository purposes.

2.3.1 Grid Serialization

To store a mesh (grid and geometry) to disk or transmit it over the network, the components must be serialized. There are many potential serializations; the following example is derived from a custom format used by the CORIE EOFS. Comments begin with a double slash and are italicized.

```
$n$            // number of nodes
$m$            // number of cells
$x_0, y_0$     // x,y coordinates for each node
$x_1, y_1$
⋮
$x_n, y_n$
// each cell is an integer $s_i$ representing the number of nodes
// followed by a sequence of $s_i$ node references.
// e.g., a triangle might be represented as 3: 0, 1, 2
$s_0$: $v_{00}, v_{01}, \ldots, v_{0s_0}$
$s_1$: $v_{10}, v_{11}, \ldots, v_{1s_1}$
⋮
$s_m$: $v_{m0}, v_{m1}, \ldots, v_{ms_m}$
```

Algorithm 1 Grid Serialization

```
Input: A 2-D grid $G$.
Input: A writable I/O stream.
Output: The grid $G$ is serialized to the I/O stream.
  write $|G_0|$ //the number of nodes
  write $|G_2|$ //the number of cells
  for each $v \in G_0$ do
    write $x[v]$, $y[v]$ //coordinates of each node
  end for
  for each c $\in G_2$ do
    $n \leftarrow$ number of nodes in $c$
    write $n$ //the number of nodes incident to this cell
    $i \leftarrow 0$
    for each $v \in G_0$ such that $v \preceq c$ do
      write $i$ //a reference to a node
      $i \leftarrow i + 1$
    end for
  end for
```

Algorithm 1 generates an appropriate serialization, and suggests the following requirements for a grid model:

1. Access to the global number of nodes.
2. Access to the global number of cells (taken to be polygons in this case).
3. Iteration over the nodes.
4. Access to data values associated with the nodes.
5. Iteration over all cells in the grid of a given dimension.
6. Iteration over nodes for a given cell.
7. Ability to impose a total order on the nodes (each node is assigned an integer i).

2.3.2 Direct Rendering

Data products are frequently visualizations; we need an algorithm to render a grid to the screen. We assume access to an appropriate function for drawing primitive graphics objects such as points, lines, and polygons.

Algorithm 2, renders the outside surface of a 3-D grid. A 2-cell is determined to be on the outside surface if it is incident to just one 3-cell.

Algorithm 2 Direct Rendering

```
Input: A grid $G$ equipped with $x, y, z$ coordinates at the nodes and a scalar value
  at each face.
Input: A function DrawPolygon accepting a floating point between 0 and 1
  representing a color and a sequence of $(x, y)$ pairs.
Input: The range of values $(l, h)$ to be assigned to colors; values lower than $l$ or
  higher than $h$ will be clamped to $l$ and $h$, respectively.
Output: The grid $G$ is drawn to the screen.
  for each $c \in G_2$ do
    if $|\{d \mid d \in G_3, c \prec d\}| = 1$ then
      $colorval \leftarrow (c.\text{scalar} - l)/(h - l)$
      $P \leftarrow$ empty sequence of coordinate pairs
      for each $v \in G_0$ such that $v \preceq c$ do
        append $(v.\text{x}, v.\text{y})$ to $P$
      end for
      DrawPolygon($colorval$, $P$)
    end if
  end for
```

The requirements for Algorithm 2 are similar to the serialization algorithm, except the grid is now 3-D, we must perform some additional manipulation of data values.

1. Iteration over all cells in the grid at a particular dimension.
2. Iteration over the faces of a cell.
3. Iteration over the nodes of a face.
4. Access to geometry data associated with each node.
5. Access to a data value associated with each face.
6. Ability to perform arithmetic on data values.

2.3.3 Smoothing

Given a noisy scalar field associated with the nodes of a grid, we can smooth the field by averaging over a local neighborhood. In this case, two nodes are in the same neighborhood if they are both incident to a common 1-cell.

Algorithm 3 Scalar Smoothing

```
Input: A grid $G$ equipped with a scalar value $h$ at the nodes.
Input: A weighting factor $w_o$ for the original value.
Input: A weighting factor $w_n$ for the neighbors.
Output: The grid $G$ is equipped with a smoothed scalar value $avgh$ at the nodes.
  for each $v \in G_0$ do
    sumh $\leftarrow 0$
    counth $\leftarrow 1$
    for each $e \in G_1$ such that $e \succeq v$ do
      for each $n \in G_0$ such that $n \preceq e$ do
        if $v \neq n$ then
          sumh $\leftarrow$ sumh $+ h[n]$
          counth $\leftarrow$ counth $+ 1$
        end if
      end for
    end for
    $v$.avgh $\leftarrow w_n \cdot$ sumh/counth $+ w_o \dot{v}$.h
  end for
```

An illustration of the smoothing algorithm appears in Figure 2.10. The node labeled "c" will be assigned the average of the values associated with the hollow nodes. These nodes are accessed by traversing the shaded edges.

The requirements for a smoothing operation involve iterating over edges associated with a node as well as nodes associated with an edge. In the previous algorithms, we iterated over lower dimensional cells associated with a higher dimensional cell only. Specifically, the requirements are as follows:

1. Iteration over the nodes in the grid.
2. Iteration over the edges to which a node is incident.
3. Iteration over the nodes incident to an edge.
4. Access to scalar data associated with each node.
5. Ability to perform arithmetic on data values.
6. Ability to update the value at each node.

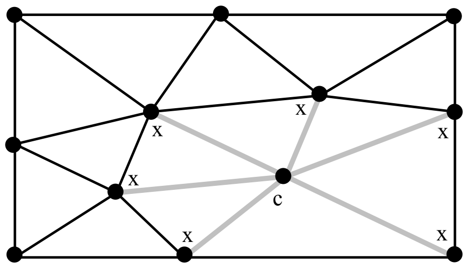


Figure 2.10: The node labeled “c” will be assigned the average of the values associated with the nodes labeled “x”. The shaded nodes are accessed by traversing the shaded edges.

2.3.4 Streamlines

Streamlines are a visualization technique for a vector field. In Figure 2.11, streamlines are used to indicate the path of blood flow at the surface of an aneurysm [LH05]. Given a grid equipped with a vector at each node, a seed point, and a time interval, a streamlines algorithm integrates the vector field and produces a sequence of points representing the path of the flow starting from the seed location. We use a simple Euler method of integration in this example, meaning the next location in the sequence is computed as the current velocity times the time interval. That is, the position vector x at time $t_0 + \Delta t$ can be expressed as

$$x(t_0 + \Delta t) = x(t_0) + v \cdot \Delta t \cdot x(t_0)$$

Alternative methods (e.g., Runge-Kutta) improve accuracy by incorporating information from a local neighborhood of velocity values.

Algorithm 4 is used for visualizing steady flows; the velocity is dependent on the position only.

Algorithm 4 searches locally for the next cell by raycasting in the direction of the vector. There are two cases to consider: 1) The ray crosses an edge of the polygon, and we should check the appropriate adjacent cell, and 2) the ray passes

Algorithm 4 Streamlines

```
Input: A gridfield $G$ equipped with vector values $(u, v)$ on the cells and $(x, y)$-
    coordinates on the nodes.
Input: A seed point $p$ equipped with $(x_0, y_0)$.
Input: A time step $\Delta t$.
Input: A maximum time $t_{\max}$.
Output: A sequence of $(x_i, y_i)$ coordinates.
    $P \leftarrow$ empty sequence of points
    next $\leftarrow c \in G_2$ such that polygon($c$) contains point($x_0, y_0$)
    $x, y \leftarrow x_0, y_0$
    $t \leftarrow 0$
 5: while $t < t_{max}$ do
      $t \leftarrow \Delta t + t$
      add $x, y$ to $P$
      $u, v \leftarrow$ next.u, next.v
      $x, y \leftarrow x + u\Delta t,\ y + v\Delta t$
      //search locally for the next cell
      //ray exits through an edge?
10:   while polygon(next) does not contain point($x, y$) do
        next $\leftarrow$ NULL
        for each $e \in G_1$ such that $e \preceq c$ do
          $p_1, p_2 \leftarrow$ [point($n$) | $n \in G_0 \wedge n \preceq e$]
          if ray($x, y, u, v$) crosses line($p_1, p_2$) then
15:         next $\leftarrow$ [$c$ | $e \preceq c \wedge c \neq$ next]
          end if
        end for
        //ray exits through a node?
        for each $n \in G_0$ such that $n \preceq c$ do
          if ray($x, y, u, v$) passes through point($n$) then
20:         next $\leftarrow$ [$d$ | $d \in G_2 \wedge n \preceq d \wedge c \neq d \wedge$ ray($x, y, u, v$) crosses polygon($d$)]
          end if
        end for
        //if no next cell was found
        if next is NULL then
          return $P$
25:     end if
      end while
    end while
```

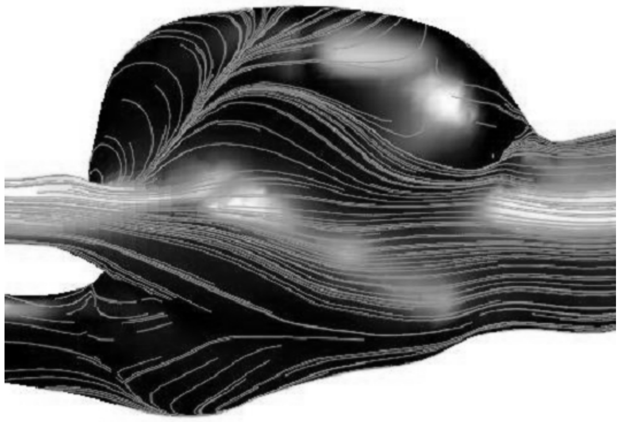


Figure 2.11: Streamlines indicating the direction of blood flow near the surface of an aneurysm. [LH05]

through a node v of the cell, in which case we need to check all cells that share the node v. In either case, we may need to continue searching from the new cell *next*.

The polygon, line, and point functions access the geometry information of a cell's nodes as in the previous algorithm. These functions create a geometric region from a topological cell. The region includes the boundary of the cell: The function polygon(c) contains the region represented by the 1-cells incident to c. Variations on this algorithm include different search methods and different integration methods. The requirements suggested by this streamlines algorithm include the following:

1. Iteration over all cells in the grid at a particular dimension.
2. Access to data associated with the nodes.
3. Access to data associated with each cell.
4. Access to a cell that shares a node with a given cell.
5. Access to a cell that shares an edge with a given cell.
6. Conversion of a cell to a geometric object.
7. Repetition of blocks of code; non-topological iteration.

2.3.5 Feature Extraction

We have argued that simulation is necessarily a coarse, bottom-up approach to scientific discovery. Scientists model systems holistically first and study specific phenomena second. A frequent operation, therefore, is to identify portions of a grid that capture a specific feature and cull the remainder. A feature can often be described as a spatially connected region for which some invariant holds throughout. For example, a vortex feature in a fluid can be defined as a region of low pressure, a region of high average vorticity, or a region of high helicity; other definitions exist [BS94a].

Each of these variants involve traversal of neighborhoods, an activity for which we have already provided examples. We therefore use the simplest method and emphasize the thresholding activity.

Algorithm 5 produces a set of gridfields, each representing a connected region of low pressure. As with the streamlines algorithm, there is an iterative structure to the algorithm in addition to the navigation of topology. We also see the runtime construction of gridfields. The requirements we may extract are:

1. Search for cells with a minimum value.
2. Iteration over nodes that satisfy a particular criteria.
3. Access to data associated with each cell.
4. Access to nodes adjacent to a node.
5. Insertion of cells and nodes into a gridfield.
6. Repetition of blocks of code.

2.3.6 From Algorithms to Recipes

In this section, we introduce two properties of algorithms useful for reasoning about how to connect algorithms together: the *stencil* for describing a neighborhood of

Algorithm 5 Feature Extraction

```
Input: A grid $G$ equipped with pressure values $p$ at the nodes.
Input: A threshold pressure $\rho$.
Output: A sequence of grids $S$, each representing a vortex.
  $V \leftarrow$ empty grid
  for each $v \in G_0$ do
    if $v$.p $< \rho$ then
      add $v$ to $V$
    end if
  end for
  $S \leftarrow$ empty sequence
  $i \leftarrow 0$
  while there are unused nodes in $V$ do
    seed $\leftarrow$ unused node with lowest pressure
    mark seed as used
    $S[i] \leftarrow$ empty grid
    add seed to $S[i]$
    for each unused node $v \in V$ adjacent to a node in $S[i]$ do
      mark $v$ as used
      add $v$ to $S[i]$
      for each $c \in G_2$ such that $v \preceq c$ do
        if the set of nodes incident to $c$ are in $S[i]$ then
          add $c$ to $S[i]$
        end if
      end for
    end for
    $i \leftarrow i + 1$
  end while
  return $S$
```

cells, and the *scheme* for describing input and output requirements of an algorithm.

Each algorithm we described represents an individual task. These tasks may be composed into a *recipe*, a sequence of coupled tasks designed to produce a single result. Composition of simpler tasks into a recipe generates new requirements related to the communication between sub-tasks. We distilled the following requirements from the algorithms in this section:

1. Iteration over cells of a particular dimension.
2. Iteration over incident cells of dimension i given a cell of dimension j.
3. Read and write access to data values associated with cells of arbitrary dimension.
4. Identification of cells that satisfy a particular condition on their data.
5. Ability to construct a new grid using cells from a given grid.
6. Repetition of operations; recursion.

What additional requirements do recipes generate? We must ensure that each sub-task has access to the data it needs. Two concepts are useful in this regard. First, we observe that many algorithms involve one or more *stencils* describing the manner in which incidence information is accessed. A stencil is a sequence of integers $s_0 s_1 \ldots s_n$ representing a pattern of iteration over a neighborhood of cells. For example, a stencil appropriate for Algorithm 3 is 010, which describes any algorithm of the following form:

```
for each cell c of dimension 0:
  for each cell f of dimension 1 to which c is incident:
    for each cell v of dimension 0 incident to f:
      do something
```

Let halo be a function for retrieving nearby cells using a stencil. The halo of a cell c with respect to a stencil σs in the context of a grid G is defined as follows:

$$\mathsf{halo}_G(c,s) = \begin{cases} \{c\}, & s = \mathsf{dim}(c) \\ \text{undefined}, & \text{otherwise} \end{cases}$$

$$\mathsf{halo}_G(c,\sigma s) = \{e \mid e \gtreqless d \text{ and } \mathsf{dim}(e) = s \text{ and } d \in \mathsf{halo}_G(c,\sigma) \text{ and } c \neq e\}$$

where s is an integer dimension and σ is a prefix of the stencil σs. For example, if σ is 01 and s is 0, σs is the stencil 010. This notation is meant to resemble language-theoretic string notation. The subscript denoting the grid my be dropped if it is apparent from the context.

In the previous pseudo-code, the halo would be only the set of cells assigned to the variable `v`, ignoring the cells assigned to the variables `c` and `f`. In Figure 2.10, $\mathsf{halo}(c, 010)$ is the set of nodes indicated by large circles marked with "x", and $\mathsf{halo}(c, 01)$ is the set of shaded edges.

Given a root cell c, a stencil σ can also be used to construct a grid whose boundary is $\mathsf{halo}(c,\sigma)$.

Definition 7. Let $G = (X, \preceq_G)$ be a grid, c be a k-cell in G, and σ be a stencil $\sigma = s_0 s_1 \ldots s_n$ such that $k = s_0$. The *stencil grid* G_c^σ is a subgrid of G defined as $F = (Y, \preceq_F)$, where $x \preceq_F y$ if $x, y \in F$ and $x \preceq_G y$, and $F = \bigcup_{i \leq n} \mathsf{halo}_G(c, s_0 s_1 \ldots s_i)$.

For example, for the grid G in Figure 2.6, G_x^{10} consists of the edge x and its incident nodes, 2 and 3. The grid G_4^{02} consists of the node 4 and the 2-cell A.

For a well-supported grid, the *closure grid* of c in G, $\overline{c}_G$, is equivalent to the stencil grid of c for the stencil $(\mathsf{dim}(c), \mathsf{dim}(c) - 1, \ldots, 0)$.

We have pointed out that each algorithm involves one or more stencils. We also observe that each algorithm accesses a certain set of attributes at each dimension, placing minimal requirements on the scheme of the input. For example, let $\mathbf{G}$ be the input to Algorithm 2. The algorithm requires that $\mathsf{sch}_0(\mathbf{G}) \supseteq \{x, y\}$ and $\mathsf{sch}_1(\mathbf{G}) \supseteq \{\text{scalar}\}$.

The scheme concept helps characterize the dependencies between sub-tasks in a recipe. Applied to an input $\mathbf{G}$, Algorithm 3 requires that $\mathsf{sch}_0(\mathbf{G}) \supseteq \{\text{x,y,h}\}$ and produces a gridfield $\mathbf{F}$ such that $\mathsf{sch}_0(\mathbf{F}) \supseteq \{\text{h}\}$. Dependencies between operations can be used to 1) statically check for errors, and 2) expose efficient evaluation plans. An operation whose inputs do not have the required attributes raises an exception. An operation p that accesses an attribute supplied by a another operation q must be performed after q. We can evaluate independent operations in any order we like.

These concepts can also be used to reason about algebraic optimization. If we wish to both smooth a gridfield and extract a feature, can we perform these operations in either order? To perform the smoothing operation, we must have access to the cells of the smoothing operation's stencil 010 and to the attribute being smoothed. To perform the feature extraction, we must have access to the attribute over which the feature predicate is defined, and we must have access to the cells defined by the feature extraction stencil, 020. We cannot, therefore, extract the feature first, since cells needed by the smoothing operation may be removed. The nodes on the boundary of the feature will not be smoothed properly since only a portion of their stencil grid is available. We will show how this situation can be rectified to allow such operations to commute in Section 2.4.6.

Some algorithms may impose other conditions on their input besides a scheme. For example, line 13 in Algorithm 4 asserts that each edge will have exactly two nodes incident to it. This check and others that cannot be cast as a condition on the scheme or stencil must be checked explicitly at runtime.

Expression and analysis of gridfield computations in terms of operators, stencils, and schema make it easier to reason about efficiency and correctness of the computation, as well as properties of the result. Informed by the distilled requirements and the reasoning tools described in this section, we are now ready to derive a set of general operators to replace task-specific algorithms.

2.4 GRIDFIELD ALGEBRA

The operators for manipulating gridfields must correctly handle both the underlying 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 that the output grid retains certain properties. Other operators are novel, such as regrid. The regrid operator maps data from one grid onto the cells of another and then aggregates to produce a single value per cell.

2.4.1 Bind

The bind operator changes the attributes of a gridfield at some rank k. In many cases, bind acts simply as a constructor, providing a gridfield with its first attribute. Attributes are modeled as functions, and functions may be applied to a subset of their domain. Therefore, we can use bind to associate an attribute (a, τ, f) to not only a grid G, but any subgrid of G.

Definition 8. Given a gridfield $\mathbf{G} = (G, g_0, \ldots, g_d)$, an integer k where $0 \leq k \leq d$, and an attribute (a, τ, f), the expression $\mathsf{bind}(\mathbf{G}, k, a)$ produces a gridfield $\mathbf{G} = (G, g_0, \ldots, g_k \cup \{a\}, \ldots, g_d)$ where the domain of f is G_k.

For the CORIE application, the simulation produces arrays for each of several variables: salinity, temperature, and velocity, among others. The bind operator allows us to read each array separately, avoiding contact with data that is not necessary for the task at hand. Others have argued the benefits of this *column-oriented* approach when working with read-mostly data [SAB+05].

2.4.2 Restrict

The restrict operator behaves like a relational select where each result satisfies some predicate. However, when data values are culled, associated cells and possibly some neighboring cells are also removed from the grid.

The predicate is defined over cells of a particular dimension, i. For each cell that fails the predicate, which nearby cells should also be removed? There are several ways we might identify these dropped cells. The *Simple* semantics excludes exactly those cells that fail to satisfy the predicate, a scheme which removes as few cells as possible. The *Drop* semantics operates at the other extreme: only cells that satisfy the predicate are retained; all others are removed. The *Extensive* semantics retains any cell sharing an incidence relationship with at least one cell that satisfies the predicate. The *Intensive* semantics, conversely, *removes* any cell that shares an incidence relationship with a cell that *fails* the predicate. The *Exact* semantics strikes a balance between the extensive and intensive semantics by considering dimension: lower-dimensional cells follow the Intensive semantics, and higher-dimensional cells follow the Extensive semantics. This choice has the following property: if a cell c is retained in the result, then so is its closure grid, preserving well-supportedness.

Let $F = (X, \preceq_F)$ be a grid and let $\mathbf{F} = (F, f_0, \ldots, f_d)$ be a gridfield of dimension d. Also, let p be a predicate $[\mathsf{sch}_i(\mathbf{F})] \rightarrow \mathsf{Bool}$ for some i, $0 \leq i \leq d$. The various semantics for the grid G in the definition $(G, f_0, \ldots, f_d) = \mathsf{restrict}(p, i, \mathbf{F})$ are defined formally below:

Simple

$$G_i = \{c \mid c \in F_i \text{ and } p(f_i[c]))\}$$
$$G_{k \neq i} = \{d \mid d \in F_k\}$$

Drop

$$G_i = \{c \mid c \in F_i \text{ and } p(f_i[c]))\}$$
$$G_{k \neq i} = \{\}$$

Extensive

$$G_i = \{c \mid c \in F_i \text{ and } p(f_i[c]))\}$$
$$G_{k \neq i} = \{d \mid d \in F_k \text{ and } \exists c.(p(f_i[c]) \text{ and } c \gtrless_F d)\}$$

Intensive

$$G_i = \{c \mid c \in F_i \text{ and } p(f_i[c]))\}$$
$$G_{k \neq i} = \{d \mid d \in F_k \text{ and } \forall c.((c \in F_i \text{ and } c \gtrless_F d) \Rightarrow p(f_i[c]))\}$$

Exact

$$G_i = \{c \mid c \in F_i \text{ and } p(f_i[c]))\}$$
$$G_{k > i} = \{d \mid d \in F_k \text{ and } \forall c.((c \in F_i \text{ and } c \gtrless_F d) \Rightarrow p(f_i[c]))\}$$
$$G_{k < i} = \{d \mid d \in F_k \text{ and } \exists c.(p(f_i[c]) \text{ and } c \gtrless d)\}$$

In each case, the incidence relation of G is defined as $\preceq_G$, where $x \preceq_G y$ if and only if $x \in X$, $y \in X$ and $x \preceq_F y$. This fact makes G a subgrid of F. Figure 2.12 illustrates the intuition behind each of these possible semantics.

Which of these semantics is an appropriate choice for our operator? The Exact semantics tend to properly capture intuition and algorithmic requirements in most cases. The primary requirement for all these possible semantics is that every i-cell in the result must satisfy the predicate. Beyond this requirement, we observe that a common stencil for operations has the form $d, d-1, \ldots, 0$, where d is the dimension of the grid. The Exact semantics preserves this stencil. Further, the Exact semantics preserves the well-supportedness property, since all cells incident to a retained cell are themselves retained. Extensive semantics lead to non-well-supported grids when $k < \mathsf{dim}(G)$, and Intensive semantics lead to non-well-supported grids when $k > 0$. A different semantics can be emulated using the accrete operator, described after restrict.

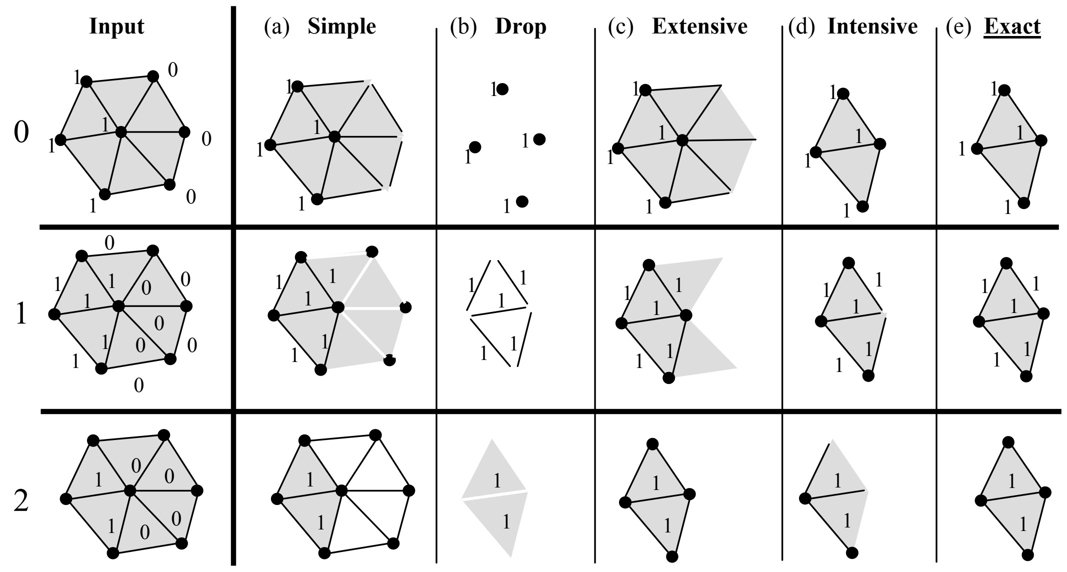


Figure 2.12: Possible semantics for restrict: (a) Simple, (b) Drop, (c) Extensive, (d) Intensive, (e) Exact. We adopt (e) Exact to preserve well-supportedness. The integers at the left indicate which dimension we are operating on; restrict is parameterized by dimension.

Definition 9. Let $F = (X, \preceq_F)$ be a grid and let $\mathbf{F} = (F, f_0, \ldots, f_d)$ be a gridfield. Also, let p be a predicate $[\mathsf{sch}_i(\mathbf{F})] \rightarrow \mathsf{Bool}$ and i be an integer such that $0 \leq i \leq d$. Then $\mathsf{restrict}(p, i, \mathbf{F})$ is a gridfield $\mathbf{G} = (G, f_0, \ldots, f_d)$ such that G is defined using the Exact semantics.

2.4.3 Accrete

The accrete operator grows a gridfield $\mathbf{E}$ by adding cells selected from a base gridfield $\mathbf{B}$. The cells selected from $\mathbf{B}$ are connected topologically to some cell that appears in both $\mathbf{E}$ and $\mathbf{B}$. The topological relationship between the cells in $\mathbf{E}$ and the cells selected from $\mathbf{B}$ are defined by a stencil. In Figure 2.13, the stencil is 120, indicating that for each 1-cell e in E, we are to find the set of 2-cells in B to which e is incident (c and d in this case) and add them to the result. Further,

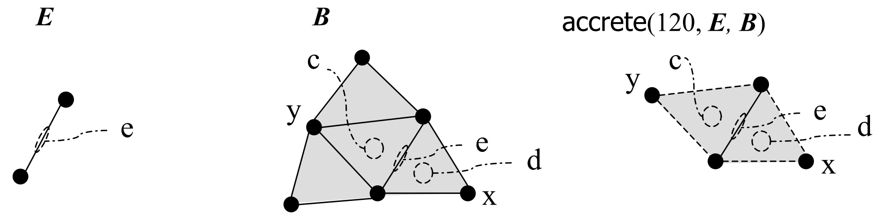


Figure 2.13: An example of the accrete operator, whose result consists of cells from E plus a neighborhood of cells from B. The neighborhood is defined by a stencil. Since the stencil is 120, the result includes the 1-cell e in E, plus any 2-cells in B to which it is incident, plus any 0-cells incident to a 2-cell already included.

---

we are next to find 0-cells in B incident to c or d and add them to the result (in this case x and y). The gridfield $\mathbf{E}$ need not be distinct from $\mathbf{B}$. When they are the same, accrete acts as a kind of selection condition on the topology rather than the data values.

The data associated with the new grid is taken from the $\mathbf{E}$ at the "root" dimension and taken from $\mathbf{B}$ at all other dimensions. For example, any data associated with the edge e in the gridfield $\mathbf{B}$ is ignored in Figure 2.13.

This operator allows us to restore a neighborhood of cells that was potentially removed by a restrict operator or a merge operator (described after accrete). The accrete operator also allows us to incrementally grow a grid, which is a requirement of Algorithm 4 and Algorithm 5.

Definition 10. Let $\mathbf{E} = (E, e_0, \ldots, e_c)$ and $\mathbf{B} = (B, b_0, \ldots, b_d)$ be gridfields. Let $\sigma = k, x_1, \ldots, x_n$ be a stencil. Then accrete$(\sigma, \mathbf{E}, \mathbf{B})$ produces a gridfield $\mathbf{G} = (G, g_0, \ldots, g_n)$ composed of stencil grids, where $G = \bigcup_{c \in E_k} B_c^\sigma$ and $g_k = e_k$ and $g_i = b_i$ for $i \neq k$.

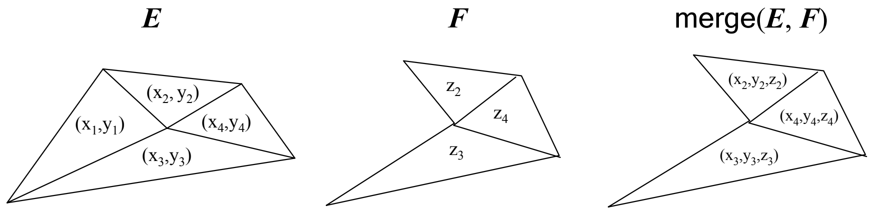


Figure 2.14: An illustration of the semantics of merge: an intersection of grids and a union of attributes.

2.4.4 Merge

The merge operator computes the intersection of two grids and retains data values defined over this intersection. Merge is not associative, since there may be name conflicts in the attributes of the two arguments. In the case of conflicts, the values from the first argument are used. A more general operator might accept a function for handling the conflicts (perhaps averaging the two values, for example), but we have found the first definition useful in practice. The intuition behind the utility of this non-associative semantics is that the merge operator restricts the attribute functions of the first argument to the domain defined by the second.

Definition 11. Let $\mathbf{E} = (E, e_0, \ldots, e_c)$ and $\mathbf{F} = (F, f_0, \ldots, f_d)$ be gridfields. Let $n = \mathsf{min}(c, d)$. Then $\mathsf{merge}(\mathbf{E}, \mathbf{F})$ produces a gridfield $\mathbf{G} = (E \cap F, h_0, \ldots, h_n)$ where $h_i = e_i \cup (f_i - e_i)$ (Recall that e_i and f_i are sets of attributes at rank i.)

The definition involves a slight abuse of notation. When defining h_i to be $e_i \cup f_i - e_i$, we are avoiding name conflicts in the result by favoring attributes from the left argument over attributes from the right argument. Each member of the set e_i is actually a triple (a, τ, f), but we assume set membership is defined by the pair (a, τ).

2.4.5 Cross Product

The cross-product operator for gridfields builds on the cross-product operator on grids.

Definition 12. Let $\mathbf{E} = (E, e_0, \ldots, e_n)$ and $\mathbf{F} = (F, f_0, \ldots, f_m)$ be gridfields. The *cross product* of $\mathbf{E}$ and $\mathbf{F}$, written $\mathbf{E} \otimes \mathbf{F}$, is a gridfield $\mathbf{G} = (E \otimes F, g_0, \ldots, g_{n+m})$ where $g_k[cd] = \bigcup_{i=0}^{k}\{e_i[c] \cup f_{k-i}[d]\}$ for each product cell cd in G_k. Naming conflicts between the attributes in e_i and f_{k-i} cause an exception.

This definition can result in a gridfield with values undefined at some cells at ranks $0 < k < d$, since there are multiple ways to form cells at that rank and each way of forming cells gives rise to different attributes. Consider gridfields $\mathbf{E}$ and $\mathbf{F}$ with the following schemes at each rank, ignoring types for the moment.

$$\begin{array}{ll} \mathsf{sch}_0(E) = (a) & \mathsf{sch}_0(F) = (w, x) \\ \mathsf{sch}_1(E) = (b, c) & \mathsf{sch}_1(F) = (z) \end{array}$$

The cross product gridfield $\mathbf{G} = \mathbf{E} \otimes \mathbf{F}$ will have the following schemes at each rank.

$$\mathsf{sch}_0(G) = (a, w, x)$$
$$\mathsf{sch}_1(G) = (a, z, b, c, w, x)$$
$$\mathsf{sch}_2(G) = (b, c, z)$$

The 1-cells in the result are generated by both $E_0 \times F_1$ and $F_0 \times E_1$. Each of these sets of cells alone would have data of different schemes; namely, (a, z) and (b, c, w, x) respectively. However, since all cells at a particular rank must have the same type in our model, we adopt a kind of "union" type: (a, z, b, c, w, x). Since some cells will not have a value for a or z, we declare these values to be $\perp$. Therefore, all tuples at rank 1 in the result gridfield $\mathbf{G}$ will be of the form

$(a, z, \perp, \perp, \perp, \perp)$ or of the form $(\perp, \perp, b, c, w, x)$. Functions that operate on tuples must handle the value $\perp$.

In the relational algebra, an analogous issue is the requirement that relations be "union-compatible" before their union can be computed. The analogous solution is to coerce them to the same type, performing an "outer union," by padding the tuples with NULL to enforce union compatibility.

2.4.6 Regrid

The regrid operator maps a *source* gridfield's cells onto a *target* gridfield's cells, then aggregates the data values bound to the mapped cells and binds the result to the target gridfield. The behavior of regrid is controlled by two functions, an *assignment* function (for mapping cells) and an *aggregation* function (for aggregating data values). The assignment function associates each cell c in the target grid with a set of cells in the source grid, which we refer to as the *preimage* of c. To perform the assignment, this function might use topological information only (e.g., a "neighbors" function that identifies adjacent cells), or it may use the attributes of the two gridfields (e.g., an "overlaps" function that uses geometry data). The aggregation function is applied separately to each preimage.

To illustrate a simple use of regrid, 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 2.15(a). One use of the regrid 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 three nodes, the chunk, in the source grid S (Figure 2.15(b)). The aggregation function can then, say, average the value at the three nodes to obtain a single value (Figure 2.15(c)).

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.

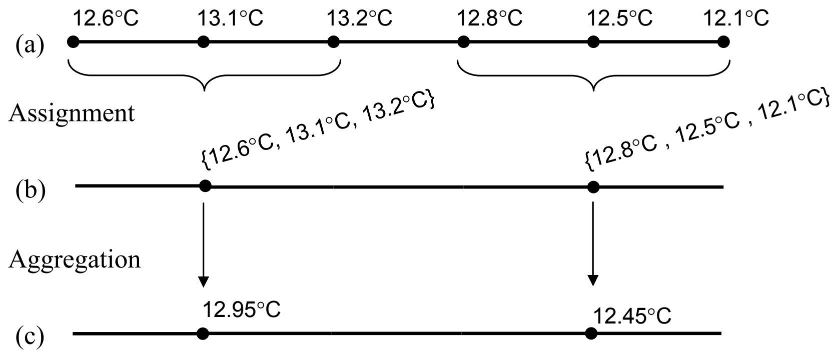


Figure 2.15: An example of the regrid operator. (a) A 1-D gridfield returning temperatures. (b) Assignment to the target grid. (c) Aggregation using arithmetic mean.

For target node i, the window function assigns source nodes $[i-k, i-k+1, \ldots, i, i+1, \ldots, i+k]$, assuming nodes are numbered linearly. 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.

Definition 13. Let $\mathbf{E} = (E, e_0, \ldots, e_c)$ be a target gridfield and $\mathbf{F} = (F, f_0, \ldots, f_d)$ be a source gridfield. Let i and j be integer dimensions such that $0 \leq i \leq c$ and $0 \leq j \leq d$. Let m be an assignment function $m : \mathcal{GF} \times \mathcal{C} \rightarrow \mathscr{P}(\mathcal{C})$, where $\mathscr{P}(\cdot)$ is the powerset function, such that when applied to the source gridfield $\mathbf{F}$ and a target cell in E_i, m returns a subset of F_j. Let a be an aggregation function $a : [\mathsf{sch}_i(\mathbf{E})] \times \mathscr{P}([\mathsf{sch}_j(\mathbf{F})]) \rightarrow \gamma$ returning a tuple of type γ. Then $\mathsf{regrid}(\mathbf{E}, i, \mathbf{F}, j, m, a)$ produces a gridfield $\mathbf{G} = (E, g_0, \ldots, g_c)$ where $g_i[x] = a(e_i[x], \{f_j[y] \mid y \in m(\mathbf{F}, x)\})$ and for $k \neq i$, $g_k = e_k$.

A regrid operator computes a new gridfield with the same grid as the target, but with new values at a particular rank i. The new values are computed using 1) the existing tuple in the target gridfield, and 2) values in the source gridfield at a particular rank j. We include the target tuple $e_i[x]$ in the definition in order to allow complex aggregation functions such as weighted averages.

To perform aggregations at multiple ranks, multiple regrid operators must be used. Although we operate on only one dimension at a time, there are no special restrictions on how the assignment function m can be defined. Specifically, m may reference data bound to other dimensions. For example, imagine we have two 2-dimensional gridfields $\mathbf{E}$ and $\mathbf{F}$ both with x, y coordinates bound to their nodes. Further, the source grid F of $\mathbf{F}$ has a temperature dataset bound to its 2-cells. A common aggregation is to "regrid" the temperature dataset by averaging the values of 2-cells that overlap a given target 2-cell [HMG94]. In this case, $i = j = 2$. However, the geometry data bound to the nodes of the grids must also be accessed to evaluate the "overlaps" predicate. Specifically, the assignment function returns the set of polygons in F that overlap a given polygon in E. Intuitively, the regrid operator mimics the functionality (in the relational algebra) of a (possibly spatial) join followed by a group-by.

Specializing and Optimizing Regrid

The strengths of the regrid operator are expressiveness and extensibility, but these characteristics can also make it difficult to optimize. To process regrid operations efficiently, we must identify and exploit special cases.

The design of the regrid operator, and indeed the algebra itself, reflects the fact that the data has a great deal more structure than a relation. A grid gives a dataset a kind of "shape" that exposes its cardinality and local organization. The operators are designed to modify this shape in a predictable manner. For example, we could have produced an inner join operator that assigned cells of a grid S to cells of another grid T using an arbitrary predicate. The result would be a set of pairs of cells rather than a grid. Instead, we exchange the predicate for a function from T to S, guaranteeing at the logical level that the result data is bound to the known grid T.

This design also allows a critical optimization opportunity: The commutativity of restrict and regrid. A restriction whose predicate does not depend on the newly aggregated values may be pushed through the regrid. That is,

$$\textsc{RestrictRegrid}\ \frac{\begin{array}{c} a \text{ has type } [\mathsf{sch}_i(\mathbf{T})] \times \mathscr{P}([\mathsf{sch}_j(\mathbf{S})]) \to \gamma \\ p \text{ has type } [\mathsf{sch}_i(\mathbf{T}) - \gamma] \to \mathit{Bool} \\ \mathsf{restrict}(p, i, \mathsf{regrid}(\mathbf{T}, i, \mathbf{S}, j, m, a)) \end{array}}{\mathsf{regrid}(\mathsf{restrict}(p, i, \mathbf{T}), i, \mathbf{S}, j, m, a)}$$

This transformation restricts the domain of the assignment function, but does not affect its answers. The analogous transformation in the relational algebra is to push a selection through a group-by if the predicate involves only the grouping attributes.

The assignment function dictates the behavior of the regrid operator, and therefore guides the processing strategy. We have identified several assignment functions that afford specialized, efficient implementations.

The most common use of the regrid operator is to simply apply an arithmetic expression to each of the tuples, generating one or more new attributes. The assignment function in this case is simply the identity function, and the aggregation function is the user-defined arithmetic expression suitably extended to singleton preimages. The apply operator supports this usage efficiently.

Definition 14. Let $\mathbf{G} = (G, g_0, g_1, \ldots, g_n)$ be a gridfield and let f be a function $[\mathsf{sch}_i(\mathbf{G})] \to \gamma$. Then $\mathsf{apply}(\mathbf{G}, i, f) = \mathsf{regrid}(\mathbf{G}, i, \mathbf{G}, i, \mathsf{id}, h)$, where id is the identity function and $h(g_i[c], \{c\}) = f(g_i[c])$.

The project operator is another specialization that uses the identity function for assignments. The project operator removes unwanted attributes from the source gridfield.

Definition 15. Let $\mathbf{G} = (G, g_0, g_1, \ldots, g_n)$ be a gridfield and let A be a set of (name, type)-pairs $\{(a_0, \tau_0), (a_1, \tau_1) \ldots, (a_m, \tau_m)\}$. Then $\mathsf{project}(\mathbf{G}, i, A)$ is equivalent to $\mathsf{regrid}(\mathbf{G}, i, \mathbf{G}, i, \mathsf{id}, f)$, where id is the identity function and $f(g_i[c], *) = \{(a, v) \mid (a, \tau) \in A \text{ and } (a, v) \in g_i[c]\}$.

The unify operator aggregates all of the values in a grid to a single value. The target grid of the unify operator is the *unit grid* consisting of a single node and no higher-dimensional cells. The assignment function used in the unify operator maps every cell in the source grid to the single node in the unit grid; the user supplies only an integer rank and an arbitrary aggregation function.

Definition 16. Let $\mathbf{G} = (G, g_0, g_1, \ldots, g_n)$ be a gridfield and let f be an aggregation function. Then $\mathsf{unify}(\mathbf{G}, i, f) = \mathsf{regrid}(\mathbf{U}, 0, \mathbf{G}, i, \mathsf{all}, f)$, where $\mathbf{U}$ is the unit gridfield $((\{c\}, \emptyset), \emptyset)$ and all assigns all i-cells in $\mathbf{G}$ to c, the only cell in the grid of $\mathbf{U}$.

We also define the accumulate operator for iterating over the cells at a given dimension and recording intermediate results of an aggregation for each cell. For example, we can call

$$\mathsf{accumulate}(\mathbf{G}, i, \text{“x=0”}, \text{“x=x+1”})$$

to assign a unique integer to each cell at domain i, beginning at zero. The first argument is a gridfield and the second argument is an integer dimension, as usual. The third argument is a *seed expression* initializing the value of an attribute “x” at the first i-cell to 0. The fourth argument is a *step expression* which tells the operator how to compute the next value from the previous value; in this case, the step expression simply increments the previous value by one.

Since the semantics of this operator relies on the physical order of the cells in the representation, we do not provide a formal definition.

The neighborhood operator can express computations over local neighborhoods of cells as defined by a stencil (see Section 2.3.6). The smoothing algorithm described in Section 2.3.3 and illustrated in Figure 2.10 is such a computation. The halo function, equipped with a stencil, takes the place of the assignment function in the general regrid operator. For example, we can use the assignment function ($\lambda c.\mathsf{halo}(c, 20)$) to assign neighboring 0-cells to each 2-cell c. (Refer to Section 2.3.6 for details.)

Definition 17. Let $\mathbf{G} = (G, g_0, \ldots, g_d)$ be a gridfield of dimension d. Let σ be a stencil $s_0 s_1 \ldots s_n$. Let a be an aggregation function. Then $\mathsf{neighborhood}(\mathbf{G}, \sigma, a) = \mathsf{regrid}(\mathbf{G}, s_0, \mathbf{G}, s_n, m, a)$, where $m(c) = \mathsf{halo}_G(c, \sigma)$.

The assignment function only depends on cells and data contained in the local neighborhood of a given cell, as defined by the stencil. In some cases, this limitation allows restrict and merge to commute with neighborhood down the source branch as well as the target branch.

Specifically, we have the following rewrite rules.

$$\textsc{RestrictNhood}\ \frac{\mathsf{restrict}(p, i, \mathsf{neighborhood}(\mathbf{G}, \sigma, a))}{\mathsf{restrict}(p, i, (\mathsf{neighborhood}(\mathsf{accrete}(\sigma, \mathsf{restrict}(p, i, \mathbf{G}), \mathbf{G}), \sigma, a)))}$$

$$\textsc{MergeNhood}\ \frac{\mathsf{merge}(\mathsf{neighborhood}(\mathbf{G}, \sigma, a), \mathbf{H})}{\mathsf{merge}(\mathsf{neighborhood}(\mathsf{accrete}(\sigma, \mathsf{merge}(\mathbf{G}, \mathbf{H}), \mathbf{G}), \sigma, a), \mathbf{H})}$$

The idea behind both of these rules is that the neighborhood operator only needs access to local information for each target cell. The restrict operator could not normally be commuted through regrid, since the assignment function might return different results on the restricted gridfield than it would on the original gridfield. However, the accrete operator ensures that the entire local neighborhood is available. The value of this rewrite is that, under certain conditions, the restrict operator can commute with the neighborhood operator, producing a less expensive recipe.

As an example, consider the grid in Figure 2.10. Let $\mathbf{G}$ be a gridfield defined over this grid, with an attribute x. The expression neighborhood($\mathbf{G}$, 010, average(x)) computes the average value of x for each local neighborhood of nodes, where a neighborhood is defined by the stencil 010. Imagine the user is only interested in the average value at one cell c, and he or she is able to define a predicate that only c satisfies, only_c. Then the user can write

$$\mathbf{R} = \mathsf{restrict}(\mathsf{only\_c}, 0, \mathsf{neighborhood}(\mathbf{G}, 010, \mathsf{average}(x)))$$

This expression can be rewritten to

$$\mathbf{A} = \mathsf{accrete}(010, \mathsf{restrict}(\mathsf{only\_c}, 0, \mathbf{G}), \mathbf{G})$$
$$\mathbf{R} = \mathsf{restrict}(\mathsf{only\_c}, 0, \mathsf{neighborhood}(\mathbf{A}, 010, \mathsf{average}(x)))$$

The intermediate result $\mathbf{A}$ contains the shaded 0-cells in Figure 2.10: The restrict operator removes everything but c, and the accrete operator includes the appropriate neighborhood. The neighborhood operator can then be evaluated for fewer cells. The last restrict is a pre-application of the predicate to shed the cells in which the user is not interested. Whether or not this rewrite is cost effective depends on the selectivity of the predicate, the cost of the aggregation function, and the overhead of each operator.

Another specialization of regrid involves a cross-product operator appearing upstream in the recipe. Recipes of the form regrid($\mathbf{A}$, bind($\mathbf{A} \otimes \mathbf{B}$, attr)) appear frequently. This fragment involves constructing a potentially large intermediate result from the cross product, binding a new attribute, then aggregating the new values back onto $\mathbf{A}$ using an assignment function that collapses the product structure of the grid. A better strategy uses iteration to avoid construction of the large intermediate product gridfield $\mathbf{A} \otimes \mathbf{B}$. For each cell in $\mathbf{A}$, we bind a subset of the data to $\mathbf{B}$, then aggregate to produce one output value. This technique involves more than just a specialized implementation of the regrid operator, as we must identify and replace a specific surrounding pattern in the overall plan.

Iterative execution of recipes is common in our system. Animated data products will call for repeated execution of a subplan of the form $\mathsf{regrid}(\mathbf{B}, \mathsf{bind}(\mathbf{A}),$ data). As data on $\mathbf{A}$ for each frame of animation is read via the bind operator, they must be interpolated onto a different grid $\mathbf{B}$ for display. The assignment function is based on the geometry of the two grids, which we assume for this example does not change over time. Therefore, recomputing the assignment for each frame is wasteful. There are two conceptual phases to the regrid operator: assignment and aggregation. To improve performance, the assignment portion of computation can be factored out of the loop structure using an analogue of a *join index*.

In a relational setting, the analogy is a join between tables R and S where the non-join attributes of S are frequently updated. Rather than recomputing the join whenever new answers are needed (and rather than maintaining an updateable materialized view), we can compute a join index: Store a pair of addresses for each pair of tuples from $R \times S$ that satisfy the join predicate.

We simulate a join index without requiring a full indexing subsystem using the regrid operator. For each cell in the target grid T, execute the assignment function, construct a collection object holding the assigned cell ids taken from the source grid, and record these collection objects as a new attribute on the target grid. The aggregation function is a collection constructor in this case.

To exploit this index attribute, we use another regrid operator with a less expensive assignment function. The assignment function only needs to know how to open up the collection and dereference the cell-id pointers stored inside. Recall that the original assignment function implemented an interpolation, requiring a search through the source gridfield to find geometrically overlapping cells. Even if we have access to an R-tree or other spatial index, having direct access to pointers saves a $\log n$ lookup for each cell in the target grid.

2.4.7 Abstract Syntax and the Fixpoint Operator

We have assumed thus far that the parameters controlling the behavior of our operators are provided as literals by the user writing the algebraic expression. We now generalize the situation by assuming the existence of a global environment in which the recipes are executed, allowing variables from this environment to appear as parameters to operators. The names of gridfields appear in this global environment. Also, the names of functions (attributes) passed to bind or appearing in predicates and functions passed to restrict or regrid also appear in this global environment.

The default global environment, which we will call the *catalog*, is a 6-tuple (GF, A, M, F, P). Let $\mathcal{S}$be a universe of symbols. These components are defined as

- $GF \subset \mathcal{S} \times \mathcal{GF}$, a set of named gridfields.
- $A \subset \mathcal{S} \times$(all attributes), a set of named attributes.
- $M \subset \mathcal{S} \times$(all assignment functions), a set of named assignment functions.
- $F \subset \mathcal{S} \times$(all aggregation functions), a set of named aggregation functions.
- $P \subset \mathcal{S} \times$(all predicates), a set of named predicates.

Although we show the catalog being indexed by symbols, the catalog may provide functions that return gridfields and attributes using numeric values. For example, the catalog for the CORIE system can return gridfields and attributes given a year, a day of the year, and a logical "timestep." Calls to these functions appear in the place of symbols at the leaves of expression trees.

Using the catalog, we can define an abstract syntax for a language implementing the gridfield algebra (using Haskell notation).

```
GFExp =
  | GridField GF
  | Bind GFExp Dim A
  | Restrict GFExp P
  | Cross GFExp GFExp
  | Accrete [Dim] GFExp GFExp
  | Merge GFExp GFExp
  | Union GFExp GFExp
  | Difference GFExp GFExp
  | Regrid GFExp Dim GFExp Dim M F
```

The symbol `Dim` is an integer representing a dimension. The notation [`Dim`] indicates a sequence of `Dim`.

Armed with the abstract syntax, we can define an eval function that walks an expression tree instantiating and evaluating the operators, accessing the catalog when necessary.

The complete abstract syntax that includes the fixpoint operator is this:

```
GFExp =
  | Hole Int                        -- Id
  | GridField GF
  | Bind GFExp Dim A
  | Restrict GFExp P
  | Cross GFExp GFExp
  | Accrete [Dim] GFExp GFExp
  | Merge GFExp GFExp
  | Regrid GFExp Dim GFExp Dim M F
  | Fixpoint GFExp GFExp Int        -- Body, Init, Id
```

The value Hole represents a variable in an expression. Each Hole is associated with exactly one instance of the fixpoint operator. This relationship between Holes and fixpoint operators is maintained by integer identifiers passed to the constructors.

The fixpoint operator repeatedly evaluates its "Body, " which may contain multiple occurrences of Hole. Prior to the first evaluation, the "Init" argument is evaluated. Any occurrences of Hole in the "Init" argument throws an exception[2]. Each subsequent evaluation replaces Hole with the result of the previous evaluation. The fixpoint operator terminates when the result is the same for two consecutive evaluations.

Let $\mathbf{E} = (F, f_0, f_1, \ldots, f_n)$ be a gridfield and let Exp be a gridfield expression containing one or more occurrences of the value Hole. Let eval be a function GFExp $\rightarrow \mathcal{GF}$ that evaluates a gridfield expression according to the semantics of the operators as defined in this chapter. Let $\text{Exp}_{id\rightarrow\mathbf{F}}$ be the gridfield expression formed from Exp by substituting the gridfield $\mathbf{F}$ for each occurrence of Hole with identifier id.

Definition 18.

$$\mathsf{fixpoint}(\mathbf{F}, \text{Exp}, id) = \begin{cases} \mathbf{F} & \text{if } \ \mathbf{F} = \mathsf{eval}(\text{Exp}_{id\rightarrow\mathbf{F}}\rangle) \\ \mathsf{fixpoint}(\mathsf{eval}(\text{Exp}_{id\rightarrow\mathbf{F}}), \text{Exp}, id) & \text{otherwise} \end{cases}$$

The fixpoint operator allows us to implement iterative recipes, one of the requirements from Section 2.3.6. We will show an example of its use in Section 2.5.

2.4.8 Feature Summary

Using only the gridfield operators defined so far, we can write recipes that express data products from several domains, including various 3-dimensional visualizations, horizontal and vertical "slices," aggregations over space and time, vorticity and gradient calculations, and 1-dimensional plots against time.

[2]We can use the Haskell type system to enforce this restriction, but we adopt the simpler abstract syntax for clarity.

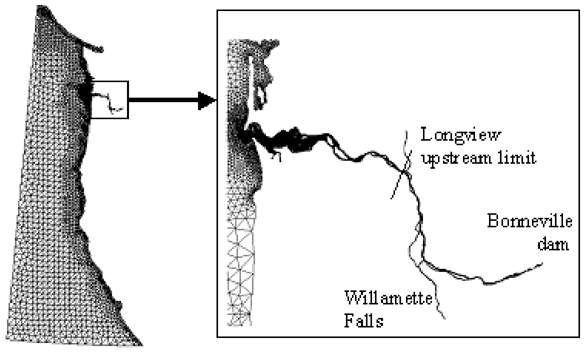
Our recipes generally involve five to ten operators. Existing custom routines to create these data products have hundreds of lines of array manipulations. Using visualization software, some burden was offloaded to library functions, but domain-specific optimizations were sacrificed. Our operators were able to re-express the semantic optimizations that were previously coded by hand.

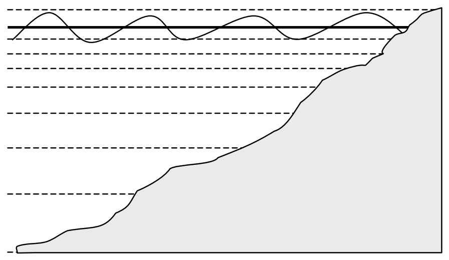
We summarize the features of our data model:

- Grids are first-class and of arbitrary dimension.
- Properties of grids can be described and reasoned about.
- The sharing of grids between datasets is explicit, as in Figure 1.5.
- Geometry is modeled as ordinary data, exposing topological equivalences between distinct geometric interpretations, e.g., different coordinate systems.
- Data can be associated with cells of any dimension, avoiding awkward conventions for associating, for instance, a cell's area with one of its nodes. For example, Figure 1.5 would be difficult to encode if we could associate data only with nodes.
- The data model captures irregular grids directly, while the cross-product operator can express the repeating patterns of regular and extruded grids.
- The regrid operator is extensible, admitting application-specific assignment and aggregation functions, as well as specialized implementations.
- The operators admit alternative implementations and accommodate different underlying gridfield representations.
- Subgrid relationships, though not always explicit, can be inferred statically.

2.5 EXAMPLES AND DISCUSSION

In this section, we exercise our model using data products from the CORIE system.





(a) (b)

Figure 2.16: A horizontal grid (a) and a vertical grid (b) used to discretize the domain of the Columbia River Estuary.

2.5.1 Example: Wetgrid

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 linear 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 normal to the earth's surface. These grids are illustrated in Figures 2.16(a) and 2.16(b), 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 2.16(b). The shaded region illustrates the bathymetry (depth) 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, binding a depth to each node. To filter out nodes in the product grid that are deeper than the river bottom, we

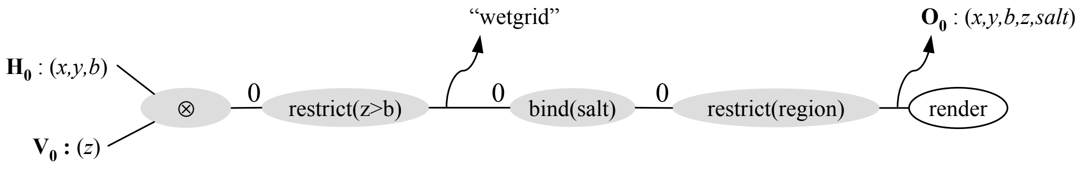


Figure 2.17: A recipe for visualizing a 3-D CORIE dataset.

---

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 gridfield **H** constructed from the 2-D horizontal grid H and tuples of the form (x, y, b) bound to the nodes. The attribute b captures the river's bathymetry at a particular location, and the attibutes x and y are geometric coordinates. We will also refer to a gridfield **V** constructed from the 1-D vertical grid V with tuples of the form (z) bound to the nodes, where z is the depth below mean sea level.

The task is to construct the grid over which the simulation outputs are defined, bind a dataset to it, restrict to a region of interest, and visualize the results. The recipe for this task is shown in Figure 2.17. 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 **H** and **V** gridfields, constructed using application-specific access methods. The notation $\mathbf{H}_{\mathbf{0}:(\mathbf{x},\mathbf{y},\mathbf{b})}$ in Figure 2.17 indicates that the functions at dimension 0 of the gridfield returns tuples of the form (x, y, b). The cross operator combines **H** and **V** to produce a different, 3-dimensional gridfield. After using restrict to filter out everything below the river bottom, we have our "wetgrid" (at the point labelled as such in Figure 2.17). After binding a salinity dataset to the wetgrid, we restrict the grid to a user-supplied region. In this example, the word "region" is shorthand for a predicate over x and y values that returns true for only those nodes positioned outside of the river's estuary. After evaluating the last restrict operator, we obtain a gridfield **O** for "ocean", which we will use in the next example.

2.5.2 Example: Plume Volume

The *plume* is the jet of fresh water that extends into the ocean at the mouth of a river. Plume dynamics are an active area of research in coastal research and oceanography with a variety of applications. For example, the plume of the Columbia River Estuary is an important salmon habitat. The fish are known to congregate near the edge of the plume; tracking the movement of this region can improve fisheries' efficiency. The volume of the plume is also of interest; large-scale oceanographic processes can be analyzed by studying plume size.

The plume is defined as the region of water outside the mouth of the river with a salt content below a given threshold. The recipe to compute the plume builds on the recipe that binds salinity to the "wetgrid" in Section 2.5.1.

Instead of restricting the wetgrid to a user-specified region, however, we want to identify the plume. We said that the plume is defined by the region 1) outside the mouth of the river and 2) with salinity less than a given threshold. We have already obtained a recipe for the region outside of the mouth of the river in Figure 2.17 based on the "region" predicate. We now assume this result has been previously computed and is available for reuse. The bottom gridfield in Figure 2.18 is the same as the result of the shaded portion of the recipe in Figure 2.17 (the gridfield **O**).

To illustrate the expressiveness of our operators, we compute the plume's volume using topological information. The first step is to compute the area of the 2-cells in **H** and the length of the 1-cells in **V**, using a regrid operator in each case.

We consider the aggregation over **H** first. The regrid operator uses **H** for both source and target gridfield, which is the meaning of the double lines connecting **H** to the operator oval. To compute the area of the 2-cells in **H**, we use an assignment function that maps nodes to the 2-cells to which they are incident, and then use an aggregation function that knows how to compute the area of a polygon given a sequence of (x, y) pairs defining its perimeter. These two functions appear as

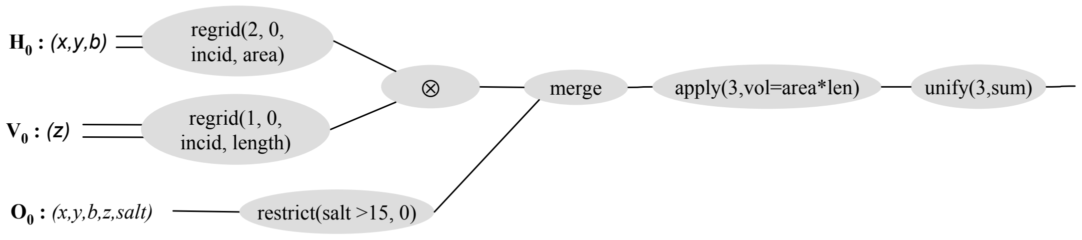


Figure 2.18: A plan for computing the volume of the *plume*, building on the wetgrid recipe in Figure 2.17. The double lines at left represent that the regrid operator taking two input gridfields that happen to be the same here.

"incid" and "area", respectively, in the arguments to the regrid operator in Figure 2.18.

The aggregation over **V** is nearly identical, except the aggregation function computes the length of a 1-cell instead of the area of a 2-cell. Once we have computed these measures for **H** and **V**, we compute the cross product of the resulting gridfields.

The gridfield **O** in Figure 2.17 also appears at the bottom of Figure 2.18 as the input to a restrict operator. The restriction defines the plume according to a salinity threshold. This result is compatible (see Section 2.2.3) with the result of the cross product, so we can merge the two gridfields and obtain reasonable results. After the merge operator, we have a gridfield representing the region outside of the river's estuary with geometry and salinity datasets bound to the 0-cells, and with length and area datasets bound to the 3-cells. We can now employ our specialized regrid operator apply to multiply area and length, resulting in each 3-cell being associated with its volume.

Finally, we use the unify operator to compute the volume of the entire plume. The unify operator, like the apply operator, is a specialization of the regrid operator. Any aggregation function may be used with unify; here we wish to sum the volumes of the individual cells.

Note that this plan assumes that the cells have straight edges. If our grid consisted of curvilinear cells, expressed by associating data to the 1-cells, then we would have a different geometric embedding, and therefore a different volume calculation. However, polyhedron formed by linearly translating a polygon through 3-dimensional space can be correctly handled by this plan, varying only the function that is applied.

2.5.3 Example: Minimum Cell Diameter

The ability to predict the running time of a simulation statically is important for scheduling computations on a computational cluster. The running time of a simulation depends on the "cardinality" of the grid used, as measured by the number of cells of a particular dimension. (The choice of dimension is not usually important, as the different measures are correlated by the correctness criteria of grid-generation packages, bounding the number of cells to which a node is incident.) The running time of a simulation may also depend on the data, as iterative solvers take longer to converge on, for example, an ill-conditioned matrix. The stability and accuracy of the solvers demand that the size of the timestep relative to the size of the smallest cell of a grid be bounded. Therefore, highly refined grids demand small timesteps, which leads to increased running time. The size of the smallest cells in the grid can therefore allow an estimate of the running time of a step of the simulation.

Figure 2.19 shows a recipe to compute the minimum diameter of any cell in a 2-D gridfield. The diameter of a cell is the diameter of the circle with the same area as the cell. This recipe demonstrates the generality of the regrid operator; each operation in the recipe is implemented using a form of regrid. Each instance of regrid exercises a particular specialized implementation as well, however. The first operator maps nodes to cells so that area can be computed; this task is implemented as an instance of the neighborhood operator. The second operator

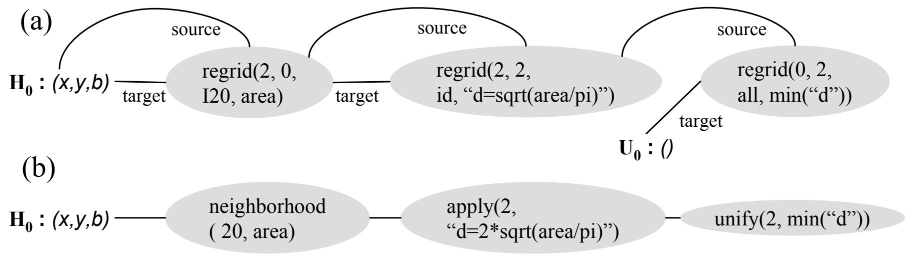


Figure 2.19: (a) A plan for computing the minimum "diameter" of the 2-cells in a gridfield using three regrid operators. (b) An equivalent plan using three specializations of the regrid operator. The minimum cell diameter is used to estimate completion time of a simulation.

---

applies an arithmetic operation to the area value. The assignment function in this case is simply the identity function, so we can use the apply operator. Finally, the results are aggregated using the unify operator, which maps all i-cells to the single node of the unit grid. The attribute d on the unit grid holds the minimum diameter of all cells in the gridfield **H**.

2.5.4 Example: Feature Extraction

Algorithm 5 in Section 2.3.5 is a pseudocode implementation for extracting connected regions from a dataset. The gridfield algebra can express this algorithm using the fixpoint operator. Figure 2.20 gives a plan in the gridfield algebra for extracting a region of low pressure from a 2-D gridfield.

The left-most restrict operator produces a gridfield containing only those nodes that are below the specified threshold, ρ. The unshaded operators compute the seed value. The apparent complexity of this sub-plan is due to fact that many cells may be associated with the minimum pressure value, and we must select one of them to seed the iteration. First we find the maximum value using the unify operator. Also, we use the apply operator to copy the attribute id to a new attribute $newid$. The

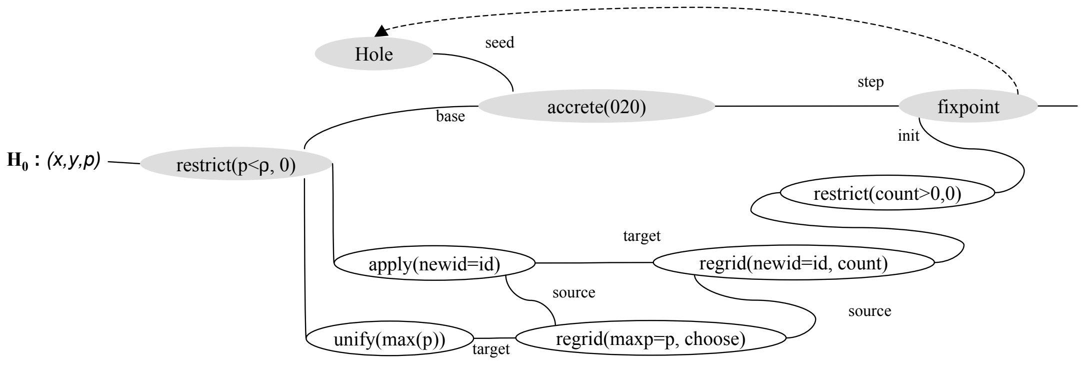


Figure 2.20: A plan for extracting a feature: a connected region with pressure values below a certain threshold.

---

id attribute is an implicit attribute uniquely identifying each cell. Next, we use the regrid operator to implement a join between the gridfield with pressure values and the unit gridfield with the max pressure value. The aggregation function choose chooses one of the matching tuples and returns it. One of the attributes in this tuple is *newid*. The next regrid associates a count of 1 with the cell matching the chosen value of *newid* and associates a count of zero with all others. Finally, we use restrict to remove all cells with a count of 0. Note that the latter two operators implement a semijoin. With the seed defined, we now iteratively grow a gridfield using fixpoint and accrete. On each iteration, the gridfield representing the feature is extended by including neighboring cells via the accrete operator with stencil 020.

2.5.5 Discussion

The examples in this section all involve data from the CORIE EOFS, but they demonstrate a range of capabilities. The first example is *constructive*, in that it is used to assemble a gridfield from constituent parts. The ability to express gridfield assembly within the gridfield algebra itself aids integration with l egacy data. We find that applications rarely have pre-packaged, self-contained, self-describing datasets ready to be read in as gridfields. Rather, the grid and each attribute may be in separate files and become available at different times. Gathering these data together for processing need not be outside the scope of the model, but it frequently is. For example, data must be loaded into a relational database before queries may be answered. Using the visualization library VTK [SML96], we were forced to design custom readers that effectively implemented cross product and bind in order to construct a gridfield for further processing. We will return to this issue in Chapter 4, where we present a mechanism for describing the ad hoc file formats used in practice and how to connect gridfields to them. We will show that generic readers for ad hoc file formats are competitive with hand-coded readers. We report empirical results from experiments with VTK in Chapter 6.

The second example performs geometric processing on the cells of the gridfield. Although the model itself does not explicitly involve geometry, functions to compute distance, area, volume, and so forth can be used as arguments to the regrid operator. In this sense, the gridfield algebra is *extensible*. Each of the data products we have encountered in practice can be expressed with the gridfield algebra and suitable assignment and aggregation functions.

The third example illustrates that gridfield algebra expressions can be used to compute scalar quantities in addition to topological structures. Scalar quantities (and, in fact, vector quantities) are captured with attributes bound to the unit grid.

The fourth example illustrates an iterative algorithm expressed as a gridfield recipe. Gridfields need not be relegated to simple functional transformation of datasets, but can be applied to complex computational tasks as well.

In the next chapter, we will discuss the implementation of the model, considering several alternative data structures to manage the topology of the grid.

Chapter 3

DATA STRUCTURES FOR GRIDFIELDS

Recall our thesis: Programs manipulating scientific datasets exhibit an algebraic structure that can be exploited to reason about their results and improve their performance. Having developed the model and shown examples of its use, the next step is to define a *reference implementation* for empirical evaluation.

In this chapter, we derive a reference implementation of gridfields, focusing on core data structures rather than recipe-level evaluation and optimization. We will derive a minimal interface from the requirements in Chapter 2, then consider alternatives for implementing the API using proposals from the literature. We find that many of these proposals are too specialized for use as a reference implementation of gridfields. We experimentally compare alternatives that are sufficiently general and select one for implementation on that basis. Details of operator implementation will be considered when they illuminate features or shortcomings of the data structure under consideration.

The requirements for manipulating gridded datasets derived informally in Chapter 2 are repeated here as a starting point for refinement.

1. Iteration over cells of a particular dimension.
2. Iteration over incident cells of dimension i given a cell of dimension j.
3. Read and write access to data values associated with cells of arbitrary dimension (though usually just one dimension at a time).
4. Identification of cells that satisfy a particular condition on their data.
5. Ability to construct a new gridfield using cells from a given gridfield.

6. Repetition of operations; recursion.

The first two requirements involve iterating over cells. The concept of an iterator offers a generic interface to a collection of cells and is widely supported by programming languages.

```
class iterator<T> {
  void next();
  T value();
  bool done();
}
```

The syntax is Generic Java, which supports polymorphism using a syntax resembling C++ templates. The iterator interface controls access to a collection of values of type `T` with three methods: `next` for advancing to the next element, `value` for retrieving the current value, and a boolean test `done` indicating that no more values are available. Equivalent formulations are possible.

A cell has identity, and two cells may therefore be compared for equality. A cell also has a dimension member of type integer. According to the model, the same cell may be used in multiple grids, so incident cells and data bound to a cell are not naturally part of the cell's definition. A grid data structure is not prevented from representing incidence information within a cell, however.

```
class Cell {
  bool equals(Cell c);
  int dimension;
}
```

A grid should support iteration over the cells at rank k. Given a k-cell c, a grid also allows iteration over the j-cells sharing an incidence relation with c. A gridfield allows access to the value of an attribute a given an k-cell c. Attribute values may also be mutated to allow efficient in-place implementations of operators.

```
class Grid {
  iterator<Cell> kcells(int k);
  iterator<Cell> incidence(int k, Cell c);
  bool incident(Cell c, Cell d);
}

class GridField public Grid {
  Value GetValue(string a, int dim, Cell c);
  void SetValue(string a, int dim, Cell c, Value v);
}
```

This core interface suffices to implement all the gridfield operators we have introduced, but a specific data structure may offer more efficient methods. For example:

- Testing membership of a cell in a grid requires a linear scan using the interface above, but can be implemented in constant time if the cell ids are known to occupy a contiguous range of integers.
- The `incident` method can be used to test two cells for incidence, but incidence relationships may be derivable from the cell identifiers. For example, an efficient representation of a 1-dimensional trajectory declares a node i to be incident to the edges i and $i + 1$. More generally, the 0-cells and 1-cells themselves might be entirely implicit and their incidence to higher dimensional cells calculable from the representation of those cells.
- A cell's adjacent cells can be retrieved through an index rather than by two calls to the `incident` method.
- The restrict operator requires a linear scan to identify cells whose bound data satisfy a predicate using this API, but a data structure may be equipped with an index for faster lookup.

In Chapter 2, we introduced compatibility between two grids, defined as the existence of a mutual supergrid. We will work with subgrids frequently, appealing to the existence of a supergrid to reason about properties. For example, if we know we have a subgrid of a well-supported grid, we need not not materialize every cell in the grid at every step of a recipe. A recipe that computes the global average temperature over a region needs access to the nodes only. The logical, unmaterialized supergrid contains all cells in the grid, however, and they can be restored later if needed. Another example: Triangles are usually represented as a sequence of nodes, and the edges are implicit, so a data structure that requires explicit edges is at a disadvantage. We can operate "edgelessly" with a 2-D gridfield with only 2-cells and 0-cells, knowing that the logical supergrid holds the edges. If they must be reified physically, then the catalog will supply them (perhaps dynamically via a view).

The number of cells accessed through the iterator returned by `incident` is limited in practice by requirements on cell *quality* [Ber98]. In two dimensions, for example, the minimum angle of a triangle is bounded to avoid "skinny" triangles that incur numerical interpolation errors. The result is that each i-cell tends to touch only a few j-cells. Calls to `incident` may therefore return an explicit set or vector for iteration.

Physical operator implementations can use the interface above to reduce dependence on a particular physical representation. This API is also exposed to authors of assignment functions passed to the regrid operator (see Section 13). To prevent brittle, unmaintainable libraries of assignment functions, we keep the grid API narrow. As described in Chapter 2, we identify common uses of the API to craft specialized implementations of the regrid operator. For example, the neighborhood operator access the local neighborhood of a cell through a stencil implemented as a small, finite number of calls to the `incidence` method.

In the remainder of the chapter, we will use the following notation to refer to the core API methods.

- C_k: Iterator for cells of dimension k.
- $I_{ij}(c)$: Iterator for the j-cells touching the i-cell c.
- $F_k(a, c)$: Access to the value of attribute a bound to cell c at dimension k.

3.1 CHARACTERIZING GRID REPRESENTATIONS

Grid data structures are typically designed to provide constant time access to incidence information with as little space overhead as possible. For example, a traversal of the n incident nodes of a polygon should require $O(n)$ time.

The model for grids we have described can be implemented directly as an *incidence graph* [Ede87]. Each cell becomes a node in the graph, and each incidence relationship becomes an edge. An incidence graph can be represented with familiar graph data structures; for example, a set of adjacency lists or an adjacency matrix of cells. These representations ignore dimension however. Operators generally access grids one dimension at a time: We want direct access to the nodes of a tetrahedron without having to search through the edges and faces. One solution is to divide the graph structure into multiple adjacency lists instead.

A fully generalized incidence graph representation has two problems: First, pathological grids can be constructed that are difficult to interpret in real applications (see Chapter 2, Section 2.2.2), and, second, accessing the incidence relation either requires time linear in the grid size (which is generally unacceptable) or significant redundancy. For example, the 0-cells might store pointers to the 2-cells so that I_{02} can be evaluated efficiently, but then 2-cells must also store pointers to 0-cells so that I_{20} can be evaluated efficiently. Incidence relationships implied by the transitive closure may also be represented redundantly.

Other grid representations proposed in the literature solve these problems by specializing on grids that obey certain properties: For example, only 2-D or 3-D grids may be allowed, or each cell must be a simplex (a triangle in 2-D or a tetrahedron in 3-D), or a d-dimensional grid must be embeddable in a d-dimensional space.

By targeting a narrower class of grids, designers can simplify the data structure (improving performance) and guarantee that only proper grids are constructed. For example, the edge-based data structures we will discuss can only represent 2-D connected, embeddable grids. Grid construction with this data structure proceeds by sewing edges together. Besides guaranteeing the embeddable and connected properties, this restriction avoids the need to build a separate object for each node, edge, and face; faces and nodes can be represented as members of the edge object. Also, the incidence relationship is implicit, so I_{10} and I_{12} can be evaluated in constant time. However, such restrictions exclude grids we encounter in our motivating applications: The CORIE grids are non-simplicial, for example.

Once a grid is constructed, verification of an arbitrary property may be difficult or even undecidable. For example, verification that a cellular complex (a formal foundation for some grid data structures) describes a three-manifold reduces to the Sphere Recognition problem. This problem is easy for $d \leq 3$ thanks to Euler's formula relating the number of nodes, edges, faces, and polyhedra in a grid. However, for $d = 4$, the problem is in NP, for $d = 5$ the problem is open, and for $d = 6$, the problem is known to be undecidable [Mar58].

So verification of grid properties may be expensive or impossible, and specialized data structures are often too restrictive for our purposes. In practice, we find that the lack of enforcement grid properties does not cause problems. Why should this be the case? The answer is that the gridfield algebra prevents low-level insertions of cells that can produce meaningless grids. Every operator except $\otimes$ produces a result that is compatible with its argument(s), meaning that properties preserved through compatibility are preserved through a class of gridfield algebra expressions. More precisely:

Proposition 5. *Given a tree T of gridfield operators from the set $\{\otimes$, union, merge, restrict, bind, accrete, regrid$\}$ in the gridfield algebra, construct a tree T' by* *applying the following transformations:*

$$\textit{merge}(\mathbf{A}, \mathbf{B}) \to \mathbf{A}$$

$$\textit{restrict}(p, i, \mathbf{A}) \to \mathbf{A}$$

$$\textit{bind}(\mathbf{A}, i, a) \to \mathbf{A}$$

$$\textit{accrete}(\sigma, \mathbf{B}, \mathbf{A}) \to \mathbf{A}$$

$$\textit{regrid}(\mathbf{A}, i, \mathbf{B}, j, m, a) \to \mathbf{A}$$

Then the grid of the result of T is a subgrid of the result of T'.

The consequence is that we can reason about the correctness of gridfield algebra expressions in terms of the correctness of their inputs. If a property is preserved through union, cross, and the subgrid relationship, then it is preserved through the gridfield algebra.

A gridfield representation must also allow access to data values bound to cells. All of the structures we discuss have some ability to associate data with cells. There are at least four ways to do so: 1) The cell's definition may include a pointer to a tuple, 2) an auxiliary structure may relate each cell with a tuple, 3) each tuple may hold a pointer to a cell, or 4) an array of cells may be positionally aligned ("parallel") with an array of tuples, an implicit representation of both 2) and 3). An advantage of the fourth solution is that more than one array of tuples can be bound to a grid; conversely, each attribute within a tuple may constitute its own positionally aligned array. This configuration allows attributes to be bound to a grid and projected out of a grid in constant time. Further, positionally aligned arrays ("parallel arrays") are a common idiom in both scientific data processing and visualization, classes of systems with which gridfields must interact. Positionally aligned arrays also do not require an explicit cell object to be constructed. For example, a 0-dimensional gridfield consisting of 100 nodes and a single attribute x requires one integer (the number of 0-cells) and 100 floating point numbers. The nodes themselves are entirely implicit.

The validity of a particular solution method for a numerical simulation depends crucially on the characteristics of the *boundary conditions*. For example, the value of the solution may be fixed at a boundary (Von Neuman boundary conditions), or the value of the derivative of the solution may be fixed (Dirichlet boundary conditions); each is solved in a different way. Data structures for manipulating gridded datasets may have a built-in representation of the boundary, or the boundary may be represented as a separate structure. The boundary of a grid may also be efficiently derivable from the full grid.

Let us summarize this introductory discussion: The gridfield model admits a direct representation based on incidence graphs. A full incidence graph is redundant and inefficient, and does not prevent the construction of pathological grids. However, we can work at times with only a portion of the full incidence graph, since the gridfield operators preserve or allow recovery of the important grid properties. Other important characteristics of a gridfield data structure include how data values are associated with the cells, and how boundaries are represented.

We will consider specialized data structures, many of which will prove too restrictive given the grids we encounter in practice. Three data structures prove worthy of further analysis and testing: the NGMap of Lienhardt [Lie94]; the indexed cell set(ICS) representation, popular for its simplicity; and our modified, tailorable incidence graph, TIGraph. We will then describe implementation details of the gridfield model with respect to these data structures. Finally, we will compare them experimentally using the restrict operator and the neighborhood operator. The restrict operator is an appropriate test case since it involves topology queries, access to bound data, and potentially the construction of a new result gridfield. The neighborhood operator is edifying since it involves repeated incidence queries to implement stencils. We conclude that the ICS representation, optionally extended with features of the TIGraph representation, offers the best balance of performance and flexibility.

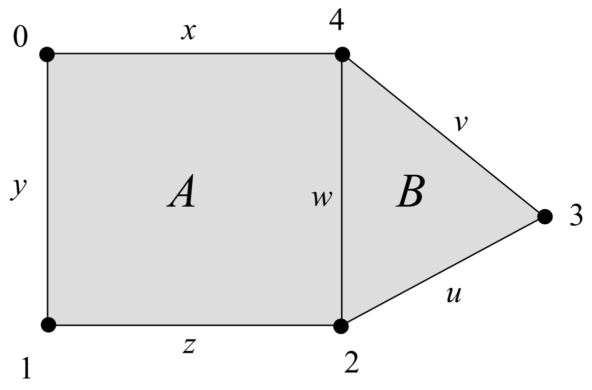


Figure 3.1: A simple grid we use to illustrate representation alternatives.

3.2 REVIEW OF GRID DATA STRUCTURES

In this section, we will use the grid in Figure 3.1 as a running example to illustrate data structures proposed in the literature.

3.2.1 Incidence Graph

The grid model can be directly implemented with an explicit incidence graph [Ede87] whose nodes are cells and whose edges are incidence relationships. We refer collectively to the all variants of this representation as TIGraph. The cells are partitioned by dimension and the incidence relation is constrained to be a partial order respecting dimension. A Hasse diagram of the incidence graph of the mesh in Figure 3.1 appears in Figure 3.2. Edges implied by the transitive closure of the partial order are omitted.

The representation of the graph edges varies among concrete TIGraph implementations. A simple adjacency matrix is likely to be space inefficient. Mesh quality metrics usually prevent more than 5-10 cells from touching any one cell, so the resulting adjacency matrix would be very sparse.

Adjacency lists (adjacency in the sense of graphs rather than grids) are more

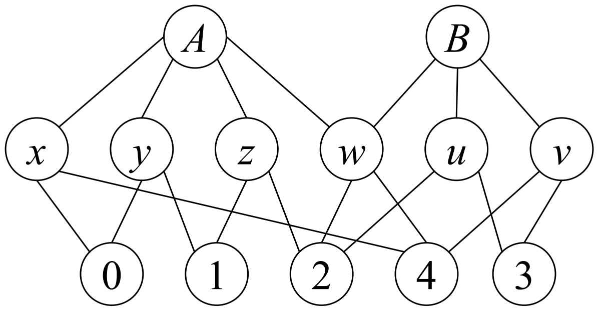


Figure 3.2: A Hasse diagram of the incidence graph of the grid in Figure 3.1. The edges are incidence relationships, though the relationships implied by the transitive closure, such as $(A, 0)$, are implicit in the vertical stratification of the graph vertices.

common, but a decision must be made about which dimensions will be explicitly linked. For example, an edge may explicitly list its incident nodes, but the nodes need not explicitly reference the edges. More complex inferences are common: The edges of a triangle can be inferred from an ordered list of nodes; neither the edges themselves nor the corresponding incidence relationships need be explicit.

A recent exemplary variation on the incidence graph is the Simplified Incidence Graph (SIG) of de Floriani, Greenfieldboyce, and Hui [FGH04]. A SIG explicitly stores $I_{i(i-1)}$ for all $0 < i \leq \mathsf{dim}(G)$, which the authors refer to as *boundary relations*, as well as a *partial co-boundary relation* (PCBR) for $0 \leq i < \mathsf{dim}(G)$. A partial co-boundary relation is a subset of $I_{i(i+1)}$; specifically, a single arbitrary $(i + 1)$-cell is stored for each i-cell. The PCBR allows upward traversal of the incidence graph (upward with respect to increasing dimension), while minimizing redundancy.

The benefits of this representation include efficient space utilization relative to other variations of the incidence graph, as well as amortized constant time access for incidence and adjacency queries (generally assumed to be a requirement for topological representations). The SIG representation is applicable only to simplicies however, so we cannot apply it directly to all the grids we find in practice.

3.2.2 Indexed Cell Set

A popular representation of unstructured grids—favored by the visualization community for its compactness—is the *indexed cell set* (ICS) [SCESL02]. ICS can be considered a variation of an incidence graph representation that specifies representations of geometric coordinates and relies on conventions for representing common cell shapes. ICS stores an explicit I_{d0} relation as well as an array of tuples representing geometric coordinates, where d is the dimension of the grid. I_{d0} is represented as an adjacency list where order matters; the k-cells for $0 < k < d$ are inferred by the API implementation (or by the application) based on the order of nodes.

Before we describe this idea in detail, consider Figure 3.3, an example of the ICS representation of the 2-D grid of Figure 3.1, extended with some geometric information implied by the illustration.

A list of geometric coordinates for each node is followed by a sequence of 2-cells. Each 2-cell consists of an integer n (the number of nodes) followed by n node indexes into the list of coordinates. The example ICS is written using ASCII values; ordinarily these numbers would be in a packed binary format.

The ICS representation is a variation on an adjacency list representation of the incidence graph, with two differences. First, it was designed for geometric models, and so includes a specification for representing geometric data as well as the topology. Second, the only portion of the incidence relation represented is between d-cells and 0-cells. The i-cells for $0 \leq i \leq d$ can sometimes be inferred from the order of the 0-cells.

For example, a 2-D grid encoded in an ICS representation is assumed to be well-supported, so the 1-cells can be derived implicitly from the order of the node

```
0 | 0.0, 1.0
1 | 0.0, 0.0
2 | 1.0, 0.0
3 | 1.6, 0.4
4 | 1.0, 1.0
  |
5 | 4; 0, 1, 2, 4
6 | 3; 2, 3, 4
```

Figure 3.3: An example of the ICS representation. The numbers delineated to the left are line numbers only. A sequence of x, y coordinate pairs is followed by a sequence of cell definitions. Each cell definition is a sequence of node references, preceded by the sequence length.

---

references for each 2-cell. A 2-cell encoded as $(2, 3, 4)$ implies the existence of a 1-cell to which nodes 2 and 3 are incident, another 1-cell to which nodes 3 and 4 are incident, and a third 1-cell to which nodes 4 and 2 are incident.

Relying on implicit representation of cells has disadvantages. Most prominently, there is not enough information in an ICS to express arbitrary grids. Cells are limited to pre-defined shapes identified by the length of the node list and the overall dimension of the grid: three nodes indicates a triangle, four nodes indicates a quadrilateral in two dimensions and a tetrahedron in three, six nodes indicates a hexagon in two dimensions and a prism with triangular bases in three, eight nodes indicates a cube in three dimensions. Most common cell types can be discerned with this convention, but one can imagine pathological cells. For example, both a prism with triangular bases and an octahedral cell are three dimensional cells represented with six nodes. The use of convention to define topology can lead to errors that are difficult to detect combinatorially. For example, an incorrect ordering of nodes representing a prism can lead to a self-intersecting degenerate

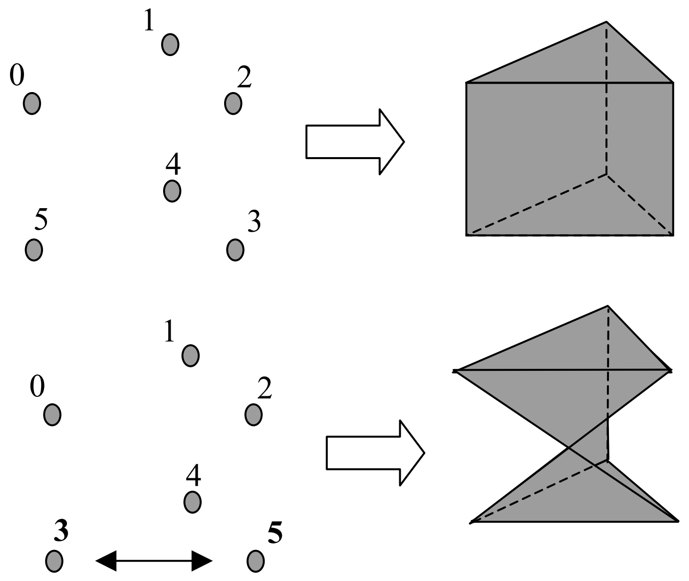


Figure 3.4: An Indexed Cell Set (ICS) representation relies on the ordering of nodes to indicate faces and edges in three dimensions. An unconventional ordering of nodes can lead to degenerate cells, a condition that is expensive to check for.

cell, as in Figure 3.4. However, a test for this condition is specific to each cell type and potentially expensive, requiring evaluation of geometric relationships between multiple pairs of nodes.

The ICS representation exploits implicit information so as to require less space than an explicit incidence relation, but can only capture well-supported grids. An incidence relation may be represented by a sequence of pairs of the form (c_2, d_0), where c_2 and d_0 are explicit references to a 2-cell and a 0-cell respectively. This representation requires $2N$ integers, where N is the number of relationships that must be modeled. The ICS representation requires $N+M$ integers, where M is the number of 2-cells. For a well-supported grid, all 2-cells have at least one incident node, so $M < N$. If $M > N$, we could not use the ICS representation.

Note that homogeneity is not enforced or required by this representation; there may be nodes that are not referenced by any 2-cell (Every 1-cell implicitly is.)

The portion of the complete incidence relation that must be handled implicitly by the API scales exponentially with dimension. For each triangle in a 2-D grid we must infer the existence of three edges. For each tetrahedron in a 3-D grid, we must infer the existence of four triangles. For an nD simplicial grid, we must infer the existence of $(n+1)!$ k-cells. However, the only information we have to determine an appropriate shape for a n-cell is the number of nodes involved and the dimension of the grid. The basic ICS representation is therefore unsuitable for representing grids of dimension higher than three.

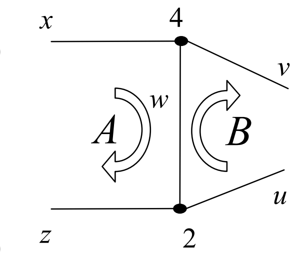
The *extended ICS* representation allows additional lists of explicit cells for dimensions less than d. For example, if only the boundary of our running example (Figure 3.1) was present and the internal edges were absent, we could add another section to the ICS representation containing only these edges:

```
2; 0, 1
2; 1, 2
2; 2, 3
2; 3, 4
2; 4, 0
```

The 2-D ICS representation, under the well-supportedness assumption, can be used to implement I_{20} and I_{21} efficiently. We can support additional methods efficiently by augmenting (or replacing) the base representation with additional information. For example, we can add inverse mappings from nodes to k-cells, efficiently implementing I_{0k}, or an *adjacency* relation associating d-cells that share an incident cell.

Since CORIE data is initially generated in an ICS representation, and visualization clients frequently use an ICS representation, convenient translation to and from an ICS representation is desirable. As mentioned, ICS is a specialization of TIGraph, where only the I_{k0} relationships are explicit. Our tested implementation of ICS is similar to the extended ICS representation, but with two distinctions: We

(a)



(b)

Vertex	Edge
4	*w*
3	*w*
2	*z*
1	*z*
0	*x*

Face	Edge
A	*y*
B	*u*

(c)

Edge	Vertices		Faces		Left Traversal		Right Traversal	
Name	Start	End	Left	Right	Pred	Succ	Pred	Succ
z	1	2	*A*	-	*w*	*y*	-	-
y	0	1	*A*	-	*z*	*x*	-	-
x	4	0	*A*	-	*y*	*w*	-	-
w	2	4	*A*	*B*	*x*	*z*	*u*	*v*
v	4	3	-	*B*	-	-	*w*	*u*
u	3	2	-	*B*	-	-	*v*	*w*

Figure 3.5: The winged-edge data structure. (a) An illustration of the information in the winged-edge record for edge w. The solid white arrows indicate left and right traversals of neighboring faces. (b) The vertex and face tables linking each node (edge) to an arbitrary touching edge. (c) The edge table of the winged edge representation. Empty fields (null pointers) capture boundary information.

---

separate geometric data from topology data, storing x and y values as column-wise attributes of nodes (see Section 3.3.1), and we extend ICS with an inverse mapping from nodes to k-cells, implementing I_{0k}. The inverse mapping is constructed automatically when an operator calls I_{0k} (which makes the first call expensive). An array of cells is encoded as a contiguous array of lists of node references, and the inverse mapping is a hash index mapping nodes to lists of k-cells.

What about I_{ij}, where $i \neq 0$ and $j \neq 0$? There are two choices: Grids generated by tools for narrow cases (2-D simplicial grids, for example) have the property that $\mathsf{nodes}(x) \subseteq \mathsf{nodes}(y) \Leftrightarrow x \preceq y$, so I_{ij} may be implemented via I_{i0} and I_{j0}. Alternatively, additional explicit I_{ij} methods can be provided.

3.2.3 Edge-based Structures

Edge-based structures can only represent 2-D surfaces. There may be additional properties that must hold for the surface being represented; we will review them informally here. An *orientable* surface has two discernible "sides;" a Möbius strip is the typical example of a non-orientable surface. A *closed* surface has no holes in its one-dimensional boundary. A *manifold* surface locally resembles Euclidean space, but may have a more complex global structure.

The winged-edge data structure [Bau72] can represent 2-D, manifold, complex, orientable, closed surfaces. As with several of the representations we will discuss, the winged-edge structure adopts the edge as the fundamental object. For each edge, a record is inserted into an edge table (Figure 3.5). Each edge record holds pointers to its left and right faces, its origin and destination nodes, and four adjacent edges. The four adjacent edges correspond to a forward and backward traversal of the left face, and a forward and backward traversal of the right face. In addition to the edge table, a node table and a face table map each node (face) to a single incident edge. From that edge, the remaining edges can be gathered via the edge table.

Iteration over edges associated with a node and edges associated with a cell are straightforward, but accessing the faces associated with a node is awkward since there is no consistent way to order the pointers. That is, a pointer to an edge does not consider which direction we are coming from. The data structure can be extended with an extra bit for each edge-pointer, but this extension complicates the implementation.

The *half-edge* data structure solves the orientation problem by splitting each edge into two *half-edges* representing the edge as well as its orientation. Each half-edge record stores a pointer to its opposing half-edge, an incident node, and a face to which it is incident. The choice between the origin and destination nodes is a matter of convention; the term *split-edge* is sometimes used to refer to the choice

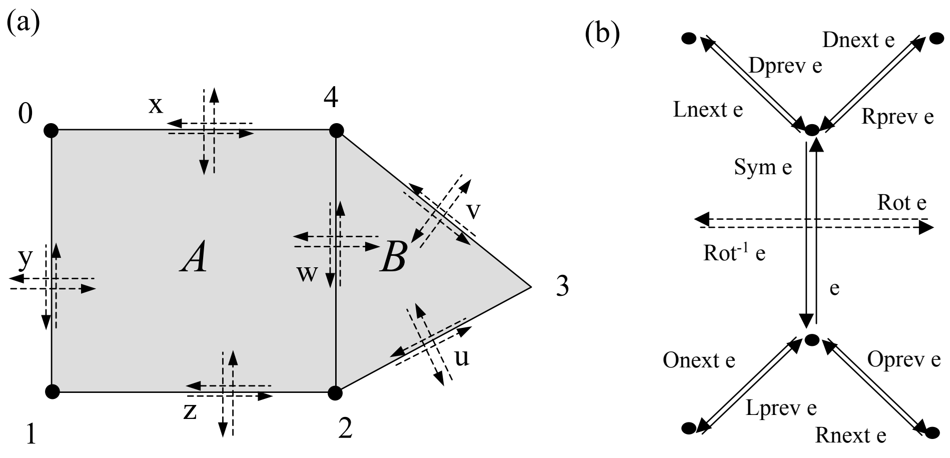


Figure 3.6: The quad-edge data structure. (a) Four quad-edge structures are used to represent each edge in the original grid. (b) The edge algebra elegantly allows access to neighboring edges in both the primal and dual grid.

of faces as the origin.

If we store half edges from node to node as well as *dual* half-edges from face to face, we have the quad-edge data structure proposed by Guibas and Stolfi in 1984 [GS95]. The quad-edge structure again adopts the edge as the primary interface to the topology information, but implements a formal edge algebra rather than prescribing a particular table of data. The edge algebra provides four ways of looking at the same edge, depending on the orientation of the beholder. If we imagine a bug standing on the edge, it may be facing the origin node, the destination node, the left face, or the right face. To represent orientation, some implementations allow the bug to flip upside down to be standing on the underside of the surface, providing four additional orientation options.

Operators for changing the bug's orientation and movement to a "next" edge allow traversal of the surface. An illustration of the quad-edge structure and the operators appears in Figure 3.6. This set of operators form an elegant algebra obeying many laws. The common operators are shown in Table 3.1, along with

Operator	Action	Implementation
Rot	Rotate 90 degrees.	Explicit pointer.
Onext	The next edge around the origin.	Explicit pointer
Flip	Flip upside down.	Set a flipped bit.
Sym	Rotate 180 degrees.	Rot^2
InvRot	Rotate -90 degrees.	Rot^3 or Flip Rot Flip
Dnext	The next edge around the destination node.	Sym Onext Sym
Lnext	The next edge around the left face.	InvRot Onext Sym
Rnext	The next edge around the right face.	Rot Onext InvRot
Oprev	The previous edge around the origin node.	Rot Onext Rot
Dprev	The previous edge around the destination node.	Rot^3 Onext Rot^3
Lprev	The previous edge around the left face.	Onext Sym
Rprev	The previous edge around the right face.	Sym Onext

Table 3.1: Operators for the Quad-Edge data structure.

their usual implementation. Of course, the algebraic equivalences allow alternative implementations. All rotations and orderings are counter-clockwise.

Although we defined the operators in terms of nodes and faces, there is actually no way to distinguish the two concepts. Put differently, the mesh and its *dual* are represented simultaneously. An edge can be described in terms of its origin and destination node; its dual edge can be described in terms of its origin and destination faces.

The physical data structure involves four oriented edge objects to represent each unoriented edge and its dual. Data can be associated with the nodes via a pointer to the origin node of each record.

The quad-edge structure can only represent 2-D, orientable, closed surfaces. Non-manifold “dangling” edges can be represented by a fixpoint for the `onext` operator, but other violations of the manifold property, specifically edges that are incident to more than two faces, cannot be represented. (Note that we are describing properties defined on the infinite, continuous, topological surface, not the discrete, combinatorial data structure.)

3.2.4 Cell Tuples

A grid G of dimension d can be represented as a set of *cell tuples* [Bri89] together with a sequence of operators $switch_i$ for $0 \leq i \leq k$. Each cell tuple $(c_k, c_{k-1}, \ldots, c_0)$ satisfies $c_k \succ c_{k-1} \succ \ldots c_0$. Each operator $switch_i$ maps a cell tuple d to another cell tuple e such that d and e differ only at dimension i. There is a unique cell tuple satisfying this condition if the underlying topological space being modeled is manifold; i.e., each k-cell is homeomorphic to the open k-ball.

The set of cell tuples representing the grid in Figure 3.1 is

$$\begin{array}{llll}
(A,z,2) & (A,z,1) & (A,y,1) & (A,y,0) \\
(A,x,0) & (A,x,4) & (A,w,4) & (A,w,2) \\
(B,w,4) & (B,v,4) & (B,v,3) & (B,u,3) \\
(B,u,2) & (B,w,2) & &
\end{array}$$

Some examples of switch operators applied to the cell tuples of representing Figure 3.1 are

$$\begin{aligned}
switch_0((A,z,2)) &= (A,z,1) \\
switch_1((A,z,2)) &= (A,w,2) \\
switch_2((A,w,2)) &= (B,w,2)
\end{aligned}$$

The boundary of a grid can be represented by allowing fixpoints for $switch_k$; defining, for example, $switch_2((A,z,2))$ to be $(A,z,2)$. Alternatively, the set of cells may be extended with an additional k-cell Ω representing the exterior of the grid, so that, for example, $switch_2((A,z,2)) = (\Omega,z,2)$. Every $(k-1)$-cell on the boundary of the grid is incident to Ω.

The formal development of the cell tuple representation stems from the *barycentric subdivison* of a grid. The barycentric subdivision B of a grid G is a grid

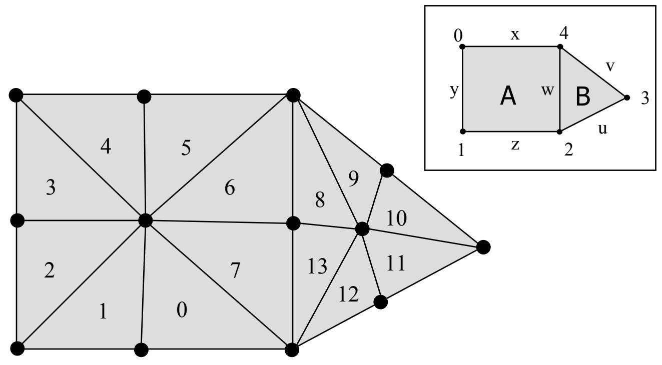


Figure 3.7: The barycentric subdivision of the grid in Figure 3.1, which is repeated in the inset. The resulting simplicial complex provides the technical machinery to develop the cell tuple data structure.

---

constructed as follows. For each cell c in G, add a 0-cell v_c to B. If $x \prec y$ in G, then add a 1-cell e_{xy} to B and assert $v_x \prec e_{xy}$ and $v_y \prec e_{xy}$. If $w \prec x \prec y$ in G, add a 2-cell t_{wxy} to B and assert the appropriate incidence relations. This process can be continued for grids of arbitrary dimension, forming a d-dimensional simplicial complex. Each d-cell in this complex corresponds to a cell tuple. The barycentric subdivision of the grid in Figure 3.1 appears in Figure 3.7. The 13 numbered d-cells correspond to the 13 cell tuples listed previously.

The switch operators, $switch_i$, link pairs of adjacent d-cells (e.g., $switch_0(2) = 3$ and $switch_2(13) = 7$ in Figure 3.7). Using the switch operators, we can define *orbits* that allow access to incident cells. For example, to visit the nodes around a face, follow the orbit defined by repeated application $\{switch_1, switch_2\}$. The concept of orbits will be revisited in the next section.

3.2.5 Generalized Combinatorial Maps

A generalization of cell tuples proposed by Lienhardt [Lie94] allows representation of some forms of non-manifold surfaces. We have defined embeddable grids as grids in which each $(d-1)$-cell cell is incident to no more than two d-cells. A 2-D grid with isolated nodes or dangling edges is embeddable but not manifold.

Lienhardt's *n-dimensional generalized combinatorial map*, or NGMap, can represent non-homogeneous grids and grids that are not strongly-connected, but cannot represent grids that branch. Using properties as we defined them in Chapter 2, we say that NGmaps can represent embeddable grids.

An NGMap representing a grid G of dimension k is a set of *darts* D and a set of involutions α_i on D. Just like a cell tuple, a dart represents the relationship between a d-cell, a $(d-1)$-cell, and so on down to a 0-cell. The term involution implies that $\alpha_i(\alpha_i(c)) = c$. These involutions are essentially "wiring" information about darts.

An illustration of an NGMap representing the grid in Figure 3.1 appears in Figure 3.8. Intuitively, α_i may be thought of as a movement from one dart to another, so that the new dart only differs at dimension i. Using the labeled darts in Figure 3.8, we can give some examples of α_i:

$$\begin{aligned} \alpha_0(2) &= 3 & \alpha_0(3) &= 2 \\ \alpha_0(9) &= 10 & \alpha_1(4) &= 3 \\ \alpha_1(10) &= 11 & \alpha_2(3) &= 3 \\ \alpha_2(7) &= 13 \end{aligned}$$

In the first example, we move from dart 2 to dart 3, staying in contact with the edge y and the face A. In the third example in the left column, we move from dart 10 to dart 11, staying in contact with the node 3 and the face B. In the last example, we move from dart 7 to dart 13, staying in contact with the node 2 and the edge w.

These involutions offer an elegant method of traversing the incidence information (and implementing the `incident` method). To visit all the edges around a

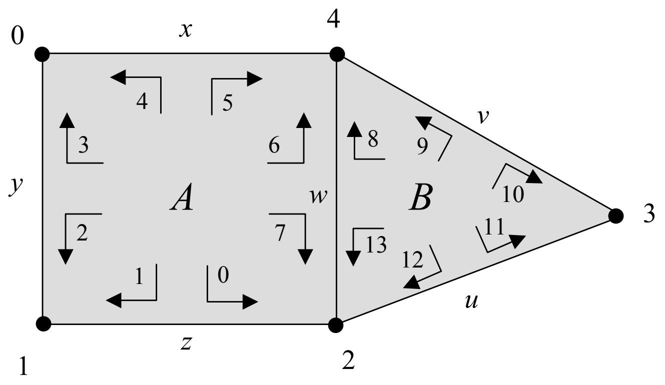


Figure 3.8: Each arrow corresponds to a dart in the NGmap representing the grid of Figure 3.1. The numbers associated with the darts correspond to the labeled *d*-cells in the barycentric subdivision of Figure 3.7.

---

face, we can successively apply the composition $\alpha_0\alpha_1$. The sequence of darts returned by this process is called an *orbit*, written $< \alpha_0\alpha_1 >$. For example, in Figure 3.8, the orbit $< \alpha_0\alpha_1 >$ beginning at the dart labeled 8 is the sequence 9, 10, 11, 12, 13, and 8 in that order. To visit all the faces adjacent to a face, we traverse the orbit $< \alpha_2\alpha_1\alpha_0\alpha_2 >$. An orbit is a set of darts rather than a set of cells, so to avoid duplication each dart must be associated with cell identifiers. An implementation may use a pointer to a data object or an explicit cell identifier.

In addition to being an involution each function α_i has the following property, ensuring that darts are sewn together completely or not at all:

$$\forall i, j.0 \leq i < j - 2, j \leq k, \alpha_i \circ \alpha_j \text{ is an involution.}$$

Both darts and cell tuples represent the *relationship* between cells rather than the cells themselves. For a closed manifold grid, darts correspond exactly to cell tuples and involutions are equivalent to the switch operators [Bri89].

In order to model only closed, manifold grids, fixpoints are allowed only for α_d. Fixpoints for α_d indicate darts on the boundary of a connected region. This restriction precludes the pathological grids iii) and iv) in Figure 2.7. In many treatments, fixpoints are allowed at other dimensions to represent solitary nodes, "dangling" edges, or other non-manifold artifacts.

Invariants (e.g., manifoldness) are enforced constructively in the implementations of four methods

$$\begin{array}{l} \mathsf{Dart}\ \textbf{mkDart}() \\ \textbf{killDart}(x : \mathsf{Dart}) \\ \textbf{sew}(i : \mathsf{int}, x : \mathsf{Dart}, y : \mathsf{Dart}) \\ \textbf{unsew}(i : \mathsf{int}, x : \mathsf{Dart}, y : \mathsf{Dart}) \end{array}$$

The method **mkDart** initializes a dart. This operation is sometimes interpreted as creating an isolated node. However, a dart really represents the *relationship* between $d + 1$ different cells (for a d-dimensional grid), so such an interpretation is incomplete. To sew darts x and y at dimension i, they both must be *free* at dimension i, meaning that both x and y are fixpoints for α_i. (Treatments that do not permit fixpoints at any dimension must define higher-level operators, based on sew and unsew, that maintain the appropriate invariant.) After sewing, we have $\alpha_i(x) = y$ and $\alpha_i(y) = x$, and the condition that $\forall i, j.0 \leq i < j - 2, j \leq k, \alpha_i \circ \alpha_j$ is an involution. This last condition prevents "partial" sewings; for example, two darts constituting an edge sewn to different faces. The method **unsew** frees the dart at the given dimension. The method **killDart** frees the dart at all dimensions, and excises it from the grid.

3.2.6 A Relational Implementation

The data structures described thus far were all designed specifically for grids. What about adapting more general data models for use with grids, such as the relational data model?

A direct representation of the grid model would consist of two relations

$$C(id, dim), \quad I(from, to)$$

where $C(i, k)$ indicates a k-cell with a numeric id and $I(c, d)$ means $x \preceq y$. The relationship between the cell tuple and NGMap representations can be expressed in the relational algebra. First, we derive the cell tuple representation using the relational algebra. Let

$$C_i = \rho_{id \to i}(\pi_{id}(\sigma_{dim=i}(C)))$$
$$C_i \bowtie_I D_j = \pi_{i,j}(C_i \bowtie_{id=from} I \bowtie_{to=id} D_j)$$

where ρ is the rename operator, π is the project operator, and σ is the select operator.

The cell tuple relation D of a grid $G = (X, I)$ can then be expressed as

$$G_d \bowtie_I G_{d-1} \bowtie_I \ldots \bowtie_I G_1 \bowtie_I G_0$$

When used to represent a d-dimensional grid, this relation has $d + 1$ columns. We assume that the column corresponding to dimension i is named ci, though we omit explicit rename operators for clarity.

The set of darts D of the NGMap $(D, \alpha_0, \alpha_1, \ldots, \alpha_d)$ is exactly the cell tuples and has scheme $D(c0, c1, \ldots, cd)$. Let each involution $\alpha_i(x)$ in the NGMap can be expressed as a relation

$$D \bowtie_{(\mathrm{c}0=\mathrm{c}0)\wedge\ldots(\mathrm{c}(i-1)=\mathrm{c}(i-1))\wedge(\mathrm{c}(i+1)=\mathrm{c}(i+1))\wedge\cdots\wedge(\mathrm{c}d=\mathrm{c}d)} D$$

that is, an equijoin on all attributes except i. For an embeddable grid, the result of this expression will have one or two tuples [Lie94], depending on whether or not the cell is part of the boundary.

The other data structures we have considered exhibit an obvious tabular structure; devising an appropriate relational implementation is straightforward.

So the relational model can adequately express the individual data structures used to represent grids. However, there are some difficulties when attempting to model a large and growing number of datasets bound to a relatively small number of grids. Four possible schemas for capturing gridfields bound to the same fully general grid are:

1. $\textsc{Var}_1$(cell, value), $\textsc{Var}_2$(cell, value), . . .
2. CellData(cell, var_1, var_2, . . .)
3. CellData(cell, var_id, value)
4. CellData(cid, var_id, value), Cell(cid, grid_id)

Schema 1 models each dataset $\textsc{Var}_i$ as a separate relation. This design agrees with the existing storage format in the CORIE system, as every simulation run generates many arrays, each stored separately. In a relational setting, however, this design suffers from several disadvantages. First, each new dataset requires a change to to the schema, making robust queries difficult to write and maintain. The CORIE system generates data-sets continuously; continuous schema changes are undesirable. Second, the grid is obscured; the fact that all of these datasets are bound to the same grid is not captured. Third, each cell definition is redundantly stored in several tables.

Schema 2 models each dataset var_i as an attribute in a single large relation. We still must modify the schema whenever new datasets are added, but the common grid between these datasets is exposed and the redundancy is removed. However, this design is constrained by table attribute limits found in many popular RDBMS. Also, a recipe is not likely to access all attributes of these very long tuples, resulting in a great deal of wasted I/O (unless you use a column-store [SAB+05]).

Schema 3 uses a single relation with only three attributes. Each tuple holds a cell definition, a dataset identifier *var_id*, and a data value. No schema changes are required to add a new dataset, though each new dataset must have the same type (e.g., integer, float). Multiple copies of the same cell are still stored, however. This design also requires multiple self-joins to bind many attributes to the same grid. Finally, the common grid between var_1, var_2, etc. is again obscured. If an erroneous "extra" cell is inserted into the grid for a particular variable, no error can be raised by the database.

Schema 4 is Schema 3 extended with an assignment of each cell to a grid. Schema 4, like Schema 3, does not require changes when new datasets appear. Schema 4 also solves the problem of inserting "illegal" cells—cid is a foreign key to the Cell relation. To reduce redundancy, we use a surrogate key (an integer) for the cell definitions. In the Cell relation, we also add a grid_id attribute so that multiple grids can be represented. Like Schema 3, however, Schema 4 will generate a very large CellData table. Further, queries will commonly align multiple variables on the same grid for processing; this task requires one less self-join than the number of variables aligned.

For another challenge, consider a product grid such as our running "wetgrid" example. The 3-D cells must be stored explicitly, or they must at least hold foreign key values to their source cells. For example, a prism is constructed as the product of a triangle and an edge. In order to associate a value with the prism, the prism itself, or the triangle and the edge, or identifiers for the triangle and the edge must be stored explicitly. In the source representation, these identifiers were encoded implicitly in the ordering of the cell sequences. We will return to this issue experimentally in Chapter 6.

These somewhat unpleasant choices for modeling grids in relational databases suggest that sets of tuples with fixed schemas are not the correct model for scientific data.

3.2.7 A Data Structure for Visualization

The data structures discussed so far were designed for general applications. Visualization libraries also require grid data structures, but their design reflects the requirements of visualization algorithms. For example, the manipulation of several attributes simultaneously is awkward; VTK uses a distinguished attribute labeled the "SCALARS" and another (composite) attribute labeled the "VECTORS." Working with multiple attributes sometimes requires promoting them to one of these distinguished roles so the algorithms may act on them. For example, if the programmer wishes to switch the view from salinity to temperature, he or she will promote the temperature attribute to the distinguished "SCALARS" position via a method call.

These libraries in general tend to emphasize manipulation of a single dataset at a time, and therefore provide little support for reasoning about the relationships between datasets. We have argued that recognizing and exploiting relationships between datasets is a good source of optimization opportunities. The focus on specific algorithms (isosurfaces, streamlines, cutting planes) instead of higher-level algebraic operations can burden the programmer, 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 (VTK) [SML96], to extract a subset of a grid (essentially a `restrict` operator), 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 regular, rectilinear grids and computes a subgrid using a range of indices for each dimension, as opposed to geometric information or other data. The operation `vtkExtractGeometry` works on a wider range of datasets, but defines regions in terms of geometric functions rather than topological connectivity. A more efficient version, `vtkExtractPolyDataGeometry`, is available for polygonal data. Another operator, `vtkThreshold` filters grids based on non-geometric attributes.

In VTK, the physical concerns of representations and algorithms are intermingled with semantic concerns such as which data are used to filter the grid. All of the operations above can be implemented using the restrict operator, possibly with the regrid 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 called OpenDX [IBM93], we found that the simple concepts we used to describe our data products in English often did not have counterparts in these libraries. Below we list some specific concepts we found weak or missing.

- Irregular Grids. Manipulating regular, rectilinear grids is easier than manipulating irregular grids consisting of arbitrary cells. The CORIE system uses both kinds of grids.
- Extruded grids. Some grids exhibit repeating patterns but are not rectilinear. Many of these grids can be expressed concisely as the cross product of two other grids.
- Shared grids. Two datasets defined over related grids should permit efficient and convenient merging into one dataset.
- Combinatorial algorithms that do not depend on a specific geometric embedding. Berti observes that combinatorial algorithms for grid manipulation are superior to geometric algorithms [Ber00].
- Regrid. Both libraries we reviewed implement specific cases of regridding, but do not provide or acknowledge the general abstraction.
- Time. We found it useful to reason about time without distinguishing it from the other dimensions. The time dimension is either not supported or poorly

integrated by the systems we investigated.

3.3 IMPLEMENTING AND TESTING GRIDFIELD STRUCTURES

For grids, we chose to implement and test the Indexed Cell Set (ICS) representation, the NGMap representation, and a novel form of the Incidence Graph representation that we refer to as the Targeted Incidence Graph, or TIGraph. ICS matches the storage formats found in practice, and is also the basis for the in-memory data structures used in the visualization tools VTK and OpenDX. An NGMap efficiently and uniformly supports many types of incidence and adjacency queries, and can model certain types of non-manifold grids. The Targeted Incidence Graph is a generalization of ICS that can be tailored to support a specific gridfield expression with a minimum of overhead. The other representations, especially edge-based representations, were effective only for low-dimensional grids with specific properties and were therefore inappropriate for our purposes.

Before describing the grid representations, we describe how the data bound to the grids are represented. This mechanism is common to all grid representations we consider.

3.3.1 Bound Attributes

The data bound to a grid can be represented as a sequence of tuples (ordered arbitrarily), or a tuple of sequences. We choose the latter representation for several reasons.

First, attributes are stored as contiguous arrays as they are generated by the CORIE simulation programs, and contiguous arrays are expected by client-side visualization libraries [IBM93, SML96] and stored in standard file formats (cf. netCDF [JS92]). A tuple-based storage representation would require preprocessing prior to manipulation, and postprocessing prior to delivery of results.

Second, tuple-based representations, such as those found in relational systems, are designed to support transaction-processing workloads with many insertions, updates, and deletions of individual tuples. The grid structure that organizes the logical tuples is not typically updated in this manner. Individual cells are not added to or removed from grids, but entire attributes are. For example, animated data products require the re-execution of gridfield recipes over several timesteps. On each iteration, we can efficiently read in a block of contiguous data and overwrite the previous values, streaming the new timestep directly into the rendering system without expensive copying into a tuple format.

Third, prefetching can mitigate one of the common complaints about column-oriented processing. The complaint is this: If a restrict operator must evaluate a complex predicate involving several attributes, a page of each attribute must be read into memory, and a line of each page must be read into the cache. If the data were organized as tuples, a single cache block would hold all the attributes for a single tuple (assuming that the block size is larger than the tuple size). However, prefetching support built into the disk controller (for the I/O system) and built into the cache hardware and compiler (for main memory) is well suited to this multiple sequential access pattern.

Fourth, the model of the overall data repository is one of "few grids, many attributes," meaning that there are many possible datasets that conform to the shape prescribed by a particular grid. For example, each timestep of a CORIE simulation conforms to the same 3-D grid. Each timestep can therefore be considered a separate attribute. The "attribute" holding data for timestep 5 and the "attribute" holding data at timestep 10 should not necessarily be zipped together and stored as a sequence of pairs simply because we encounter a recipe that requires simultaneous access to both. Also, new attributes conforming to old grids are generated continuously: Should we continuously reorganize the prior data to maintain a tuple-based format? Instead, we provide the ability to retrieve these attributes from the repository on demand using the bind operator.

In the logical model, data is associated with cells via a function mapping cells to tuples. In the physical representation, attributes are represented as arrays, and cell ids are represented by positions in these arrays. A *namespace*, then, is a function from logical cell ids to physical positions in the bound data.

Imagine we have a gridfield $\mathbf{G}$ from which we want to derive a subgrid $\mathbf{G}'$ using the restrict operator or by using the merge operator and another grid. The gridfield $\mathbf{G}$ is equipped with a namespace namespace($\mathbf{G}$) that maps each cell id in grid($\mathbf{G}$) to a position in the attributes of $\mathbf{G}$. How should we represent the new subgrid $\mathbf{G}'$? Two choices seem natural: an *independent* representation and a *dependent* representation. An independent representation requires that we derive a new grid G', derive a new set of attributes D', and derive a new namespace to relate the cells to attribute values. A dependent representation derives a new grid G', but reuses the attributes and namespace of the original gridfield. In the former case, the operator is more expensive to evaluate due to copying, but the result will be smaller in memory and the supergrid's memory may be reclaimed. In the latter case, the operator is less expensive (since only the cells need to be copied—not the bound data), but we must keep track of the dependency between the two grids and be careful not to free memory in use.

We will make use of the following physical operators involving namespaces in the algorithms that follow.

RangeRestrict

The rangerestrict operator filters a set of attributes (the range) by copying the data associated with a particular subgrid (the domain). Each attribute is logically a function from a domain (k-cells in a grid) to a range (a tuple type). Given a subset of the domain, this operator identifies and copies the associated restricted range at the physical level. The rangerestrict operator is used in the implementation of the logical restrict and merge operators.

NewNamespace

The newnamespace operator maps all integer cell ids to a contiguous range of integers beginning with 0. In a sense, the ids are "swizzled" such that they can be used as direct pointers. Any I_{ij} links may hold references to the old ids that must be updated; this operator takes care of this task as well. This transformation allows the cells to represented implicitly thenceforth.

3.3.2 Cells and Grids

Having discussed how the attributes are represented, we now discuss the details of how cells, topology, and grids are represented for our three chosen representations.

Indexed Cell Set

The data structure we use in memory for our ICS representation appears in Figure 3.9. The record at the top holds a sequence of pairs of the form (G_i, f_i), where G_i is an array of cells and f_i is a sequence of *attributes*. An attribute is simply a named array of primitive numeric values. In Figure 3.9, the sequence $a0_i, \ldots, an_i$ represents the names of the attributes, and the arrows represent pointers to arrays of actual data. Cells of dimension k are aligned positionally with the attribute arrays.

Individual cells are stored as a sequence of nodes. A hash index (H in Figure 3.9) maps a cell's definition (a sequence of nodes) to its ordinal position. The cells of a particular dimension are stored contiguously. This index, along with a cell's dimension, 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 for that particular rank. The hash function used is based on the dimension of the cell and the first

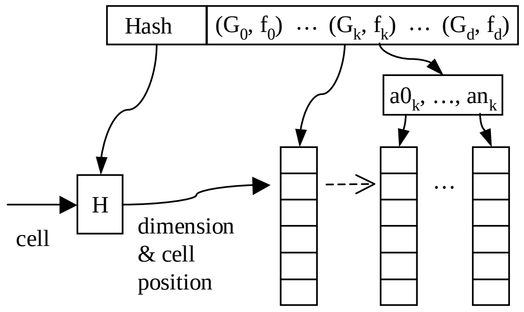

Hash
$(G_0, f_0) \cdots (G_k, f_k) \cdots (G_d, f_d)$
$a0_k, \ldots, an_k$
H
cell
dimension & cell position
...

Figure 3.9: Internal representation of a gridfield for the ICS representation.

node in its definition, exploiting the fact that a node is seldom incident to more than four or five cells. Our tests show that this hash function generates very few collisions while offering fast evaluation time.

Tailorable Incidence Graph

Our incidence graph representation, referred to as TIGraph, stores a set of cell identifiers for each dimension $0 \leq i \leq d$. Each cell identifier is used as an offset into the bound attributes of the full gridfield, as with the ICS representation. In this way, we can represent both independent gridfields with their own namespace or dependent grids that use the namespace of a supergrid. The cell identifiers at dimension k in an independent grid are a contiguous range of integers from 0 to $|G_k|$, and can therefore be represented implicitly.

To represent topology, we store a set of hash indexes I_{ij}, called *links*, mapping each cell of dimension i to a sequence of cells of dimension j. Since we work with sequences instead of sets, we can preserve the ordering information of the ICS representation.

The key feature of this representation is that not all cell sets need be populated, and not all links need exist; the representation is tailored to a particular recipe. At runtime, we access only the cells and links required by the recipe. For example, interpolation onto the centroid of a cell requires the nodes but not the edges.

The ICS representation is a special case of the TIGraph representation, with a set of d-cells, a set of 0-cells, and a single link I_{d0}.

To analyze a recipe, we consider 1) the explicit cell sets and links that are already part of the grid, and 2) the links we can derive at runtime. For example, given a link I_{20}, we can build the inverted link I_{02}. Given a TIGraph representation with an explicit I_{20} link, we can use the ordering information to derive I_{21}.

Algorithm 6 implements the restrict operator and illustrates the use of this representation. This version creates a new independent TIGraph with its own namespace for the result.

Algorithms implementing restrict have four phases: 1) identify the cells that satisfy the predicate, 2) handle the higher-dimensional cells, 3) handle the lower-dimensional cells, and 4) update any references to missing cells (i.e., use the new-namespace operator).

Phase 1) is written here as a simple linear traversal of the cells. A predicate is represented as a string involving comparison operators, boolean operators, and arithmetic expressions. This string is parsed and compiled to bytecodes for efficient runtime evaluation using a specialized library [Nie]. An index on attributes involved in the predicate, if available, could also be used in this phase.

For phase 2), Algorithm 6 uses the *backward* links I_{ki} for $i < k \leq \mathsf{dim}(G)$ to join the cells in G_k with the cells in R_i. Variations on this algorithm could use the forward links if they exist to navigate from the cells in R_i to the cells in G_k. The flexibility of this representation is that different physical plans can be chosen based on the data that is available.

We cannot use the accrete operator to implement phase 2), because the semantics of the restrict specify that cells are passed through to the result only if *all* of their incident i-cells satisfy the predicate. The accrete operator does not implement this universal quantification, so we implement it “manually” here.

Algorithm 6 RestrictTIGraph

Input: A gridfield $\mathbf{G} = (G, g_0, g_1, \ldots, g_n)$, where G is implemented as an TIGraph, and a dimension i, a predicate $p : [\mathsf{sch}_i(\mathbf{G})] \rightarrow \mathsf{Bool}$.

Output: A gridfield restricted to those cells satisfying the predicate, along with their neighborhoods defined by the semantics of restrict.

```
$G \leftarrow \mathsf{grid}(\mathbf{G})$
$R \leftarrow$ empty Grid
//add the cells that satisfy the predicate.
for each $c \in G_i$ do
  if $p(f_i[c])$ then
    add $c$ to $R_i$
  end if
end for
//handle the higher dimensional cells.
for each $i < k \leq \mathsf{dim}(()G)$ do
  for each $c \in G_k$ do
    if $I_{ki}(c) \subset R_i$ then
      add $c$ to $R_k$
      for each $d \in I_{ki}(c)$ do
        assert $c \prec d$ in $R$
      end for
    end if
  end for
end for
//handle the lower dimensional cells.
for each $0 \leq k < i$ do
  for each $c \in G_k$ do
    if $I_{ki}(c) \cap R_i$ is not empty then
      add $c$ to $R_k$
      for each $d \in I_{ki}(c)$ do
        assert $c \prec d$ in $R$
      end for
    end if
  end for
end for
$\mathbf{H} \leftarrow \mathsf{rangerestrict}(R, g_0, g_1, \ldots, g_n)$
return $\mathsf{newnamespace}(\mathbf{H})$
```

Phase 3) performs a similar task to phase 2), except that the lower dimensional cells are existentially quantified.

Phase 4) uses the newnamespace operator to assign new identifiers to newly-created s-cell and update references to the old ones.

Generalized Combinatorial Map

In our NGMap representation, a dart is an instance of the class `Dart`, and each function α_i is represented by a pointer to another `Dart`. These pointers are stored in an array whose size is determined by a compile-time template parameter, the dimension of the grid.

In an NGMap, each k-cell is represented implicitly as an orbit of darts. To iterate over all cells of dimension k, we traverse the darts and return an iterator for each unique instance of the orbit $< \alpha_0, \ldots, \alpha_{k-1}, \alpha_{k+1}, \ldots, \alpha_d >$. Each visited dart is marked to ensure termination and prevent duplicates.

In C++, we implement these orbits in an efficient and generic way using the iterator idiom and template classes. The involutions to be traversed in the orbit are supplied at compile time, allowing a specialized implementation to be emitted by the compiler. The general algorithm for traversing an orbit involves a breadth-first search through the graph represented by the involutions. The frontier of the breadth-first search must be maintained with a stack and visited darts must be marked. However, the orbit $< \alpha_i, \alpha_{i+1} >$ can be implemented as a loop for d-dimensional manifold grid (where $d \geq 2$), alternatively navigating α_i and α_{i+1}. For such a grid, each dart is guaranteed to be visited only once.

Although orbits provide a correct and complete representation of k-cells, retrieving all orbits for k-cells of fixed dimension k requires maintaining state. The algorithm consists of iterating over the darts, returning orbits as they are encountered. Each dart involved in an orbit is marked as it is encountered; marked darts are skipped when looking for the next orbit. The algorithm to iterate over cells of dimension k is

```
for each dart d:
   if unmarked(d):
      cell = Orbit(k,d)
      for e in cell:
         mark(e)
      yield cell
```

where `Orbit(k,d)` returns an iterator over the darts in the orbit

$$< \alpha_0, \alpha_1, \ldots, \alpha_{k-1}, \alpha_{k+1}, \ldots, \alpha_{\mathsf{dim}(G)} >$$

Set-like operations on grids require 1) cell identity and 2) efficient random access to cells. A pure NGMap implementation represents a cell only implicitly as an orbit. Darts cannot provide cell identity across multiple grids. A single dart cannot be reused in multiple grids, since the pointers implementing the α involutions would need to be different. The involutions for a cell c present in two grids may be different in two ways: $\alpha_i(c)$ may be a fixpoint in one grid but not the other, or it may point to a different dart altogether. Compatibility (see Chapter 2) between grids prevents the latter case, but not the former. The former case occurs when the dart c is on the boundary of a subgrid but in the interior of a supergrid.

To represent cells, then, we extend the dart implementation with an array of cell ids for each dimension. This extension allows identity to be traced across multiple grids. However, random access to cells by id still requires a linear traversal of the darts.

To solve this problem, we extend the NGMap implementation with an index mapping each cell id to a *key dart* for each dimension requiring iteration. This index directly supports iteration over cells. To then access topology information, operators can follow the pointers of the key dart. The cell id also serves as an offset into each attribute array at a particular dimension. For example, a node with id 5 is associated with the fifth value in each attribute bound at rank 0.

We have described three structures in our extended NGMap implementation:

1. The *NGMap* itself representing the grid as a set of darts. Each dart is equipped with two arrays of length $\mathsf{dim}(G)$: an array of cell ids for accessing data and an array of pointers to darts for accessing topology.
2. The *field* representing the bound data as a set of attribute arrays for each dimension. Each of these arrays is indexed by cell ids.
3. The *key index* mapping each cell id to a key dart.

With these three structures, operators can access cell ids from darts, data from cell ids, and darts from cell ids, respectively.

Algorithm 7 implements the restrict operator over our NGMap representation and illustrates maintenance of the involution pointers. Let a j-cell that does not satisfy the predicate be referred to as a *failed* cell. The algorithm marks each dart involved in the orbit of a failed cell. Further, to implement the Exact semantics of restrict, the algorithm visits the darts involved in the orbits of neighboring k-cells, $k > j$, and marks them as well.

The orbit representing a k-cell c is a superset of the orbit representing a j-cell c' if $c' \preceq c$, for $k > j$. So in order to cull all k-cells to which a failed cell is incident, it is sufficient to cull the darts involved in each k-cell to which the failed cell is incident.

Once the new set of darts is established and properly sewn, the associated data must be copied as well. The rangerestrict operator (line 31) performs this task while building an associative array mapping the old cell ids to the new cell ids. The newnamespace operator (line 32) uses this associative array to 1) update the old cell ids held by each dart, and 2) build the key index mapping each cell id to a key dart.

Algorithm 7 RestrictnGmap

```
Input: A gridfield $\mathbf{G} = (D, g_0, g_1, \ldots, g_n)$ (where $D$ is a set of darts), a predicate
    $p : [\mathsf{sch}_i(\mathbf{G})] \rightarrow \mathsf{Bool}$, and a dimension $i$.
Output: A gridfield restricted to those cells satisfying the predicate, along with
    their neighborhoods defined by the semantics of restrict.
  map $\leftarrow$ an associative array of type Dart $\rightarrow$ Dart
  for each $x \in D$ do
    //mark darts that are associated with a cell to be culled.
    if not $p(f_i[x])$ then
      mark($x$)
      for each $e \in \langle \alpha_0, \ldots, \alpha_{k-1}, \alpha_{k+1}, \ldots, \alpha_d \rangle(d)$ do
        for each $f \in \langle \alpha_0, \ldots, \alpha_{d-1} \rangle(e)$ do
          mark($f$)
        end for
      end for
    end if
  end for
  //copy the unmarked darts
  for each $x \in D$ do
    if $x$ is not marked then
      map[$x$] $\leftarrow$ a copy of $x$
    end if
  end for
  //sew things up correctly
  for each $x \in$ map do
    for each $0 \leq i \leq \mathsf{dim}(G)$ do
      if $\alpha_i(x)$ is in map then
        sew(map[$x$], map[$\alpha_i(x)$])
      else
        //the dart is on the boundary of the new grid
        sew(map[$x$], map[$x$])
      end if
    end for
  end for
  $R \leftarrow (\{y \mid y \in \text{map}\}, g_0, g_1 \ldots g_n)$
  $\mathbf{H} \leftarrow \mathsf{rangerestrict}(R, g_0, g_1, \ldots, g_n)$
  return $\mathsf{newnamespace}(\mathbf{H})$
```

3.4 EXPERIMENTS

We evaluated the implemented representations using the restrict operator and the neighborhood operator over two different datasets. We find that the NGMap representation does not scale as well as the ICS and the more general TIGraph representations.

The experiments are carried out on a platform designed to be identical to the cluster nodes used on the CORIE project. The computer has two 2.4GHz CPUs and 4GB of RAM connected with a 400MHz front side bus. There are two SATA disks in a RAID 0 configuration; however, I/O is not exercised in these experiments.

The machine is representative of commodity server machines found in clusters and used for a variety of applications. Gridfield algebra expressions are intended to be evaluated as queries on the server side rather than on a scientist's desktop, so this machine is representative of the platforms used in practice.

The CORIE dataset describes the estuary and surrounding ocean of the Columbia River. For this experiment we use just the horizontal gridfield H describing the surface of the water with three attributes bound to the 0-cells: x, y, $elevation$. The attribute $elevation$ gives the vertical offset of the bottom of the ocean from the mean sea level (msl) in meters.

The Quake dataset was generated by the Quake project at Carnegie Mellon University and describes the San Fernando Valley in Southern California [ABB+03]. The 3-D grid consists entirely of tetrahedra. Three attributes x, y, z bound to the 0-cells provide the geometric coordinates. Three additional attributes $swave$, $pwave$, and $density$ bound to the 3-cells are material properties of the earth in the region.

3.4.1 Restrict Operator over ICS, NGMap, and TIGraph

The first experiment evaluates the restrict operator over the x and y coordinates, restricting the grid to a bounding box of varying size. The gridfield has 29602 0-cells and 55081 2-cells, some of which are triangles and some of which are quadrilaterals.

The TIGraph representation is tailored to represent only the I_{20} link. Algorithm 6 uses only the forward links I_{ij} for $i > j$ and therefore requires no other I indexes to be built. In this sense, the ICS and TIGraph implementations behave very similarly.

Figure 3.10 shows the relationship between running time of the recipe and the selectivity of the bounding box, defined as the number of 0-cells in the result. The time to read in the data from disk is not included. Since the data is represented on disk in an ICS format, the preprocessing required to construct an NGMap representation would bias the results. If the NGMap representation is more efficient once constructed, we would consider storing gridfields in a compatible format on disk. Each representation scales linearly with respect to the selectivity of the bounding box. All representations evaluate the operator over this small dataset at near-interactive speeds. However, the NGMap representation scales with a higher constant than either the ICS or TIGraph representations. The performance of the NGMap representation is worse for two reasons: First, since topology is represented completely with an NGMap, more information needs to be copied to the result during processing. Second, the algorithm requires two passes over the darts: one to identify the darts that belong in the result and a second to sew them up correctly.

The number of darts that must be stored and manipulated scales super-linearly with dimension. We therefore expect that the performance ratio of NGMap to TIGraph (and ICS) to get worse in higher dimensions. For two other experiments involving the Quake dataset, we see that this is indeed the case. In Figure 3.11, we see the ratio of runtime using NGMap to runtime using TIGraph for three

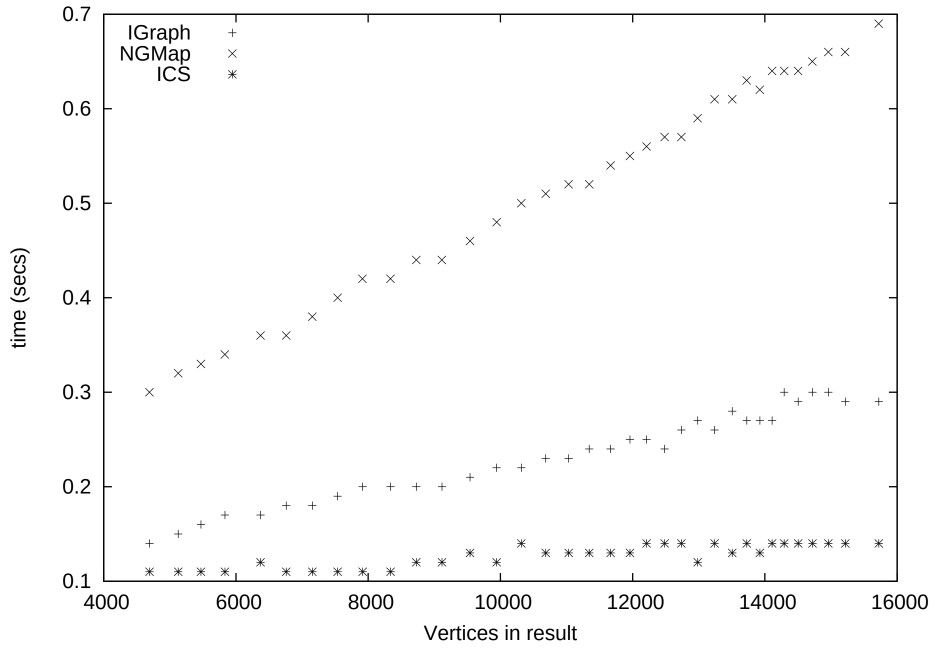


Figure 3.10: Performance of the restrict operator using three different representations. The NGMap representation implements all incidence relationships and requires more data movement. The TIGraph and ICS representations admit algorithms that only need a single incidence relationship: I_{20}.

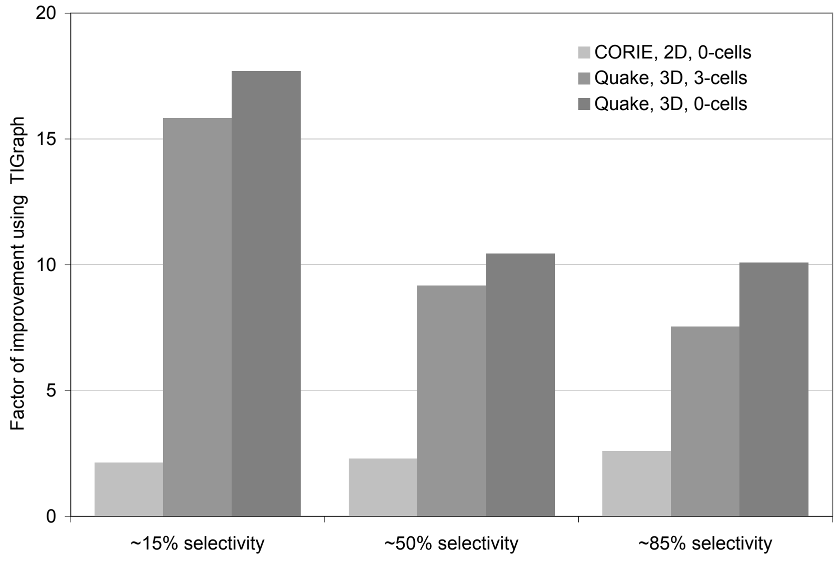


Figure 3.11: Performance improvement using TIGraph for three different situations at three different selectivities. The performance penalty using NGMap increases with dimension, especially at low selectivities.

---

experiments at three different selectivities. The first is the same experiment as in Figure 3.10, using the CORIE dataset. The second is a similar bounding box experiment using the Quake dataset. The third also uses the Quake dataset but applies the predicate to the 3-cells instead of the 0-cells. The results show that in three dimensions, the performance improvement using TIGraph is considerable. Also, at low selectivities, there is less overhead from inserting data into the hash indexes of the TIGraph representation, and the performance improvement over NGMap is even better.

Since the ICS representation models incidence as a subset relationship between shared nodes, it cannot model completely general grids. In this experiment, we use the ICS representation extended with links as necessary. This extension effectively unifies the ICS representation and the TIGraph representation, so we omit the ICS results in Figure 3.11.

3.4.2 Neighborhood Operator over NGMap and TIGraph

The NGMap representation was designed for quick access to incidence information, but the restrict operator involves copying portions of the cell and attribute data to a new structure. The neighborhood operator is intuitively better suited for the NGMap representation, since it navigates the topology but does not require any modifications or copying. However, we find that the NGMap is limited when working in higher dimensions.

Recall that the neighborhood operator uses a stencil to identify neighboring cells, then aggregates the values associated with those cells. We test the call $\mathsf{neighborhood}(303, \mathsf{avg}(swave))$, which indicates that for each 3-cell, we compute the average value of the *swave* attribute of all adjacent 3-cells. The 3-cells we want to access share a node with the target cell; we could use the stencil 323 to access adjacent cells that share a face with the target cell.

The efficiency of NGMap when traversing stencils depends critically on the *orbit* used to implement the traversal. For example, consider the orbit $<\alpha_1\alpha_2>$, which means "visit every dart reachable through successive applications of the function $\alpha_1 \circ \alpha_2$." For a 2-D grid, this orbit traverses the 1-cells and 2-cells around a 0-cell. When the grid is manifold, the number of 2-cells visited is the same as the number of 1-cells visited, and, most importantly, the same as the number of darts visited.

For example, consider the grid in Figure 3.12. The bold lines and large black circles represent 1-cells and 0-cells in the original grid, respectively. The large triangles bounded by heavy lines represent 2-cells in the original grid. The light lines and small gray circles represent 1-cells and 0-cells in the barycentric subdivision of the original grid. Recall that in the barycentric subdivision, each 2-cell corresponds to a dart in the NGMap. Each large triangle contains 6 small triangles corresponding to the 6 darts needed to represent its topology. Beginning at the white dart labeled "s," we can visit the other white darts in a counter-clockwise order through successive applications of $\alpha_1\alpha_2$ without visiting the same dart twice.

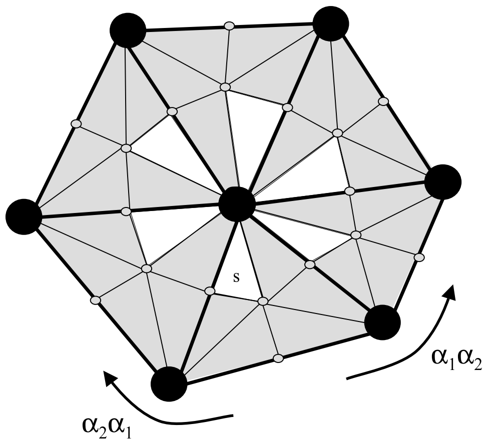


Figure 3.12: An efficient traversal of cells neighboring the center node. The traversal relies on the property that each dart visited corresponds to a unique cell. The bold lines and large black circles represent 1-cells and 0-cells in the original grid, respectively. The large triangles bounded by heavy lines represent 2-cells in the original grid. The light lines and small gray circles represent 1-cells and 0-cells in the barycentric subdivision of the original grid (see Section 3.2.4). Each large triangle contains 6 small triangles corresponding to the 6 darts needed to represent its topology. Beginning at the white dart labeled “s,” we can visit the other white darts in a counter-clockwise order through successive applications of $\alpha_1\alpha_2$. For many traversals, especially in 3-D, this efficient iteration with no repeat visits cannot be performed.

Similarly, we could traverse the white darts in clockwise order through successive applications of $\alpha_2\alpha_1$ without repetition. Each dart we visit corresponds to a unique 1-cell and a unique 2-cell in the original grid to which the middle node is incident. Further, from each white dart d we can access an adjacent node using α_0. This feature allows us to visit the cells required by the stencil 010 by dereferencing 3 pointers per dart. For most stencils, however, especially those in three dimensions, we cannot guarantee that any particular orbit will visit each required cell once and once only. We therefore must implement a breadth first search of neighboring cells, collecting the visited cells into a set. The performance difference

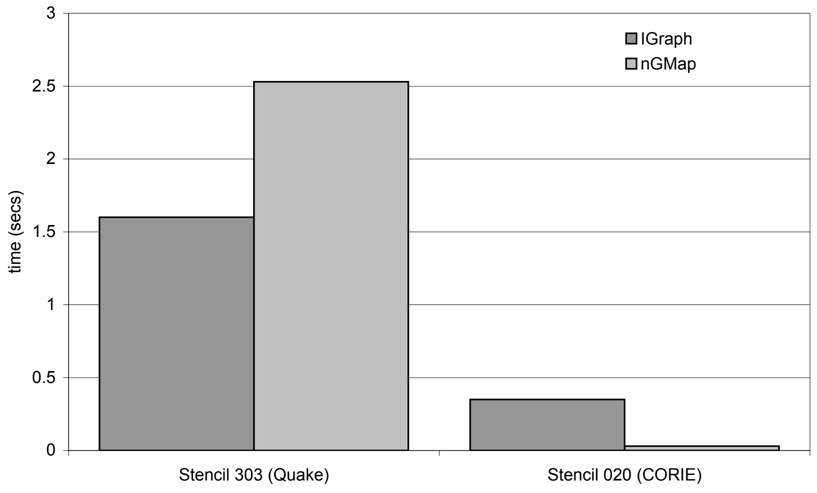


Figure 3.13: NGMap outperforms TIGraph when an efficient orbit can be found, as in the 2-D case. Most access patterns do not admit an efficient orbit, however.

---

between the traversal and search orbit algorithms is dramatic, as we see in Figure 3.13. For the 2-D stencil 010, NGMap significantly outperforms TIGraph. For the 3-D stencil 303 however, no efficient traversal can be found, and we must use a breadth-first search to collect the adjacent cells.

In the literature, most examples and results applicable to NGMap involve 2-D grids. In three dimensions the elegance and performance of using NGmaps is questionable.

3.5 CONCLUSIONS

The work in this chapter resulted in a C++ implementation of the algebra making heavy use of the Standard Template Library. Python interfaces to the C++ objects were written to allow recipe-level manipulation and optimization to be performed more conveniently. The code base consists of about 40000 lines of combined C++ and Python.

In this chapter, we investigated physical data structures for representing gridfields. We refined the requirements of Chapter 2 and produced a simple API that all grid data structures should implement. We discussed some subtle issues facing the designer, including binding data to the cells at multiple ranks and representing the boundary of grids.

Next, we reviewed proposals for grid data structures found in the literature. Most of these were designed to target specific kinds of grids, such as 2-D or simplex grids; such restrictions preclude their use in most applications we have encountered.

Three classes of data structures passed the first round of evaluation, surviving to be implemented for further analysis: the the indexed cell set (ICS), the tailorable incidence graph (TIGraph), and the n-dimensional generalized combinatorial map (NGMap). A TIGraph allows arbitrary subsets of well-supported grids; we say it can be tailored for specific purpose. For example, if a recipe does not involve the 1-cells, they need not be included in a TIGraph. Correctness guarantees offered by more restrictive data structures cannot be proved for a TIGraph. However, the grid-level operations of the algebra prevent the construction of pathological grids. Specifically, every result grid can be related to the source grids through Proposition 5.

The ICS representation is actually a specialization of the TIGraph, where the only incidence relations supported are of the form I_{i0} or I_{0j}. Some grids allow a surrogate definition of incidence based on the nodes: $x \preceq y \Leftrightarrow \mathsf{nodes}(x) \subseteq \mathsf{nodes}(y)$. If this property does not hold, then a recipe that accesses the missing incidence information cannot be evaluated without extending the ICS representation to the TIGraph representation.

The NGMap data structure offers an elegant solution for managing topology information, but suffers scalability problems with respect to dimension.

After describing our implementations, we tested the performance of individual operator algorithms on each of the three data structures. We find that the ability to tailor the representation to a specific recipe offers significant savings.

In conclusion: We chose the ICS representation as a reference implementation of the gridfield algebra. Its limitations are not encountered in the applications we considered, and it can be readily extended to the TIGraph representation where necessary.

Chapter 4

RETROFITTING GRIDFIELDS ONTO EXISTING DATA

The data we wish to manipulate with the gridfield model are usually stored as a collection of packed binary files arranged in an intricate directory structure, rather than in a database. The decomposition of data into files is designed to simplify or optimize writing rather than reading. The natural grouping of data for analysis tasks may be rather different than the one chosen for convenient writing. For example, if each simulated variable is stored as a separate array in memory, then each might become a separate file on disk. In contrast, an analysis task to compare two simulated variables would prefer them to be in the same file. A parallel simulation will decompose data spatially; each portion of the decomposed grid will become a separate file. Analysis tasks must accommodate this file boundary.

In this chapter, we show how to expose these native data sources as gridfields without preprocessing, bulk-loading, or other prohibitively expensive operations. We describe native directory structures and file content using a simple schema language based on nested, variable-length arrays. Equipped with a schema, logical datasets may be accessed by name rather than by location, providing physical data independence. Further, no custom access methods need to be written by application programmers; files are both described and queried independently and declaratively.

We explore a variety of techniques to improve performance. Since each file may be large and one logical dataset may span multiple files, only the relevant portions of each file are loaded into memory. We use the file's schema to deduce an efficient plan for striding through large files and reading these relevant portions.

We test these schema-level techniques by using optimizations such as code generation, memory-mapped files, and partial evaluation, and find that generic optimized access methods can be competitive with hand-written access methods.

4.1 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) [ND05], 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." The data involved can be logically described in terms of gridfields, but there is no language to describe the physical structures: directories, files, binary file content. In the interest of accelerating this kind of federation effort, we advocate inserting a layer of indirection between gridfield applications and the native source data as an alternative to mass conversion of existing data.

The salient feature of the gridfield model is that the grid structure of the datasets is explicit. Traditionally, gridded datasets are stored and manipulated as a collection of arrays; the logical grid structure appeared only in the code itself. By making this hidden grid structure explicit, 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 current 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 to allow access to these data via gridfields 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 RDBMS 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 and directory structure we encountered. To generate a gridfield, routines to iterate over multiple files are layered on routines to interpret each file's format. The results are used to assemble gridfield objects suitable for manipulation with the gridfield algebra. Creation of these routines became repetitive enough to motivate a more general abstraction [HM04c]. In this chapter, 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 or direct image generation.

The model we describe in this chapter simplifies the task of maintaining "readers" for creating gridfields from source data. Data creators describe their file formats and directory structures declaratively, while application writers can access the parts of these files they need by name rather than by location. Given a description of the filesystem and a set of requested names, the software will generate appropriate access methods.

We again refer to the CORIE Environmental Observation and Forecasting System as a source of examples. In a particular run of a CORIE simulation, 3-D spatial datasets are produced at regular intervals of simulated time, for each of several physical variables. Each of these variables are stored in a separate file. 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—calibration runs as opposed to final results. For example, the run directory names in the middle column of Figure 4.3(a) in Section 4.4.1 contain the week and the year, while the names of the checkpoint files in the right-most column contain the day of the week. Every application accessing these data must understand the semantics of directory and file 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, directory and file naming, 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 be inconvenient 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 with 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 file path.

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 4.3(a) has a separate file for each day of the week (per variable). In this case, the day of the week and the week number are not stored within the file, and are therefore inaccessible to tools that reason only about a file's content [MC03, WB03]. This representation reflects the manner in which the data was generated: A checkpoint file was recorded by the simulation model 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 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 is of the form

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

The boundary between directory-level data and file content data is not apparent from 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, a user writes a *schema file* 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 pseudo-code above with the following *extraction query*:

```
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. Application writers are provided two forms of physical data independence: 1) they can access data by name rather than by location, and 2) they can ignore file boundaries.

The layers of abstraction required to provide these forms of physical data independence can degrade performance, especially when accessing multiple gigabyte-scale files with a single expression. The naïve evaluation strategy resembles the pseudo-code above: nested iteration over every value in the file. Values included in the query are copied and all others are skipped. The techniques we use to improve performance fall in two categories: 1) analytical techniques that reduce the number of load and copy operations, and 2) operational techniques that can reduce the overhead from various sources. We consider the following techniques to improve performance over the performance obtained with this basic implementation

Analytic: Static Optimization By inspecting the schema, we identify arrays with known sizes that can be skipped or returned with a single read call.

Analytic: Dynamic Optimization Re-analyze the schema after reading some preliminary information from the source file.

Analytic: Schema Transformation Simplify the schema with respect to a particular query so that the same data can be accessed with fewer operations.

Operational: Code Generation Generate and compile programs that read the data for a particular query to reduce the work to be done at runtime.

Operational: Partial Evaluation Assume that many files share similar structure and use the results of a dynamic optimization on one file to improve performance for reading other files.

Operational: Memory-mapped files Reduce the number of system calls executed by allowing the operating system to automatically load pages of the file as they are accessed.

In summary, this chapter presents

- A data model for describing binary file formats.
- A complementary data model for describing data embedded in directory structures and file names. We refer to these two mini-models jointly as the *Native Data Model*.
- Optimization techniques to efficiently evaluate queries over native content.
- Experimental evidence based on real data showing that suitably optimized generic access methods can perform competitively with hand-coded access methods.

In Sections 4.2 and 4.3, we present related work and introduce a running example, respectively. In Section 4.4, we give examples of schema files for accessing binary file content as well as data encoded in the directory structure. In Sections 4.5 and 4.6, we focus on evaluation techniques and experimental results, respectively, for accessing binary content. We end by discussing conclusions and possible extensions.

4.2 RELATED WORK

The Datascript language [Bac02] is used to describe binary file formats similar to ours. Features include discriminated union types, constraints, arbitrary offsets, and variable-length arrays. However, types spanning multiple files are not considered, and there is no consideration for efficient I/O strategies for large files. Datascript also aims to faithfully recreate the on-disk types in memory; the two data models, both based on C, are identical. We use two different data models: a simple array-based data model in memory and a more complex data model for describing the realities of file formats.

The PADS system [FG05] is designed for "processing ad hoc data." Like Datascript, PADS maps on-disk types into C-style structs, unions, and arrays. Given a file description, PADS generates parsers for each type. PADS parsers are designed to work not only on files but on streams of records, such as those found in telecommunications applications. PADS features include sophisticated error-handling for non-conforming data, recursive types, union types, support for both ASCII and binary data, and accumulators and histogram operations for collecting summaries of data streams. Our work differs from PADS in the access patterns supported: Our on-disk types are large enough to span disk blocks and files; applications need the ability to extract only the relevant portion of these large types with an efficient I/O strategy. The PADS system is designed for processing smaller data types, but at high speeds. Applications using PADS expect to touch every record, but the complicated code to read, interpret, and validate the record is handled by PADS.

PADX [FFGM06] couples an XQuery engine to the PADS system to allow XQuery expressions to be evaluated over non-XML data. The access patterns considered in PADX are *bulk-load*, in which all data is read into memory (and converted to XML) prior to query evaluation, *on-demand*, in which PADS data is loaded when the the XQuery engine first references the relevant XML node, and *paged*, in which files are assumed to have the form HR^*F; a header followed by some number of small records followed by a footer. The file formats we work with contain large arrays with variable lengths; the appropriate unit of iteration cannot be deduced statically. We will return to this issue when we discuss query evaluation.

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 [SHT$^+$77] 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 [Bat85]. Our work similarly provides a description of file structures in terms of arrays.

The Binary Format Description Language (BFD) [MC03] 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 [WB03] library can be used to access binary data 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 adds support for nested arrays, but only if their length is fixed.

The External Data Representation standard (XDR) [Sri95] is a data format language focused on machine-level number-representation issues. Variable-length arrays in XDR must have homogeneous elements (i.e., their elements are not variable-length), and their lengths must be encoded directly prior to the first element. Further, XDR obviously does not describe directory structures, preventing access to datasets that span multiple files.

Platforms for scientific query and analysis include AQSIM [LCA+03] and the Active Data Repository [KaC+01]. Both of these systems require preprocessing of data repositories in order to construct indices, compute statistics, or to bulk load data into a managed environment. Other systems such as Chimera [FVWZ02] and Godiva [MWN+04] supervise the execution of data access programs, but rely on users to write them in the first place. Our techniques are used to derive an efficient reader given only a set of queries and a declarative description of the data.

One technique we use to optimize our access methods is inspired by partial evaluation. Partial evaluation has been studied extensively for a variety of programming languages [And94, Bon93, BW93, Thi98, BGZ94], sometimes specifically in the context of scientific computing [BW90, BS94b]. Partial evaluation exploits the fact that some arguments to a function become available sooner than others. Instead of waiting for all arguments to become available, the function may be partially evaluated with available arguments to generate a new function. The new function is hopefully more efficient at runtime. Similarly, our query engine consumes a query, a schema, and a portion of a file instance and produces a more efficient function for extracting the query result. This generated function can be applied to other files with a similar structure to amortize the cost of its creation.

The generalization of partial evaluation to multi-stage programming [TS00] is relevant here as well—an access method may be specialized multiple times as increasingly restrictive assumptions are made about the files it is applied to. For example, a schema for a record-based file format might have a header, some number of variable-length records, and a footer. A specialized schema for files with similar headers may be derived such that the header can be skipped during reads. A further specialization for fixed length records may be subsequently derived, allowing efficient random access to any record.

Our schema files assign a type to large binary files. Since the length of the arrays depend on values in the file itself, the files are dependently typed. More specifically, they are instances of *dependent sum types*, where $\sum xA.B$ is the type of pairs with the first element of type A and the second element of type B, and B depends on the value x. Using languages such as Cayenne [Aug98] and Epigram [MM04], or provers such as Agda [Coq06], one can prove properties about the results of dependently-typed computations at compile-time; a frequent application of this facility is to bound the size of the result of array computations.

4.3 RUNNING EXAMPLE

The CORIE system uses several different forms of grid to model coastal and ocean waters. One early form of grid used to describe the Columbia River Estuary and surrounding waters was constructed as follows: A 2-dimensional unstructured grid models the surface of the water around the mouth of the Columbia River Estuary (Figure 4.1(a)). This horizontal grid is repeated at each depth in a 1-dimensional structured grid, creating a 3-dimensional extruded grid. The sloping bathymetry of the river causes many of the grid cells of this 3-dimensional grid to be positioned underground. Figure 4.1(b) illustrates the situation. Each dotted line represents a copy of the horizontal surface grid repeated at a particular depth and oriented perpendicularly to the plane of the page. The shaded region represents the bottom of the river. The horizontal levels towards the bottom contain fewer valid "wet"

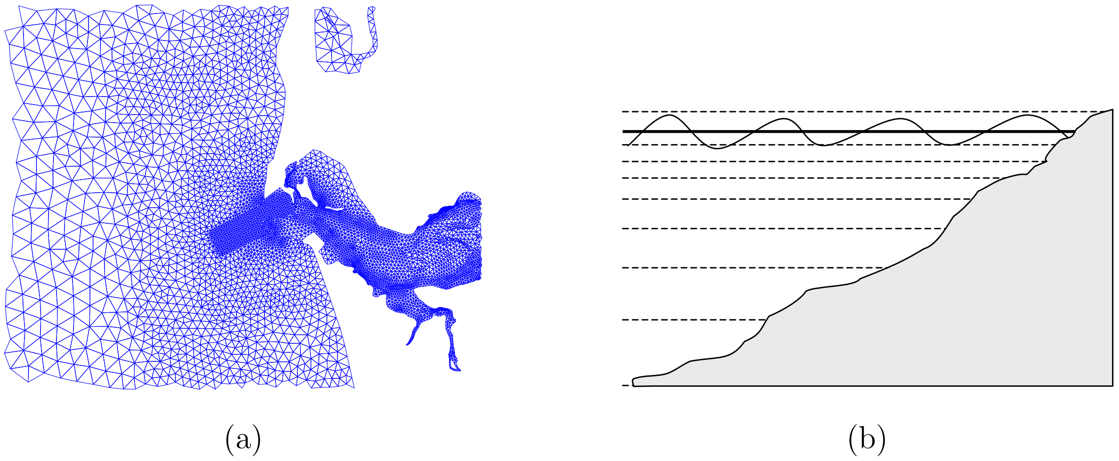


Figure 4.1: (a) The horizontal unstructured CORIE grid. (b) Illustration of the river's bathymetry. The shaded region is underground.

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:

$$G = \mathsf{bind}(salt, 0, \mathsf{restrict}(b < z, \mathsf{cross}(H, V))) \tag{4.1}$$

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 4.1(b)). The bind operator reads in an array named *salt* and attaches it as an attribute of the grid's 0-cells.

In Equation 4.1, the names *salt*, H, and V are assumed to refer to an attribute (an array), a gridfield, and another gridfield, respectively. How are these gridfields and attributes constructed from the raw data on disk? Examples of code to assemble gridfields appear in Figure 4.2. The quoted, path-like expressions in Figure 4.2 are *extraction queries* over the schema in Figure 4.7. These expressions will be

```
0:  context = “week[=01]/year[=2006]/day[=1]/”

1:  H = Gridfield()
2:  H.cells[0] = idrange(0, context + “/nodes”)
3:  H.cells[2] = “context/cells”
4:  Bind(H, 0, “x”,    context + “/nodedata/x”)
5:  Bind(H, 0, “y”,    context + “/nodedata/y”)
6:  Bind(H, 0, “bath”, context + “/nodedata/b”)

7:  V = GridField()
8:  V.cells[0] = idrange(0, context + “/levels”)
9:  Bind(V, 0, context + “/zcor”)

10: G = Restrict( “b < v”, 0, Cross( H, V ) )
11: Bind(G, 0, “salt”, context + “/timedata[0]/horizontal/depths/vector/val”)
```

Figure 4.2: Examples of gridfield assembly. These statements constitute a program for constructing gridfields from base components.

---

described in Section 4.4. Line 0 assigns an extraction query to a variable `context`, which is used as a prefix in the other extraction queries. Extraction queries used to specify an attribute (e.g., line 4) must return a one-dimensional array. Extraction queries used to specify a set of cells (e.g., line 3) may have a more complicated form corresponding to one of the representations discussed in Chapter 3. For example, a set of polygonal cells can be represented as an array of arrays, where the elements of the inner array represent cell nodes.

The 0-cells of the grid are often specified implicitly, using the function `idrange` (lines 2 and 8), which generates a contiguous range of integer cell ids given a starting id and stopping id. Cells of higher dimensions are defined as sequences of integer references to 0-cells. A triangle will have three references, a quadrilateral will have four, and so on. Attributes are also specified using extraction queries. To bind an attribute x to cells of dimension 0, we use a form of the bind operator, passing it as in line 4 of the figure.

Gridfields can also be constructed through expressions in the gridfield algebra, as in line 10 of the figure. Line 10 specifies the same gridfield as in Equation 4.1.

4.4 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.

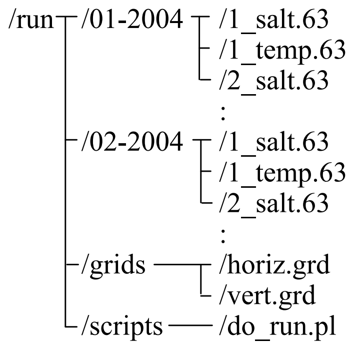
A filesystem-based data repository is described via a collection of declarations housed in a schema file. There are two types of declarations. *FileType* declarations describe relevant directory structures and allow access to data encoded within file and directory names. *BlockType* declarations describe the layout of portions of binary files. Application writers access data using *extraction queries* that abstract some details of file organization, data format, and the boundary between file name metadata and file content.

4.4.1 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` [Nat06]. The string "20030803" is evidently a date, but the other fields require some external information to parse[1]. A schema file gives data administrators a place to put this external information. Schema files provide executable documentation for one's filesystem organization conventions. Moreover, schema files allow data administrators to provide access to data unilaterally; they need not consult application writers to find out how the data is to be used.

Consider the filesystem in Figure 4.3(a). The root directory `run` contains directories with simulation results for each week of 2004 in the form `<week number>-2004`. Each week directory contains 14 files, one for each variable (salinity, temperature) and day of the week (1-7).

[1]The number 215 is the day of the year, and the number 1800 is the start time—6:00 pm.

(a)


(b)
```
1: FileType week/year = /run/%i-%i/

2: FileType saltday = %i_salt.63

3: FileType tempday = %i_temp.63
```

Figure 4.3: (a) Simulation results stored on an ordinary filesystem. (b) FileType declarations describing the filesystem.

To access the data embedded in these file names, we write an expression in the style of the Unix `scanf` command.

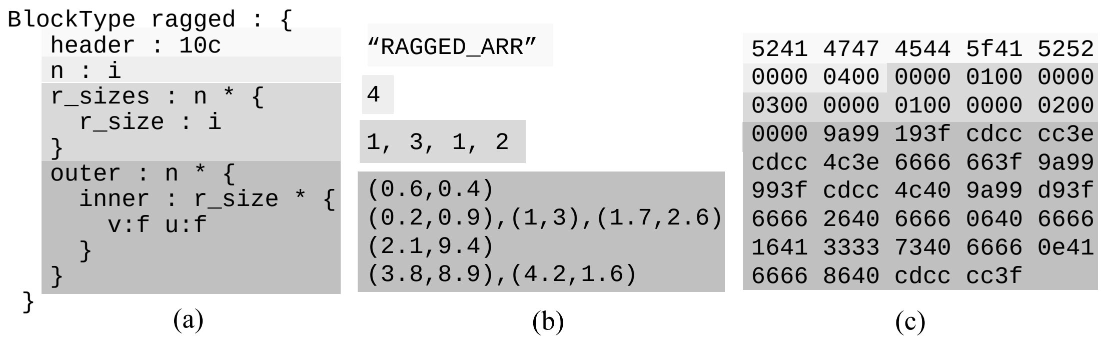
```
week/year/saltday = /runs/%i-%i/%i_salt.63
```

The left-hand side of this expression is a tuple of variable names formatted in a path-like notation. The right-hand side is a pattern matched against the set of all files in some filesystem context. The variable names on the left will be bound to values generated by a successful match of the pattern on the right. Each match will generate a sequence of values by evaluating the pattern in a particular filesystem context. For example, the variable `week` corresponds to the sequence $(01, 02)$ when the pattern is evaluated in the context of Figure 4.3(a). Note that the sequence order is determined by the manner in which the directory is traversed by the system calls for a particular OS.

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

```
week/year/saltday
```

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



```
BlockType ragged : {
   header : 10c
   n : i
   r_sizes : n * {
     r_size : i
   }
   outer : n * {
     inner : r_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 4.4: (a) A schema file for extracting binary file content. (b) An ASCII interpretation of a file instance. (c) A hexadecimal representation. Each color of shading represents a different logical component in the schema.

XPath-like conditions:

```
week[=04]/year/saltday[<3]
```

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

```
week[0:2]/year/saltday[1:3]
```

selects the zeroth and first `week`, and the first and second `saltday` files. The indices refer to the ordering of the files as returned by the operating system.

These expressions are passed as strings to an evaluation function by the programmer; the result is an array of typed values in the native language. The prototype is implemented in Python using the `numarray` extension [GMWH].

Note that FileType declarations are not linked to each other; a schema does not prescribe an overall directory hierarchy. Different queries may express different sequences of FileTypes. Any, all, or none of these sequences may be valid with respect to a particular filesystem context. For example, an individual scientist may have several "loose" `salt63` files stored in his or her home directory. Queries can then reference those files directly, without having to first navigate through a run directory.

BlockTypes, 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.

4.4.2 Data From Binary Files

Scientists frequently use packed binary encodings of large datasets to conserve space and improve performance. In this section we describe a model of this data and its interface with the model of the directory structure. We model the content of binary files as a sequence of one or more *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, or an anonymous *struct* component. Our examples will use the primitive typecodes 'f', 'i', and '*n*c', representing floating point numbers, integers, and character arrays of fixed length '*n*'. 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 through nesting.

A file format for storing a simple ragged array is described in Figure 4.4(a), an instance of the file, in ASCII, is shown in Figure 4.4(b), and a hexadecimal representation is shown in Figure 4.4(c). The root component in Figure 4.4(a) 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 `r_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 using either. In Figure 4.4(a), 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.

Components referenced in a length expression may appear anywhere in the file prior to the reference. This freedom generalizes other binary description formats that require that the length of a variable-length array be defined immediately prior to the array's elements [MC03, WB03]. Many formats, including netCDF [JS92], HDF [hdf04], and CORIE's own internal format require this generalization.

Scanning the raw file instance in Figure 4.4(c) sequentially, we can interpret the data as in Figure 4.4(b). First, we encounter a ten-character header "RAGGED_ARR", then the integer 4, then four integers 1, 3, 1, 2, and finally a longer sequence of floating point numbers.

The length of the array `inner` is a reference to the integer component `r_size`, which itself is a sub-component of an array `r_sizes`. For each element of `outer`, a different size is specified by indexing into the array `r_sizes`. The portion of the instance in Figure 4.4(b) 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 `r_sizes` have the same expression for their array length (the expression '`n`'), there is no ambiguity as to which particular element of `r_sizes` is being referenced.

The files used by scientists are effectively dependently typed—the type of the file depends on data appearing in the file. Dependently typed files allow enormous flexibility in describing a file format, but makes it difficult to derive an efficient reader in the general case. Systems such as PADS [FG05] and BinXML [WB03] also support dependent types, but the application is generally in charge of accessing the data efficiently.

4.4.3 Extraction Queries

Programmers can access file data by writing path expressions constructed as a sequence of named components from the schema. At each nesting level, an array expression may be used to further restrict which values should be returned. An individual array element can be specified through conventional integer indexing. An array can also be "sliced" to produce another array as in APL or Matlab. All of the following expressions are valid extraction queries over the schema in Figure 4.4(a).

```
ragged/n
ragged/outer/inner[1]
ragged/outer/inner/u
ragged/outer[0:4]/inner/u
ragged/outer/inner[0:1]/v
```

Given a description of a file's content, how do we load the data required by an application into memory? A component can be naturally interpreted as a type in C: A sequence of components becomes a C struct, an array component becomes a C array, or a primitive component becomes the appropriate primitive type in C. This translation suggests a natural way of reading the data into memory: Create the appropriate type for the entire file's content and populate it when the file is read. Extraction queries can be evaluated over this structure to copy out the desired data.

This approach is inadequate for two reasons: First, files may be large and applications may access multiple files; the entirety of all these files may not fit in memory. Second, applications must accommodate the C type system (as with the PADS system [FG05]), a restriction that may preclude the use of other high-level languages.

A solution to the first problem is to allow applications to request a specific portion of the file and have the software figure out the best way to extract that portion from the file and load it in memory. A solution to the second problem is to restrict the “requestable portions” to simple structures. We solve both problems by providing a simple query language where each result is a one-dimensional array whose element type is a primitive.

The ability to extract a component by name rather than by location in the file provides physical data independence. For example, the components `u` and `v` in Figure 4.4 could appear in reverse order without affecting applications. However, applications are not insulated from all changes in physical organization—if `outer` and `inner` were transposed such that `outer` was nested inside of `inner`, all but the first extraction query would throw an error.

4.5 EVALUATING QUERIES OVER BINARY DATA

In this section we describe the engine for evaluating extraction queries. We wrote the prototype in Python extended with the numarray library for handling large numeric arrays efficiently [GMWH]. C code was emitted for the code generation experiments. After writing a schema file, users can parse and process it by calling Python functions. Figure 4.5(a) illustrates the internal representation of the schema file in Figure 4.4(a). Each oval corresponds to a component object and each arrow corresponds to a reference to an object. Figure 4.5(b) illustrates an instance of the schema. The dashed ovals represent individual data values from a particular file. Query answers are a sequence formed from a subset of these values. For example, the query `outer[1].inner[0].u` returns the value u_{21} in Figure 4.5(b) and the query `outer.inner.v` returns all the values in Figure 4.5(b) with labels of the form v_{ij}.

In order to read a datum from a file, we must know its position within the file and its size in bytes. The position of any datum is the sum of the sizes of the data that appear before it. The size of a primitive value is defined by its type (e.g., 4 bytes for floats and integers). The size of an array is its length multiplied by

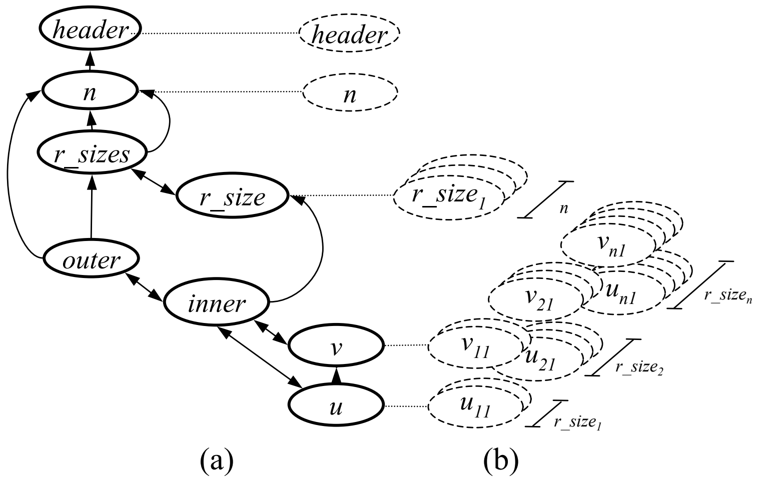


Figure 4.5: (a) Internal representation of a schema file. (b) An illustrated instance of a schema.

the size of its element type. The length of an array may depend on other data appearing earlier in the file. This situation can lead to significant complexity in file formats.

We now illustrate a naïve computation of size and position using the schema and instance of Figure 4.5. For a given file, let $\mathsf{pos}(x)$ be the offset of the datum x, and $\mathsf{size}(x)$ be the total number of bytes occupied by the datum x. Suppose we wish to read the datum u_{xy} in Figure 4.5(b) in order to answer the query `outer[x].inner[y].u`. The required computation can be expressed as follows.

$$\begin{aligned}
\mathsf{pos}(u_{xy}) &= \mathsf{pos}(v_{11}) \\
&\quad + \sum_{i=1}^{x-1} \sum_{j=1}^{r\_size_i} (\mathsf{size}(v_{ij}) + \mathsf{size}(u_{ij})) \\
&\quad + \sum_{j=1}^{y-1} (\mathsf{size}(v_{xj}) + \mathsf{size}(u_{xj})) \\
&= \mathsf{pos}(v_{11}) + \sum_{i=1}^{x-1} 8(r\_size_i) + 8(y-1)
\end{aligned}$$

where

$$\begin{aligned}
\mathsf{pos}(v_{11}) &= \mathsf{size}(header) + \mathsf{size}(x) \\
&\quad + \sum_{i=1}^{x} \mathsf{size}(r\_size_i) \\
&= 14 + 4x
\end{aligned}$$

This computation can be simplified, as shown, by exploiting static information such as the size of primitive types and constant array sizes supplied in the schema. (The latter case does not appear in this example.) We refer to this simplification process as *static optimization*. Once the position is computed, we can invoke a system call to seek there and read the value.

To evaluate the query `outer.inner.u`, we could repeat the computation above for each value u_{ij} in Figure 4.5(b). However, this strategy leads to millions of system calls. To improve performance, we want to minimize the number of read calls required.

The unknown quantities in the computation above, the value of n and the values of the array r_sizes, prevent us from completing the computation statically. If we read these data first, we can derive the position and size of every other datum in the file, and read larger blocks of data at a time. We refer to the process of prereading certain data to further simplify the computations as *dynamic optimization*, since the process requires a particular file instance.

Static and dynamic optimization allow us to partially complete the size and position computations required to evaluate queries. We offer two means of exploiting these computations. The first is by annotating the schema graph representation, in memory, with size and position information as it becomes available. The annotated schema graph can then be used to evaluate queries more efficiently. We liken this evaluation method to an *interpreter*. The second method is to generate programs able to evaluate queries efficiently. This method evaluates queries using a *compiled program*.

In order to perform dynamic evaluation, we must physically read values from a file instance. Using the interpretation method of evaluation, we can perform dynamic optimization on the fly as a query is received. However, we cannot generate, compile, and execute programs at run time in response to user queries without incurring significant cost. The reader might assume that dynamic optimization is therefore not available for use by the compiled program method of evaluation. However, we find in practice that many file instances share the same structure, even if they are not guaranteed to do so. Therefore, we can use a typical file instance, called a *file exemplar*, to perform dynamic optimization and generate programs specialized for answering queries over files that share structure with the file exemplar. Such a program would also include checks that the current file matches the exemplar in the appropriate components.

The programmer is responsible for supplying a file exemplar and specifying the components that will be read and used for specialization. For example, consider a set of files S conforming to the schema of Figure 4.4(a). Consider further that a substantial subset of these files $R \subset S$ agree on their values for n and r_sizes. The programmer can provide a file $r \in R$ as an exemplar, and specify the components n and r_sizes, and the system will generate an optimized program able to efficiently answer queries over all the members of R.

The use of a file exemplar implies two different binding times: one for reading some structural information about the file and another for reading the remainder. The use of multiple binding times suggests that we are exploiting a form of partial evaluation.

The evaluation strategies we have described so far assume that the system calls `seek` and `read` are used to perform the actual file I/O. Query evaluation may potentially make many of these expensive system calls. An alternative is to use *memory mapped files* and allow the operating system to manage the loading of data from disk to memory. Memory mapped files can be significantly faster than "manual" I/O. Database systems typically do not use memory mapped files, since they tend to know much more about the data than does the operating system, and can therefore make better decisions about when to load and evict pages. Paging policies for files whose structure is not available until runtime, however, are more difficult to determine. We find that the use of memory-mapped files improves performance by around 30%. The experimental results in Section 4.6 are all reported using memory-mapped files, except in the hand-coded case.

4.5.1 Schema Transformation

By reading certain components early, we can partially evaluate the computations required to answer queries. Another approach to optimization we have explored is to transform and simplify a schema in response to a particular query. Given a query Q over a schema s describing a file instance f, there exist a set of schemas $s_0, s_1, \ldots, s_n$ that return the same result: $Q[s_0] = Q[s_1] = \cdots = Q[s_n]$. Some schemas require fewer read calls.

Consider the query `outer[0:2].inner.u` over the schema of Figure 4.4(a). In Figure 4.6(a), we show a transformed version of the schema. We have blended the two components `u` and `v` into a single component named `uv`. Further, the nested,

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

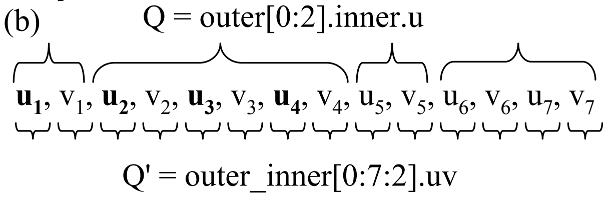
(b)

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 4.6: (a) A transformation of the schema in Figure 4.4(a). (b) A transformed query compatible with the transformed schema.

---

ragged array has been flattened into a one 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 4.6(b), we have a sequence of values representing the contents of the schema instance in Figure 4.4(b). 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).

This transformation allows us to treat a nested array with a complex element type as a one-dimensional array with a primitive element type. The reason this transformation is useful is that our query evaluation engine is already capable of reading, slicing, and returning one-dimensional arrays. We find it easier to “program” the query evaluator by changing the schema than to empower it with the full types system used to describe data.

We apply schema transformation rules after dynamic optimization. As we recursively descend the schema graph reading data, we may apply "temporary" schema transformation rules multiple times. The schema and query rewrites, along with any conditions for their application, are described by the following rules. The notation $S = \mathit{expr}$ ($Q = \mathit{expr}$) means that the schema (query) being considered is expr. The rules below can be read "if the schema is sexpr, and the query is qexpr, then an equivalent schema and query are sexpr' and qexpr'"

$$\textsc{enstruct}\ \frac{S = a : n * \{b : m * \{x : t\}\} \qquad Q = a[s_a : e_a].b[s_b : e_b].x}{\begin{array}{c} S = a : n * \{\mathit{prefix},\ b : (e_b - s_b) * \{x : t\},\ \mathit{suffix}\} \\ Q = a[s_a : e_a].b.x \end{array}}$$

$$\textsc{destruct}\ \frac{\begin{array}{c}\mathsf{sizeof}(t_i) = \mathsf{sizeof}(t_j) \text{ for all } i, j \\ S = a : n * \{x_0 : t_0, x_1 : t_1, \ldots x_m : t_m\} \qquad Q = a[s : e].x_i\end{array}}{S = a : (nm) * \{x : t_i\} \qquad Q = a[s + i : e * i : i]}$$

$$\textsc{flatten}\ \frac{S = a : n * \{b : m * \{x : t\}\} \qquad Q = a[s_a : e_a].b.x}{S = ab : (n * m) * \{x0 : t\} \qquad Q = ab[s_a * m : e_a * m].x}$$

$$\textsc{group}\ \frac{S = a : n * \{x : t\} \qquad Q = a.x}{S = x : (\mathsf{sizeof}(t)n)c\} \qquad Q = x}$$

The rule ENSTRUCT breaks a nested array into a skippable prefix, a slice to read, and a skippable suffix, and transforms the query to access the slice portion only. The prefix is described by the schema $\mathit{prefix} : (\mathsf{sizeof}(t)s_b)c$ and the suffix is described by the schema $\mathit{suffix} : (\mathsf{sizeof}(t)(m - e_b))c$. The query evaluator no longer needs to iterate over each element of the outer array and each element of the inner array, but can stride over portions of the array that do not need to be returned.

The rule DESTRUCT converts an array with a complex element type into an array with a simple element type as in Figure 4.6. DESTRUCT is a generalization of FLATTEN. The benefit in this case is that an array with a simple element type is read all at once and then sliced.

The rule FLATTEN converts a nested array with a primitive element type into a one-dimensional array. This rule allows a contiguous block to be read and returned with one call.

The rule GROUP converts an array with a primitive element type into a block of uninterpreted bytes. An array can be read with a single call, but arrays incur some overhead since elements are interpreted as typed values. GROUP avoids this overhead and just copies the bytes into memory directly.

These rules provide the basis for optimizing extraction queries using schema transformations. The list of rules is probably not complete; we expect more rules to be discovered for various situations. The key idea is that thinking in terms of transforming the schema as well as the queries seems like a promising conceptual framework for optimizing extraction queries.

4.6 EXPERIMENTAL RESULTS

Figure 4.7 shows the schema for the simulation result files from the CORIE project 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 to 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 daily.

The header portion of each file, shown in plain type, contains time-independent information, including the horizontal and vertical grids and the river's bathymetry. The time-varying portion of the file, highlighted in bold type, is a nested array

```
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 } }
timedata : nsteps * {
  tstamp : f, tstep : i
  surfdata : nodes * { surf : i  }
  horizontal : nodes * {
    depths : (levels - bathymetry + 1) * {
      vector : rank * { val : f }
    }
  }
}
```

Figure 4.7: A BinaryBlock describing the CORIE simulation output. The bold text indicates the time-varying portion of the dataset.

component with four layers of nesting over the actual values of the simulated variables: `timedata`, `horizontal`, `depths`, and `vector`. The array `surf`, similar to the array `bathymetry`, contains indices 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, so we need a separate surface array for each timestep.

We compare the performance of four representative queries using seven different techniques. The queries are listed in Table 4.1 and the results are presented in Table 4.2. Query 1 extracts the first complete timestep of data from the file, copying it to an array in memory. Query 2 accesses all timesteps, but extracts only a relatively small portion of the horizontal data; 2999 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 proved most challenging for our software. Query 4 extracts the surface information for timesteps 30 through 44. This query does not directly involve any variable-length arrays and is therefore easier to compute for most techniques. Also, this query also had the smallest result size.

The experiments were conducted on a platform almost identical to the nodes of the cluster used to run the CORIE simulations. The machine runs Linux on a 2.4 GHz processor with 4GB of main memory. We report the average of eight experiments run on two different days with the machine unencumbered. The variance was less than 1% in all cases, demonstrating stability of the results.

The simplest technique we tested was to use the internal representation produced by the static optimizer directly (Experiment (a) in Table 4.2). 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 4.2 uses a small class of schema and query transformations (see Section 4.5). 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 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 of the relevant section, 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

Table 4.1: 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

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 4.2) 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 incorporate the general form of this technique. Of course, the hand-coded reader will operate correctly only on files with this exact schema.

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 may 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

Table 4.2: Response time in seconds by query.

Experiment	Q1	Q2	Q3	Q4
(a) interpreted, static	>500	>500	>500	>500
(b) interpreted, dynamic	28.1	138.	284.	8.64
(c) same as (b), plus transformations	0.83	1.	86.	0.85
(d) same as (c), runtime only	0.02	0.24	85.	0.26
(e) compiled, static	65.	104.	>500	3.2
(f) compiled, dynamic	4.7	22.	32.	2.6
(g) by hand	0.02	0.5	0.6	0.02

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 the 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.

4.7 CONCLUSIONS

In our experience, scientists tend to prefer lightweight tools that they can deploy incrementally. Transformative solutions such as relational databases require a significant up-front investment of resources. Since the gridfield model represents a significant departure from the status quo, we have explored ways of “softening the blow” and improving the odds of adoption.

Specifically, we have described simple software layers that hide the complexity and diversity of existing file formats and directory structures without compromising performance. We prefer this process of “retrofitting” gridfields onto existing datasets to the alternatives. 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. Our techniques can facilitate data sharing between research groups and institutions with heterogeneous data formats, and make it easier to leverage the gridfield model.

Chapter 5

APPLICATIONS AND INTERFACES

We have considered three distinct aspects of the gridfield approach: the model (Chapter 2), alternative data structures with which to implement the model (Chapter 3) and an application of these data structures to existing, ad hoc, file-based data (Chapter 4). In this chapter, we synthesize these ideas and argue that gridfields offer conceptual economy, optimization opportunities, and a separation of of logical and physical concerns at several stages of application development. We describe narratively how these benefits are realized by tracing the the evolution of the model and implementation from an abstract reasoning tool, to a programming aid, and finally to a gridfield-powered data-visualization application.

Section 5.1 describes gridfields as an abstract language useful for reasoning about complicated gridded datasets even in the absence of an implementation. Section 5.2 describes our experience programming with gridfields, and how this experience led to model-level refinements. Section 5.3 describes how the various way gridfield expressions can be evaluated.

5.1 REASONING WITH GRIDFIELDS

Gridfields emerged first as a data-product description language that bridged the gap between English and code. For example, the evolution of the CORIE Environmental Observation and Forecasting System has been punctuated by major releases involving a significant change in the structure of the simulation mesh. All stages of the forecast pipeline are affected by these changes: processing of

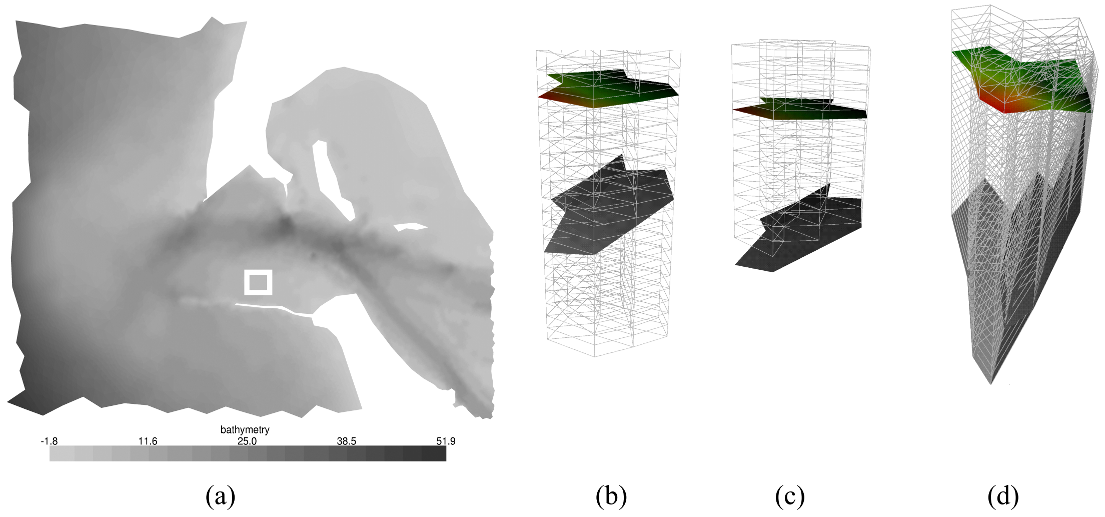


Figure 5.1: The CORIE grid evolved over time, requiring significant changes to the analysis pipeline. (a) The mouth of the Columbia River Estuary shaded by bathymetry with a highlighted region in white. (b) The original CORIE grid extended below the bathymetry. (c) To save space, these invalid values were culled, complicating the file format. (d) A newer model uses a σ-grid, with a fixed number of depth levels that follow the contours of the bathymetry.

---

atmospheric and river inputs, the finite element methods used in the simulation kernel, and generation of data products. Gridfields became a useful way to characterize each of these components, allowing us to conduct a "sensitivity" analysis of the overall system. If the timestep is changed, which programs should be affected? Which programs rely on the fact that the grid only involves triangles? Figure 5.1 illustrates a particularly inconvenient series of changes to the grid structure. Figure 5.1(a) is an illustration of the area around the mouth of the Columbia River shaded by depth measured as meters below Mean Sea Level (oriented downward). Figure 5.1(b), (c), and (d) are sections cut from three different grids modeling the same region, highlighted in (a) by a white rectangle. In each illustration, the transluscent surface represents the water's surface at a particular timestep, and the solid surface represents bathymetry. The earliest grid used by CORIE had the form of Figure 5.1(b). This grid extends below the bathymetry; a distinguished value (-99) is stored for these underground positions to indicate they are invalid. The distinguished value is also used to indicate mesh nodes positioned above the water's surface.

The gridfield expression to construct this grid appears in Figure 5.2(b). Figure 5.2(a) is a reminder of the shape of the grids H and V. The vertical grid V has z-coordinates bound statically—the z-coordinates do not change over time, though the status of nodes as valid or invalid does change as the water's elevation changes.

The largest z-coordinates in the vertical grid, corresponding to the deepest parts of the ocean, are only "wet" for a very small portion of the overall grid. Elsewhere, they are positioned underground. In the region around the mouth of the Columbia river shown in Figure 5.1(a), most of the vertical grid V is invalid—the river's mouth is very shallow relative to the ocean depths. The H grid in this region is also highly refined to accurately capture the complex interaction between the tides, fresh and saltwater mixing, and irregular shoreline. Consequently, most of the nodes in the overall product grid $H \otimes V$ are positioned underground. Storing -99 for each invalid position is therefore rather wasteful. As inexpensive as secondary storage has become, the simulation outputs are regularly deleted to make room for more. Freeing almost 50% of one's space with no loss of information is therefore an attractive option.

To recover this wasted space, the CORIE grid evolved into the form in Figure 5.1(c). Now, the positions below the bottom of the river are culled from the grid itself rather than simply marked as invalid. Programs manipulating grids and attributes as raw arrays were significantly affected by this change: They must now accommodate "ragged" arrays that do not have a rectangular shape. Unfortunately, the full-sized rectangular arrays tend to reappear during processing, since tools such as MATLAB, netCDF, HDF, and APL are not well suited to manipulating ragged arrays. Using gridfields, the new grid can be expressed by adding a single restrict operator after the cross product, as in Figure 5.2(c). The result is no more troublesome to reason about than the original grid.

Although the grid in Figure 5.1(c) wastes less space, the static z-coordinates still make it difficult to increase resolution in shallow areas. It helps that the z-levels need not be distributed uniformly: Their density is higher near the surface than near the bottom to capture high-frequency variation in the shallow waters. However, this resolution is wasteful in the open ocean where the low frequency features such as global currents and seasonality dominate at all depths.

To better fit the multi-scale domain, a fine grid near the mouth of the river and a coarse grid in the open ocean is desirable. A similar problem motivates the use of the unstructured horizontal grid to discretize the (x, y)-plane parallel to the Earth's surface. Many small triangles and quadrilaterals with areas no more than a few square meters are used to model the estuary, while large triangles with areas of several square kilometers are used in the open ocean. An analogous solution exists for the depth-resolution problem: Convert to a full three-dimensional unstructured grid consisting of tetrahedra. However, the currents in the ocean and river are predominately lateral; this asymmetry between (x, y) directions and the z direction is exploited in the simulation to reduce running time. Another solution is to retain the prism shape of the cells, but to relax the condition that the edges of the prisms must be aligned with coordinate axes. Instead of a stair-step pattern at the bottom of the river and ocean, the grid fits flush, as in Figure 5.1(d). This configuration is referred to as a σ-grid. The σ-grid also follows the surface of the water, such that the z-coordinates bound to the nodes are now time-dependent.

The value σ ranges from 0 to 1 and represents a proportion of the total depth rather than a fixed z-coordinate. Therefore, the grid is topologically similar to Figure 5.1(a), in that it is formed from the cross product of the horizontal grid

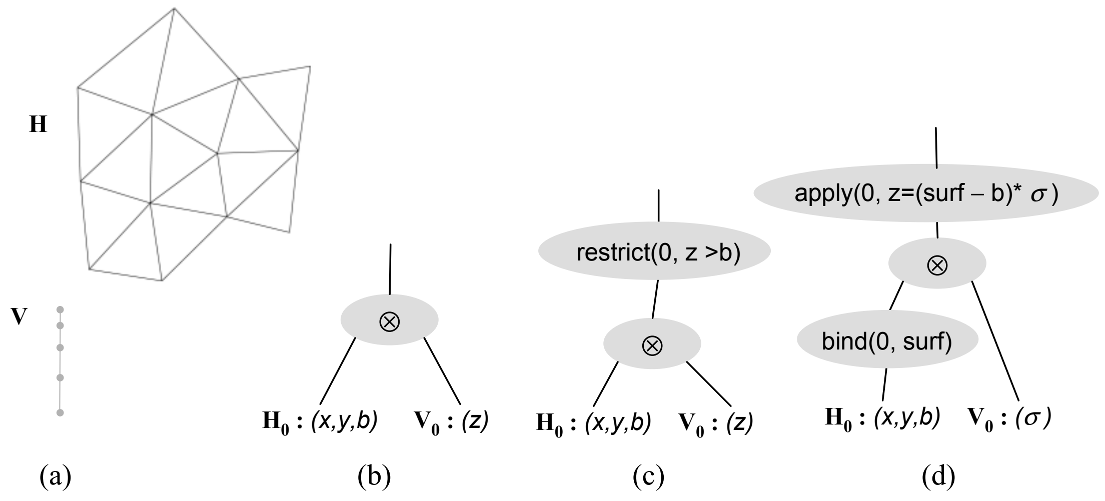


Figure 5.2: The structural differences between CORIE grids are expressed succinctly using gridfields. (a) The form of the horizontal and vertical grids. Plans (b), (c), and (d) implement the construction of the grids shows in Figure 5.1(b), (c), and (d), respectively.

---

and the vertical grid. However, the geometry of this product grid (specifically, the z-coordinate) depends on not just the node in the vertical grid, but the bathymetry and the water's surface as well as the node in the vertical grid. Remarkably, the surface, and hence the geometry of the grid, now changes over time. The gridfield expression in Figure 5.2(d) describes how the sigma grid is constructed. The z-coordinates now depend on the water's surface, which varies with time. We therefore bind the surface attribute before computing the cross product. The calculation of the z-coordinate appears as an apply operator with the expression $z = (b - surf) * \sigma$, where b is the bathymetry (the surface value is subtracted from the bathymetry since the z-axis is oriented downward).

The opaque surface representing bathymetry in Figure 5.1(d) appears to be different than in Figure 5.1(b) and Figure 5.1(c). Indeed, the bathymetry can be modeled more accurately with the σ-grid since the interpolation errors introduced by the stair-step patterns are avoided. Hence the sharp peak in the Figure 5.1(c), which was suppressed in Figure 5.1(b) and (c).

With each change to the grid, a gridfield expression that captures the change precisely was straightforward to write down and reason about. Even before a gridfield implementation was available, such descriptions served to aid the discussion about consequences of grid changes. Once the grid changes were fully implemented, the gridfield diagrams served as documentation—"before and after" pictures to orient visitors and incoming students.

5.2 PROGRAMMING WITH GRIDFIELDS

After successfully deploying the gridfield model as a reasoning tool, we implemented the algebra in C++. The grid, the attribute, the gridfield, and each operator was implemented with a separate class, organized into an appropriate inheritance hierarchy. Recipes were constructed by wiring together trees of operator objects, then calling the `getResult` method. The algorithms implementing the operators recursively called the `getResult` method on their children; at the leaves, the scan and bind operators accessed files directly. The results were piped into VTK for direct visualization. More complex visualizations could be constructed by drawing multiple gridfields in the same scene.

Once implemented, gridfields provided a narrow interface with which to manipulate scientific simulation results. By working with only a few operators, programmers could incrementally and interactively develop data product recipes without continually researching documentation. Since the algebra is closed, i.e., all operators consume and produce gridfields, most results can be inspected visually with a single function call and two arguments: the root gridfield operator and an attribute name with which to color the visualization. This capability proved useful during debugging and testing.

In this section, we describe three examples of programming with gridfields and use them to motivate design decisions in the gridfield model.

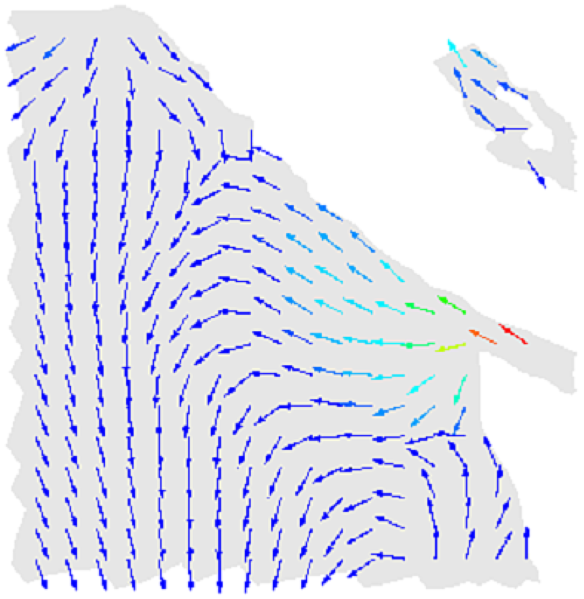


Figure 5.3: A visualization of average velocity over time interpolated to a particular depth.

5.2.1 Persistent Eddies

Figure 5.3 shows a common vector field visualization technique: associate an arrow glyph with each node or cell oriented in the direction of the flow. To investigate persistent eddies in Vancouver BC driven by the Fraser River, one student needed to compute and visualize the average velocity at various depths over a period of several weeks. This data product requires access to around 10GB per week of simulated time.

The average velocity data product motivated some extensions to the data model and the execution model. First, the original gridfield model allowed attributes to be bound at only one dimension at a time. To work with data at multiple dimensions, the programmer would have to use the merge operator and the regrid operator. For example, imagine a programmer wants to find the air-surface flux values, a function of the 2-cells, within a custom bounding box. The restriction to a bounding box

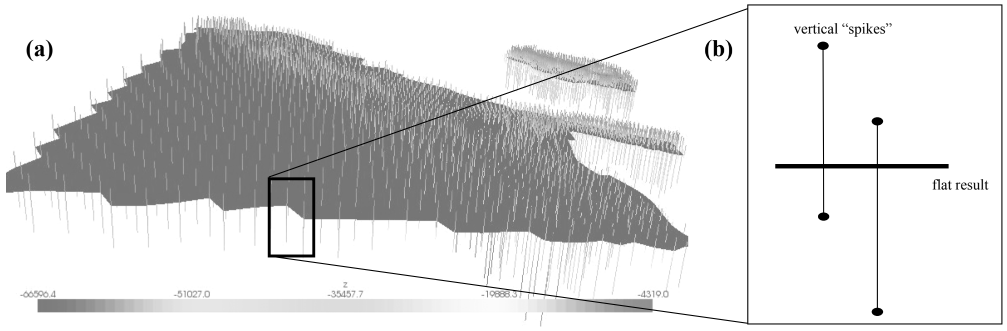


Figure 5.4: Each node in the flat shaded result 2-D gridfield depends on two values from the full 3-D gridfield: the value directly above and the value directly below. The close-up at (b) illustrates the intersection of the flat result with two vertical spikes.

---

requires a gridfield on 0-cells with bound attributes representing x and y. The restricted gridfield can then be merged with the flux-carrying gridfield over 2-cells.

The one-rank-only gridfield model simplified the definition and implementation of the individual operators, but queries became unnecessarily complicated due to frequent "rank-switching" operations. For example, both recipes in Figure 5.5 and Figure 5.6 compute the interpolation of a dataset at an arbitrary depth in the σ-grid of Figure 5.1(a). Figure 5.5 uses the one-rank-only model and Figure 5.6 uses the current multi-rank model. Besides requiring more operators, Figure 5.5 requires processing along two distinct paths, corresponding to processing at two different ranks. In Figure 5.6, both ranks can be processed using a non-branching sequence of operators.

Specifically, the recipes in Figure 5.5 and Figure 5.6 interpolate the velocity field at a particular depth for a particular user-defined region. The result must be a flat 2-D gridfield at a fixed, user-defined depth (3 meters below Mean Sea Level in this example). However, the data we are interpolating from is bound to the σ-grid of Figure 5.1(d), whose cells follow the bathymetry of the river. Each node in the flat result grid intersects a single vertical 1-cell.

Figure 5.4(a) is an intermediate result displaying two gridfields: one is the 2-D result gridfield at a fixed depth, and the second is a collection of vertically oriented 1-cells to which the data to be interpolated has been bound. To gather these "spikes," we use the accrete operator to remove all cells except the nodes from the horizontal gridfield **H**. Then we compute the cross product of this zero-dimensional gridfield with the one-dimensional gridfield **V**. The gridfield **V** has been altered, however: Each 1-cell has two incident nodes with associated σ values. The lower of the two is mapped as "minsigma" and the higher of the two is mapped is "maxsigma." Specifically, the two σ values are taken from the two 0-cells incident to a given 1-cell (Figure 5.4(b)). The regrid operator accomplishes this alteration with an assignment function mapping to each 1-cell c the set of incident nodes $\{n \mid n \prec c\}$, and selecting the minimum and maximum value from this set. The value σ multiplied by the difference between the water's surface and the water's bathymetry produces a z-coordinate suitable for comparison with the user's selected depth. A restrict operator finds those 1-cells whose minimum and maximum z coordinates form an interval that contains the user's chosen depth. The result of this restrict operator are the "spikes" in Figure 5.4. The velocity components u and v are bound to the spikes, then interpolated onto the result grid using another instance of regrid.

The refinement of the gridfield model to allow data bound to multiple ranks simultaneously sometimes led to simpler recipes, as this example demonstrates. This example also demonstrates the use of gridfields as a visual algorithm development environment. Efficient recipes can be derived by reasoning about and visualizing the intermediate results. Finally, this example demonstrates the utility in thinking about the grid separately from the bound data—the interpolation algorithm has a topological component that can be worked out independently from how the data is organized.

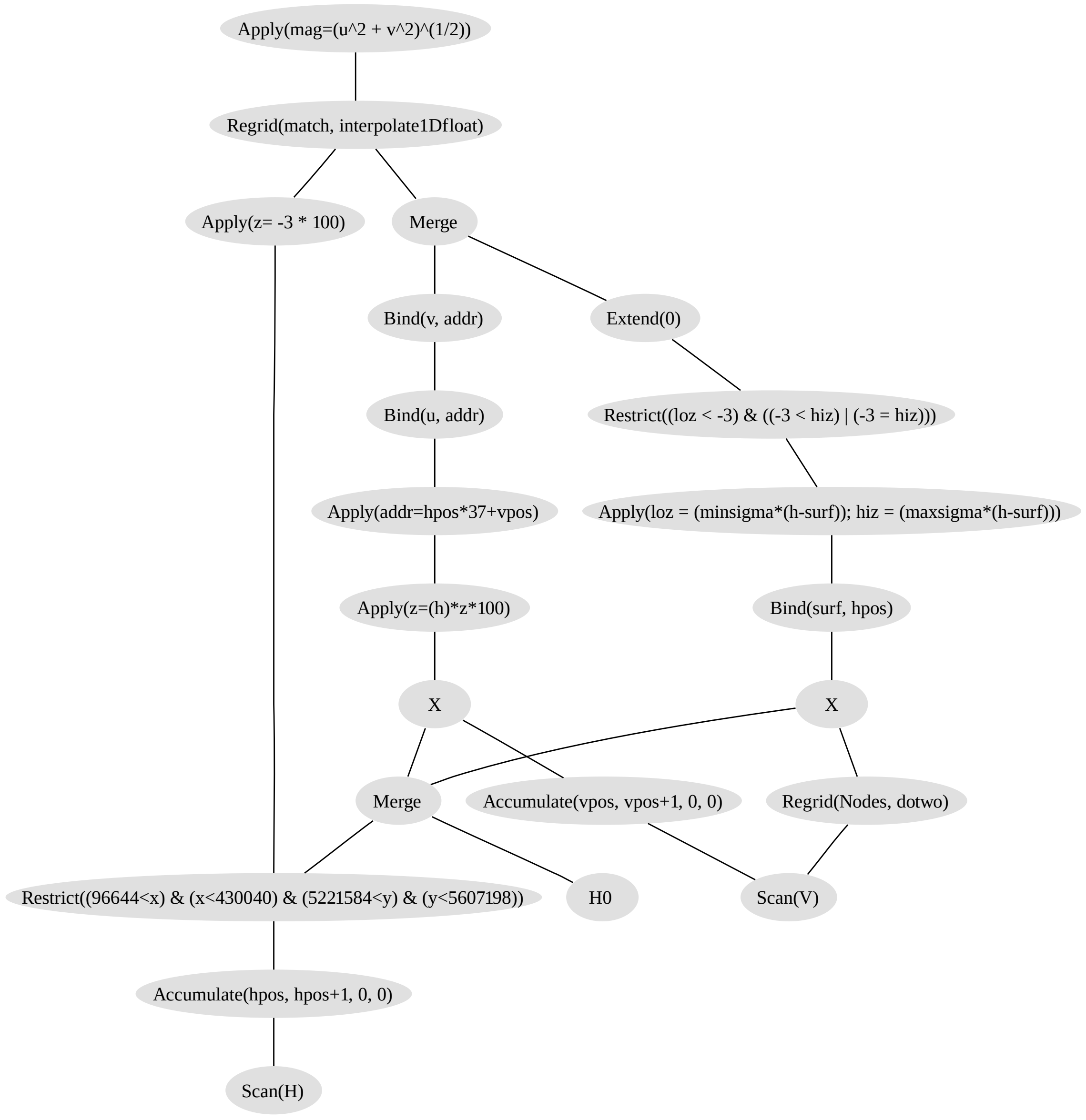

Apply(mag=(u^2 + v^2)^(1/2))
Regrid(match, interpolate1Dfloat)
Apply(z= -3 * 100)
Merge
Bind(v, addr)
Extend(0)
Bind(u, addr)
Restrict((loz < -3) & ((-3 < hiz) | (-3 = hiz)))
Apply(addr=hpos*37+vpos)
Apply(loz = (minsigma*(h-surf)); hiz = (maxsigma*(h-surf)))
Apply(z=(h)*z*100)
Bind(surf, hpos)
X
X
Merge
Accumulate(vpos, vpos+1, 0, 0)
Regrid(Nodes, dotwo)
Restrict((96644<x) & (x<430040) & (5221584<y) & (y<5607198))
H0
Scan(V)
Accumulate(hpos, hpos+1, 0, 0)
Scan(H)

Figure 5.5: This recipe for interpolating values at a particular depth is complicated by the movement of data from one rank to another. Recipes such as this motivated the current multi-rank gridfield data model. The accumulate operator (used near the leaves) is a specialization of the regrid operator used to assign an identifier to each cell (see Section 2.4.6).

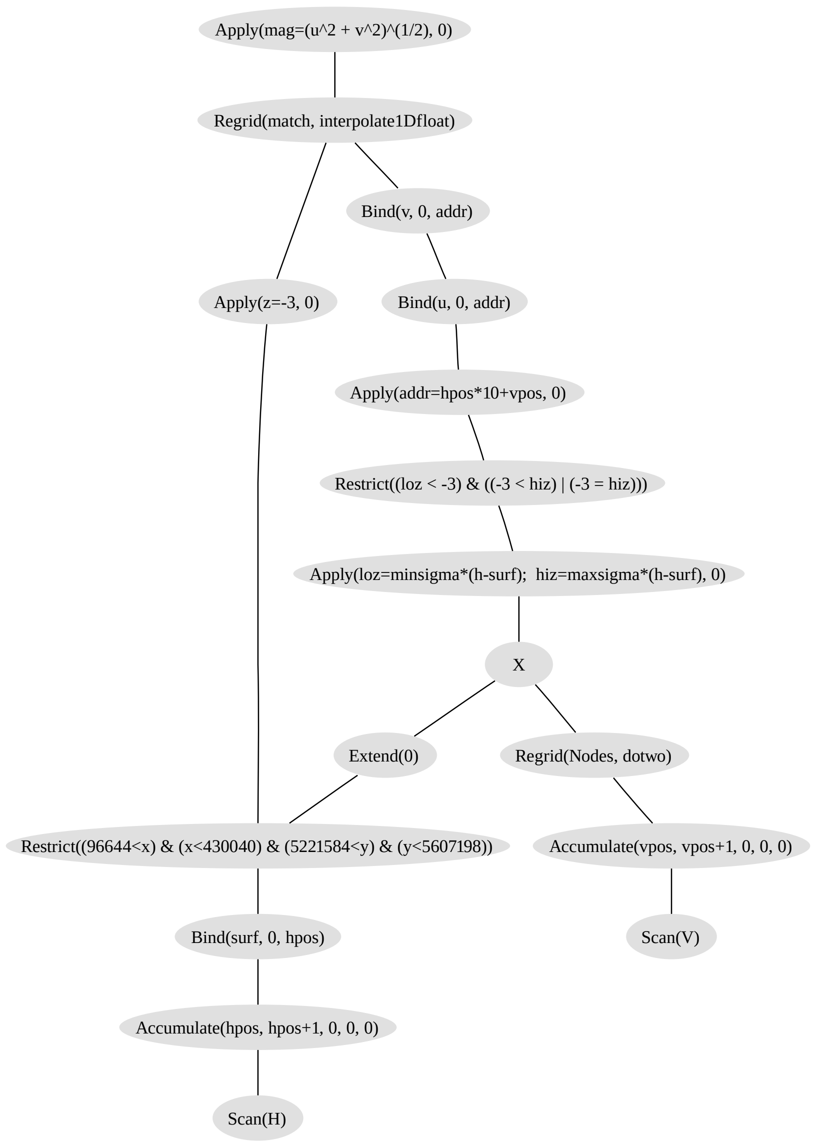

Apply(mag=(u^2 + v^2)^(1/2), 0)
Regrid(match, interpolate1Dfloat)
Bind(v, 0, addr)
Apply(z=-3, 0)
Bind(u, 0, addr)
Apply(addr=hpos*10+vpos, 0)
Restrict((loz < -3) & ((-3 < hiz) | (-3 = hiz)))
Apply(loz=minsigma*(h-surf); hiz=maxsigma*(h-surf), 0)
X
Extend(0)
Regrid(Nodes, dotwo)
Restrict((96644<x) & (x<430040) & (5221584<y) & (y<5607198))
Accumulate(vpos, vpos+1, 0, 0, 0)
Bind(surf, 0, hpos)
Scan(V)
Accumulate(hpos, hpos+1, 0, 0, 0)
Scan(H)

Figure 5.6: In this recipe for interpolating values at a particular depth, all ranks can be processed with a non-branching sequence of operators. This recipe computes the interpolation of data at a particular depth over the σ-grid in Figure 5.1. Recipes such as this motivated the current multi-rank gridfield data model. The accumulate operator (used near the leaves) is a specialization of the regrid operator used to assign an identifier to each cell (see Section 2.4.6).

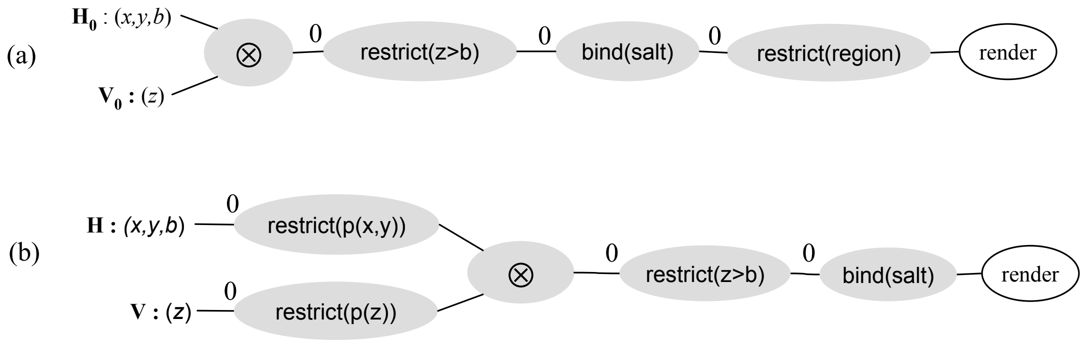


Figure 5.7: (a) The original recipe for visualizing a 3-D CORIE dataset. (b) An optimized equivalent recipe. The integer decorations indicate the dimension argument passed to the operator immediately to the right.

5.2.2 Pushing **restricts**

Recall the recipe of Figure 2.17, repeated in Figure 5.7(a), that computes a 3-D salinity gridfield and zooms in 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 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. In this section, we give a simple example of the solution to this problem; in Section 5.3.2, we will describe the general form of the technique.

To solve the 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 **H** and **V**, respectively[1] 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 **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 **H** stores bathymetry information for the river. Specifically, b is an index into the gridfield **V**. Since **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 5.7. The potentially highly selective restricts on x, y and z are evaluated prior to the cross product.

5.2.3 Optimizing Slice Products

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 5.8. One way to express the horizontal slice data product is to use the same plan as in Figure 2.17, but restrict the z dimension to a single value. As in Section 5.2.2, we could push the restricts through the cross. This time, though, we observe that restricting **V** to a single node produces the unit grid. Since the unit grid is the identity for the cross-product operator, we can remove the cross-product operator altogether.

[1]This restrict involves attributes from both **H** and **V** and does therefore not commute with the cross product.

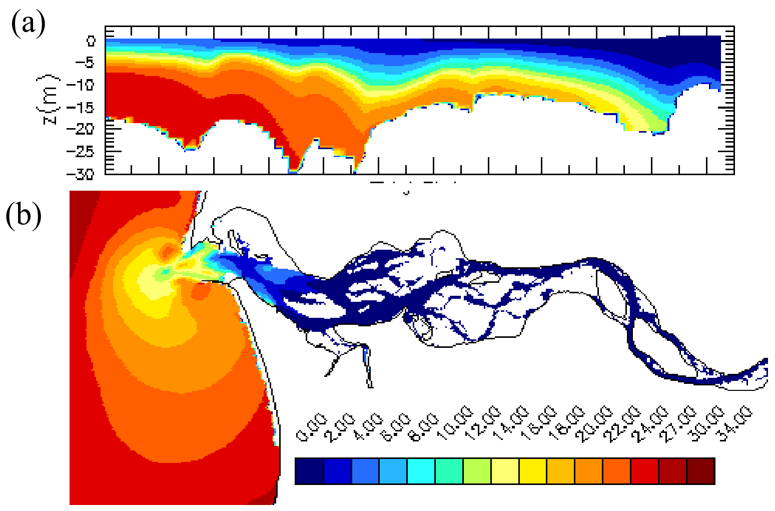


Figure 5.8: (a) A vertical slice data product. (b) A horizontal slice data product.

This optimization is unavailable to systems that cannot reason about grids alge-braically. We have not only produced a more efficient recipe, but we have also naturally expressed a critical correctness criteria: The output grid is 2-D and identical to one of the input grids. 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, and that the depths do not vary over time. We could relax these assumptions by using a regrid 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 plan. 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 5.9(a). We can connect these points to form a 1-D grid, P. A cross product with the vertical grid gives us a grid

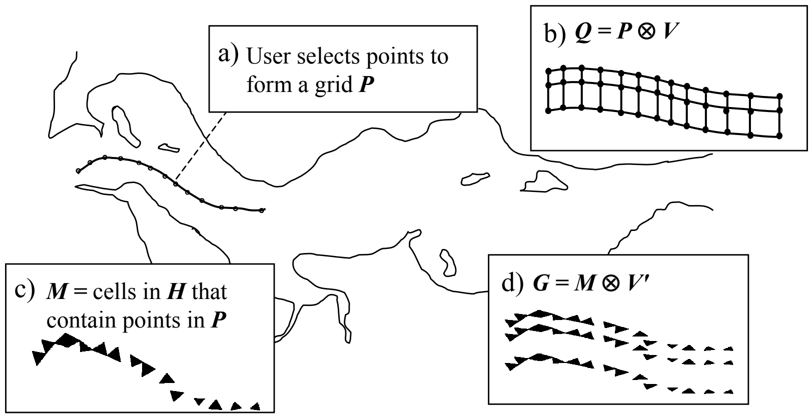

a) User selects points to form a grid P
b) $Q = P \otimes V$
c) M = cells in H that contain points in P
d) $G = M \otimes V'$

Figure 5.9: A summary of the four intermediate steps in an efficient "vertical slice" recipe.

---

representing the 2-D slice, $P \otimes V$ (Figure 5.9(b)).

Using the Visualization Toolkit library (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 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 regrid operator followed by a restrict, producing a grid M (Figure 5.9(c)).

Since the grid M is a restriction of the grid H, we can commute the restrict operator (as we did previously) to construct a 3-D grid (not shown in Figure 5.9) 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 using the accrete operator, leaving only the nodes. Call this grid V'. The grid $M \otimes V'$ consists of “stacks” of 2-D triangles (Figure 5.9(d)), rather than a connected set of 3-D prisms. To guarantee the same output, we exploit a property of bi-linear interpolation: Interpolation on the base of the 3-D prism is the same as interpolation on the 2-D triangle alone, with the latter being much less costly.

By working primarily in two dimensions and reasoning about the product structure of the grid we were able to produce a less expensive plan. Lowering the dimension of the intermediate results saves time since 1) higher dimensional gridfields tend to have more cells, and 2) algorithms for manipulating 3-D cells are more expensive than their 2-D counterparts. We will analyze the performance of this optimization in Chapter 6.

This optimization is rather specialized to a particular situation and would be difficult to apply automatically. However, it demonstrates that reasoning at the grid level can expose such optimizations, and that they can be expressed with the gridfield algebra.

Real images from the steps of the algorithm illustrated in Figure 5.9 are shown in Figure 5.10. The actual intermediate results are easy to visualize since they are the same type of object as the final results; i.e., the gridfield algebra is closed.

5.2.4 Plume Front

The dominant feature of the interface between the Columbia River and the Pacific Ocean is a large, persistent plume of fresh water. The gradual mixing of fresh and salty water (as well as cold and warm water) provides an attractive habitat to various organisms, especially salmon. The economics and science of the salmon industry is an important issue in the Pacific Northwest. Models such as CORIE can be used to perform investigative analysis of the consequences of policy decisions. The effect of river dredging, factory run-off (both as a chemical and temperature pollutant), and even climate change can be studied in the context of simulation.

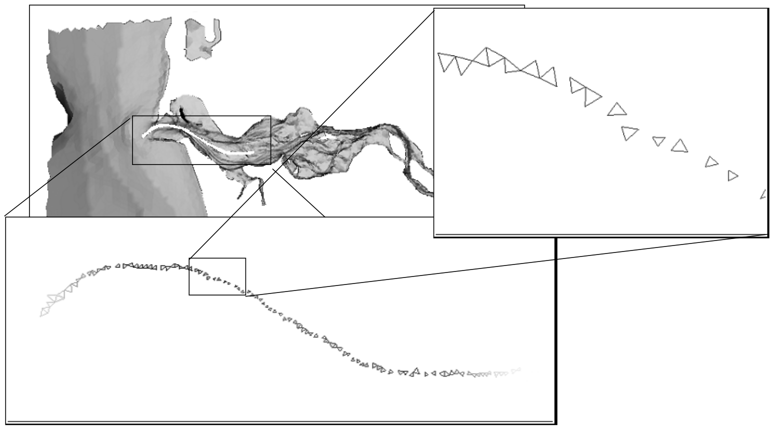


Figure 5.10: The polygons containing user-selected points for computing a transect.

Salmon tend to congregate at the forward edge of the plume. An immediate question is how to define the "forward edge." While the ultimate answer is left up to the scientists, gridfields provide a simple way of testing various definitions. One proposal was to identify regions outside the mouth of the river where the magnitude of the gradient of the salinity field is above a given threshold. There was concern that this definition would be too simplistic, but rather than speculate, we designed a data product that computed the gradient and visualized the results.

In Figure 5.11(a), the lightly shaded areas represent high gradient and the heavily shaded areas represent lower gradient. The vaguely circular shape of the plume is clearly visible, as are the large eddies forming in the "wings" of the plume. The mouth of the river (labeled) is especially turbulent and powers significant mixing between salty ocean water and fresh river water. The gradient of the salinity field therefore reaches extremely high magnitudes in this region. To improve the visualization, gradient values above a given threshold g_{max} are set to g_{max}. In Figure 5.11(b), we illustrate this gradient calculation implemented using the regrid operator. For each node in the grid, we compute the sum of all vectors v_x, where v_x points in the direction of an adjacent node x and has magnitude equal to the

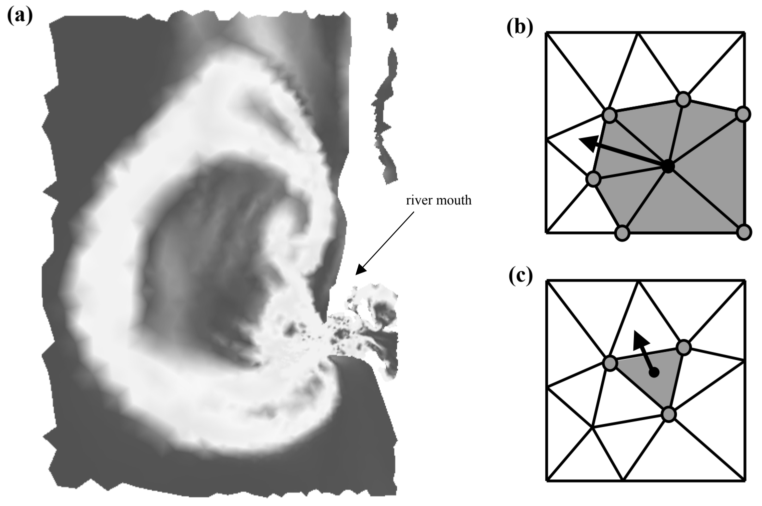


Figure 5.11: (a) The plume front computed as a low-pass filter on the gradient magnitude. The gradient is calculated using the regrid operator either (b) on each node using the values at adjacent nodes, or (c) on each cell using incident nodes. Method (b) was used to generate the image at (a).

salinity value at x. An alternative appears in Figure 5.11(c). The gradient can be computed at the center of each 2-cell rather than at the nodes. There are two advantages of (c). 1) Due to our choice of representation, navigating I_{20} is more efficient than finding adjacent nodes by navigating I_{01} and then I_{10} (or I_{02} and then I_{20}). 2) The gradient is simply the slope of the plane equation for the triangle, a quantity that can be calculated directly. The disadvantages are that 1) only triangles unambiguously define a plane (and there are quadrilaterals to consider), 2) fewer nodes are used in the calculation, leading to a less accurate approximation, and 3) a custom aggregation function is required: It must compute the cell center of a polygon and derive the plane equation. In (b), the aggregation function is simply a vector sum, a function already included as part of the core gridfield library.

In this example, gridfields provided low-level control over the important part of the computation—the method of approximating the gradient—without requiring

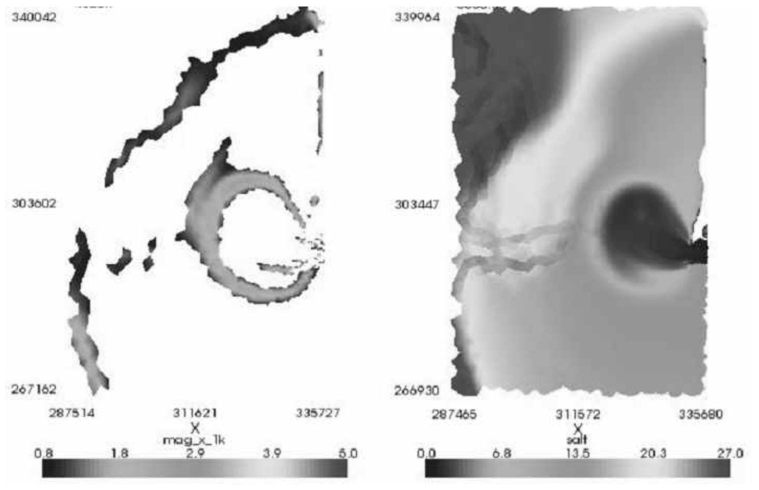


Figure 5.12: A finished data product derived from the same recipe used in Figure 5.11. The left panel represents only those cells where the salinity gradient is above a threshold, while the right panel is colored by salinity itself.

---

excessive attention on the more mundane parts—finding a particular region of the larger grid, interpolating at a particular depth, and gathering the relevant variables from different files.

A variation on this product appear in Figure 5.12. Here the panel at the left is colored by the gradient and the panel at the right is colored by salinity itself facilitating a side-by-side comparison. Only the cells with a gradient above the threshold are visible; the rest having been removed prior to rendering using the restrict operator. Both Figure 5.11 and Figure 5.12 are animated to show plume development through time.

5.3 EVALUATING GRIDFIELD EXPRESSIONS

We have described the utility of gridfields at two stages of maturity: first as an abstract language for describing and reasoning about the logical structure of gridded datasets, and second as an interactive data product development environment.

Now we describe the various ways gridfield expressions are evaluated.

Gridfield programmers build recipes by calling operator constructors with the appropriate parameters. Although the core data structures and operators are written in C++ for performance, we provide a Python interface to them to facilitate recipe manipulation and rapid development. The techniques described in this section are all implemented in the Python layer.

Consider the following Python syntax of a gridfield algebra expression:

```
1  context = {‘‘year’’:‘‘2006’’, ‘‘day’’:134, ‘‘timestep’’:0}
2  Sh = Scan(context, ‘‘H’’)
3  Sv = Scan(context, ‘‘V’’)
4  C  = Cross(Sh, Sv)
5  R  = Restrict(C, 0, ‘‘z > b’’)
6  B  = Bind(context, ‘‘salt, temp’’, 0, R)
```

Lines 1-5 describe the mesh in Figure 5.1(c), and line 6 binds two attributes to the 0-cells. Line 1 establishes a context identifying particular datasets in the catalog (for which the model and implementation were described in Chapters 2 and 4 respectively). A context is a Python dictionary mapping properties to values. This context is passed to the scan operator to inform the catalog which gridfield named “H” the programmer is interested in. When the scan operator is evaluated, the catalog may construct a filename, execute a query over a relational database, or use any other means available to resolve the name.

After the two scan operators are constructed, they are passed as arguments to the cross operator. After using restrict to remove invalid cells, the salt and temp attributes can be bound. The context object passed to bind need not be the same as the one passed to the scan operators. The CORIE grid changes only infrequently, but many copies of each grid exist on the filesystem. Therefore, it may be appropriate to bind this week’s attributes to last week’s copy of H. The catalog is assumed to be able to resolve these equivalences.

The gridfield expression can be evaluated in two ways: *query mode* and *interactive mode*. In query mode, an application constructs a tree of operators as above and submits the tree to the gridfield server for evaluation. The server is free to transform the recipe prior to and during evaluation. Once evaluated, the results are shipped back to the application.

In interactive mode, the application may request the result of any operator in the tree at any time by calling the `getResult` method. When a result is requested, the operator checks to see if a result has already been computed. If not, the operator calls `getResult` on each of its input operators recursively. If so, the operator checks to see whether the previously computed result is *stale*. The result of operator A is stale if any operator in the subtree rooted at A has been modified since the result was generated. This condition requires keeping track of the time each result is computed as well as a *dirty* flag.

The dirty flag is set whenever the operator's arguments are updated. Most arguments to operators may be changed after initial construction through `SetP` methods, where `P` is the name of the parameter. Interactive mode is similar to the evaluation model of visualization libraries such OpenDX and VTK. Each node in the pipeline is only re-evaluated if it has changed.

Using interactive mode, one can construct a pipeline to display the average salinity for Monday, then repeatedly change the context of the bind operator to see the results for Tuesday, Wednesday, and Thursday.

The result of any operator is only guaranteed valid until a downstream operator is recomputed through a call to `getResult`. This weak guarantee allows aggressive sharing of sets of cells, incidence relations, and attributes. We collectively refer to these three kinds of objects as *components*.

Each physical operator logically creates a new gridfield from a) components of its inputs, and b) new components constructed from scratch. For example,

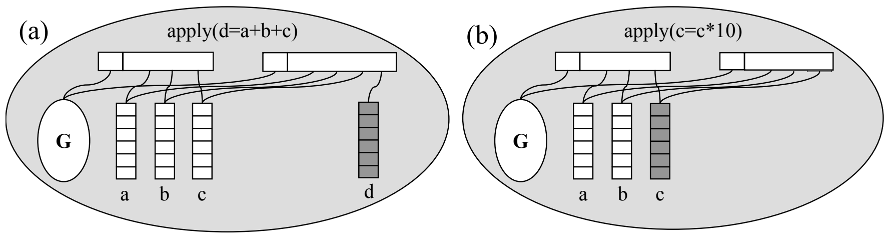


Figure 5.13: Operators (a) can share or (b) destructively update grids and attributes (the latter is done only when safe).

apply(tempC=(tempF-32)*9/5, 0, G) creates a new attribute tempC representing temperature in degrees Celsius. The cell arrays, incidence relationships, and other attributes appear in the result unmodified, while the tempC attribute is new. The restrict operator, however, is an example of an operator that may modify all components of a gridfield.

Each component in the result gridfield may be created according to one of two methods: 1) by allocating a new component, or 2) by destructively updating an input component. To evaluate a restrict operator, for example, cells whose bound attributes do not satisfy the predicate may be removed from the input grid directly, or cells whose attributes do satisfy the predicate may be copied into a newly allocated structure. Figure 5.13 illustrates the distinction between these two methods in the context of the apply operator.

In query mode, the intermediate results of a long gridfield expression are not visible to the application, so the system will try to use destructive update to conserve memory and avoid expensive copying. In interactive mode, the application may hold references to upstream results, so nothing is destructively updated.

The term "mode" suggests that the programmer explicitly declares which mode he or she will be using. In fact, the situation is simple: Execution proceeds in interactive mode except in the following circumstances.

1. The programmer *optimizes* a recipe, or
2. The recipe root is a fetch operator.

To optimize a recipe, the programmer calls the optimize function passing the recipe as the only argument. The return value is a recipe different from the input in two ways. First, the optimize function may have applied algebraic optimizations, and second, the operators, when evaluated, will aggressively share components. The optimized recipe is still composed of operators— as opposed to having been compiled to an object language—but it is considered opaque to the programmer. Preventing all access to the intermediate results in practice is difficult, since Python objects do not enforce encapsulation. The optimized recipe is therefore not truly opaque, but no guarantees are made about the validity of intermediate results.

The fetch operator is used in a recipe as any other operator. A logical no-op, the fetch operator accepts an input operator and a URL. When the `getResult` method is called, fetch marshals its subtree and ships it to a remote server (designated by the URL) for evaluation via the XML-RPC protocol. The gridfield server receives the message, compiles the recipe, evaluates it, and ships the result back to the caller in a packed binary form. Any other fetch operators in the transmitted recipe are evaluated similarly, with the initial server acting as the client. Simple distributed evaluation of gridfield plans is therefore possible. Although the fetch operator has been used manually in this work, the capability admits simple distributed optimization, where fetch operators are injected into a recipe automatically by the system in order to balance load or "search" for data requested but not available locally.

Iterative recipes pose a problem for fetch; as described, the communication of intermediate results is one-way—from the server back to the client. The fetch operator inspects its subtree before sending it to the server—if any operators have been previously evaluated, meaning that their result is already materialized locally, an exception is raised.

5.3.1 Catalog Interface: Scan and Bind

Client programs manipulating gridfields should be guaranteed some measure of safety: Attempts to bind a time-dependent attribute to a spatial grid, for example, should return an error, preferably as early as possible. Attempts to refer to attributes or gridfields that do not exist should also be handled neatly. An RDBMS solves this problem by requiring that one's data be reformatted and inserted into a managed environment before being accessed. Subsequent queries are guaranteed to be "safe." Our goal of a lightweight system enabling access to native data requires that there exist a layer of software that operates in an *unsafe* environment—there is no ultimate authority that can tell us if a particular array can be bound to a particular grid safely.

The catalog provides a safe interface to the data stored in the unsafe environment. The catalog exposes a set of *Base Gridfields* (BGs) against which clients write their recipes. Clients can therefore refer to gridfields and attributes by name rather than by location, achieving a form of data independence. In Chapter 4, we described the tools needed to implement a catalog: a language for extracting arrays from a filesystem for use with scan and bind, and how gridfield expressions could be collected in a schema file to provide BGs.

Base Gridfields are similar to views in relational databases when the base tables are not accessible for security reasons. Safety rather than security is the motivation for encapsulating the source data behind the BG interface.

To ensure a robust BG interface, the bind operator should be used only in the definition of a BG, and not in client recipes. The bind operator identifies an arbitrary attribute and an arbitrary grid, and attempts to bind the former to the latter. To ensure the safety of this operation, the catalog must know which attributes can be bound to which grids. One approach to provide this service would be to maintain a list of (grid, attribute)-pairs, and refer to this list when type checking recipes prior to evaluation. However, the semantics of bind also allow an attribute a of a grid G to be bound to any subgrid of G, G'. Since it is not feasible to explicitly list every possible subgrid of G, we must have some means of detecting when a grid A is a subgrid of a grid B.

One solution is this: Let α be the gridfield recipe whose result we wish to bind an attribute a to, and let β be the gridfield expression whose result the attribute a may be legally bound. We can simply evaluate α and β to produce two gridfields $\mathbf{A}$ and $\mathbf{B}$, respectively, and explicitly check that $\mathbf{A} \subset \mathbf{B}$. There are two problems with this solution. First, the cost of a runtime check of the subgrid relationship is prohibitive, and second, optimization of the recipe is not possible, since each new recipe generated would have to be evaluated to check safety! The alternative is a syntactic check that compares α and β directly to see if α is guaranteed to produce a subgrid of β in every case. Unfortunately, such a test is undecidable in the general case. To see why, consider that we allow arbitrary arithmetic expressions as predicates in the restrict operator. Even two very simple recipes require a test to see if one predicate implies another. For example:

$$\mathbf{A} = \mathsf{restrict}(p, 0, \mathsf{scan}(X))$$

$$\mathbf{B} = \mathsf{restrict}(q, 0, \mathsf{scan}(X))$$

In this example, $\mathbf{A}$ is a subgrid of $\mathbf{B}$ if and only if $p \Rightarrow q$. Since p and q may involve arbitrary arithmetic, the test is undecidable.

Despite the undecidability of the general case, there are many cases in which a subgrid relationship can be determined. In fact, due to the formulation of the gridfield algebra, specifically with regard to the regrid operator, the grid of every intermediate result can be related to the grid of the inputs to a recipe.

Recall that the regrid operator associates a set of cells in a source grid with a each cell in a target grid. The regrid operator often serves the same purpose as a join in the relational algebra: relating two sets on some condition over their elements. However, since it is formulated as a function from target to source, the

Expression ($\mathbf{R} =$)	$\mathsf{grid}(\mathbf{R})$	$\mathsf{sch}_i(\mathbf{R})$	Bounds on $\lvert R_i \rvert$
$\mathsf{bind}(\mathbf{G}, k, a)$	G	$\mathsf{sch}_k(\mathbf{G}) \cup \{(a, \tau)\}$	$\lvert R_i\rvert = \lvert G_i\rvert$
$\mathsf{restrict}(p, i, \mathbf{G})$	$R \subseteq G$	$\mathsf{sch}_i(\mathbf{G})$	$\lvert R_i\rvert < \lvert G_i\rvert$
$\mathbf{A} \otimes \mathbf{B}$	$A \otimes B$	$\mathsf{sch}_i(\mathbf{A})$ $\cup \; \mathsf{sch}_{\mathsf{dim}(R)-i}(\mathbf{B})$	$\lvert R_i\rvert = \sum_{j=0}^{i} \lvert A_j\rvert * \lvert B_{i-j}\rvert$
$\mathsf{merge}(\mathbf{A}, \mathbf{B})$	$A \cap B$	$\mathsf{sch}_i(\mathbf{A}) \cup \mathsf{sch}_i(\mathbf{B})$	$\lvert R_i\rvert < \lvert A_i\rvert \wedge \lvert R_i\rvert < \lvert B_i\rvert$
$\mathsf{regrid}(\mathbf{T}, i, \mathbf{S}, j, m, f)$	T	$\mathsf{sch}_i(\mathbf{T}) \cup t[f]$	$\lvert R_i\rvert = \lvert T_i\rvert$
$\mathbf{A} \cup \mathbf{B}$	$A \cup B$	$\mathsf{sch}_i(\mathbf{A}) = \mathsf{sch}_i(\mathbf{B})$	$\lvert R_i\rvert \leq \lvert A_i\rvert + \lvert B_i\rvert$
$\mathsf{accrete}(s, \mathbf{A}, \mathbf{B})$	$R \subseteq A \cup B$	$\begin{cases} \mathsf{sch}_{s_i}(A), & i = 0 \\ \mathsf{sch}_{s_i}(B), & i > 0 \end{cases}$	$\lvert R_i\rvert < \lvert A_i\rvert + \lvert B_i\rvert$

Table 5.1: Propagation of grid, scheme, and size through the gridfield operators.

result grid is always the same as the input target grid—only the attributes are different. Therefore, the grid of the output is predictable statically.

A prerequisite to optimization (and indeed, any kind of gainful reasoning) is the ability to trace the propagation of properties inductively through each operator. Table 5.1 shows the grid and scheme of the output of each of several operators in the language.

The algebra allows us to define a partial order on the grids of the intermediate results with the $\subset$ relationship. Although we cannot derive a total order, most grids are comparable using the $\subset$ relationship, primarily due to the formulation of regrid. The cardinality of the intermediate results can also be bounded from this relationship, as illustrated in the right-most column of Table 5.1. We estimate the size of a gridfield as the sum of the sizes at each rank, computed as the cardinality at rank i times the *arity* of the tuple in the function's type. The arity of a tuple type is the number of elements in the tuple. The arity of other primitive types is one. More precisely,

$$size((G, k, f)) \cong |G_k| * arity(f)$$

Now, some (assignment function, aggregation function)-pairs produce complex objects rather than primitive values. For example, a frequent idiom is to group all the nodes of a cell together and then generate a polygon object from them for further processing. However, the size of the complex objects are bounded, in practice, by grid quality measures. Constraints on the shape of the cells are reflected in the number of neighboring cells. While certainly possible to define pathological assignment functions, such cases appear to be rare and only affect our ability to estimate the size of intermediate results.

Instead of providing a guarantee of bind safety, then, we propose that bind operators only appear in BG definitions behind the catalog interface. Client recipes written against the BGs are then insulated from the physical characteristics of the source data.

The theme of the BG approach is that data administrators erect a safer interface using the gridfield model instead of bulk loading data into a managed environment or writing wrappers in a separate language outside the model. In a sense, the gridfield language is both a Data Definition Language (DDL) and a Data Manipulation Language (DML). To summarize, accessing unmanaged data is dirty work, but someone has to do it—gridfields are the natural choice.

5.3.2 Optimization in the Presence of bind

With all bind operators appearing behind the BG interface, we have argued that the clients are shielded from a particular source of complexity when writing recipes. The simplest way to evaluate recipes over BGs would be to 1) evaluate all BGs, then 2) evaluate the user recipes.

Recall that relational queries involving views can be optimized as a whole after substituting the view definition in for the view name. Recipes involving BGs should be amenable to similar optimization. In this section, we consider a simple and prolific optimization: pushing restrictions to the leaves of a recipe in the presence of the bind operator.

The bind operator, logically, accepts a function from cells to attribute values. Physically, each function is implemented as an array. Each array can be bound to one dimension of its *natural* grid; namely, the grid to which it is bound in the BG definition in which it appears. When associating an array with its natural grid G, bind may be implemented as a "zip" of an array of cells with an array of values. When binding an array to a subgrid of G, bind must use a more complicated algorithm.

For example, consider Figure 5.14. Gridfield **G** uses the natural grid of the attribute x. In Figure 5.14(a), the cells of G_0 are aligned with the values of x, and the bind operator can implement the simple "zip" algorithm. However, if we commute a restrict operator through the bind, we can no longer perform a "zip" since we must skip over the missing cells. Of course, the problem is easily remedied in this case: Simply record the ordinal positions of the cells before computing the restrict, then have the bind operator use the ordinal positions to implement a "merge" algorithm. In general, once the ordinal positions are materialized, we can safely bind an attribute x to a subgrid of its natural grid G.

All logical bind operators appearing in a BG are instances of the physical zipbind operator. The requirements to use zipbind are that the attribute values are a) one-to-one with the cells of a grid G to which they will be bound and b) arranged in the correct order. When binding to a subgrid of G, condition a) is violated, and we must use the physical mergebind operator. The relevant rewrite rules are as follows.

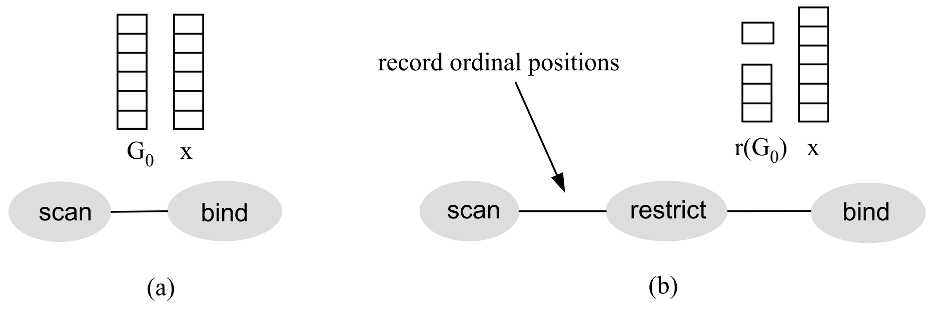


Figure 5.14: An array can be bound to its "natural" grid gridfield bound.

$$\textsc{ZipMerge}\ \frac{\mathsf{zipbind}(k, v, \mathbf{G}) \qquad (pos, \texttt{int}) \in \mathsf{sch}_k(\mathbf{G}) \qquad \forall c \in G_k,\ pos(c) \geq 0 \text{ and } pos(c) < |G_k|}{\mathsf{mergebind}(d, v, pos, \mathbf{G})}$$

$$\textsc{RestrictBind}\ \frac{\mathsf{restrict}(p, i, \mathsf{mergebind}(k, v, pos, \mathbf{G})) \qquad p : \tau \to \mathsf{Bool} \qquad v \notin \tau}{\mathsf{mergebind}(k, v, pos, \mathsf{restrict}(p, i, \mathbf{G}))}$$

The ZipMerge rule performs a bounds check on the arrays and ensures that the *pos* attribute is of the correct type (`int`). If these conditions are met, then the (slower) mergebind can be substituted for the (faster) zipbind. Once the zipbind is replaced with an instance of mergebind, a restrict operator can be safely commuted through it using the RestrictBind rule.

In some cases, the ordinal positions of the cells of an operator's result can be computed from static information about its input. In such cases, subgrid-inducing operators (restrict and merge) can be commuted through bind, frequently resulting in a more efficient recipe.

In Table 5.1, all the operators produce an equal grid or a subgrid of one of their inputs except for cross and disjoint union. For these two operators, we must provide an explicit calculation of the ordinal positions in the output using information about the inputs.

Cross Product Ordinals

Consider a recipe of the form $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$. For each cell c in G_k, there is a cell a in the grid A and a cell b in the grid B such that $c = (a, b)$. The position of c in G_k can be expressed as follows.

$$pos_C(c) = \sum_{j=0}^{k-1} |A_j||B_{k-1-j}| + pos_A(a)|B_{k-1-\mathsf{dim}(a)}| + pos_B(b) \tag{5.1}$$

where $pos_X(x)$ is the position of a k-cell x in X_k.

In addition to the cardinality of of the inputs grids at different ranks, this expression involves the dimension of a, the position of a in $A_{dim(a)}$, and the position of b in $B_{k-\mathsf{dim}(a)}$. All three of these values can be recorded explicitly prior to the cross. The dimension can be recorded and stored efficiently at each rank k with a constant-valued attribute. These values are carried through the cross product operation and are available as attributes of c. In some cases, the dimension may not be necessary: If $k = 0$, then $\mathsf{dim}(a) = 0$, and if $k = \mathsf{dim}(A) + \mathsf{dim}(B)$ then $\mathsf{dim}(a) = \mathsf{dim}(A)$.

The extra attributes required to calculate and store the position information can be computed with gridfield operators, as shown in rewrite rule CrossCalc. The extra attributes can be hidden from the client using the $\mathsf{project}$ operator, if necessary.

CrossCalc

$$\frac{\mathbf{G} = \mathbf{A} \otimes \mathbf{B}}{\mathbf{G} = \mathsf{apply}(pos_C = f(pos_A, pos_B), k, A' \otimes B')}$$

where $f(pos_A, pos_B)$ is the right-hand side of Equation 5.1, $A' = \mathsf{apply}(pos_A = id, k, \mathbf{A})$, $B' = \mathsf{apply}(pos_B = id, k, \mathbf{B})$, and id is a distinguished, implicit attribute holding a cell's position in the current grid.

The result is that we can push restrict operators through bind, and then through a cross operator in order to evaluate them as close to the leaves of a recipe as possible. The BGs can be defined at design time to ensure the safety of the bind operator, but user restrictions can be exploited to reduce the size of intermediate results and improve performance.

Union

The ordinal positions of cells in the union of two grids $C = A \cup B$ cannot be expressed in terms of the positions of cells in A and the positions of cells in B unless A and B are known to be disjoint. Consider a recipe of the form $\mathbf{C} = \mathbf{A} \sqcup \mathbf{B}$, where $\sqcup$ represents the disjoint union operator. The calculation of the ordinal positions in the result is:

$$pos_C(c) = \begin{cases} pos_A(a) & \text{if } c \in A_k; \\ |A_k| + pos_B(b) & \text{otherwise.} \end{cases} \tag{5.2}$$

Equation 5.2 involves the positions of the cells in each argument, as with cross, as well as a means of discriminating between cells originating in A and cells originating in B. A key representing the originating gridfield can be recorded as a constant attribute using the apply operator.

DUnionCalc

$$\frac{\mathbf{G} = \mathbf{A} \sqcup \mathbf{B}}{\mathbf{G} = \mathsf{apply}(pos_C = g(pos_A, pos_B), k, \mathsf{apply}(key = 0, k, \mathbf{A}) \sqcup \mathsf{apply}(key = 1, k, \mathbf{B}))}$$

where $g(pos_A, pos_B)$ is the right-hand side of Equation 5.2, and id is as before. The expressions $key = 0$ and $key = 1$ are passed to the apply operator to indicate a new attribute called *key* with a constant value should be allocated. In the union gridfield, the *key* attribute indicates the source of each k-cell—**A** or **B**. With this information, we can use Equation 5.2 to compute the appropriate positions.

If a BG is defined with only cross, bind, and regrid, then a restrict operator applied at the root can be pushed down through the recipe limited only by the attributes mentioned in the predicate. If a BG definition involves other operators, the calculation of ordinal positions needed by bind may still be possible, but cannot necessarily be derived automatically and must be provided by the BG author.

5.3.3 Animation and Parallelism

The multidimensional scientific datasets we have encountered can be assembled in the gridfield algebra with a recipe of the form $R(A \otimes B \otimes \dots)$, where R is a recipe involving restrictions, binds, merges, or regrids. For example, an animation of the Earthquake in the San Fernando valley [ABB+03] is the cross product of a 0-D gridfield **T** representing timesteps, and a 3-D gridfield **G** representing the rock and soil in the area. The displacement of the earth, a time-dependent attribute is expressed as a bind operation on this 4-D gridfield.

$$\mathsf{bind}(0, \texttt{disp}, \mathbf{T} \otimes \mathbf{G})$$

This formulation is easy to express, but direct evaluation would be inefficient. A large 4-D intermediate result must be constructed, only to be iterated over by the client to produce the desired animation.

As an alternative, we allow a cross operator to be evaluated iteratively to improve performance. Recall that all cells in a homogeneous grid are incident to a cell of maximal dimension. A homogeneous grid can be expressed as a union of closure grids as follows:

$$\textsc{Decompose}\ \frac{G \text{ is homogeneous}}{G = \bigcup_{c \in G_{\mathsf{dim}(G)}} (\overline{c}_G)}$$

where $\overline{c}_G$ is the closure grid of a d-cell c and $d = \mathsf{dim}(G)$; that is, the grid formed from the cell c and all cells incident to c. This rule can be lifted to gridfields: the data bound to a cell in $\overline{c}_G$ is just the data bound to the same cell in $\mathbf{G}$.

There are many ways we might decompose a grid into a union of smaller grids, but this particular decomposition is remarkably easy to implement in many practical cases. If $\mathbf{G}$ is a 0-D gridfield, then Decompose represents a partition of the grid into $|G_0|$ grids with one 0-cell each, which we can implement by simply iterating over the d-cells, where $d = 0$. If $\mathbf{G}$ is represented using ICS (see Chapter 3) and $\mathsf{dim}(\mathbf{G}) = d$, then each d-cell is represented as a sequence of nodes, and the intermediate cells are represented implicitly. Therefore, given a d-cell c, $\overline{c}_G$ is immediately available. We can implement the decomposition just by iterating over the d-cells.

The other gridfield operators commute with this decomposition, subject to some restrictions.

$$\textsc{DecLeft}\ \frac{\mathbf{G} = \mathbf{A} \otimes \mathbf{B} \qquad \mathbf{A} \text{ is homogeneous}}{\mathbf{G} = \bigcup_{a \in A_{\mathsf{dim}(A)}} (\bar{a} \otimes \mathbf{B})}$$

$$\textsc{DecRight}\ \frac{\mathbf{G} = \mathbf{A} \otimes \mathbf{B} \qquad \mathbf{B} \text{ is homogeneous}}{\mathbf{G} = \bigcup_{b \in B_{\mathsf{dim}(B)}} (\mathbf{A} \otimes \bar{b})}$$

$$\textsc{DecBind}\ \frac{\mathbf{G} = \mathsf{bind}(a, i, \mathbf{A}) \qquad \mathbf{A} \text{ is homogeneous}}{\mathbf{G} = \displaystyle\bigcup_{a \in A_{\mathsf{dim}(A)}} \mathsf{bind}(a, i, \bar{a})}$$

$$\textsc{DecRestrict}\ \frac{\mathbf{G} = \mathsf{restrict}(p, i, \mathbf{A}) \qquad \mathbf{A} \text{ is homogeneous}}{\mathbf{G} = \displaystyle\bigcup_{a \in A_{\mathsf{dim}(A)}} \mathsf{restrict}(p, i, \bar{a})}$$

$$\textsc{DecMerge}\ \frac{\mathbf{G} = \mathsf{merge}(\mathbf{A}, \mathbf{B}) \qquad \mathbf{A} \text{ is homogeneous}}{\mathbf{G} = \displaystyle\bigcup_{a \in A_{\mathsf{dim}(A)}} \mathsf{merge}(\bar{a}, \mathbf{B})}$$

$$\textsc{DecAccrete}\ \frac{\begin{array}{c}\mathbf{G} = \mathsf{accrete}((x_0, x_i, \ldots, x_n), \mathbf{A}, \mathbf{B}) \\ \text{if } j > i \text{ then } x_j < x_i \qquad \mathbf{A} \text{ is homogeneous}\end{array}}{\mathbf{G} = \displaystyle\bigcup_{a \in A_{\mathsf{dim}(A)}} \mathsf{accrete}((x_0, x_1, \ldots, x_n), \bar{a}, \mathbf{B})}$$

$$\textsc{DecTarget}\ \frac{\begin{array}{c}\mathbf{G} = \mathsf{regrid}(\mathbf{E}, i, \mathbf{F}, j, f_{\mathsf{assign}}, g_{\mathsf{aggregate}}) \\ i = \mathsf{dim}(E) \qquad \mathbf{E} \text{ is homogeneous}\end{array}}{\mathbf{G} = \displaystyle\bigcup_{x \in E_i} \mathsf{regrid}(\bar{x}, i, \mathbf{F}, j, f_{\mathsf{assign}}, g_{\mathsf{aggregate}})}$$

$$\textsc{DecSource}\ \frac{\begin{array}{c}\mathbf{G} = \mathsf{regrid}(\mathbf{E}, i, \mathbf{F}, j, f_{\mathsf{assign}}, g_{\mathsf{aggregate}}) \\ i = \mathsf{dim}(E) \qquad j = \mathsf{dim}(F) \qquad \mathbf{E}, \mathbf{F} \text{ are homogeneous}\end{array}}{\mathbf{G} = \displaystyle\bigcup_{x \in E_i} \mathsf{regrid}(\overline{x}_E, i, \overline{f_{\mathsf{assign}}(x)}_F, j, f_{\mathsf{assign}}, g_{\mathsf{aggregate}})}$$

The cross, bind, restrict, and merge operators naturally commute with decomposition, as captured in rules DecLeft, DecRight, DecBind, DecRestrict, DecMerge. The accrete operator only commutes with union if the stencil is a subset of the closure of a d-cell, for example 20 or 321 for a 3-D gridfield (Rule DecAccrete). This condition is met when the sequence of integers defining the stencil is monotonically decreasing. The regrid operator will also commute with decomposition if the decomposed gridfield is the target (Rule DecTarget). If the decomposed gridfield is the source, then commutativity requires that the cells returned by the assignment function f must appear in the source grid, which we indicate with a slight abuse of the closure grid notation extended to a set of cells rather than an individual cell (Rule DecSource). Further, decomposing the source without decomposing the target would not admit an iterative evaluation strategy, so the rule decomposes both.

Decomposition is used to break a recipe into independent computations. Used internally on the server, decomposition offers flexibility in the physical evaluation: Each computation can be evaluated in a separate thread to hide I/O or on a separate machine for parallelism.

If every operator in a user's recipe commutes with decomposition, then the results can be streamed back to the client, one piece at a time. However, how can the user indicate which gridfields are to be decomposed and which gridfields are to be collected whole? As an alternative to the compile function, we provide a iterate function that accepts two arguments: a set of gridfield names and a recipe involving those names. For example, the call

$$\mathsf{iterate}(\{\text{“}\mathbf{T}\text{”}\}, \mathsf{bind}(0, \mathtt{disp}, \mathbf{T} \otimes \mathbf{G})$$

indicates that the gridfield $\mathbf{T}$ is to be decomposed. If all the operators in the recipe commute with the decomposition, then the results can be returned efficiently, one piece at a time. The user simply calls `getResult` repeatedly. If the recipe does not commute with decomposition, then the full cross product will need to be constructed and decomposed in a separate step.

Besides being used to stream results back to the client, the iterate operator can be used to evaluate aggregations over a product gridfield efficiently. Consider the following recipe:

$$\mathsf{regrid}(\mathbf{A}, i, \mathsf{bind}(0, \mathsf{disp}, \mathbf{A} \otimes \mathbf{B}), i, (\lambda x \,.\, x \otimes \mathbf{B}), \mathsf{sum})$$

If **A** and **B** represent rows and columns of a matrix, then this expression sums the values in each row. Rather than compute the recipe in the order given, we can use the rule DecTarget to produce a more efficient iterative recipe (if the rule's conditions are met):

$$\bigcup_{c \in A_i} \mathsf{regrid}(\bar{c}, i, \mathsf{bind}(0, \mathsf{disp}, \bar{c} \otimes \mathbf{B}), j, (\lambda x \,.\, x \otimes \mathbf{B}), \mathsf{sum})$$

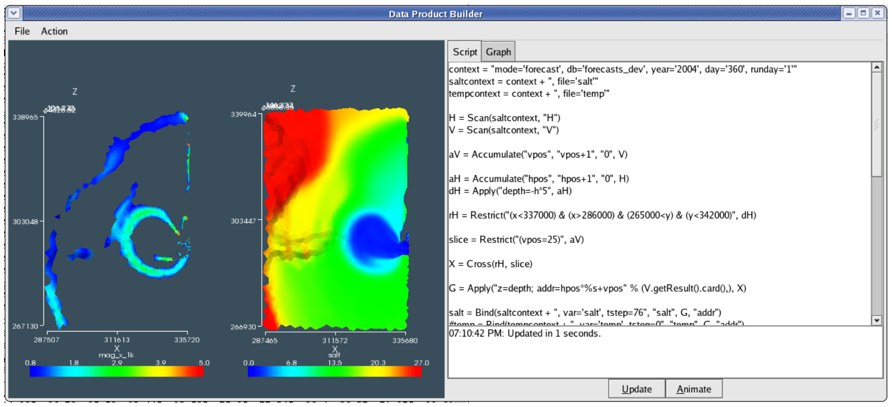
This expression divides the target grid **A** into a set of smaller grids, one for each cell $c \in A$. With this recipe, each portion of the decomposed grid can be evaluated independently.

In summary, decomposition provides semantics for the efficient iterative recipes used in aggregation, animation, and parallel evaluation.

5.3.4 Conclusion: A Canonical Gridfield Application

The evolution of the gridfield model and implementation led to the release of a standard gridfield visualization application for building data products, DPBuilder. Figure 5.15 shows a screenshot of DPBuilder. The image on the left shows the current data product, as rendered by translating a gridfield into a a VTK object. On the right is the script window in which gridfield recipes are written. At any time, the user can view the current plan graphically by clicking the "Graph" tab. This tab displays the current recipe in the style of Figure 5.5. We consider graphical editing of the recipes as future work.

As the user writes a gridfield recipe, he or she can inspect intermediate results by calling the inspect function and passing it a list of gridfields. This function



```
context = "mode='forecast', db='forecasts_dev', year='2004', day='360', runday='1'"
saltcontext = context + ", file='salt'"
tempcontext = context + ", file='temp'"

H = Scan(saltcontext, "H")
V = Scan(saltcontext, "V")

aV = Accumulate("vpos", "vpos+1", "0", V)

aH = Accumulate("hpos", "hpos+1", "0", H)
dH = Apply("depth=-h*5", aH)

rH = Restrict("(x<337000) & (x>286000) & (265000<y) & (y<342000)", dH)

slice = Restrict("(vpos=25)", aV)

X = Cross(rH, slice)

G = Apply("z=depth; addr=hpos*%s+vpos" % (V.getResult().card(),), X)

salt = Bind(saltcontext + ", var='salt', tstep=76", "salt", G, "addr")
#temp = Bind(tempcontext + ", var='temp', tstep=0", "temp", G, "addr")
```

```
07:10:42 PM: Updated in 1 seconds.
```

Figure 5.15: A screenshot from a prototypical gridfield application, DPBuilder. Gridfield expressions written at right drive the visualization at left.

---

translates each gridfield to a VTK object and renders it. To encourage its use as a debugging tool, the `inspect` function uses heuristics to derive a suitable visualization automatically based on the dimension of the grid and the attributes available. For example, attributes named x, y, or z are interpreted geometrically. Other attributes may be interpreted geometrically if these attributes are not available. The most recently added attribute is mapped to color, under the assumption that the most recent calculation is the motivation for calling `inspect`. Gridfields of dimension 0 are translated to a cloud of points in VTK, rendered as circular glyphs. Vector attributes, either identified heuristically by name or specified explicitly by the user, are used to produce arrow glyphs as in Figure 5.3.

When multiple gridfields are passed to `inspect`, by default, they are superimposed as in Figure 5.4. The user may also specify an on-screen layout as in Figure 5.15 with a list of coordinate pairs the same length as the list of gridfields; e.g., use the user would pass [(0,0), (0,1)] to generate the layout in Figure 5.15, or [(0,0),(1,0)] to arrange the two images in a single column. Arbitrary drag-and-drop placement of gridfield visualizations is a feature left as future work.

The recipes may be evaluated locally, or they may be shipped to a server using

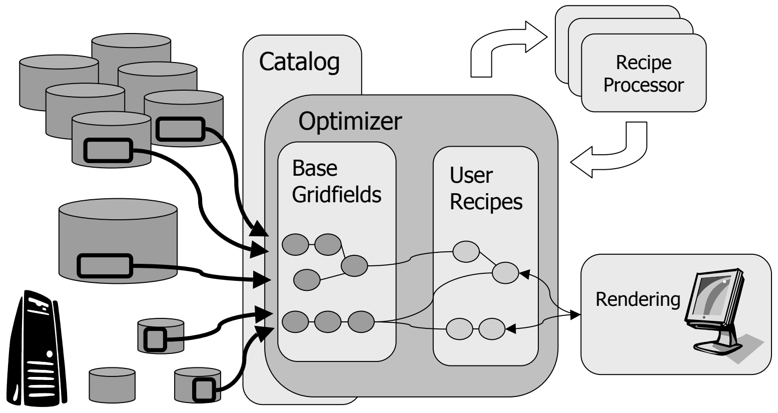


Figure 5.16: The server-side architecture supporting DPBuilder. User recipes are written against Base Gridfields and can be optimized as a whole. Currently, most optimizations are encoded explicitly within the BG definitions.

---

the fetch operator. The server architecture is shown in Figure 5.16. The BGs may reference data stored in a variety of sources (the disk icons at the left of Figure 5.16). User recipes reference BGs defined on the server. The BGs are then dereferenced during evaluation and the recipe can be optimized as a whole. The only optimization rule that is applied automatically is to push the restrict operator down toward the leaves. Decomposition applied at the server must be manually specified within a BG definition. When a gridfield is decomposed, each independent computation can be evaluated in a separate thread, or potentially a separate computer, illustrated by the multiple boxes in the upper right of Figure 5.16. Each of these processes sends their data back to the client.

In this chapter, we have demonstrated the conceptual and practical utility of the gridfield model by applying it to various problems of expression, evaluation, and optimization. We have also described an architecture for deploying gridfields in a lightweight manner, allowing multiple modes of programming. Finally, we briefly described a prototype application powered by gridfields.

Chapter 6

EXPERIMENTAL EVALUATION

We position gridfields between personal visualization applications and monolithic database solutions and evaluate expressiveness and performance relative to each. We find visualization applications too highly specialized; we say they are *algorithm-oriented* rather than model-oriented. Consequently, they provide excellent performance for specialized tasks, but obscure logical similarities and thwart optimization. Relational databases are model-oriented, but exhibit a significant barrier to entry with respect to schema design, data loading, and performance tuning. Remarkably, we do not find the resulting performance particularly good, even if this effort is invested.

Figure 6.1 illustrates our expectation of how scientific simulation applications evolve from a single researcher's desktop experiments to a project involving many users, many variations of the core model, and many applications. Over time, more and more users become interested in running simulations and analyzing results. The simulations themselves may generate new kinds of grids (Chapter 5), or new kinds of grids may appear as intermediate results during the analysis phase. Gridfields are general enough to accommodate this increase in diversity, but lightweight enough to integrate with users' favorite tools. We say that gridfields offer a smoother return on investment curve than heavyweight, transformative solutions such as relational databases [HMB07].

When developing a new model of a physical process, a personal visualization application is sufficient: The developer scrutinizes each result before adjusting parameters, working out bugs, and repeating the run. As the code matures, more

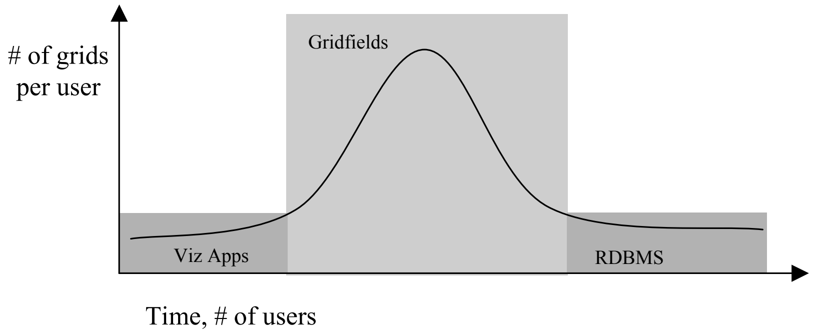


Figure 6.1: As the number of users increases over time, the number of grids to manipulate increases beyond what can be comfortably handled with a desktop visualization application. For a lasting, successful project, the number of new users will overtake the number of new grids, and the costs associated with a comprehensive relational database solution may then be justifiable. Gridfields bridge the gap between personal visualization applications and comprehensive database solutions.

---

potential applications emerge, and results are generated at a faster rate. Interest in perfect results for a narrowly defined problem shifts to interest in "coverage" of a class of related problems: ("Now that we have successfully modeled the fetal heart at 10 weeks, we need to investigate all stages of growth.") The rate of dataset production increases beyond the capacity of one-file-at-a-time tools. The analysis of individual datasets is extended with comparisons between datasets. Empirical data, usually in a different format, are compared with simulation results. These new uses require logical and physical changes to the data inventory, as described in Chapter 5.

Consider whether desktop visualization applications accommodate this increasing grid diversity. In Chapter 3, we described how the Visualization Toolkit (VTK) is organized by algorithm, each encoded as a C++ class inheriting from an abstract "Filter" class. The consequence is that different types of grid tend to require different algorithms and data structures. We gave a concrete example: 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 regular, rectilinear grids and computes a subgrid using a range of indices for each dimension, as opposed to geometric information or other data. The operation `vtkExtractGeometry` works on a wider range of datasets, but defines regions in terms of geometric functions rather than topological connectivity. A more efficient version, `vtkExtractPolyDataGeometry`, is available for polygonal data. Another operator, `vtkThreshold` filters grids based on non-geometric attributes.

Gridfields provide a uniform way of describing and manipulating different types of grids. As the number of grids and grid types increases over time, the number of algorithms and data structures used to manage them increases as well. The conceptual economy of the gridfield model becomes increasingly attractive.

Over time, the requirements for the simulation application may converge, and the number of new users grows faster than the number of new grids. A few important and popular grids (corresponding to a few important and popular types of experiment) may emerge as candidates for implementation in a comprehensive database solution [HG05, KaC$^+$01]. However, there are some significant obstacles to overcome before a high-performance database-backed visualization system can be realized.

First, either the model program must be rewired to insert results directly into the database, or the results must be loaded into the database before query and subsequent visualization is possible. Large load operations are rather expensive with modern relational databases, perhaps since they are optimized for Online Transaction Processing (OLTP) traffic. Systems designed for Extract, Transform and Load (ETL) operations offer improvement, but are rather hardware dependent. For example, UNISYS and SAS set ETL benchmarking records in 2005 by loading 324M tuples (77.2GB) in 3 hours—on a platform with 64GB of RAM, 16 processors, and 8 2GB/s Fibre Channel HBAs accessing a pair of storage appliances each with 8GB of cache and 45 drives operating at 15k RPM [etl05]. This system was obviously designed with high-performance I/O in mind. Second, good performance once the data is loaded is not guaranteed, as it depends crucially on the interaction between specialized hardware, physical database design, and database configuration settings, as we will discuss. Third, although the number of grids in use stabilizes, the absolute number of grids may still be high. A fixed database schema that can accommodate many different grid types can be elusive.

In this chapter, we present experimental evidence that gridfields are the best choice for the "adolescent" simulation applications characterized by grid size and diversity. We will justify this claim by analyzing the performance of gridfield recipes for different types of grids, each with millions of cells. Specifically, we consider highly-regular grids that admit specialized algorithms, partially-regular grids from the CORIE system, and irregular grids from the medical and geological domains.

All of the experiments were run on a Linux i386 2.4 GHz processor with 2GB of RAM available. This machine is nearly identical to one node of the cluster on which the CORIE simulations are executed.

6.1 HIGHLY REGULAR DATASETS

We test a restrict-like task with both VTK and gridfields on a variety of datasets. These datasets are highly regular—they consist solely of rectilinear cells of homogeneous dimension. In VTK, these datasets may be encoded using the class `vtkStructuredPoints`. The topology and geometry of instances of this class are entirely implicit; the programmer need only specify an origin vector, a spacing vector, and a dimensions vector to determine the geometry and the topology. This implicit representation leads to very efficient manipulation for some tasks.

Gridfields are designed to accommodate all types of grids, but the paucity of tools for handling unstructured grids was a primary motivation. Using gridfields, a structured grid is expressed as the product of simpler grids. The `cross` operator, when evaluated, eagerly generates the full grid and attributes specified by the model. An alternative would be a "lazy" evaluation, where the gridfield produced would be an instance of a special ProductGrid class. The significant cost of evaluating the product operator would be postponed, but not necessarily eliminated. Lazy evaluation only improves performance if the postponed work can be ultimately avoided.

VTK assumes that most tasks will not require each cell to be explicitly defined. For example, to inspect the internal structure of a 3-D volume, we may compute an *isosurface* for a few different values. For each cell through which the isosurface passes, a properly oriented polygon is appended to the output. The edges, faces, and cells of the input need not be materialized. To visualize the wireframe mesh, the edges of the input need to be computed, but the faces and 3-cells do not. Volume rendering techniques assign opacity values to each cell in the volume and add up their contributions to produce the final image. These techniques may require explicit 3-cells, but not necessarily explicit edges or faces.

In VTK, a separate filter class exists for each of these cases. To view the wireframe mesh, the programmer uses the class `vtkExtractEdges`, instances of which accept any dataset as input and produce a collection of edges as output—an instance of `vtkPolyData`.

Our approach is to express these different cases algebraically, emphasizing their similarities. For example, each instance of `vtkStructuredPoints` can be expressed as the product of gridfields:

$$\mathbf{R} = \mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C})$$

To visualize the 1-cells as a mesh, we want to render the edges but suppress the faces and 3-cells. In VTK, we would use `vtkExtractEdges`; in the gridfield algebra, we use the accrete operator:

$$\begin{aligned}\mathbf{X} &= \mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C}) \\ \mathbf{R} &= \mathsf{accrete}(1, \mathbf{X}) \end{aligned} \tag{6.1}$$

This expression applies the simple stencil 1 to indicate that the output should include only 1-cells. The cross operator, however, is fully evaluated even though we are only interested in the 1-cells. An equivalent but perhaps less expensive formulation is

$$\begin{aligned}\mathbf{X} &= \mathsf{accrete}(10, \mathbf{A} \otimes \mathbf{B}) \\ \mathbf{R} &= \mathsf{accrete}(1, \mathbf{X} \otimes \mathbf{C}) \end{aligned} \tag{6.2}$$

The accrete operator is used in manner reminiscent of the relational project operator: the complete dataset is projected onto the portion involving the 1-cells. This pattern appears frequently enough that we allow the cross operator to be programmed to avoid computing cells at particular dimensions. We refer to this implementation of cross as *projecting* cross. Let $\mathbf{A} \otimes_s \mathbf{B} = \mathsf{accrete}(s, \mathbf{A} \otimes \mathbf{B})$. The new expression is

$$\mathbf{R} = (\mathbf{A} \otimes_{10} \mathbf{B}) \otimes_1 \mathbf{C} \tag{6.3}$$

This expression requires fewer cells and incidence relationships to be allocated and is therefore cheaper.

Sample visualizations and sizes of the datasets tested appear in Figure 6.2; they will be referred to in the text by the labels assigned in the figure. *Carotid* is an MRI scan of a carotid artery [vtk]. *Fuel* is a simulation of fuel injection into a combustion chamber [volb]. *IronProt* is a simulation of the electrostatic potential in an iron-related protein [vtk]. *Hydrogen* is a simulation of the electron probability

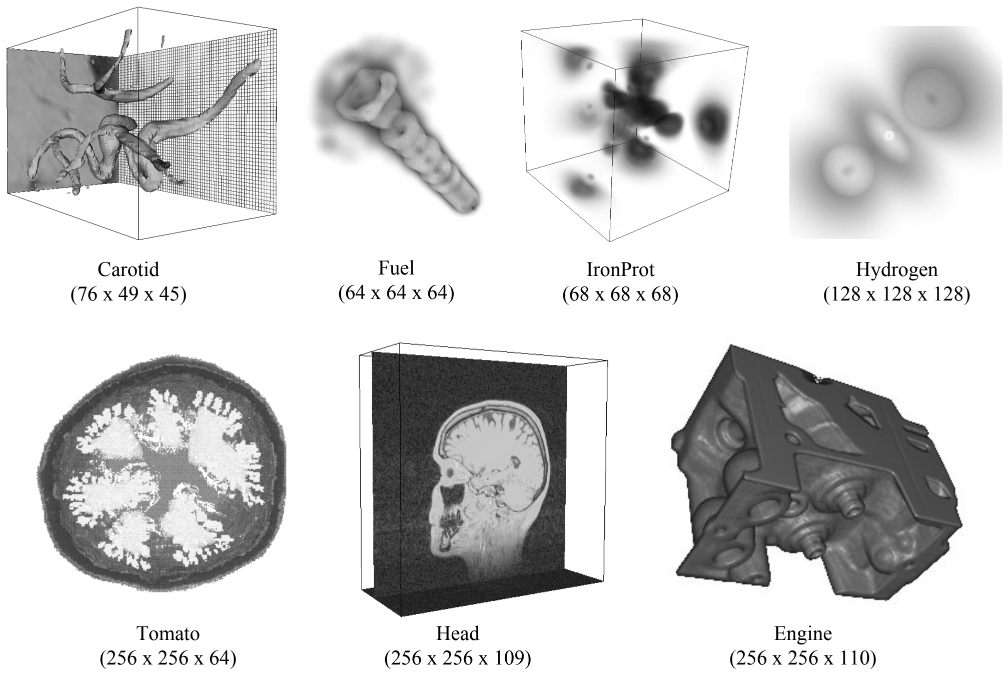


Figure 6.2: Seven sample datasets, each formed as the cross product of 3 linear grids.

---

function around a hydrogen atom [volb]. *Tomato* is a CT scan of a tomato [who]. *Head* is an MRI scan of a human head taken on the Siemens Magnetom [unc]. *Engine* is a model of an engine block provided by General Electric [vola].

In Figure 6.3, we compare the performance of the `vtkExtractEdges` filter against the expressions in Equations 6.1 (AccreteCross), 6.2 (CrossAccrete), and 6.3 (FastCross).

The gridfield formulations of the task are more efficient than the special VTK class `vtkExtractEdges`. The output of `vtkExtractEdges` is an instance of `vtkPolyData`, a class designed to hold any combination of points, lines, polygons, and polyhedra. The insertion of millions of edges into this structure one at a time is slower than our block allocation of memory inside the cross operator. Knowledge of the algebraic structure of the dataset exposes the resources required to create and store the cells and topology. This knowledge translates into a performance

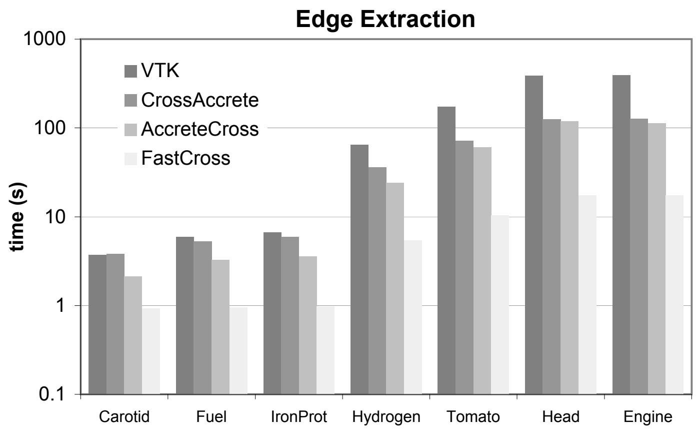


Figure 6.3: A comparison of three gridfield recipes and a specialized VTK algorithm for extracting the edges of a mesh. The y-axis is log-scaled. VTK requires the use of a special single-purpose class to perform this task (`vtkExtractEdges`) but performs worse than the gridfield expressions on all datasets except Carotid.

---

advantage for this task.

What of the other kinds of tasks introduced at the start of this discussion—those that do not require explicit enumeration of cells? Consider a simple operation to extract a rectilinear "subslab" of the 3-D dataset by providing a bounding box of the form $(x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max})$. A subslab of a `vtkStructuredPoints` is another instance of `vtkStructuredPoints`; no cells need to be explicitly constructed. The implicit representation of `vtkStructuredPoints`, corresponding to a lazy evaluation of the cross operator, should perform better in this case. Simply put, it does. The benefits are illustrated by the results in Figure 6.4. However, these results also reveal that the performance of the explicit, eagerly-evaluated gridfield approach is sensitive to algebraic optimizations.

We use three gridfield formulations of the subslab task. In each case, we push the restrict operators through the cross operator, as permitted by the algebra. The formulation using this optimization alone is labeled "gffull" in Figure 6.4.

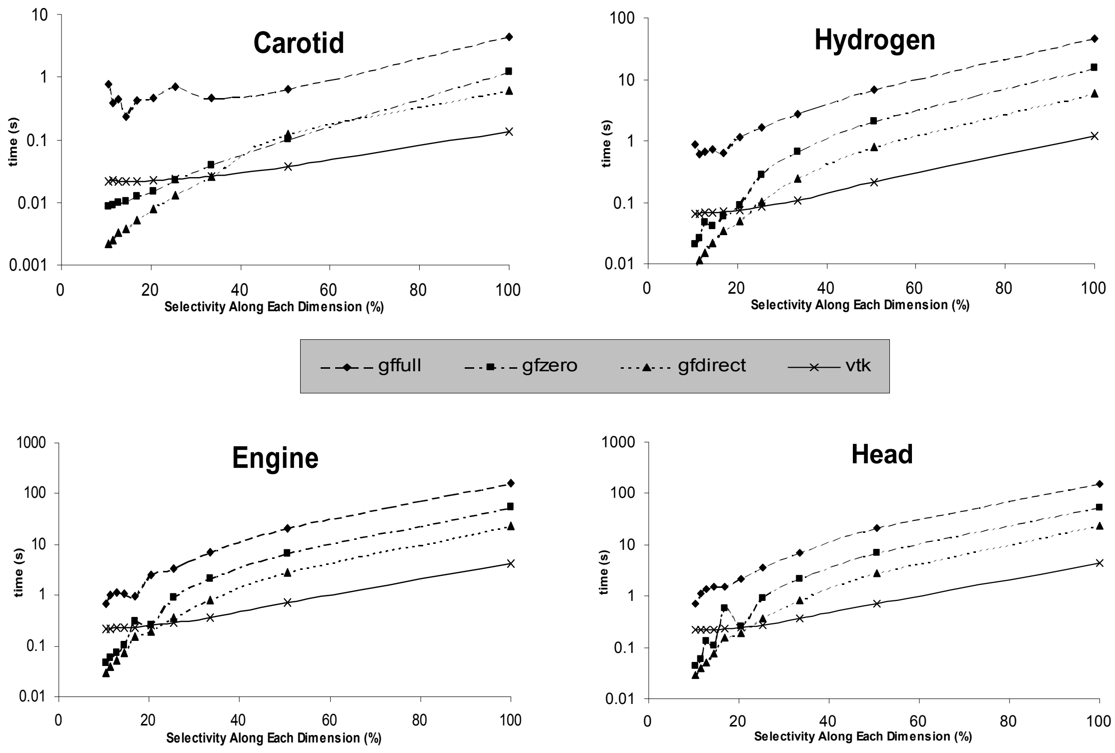


Figure 6.4: Completion time for a "subslab" operation, where each dimension is restricted by the same percentage. The result size is thus proportional to $(\frac{x}{100})^3$, where x is the selectivity on the x-axis. The vertical axis is log scaled: The fully implicit representation of VTK is far superior for this task, though gridfield optimizations make a significant difference.

The results labeled "gfzero" uses projecting cross to construct only the 0-cells. This formulation is useful if the topology is unimportant: perhaps the result is to be converted to a set of parallel multidimensional arrays, an instance of `vtkStructuredPoints` for further visualization, or just a cloud of points for storage in a relational database. The results labeled "gfdirect" were generated without the use of the cross operator. Consider that a 0-D grid G constructed as the product of 3 0-D grids $A \otimes B \otimes C$ is still just a 0-D grid. We can therefore remove the cross operator and produce the result grid directly as a set of $|A_0||B_0||C_0|$ 0-cells. Further, we can produce any subslab $A' \otimes B' \otimes C'$ of the original product grid since we know the sizes of each dimension $|A'|$, $|B'|$, and $|C'|$. To bind attributes to the subslab, however, we have more work to do: For each cell in the subslab, we must carefully calculate its address in the original grid G.

Let $G = X_{(0)} \otimes X_{(1)} \otimes \ldots X_{(N)}$ denote the product of N 0-D grids (parenthesized subscripts are used since X_i notation was defined to to the the set of i-cells in the grid X). Let $([a_0, b_0), [a_1, b_1), \ldots, [a_{N-1}, b_{N-1}))$ denote the bounding box of the N-D subslab of G, where $0 \leq a_i < |X_i|$ and $0 \leq b_i < |X_i|$, and $a_i < b_i$. Let x be the address of an element in the subslab's coordinates and $f(x)$ be the address in G. Finally, assume that the dimensions are listed in decreasing order of their rate of change with respect to address; i.e. listing rows before columns implies a row-major format. Then $f(x)$ is

$$f(x) = \sum_{d=0}^{N-1} \left[\left(\prod_{i=d+1}^{N-1} |X_i| \right) \left(\left\lfloor \frac{x \mod \prod_{i=d}^{N-1} b_i - a_i}{\prod_{i=d+1}^{N-1} b_i - a_i} \right\rfloor + a_d \right) \right] \tag{6.4}$$

where $x > y$ implies $\sum_x^y f(i) = 0$ and $\prod_x^y f(i) = 1$.

For example, consider a 0-D grid $G = X \otimes Y$, where $|X_0| = 3$ and $|Y_0| = 4$. Let $([0, 2), [2, 3))$ be the bounding box of the subslab of interest, as in Figure 6.5. For a position x in the subslab, the corresponding position in the original grid is

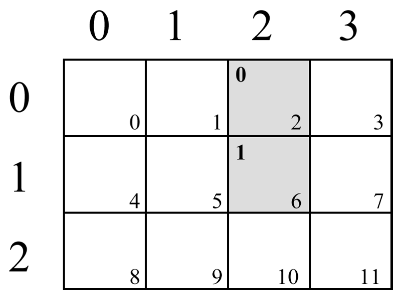


Figure 6.5: The address of a cell in the original grid (the numbers in the lower right of each cell) can be calculated from its address in the subslab (the boldface numbers in the upper left of each shaded cell) and the subslab's bounding box.

$$
\begin{aligned}
f(x) &= \left[(4)\left(\frac{x \mod (2-0)(3-2)}{(3-2)} + 0\right)\right] + [x \mod (3-2) + 2] \\
&= 4(x \mod 2)) + x + 2
\end{aligned}
$$

Although this expression allows direct computation of the positions of the cells and obviates the need to evaluate a cross operator, it is also rather complex, and consequently, expensive to evaluate. The results in Figure 6.4 demonstrate the superiority of the implicit implementation for these highly regular datasets. Considering the log scale, we see that only the most efficient gridfield formulation, "gfdirect" performs within a factor of 5 of the VTK implementation. A fully-implicit representation of product gridfields, similar to the representation used by VTK, would be more effective for this task than the explicit representation we use.

Another implementation of the cross-product operator exploits the fact that it is followed immediately by a restrict. In the relational algebra, a join is semantically equivalent to a cross product followed by a select. We can create an analogous *join* operator that evaluates the restrict as the cross product is computed, never materializing cells that will not be in the final result.

We compare the projecting implementation, the join implementation, and the

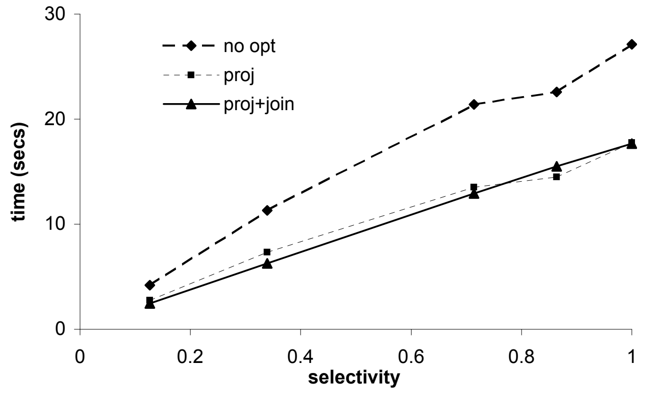


Figure 6.6: Comparing implementations of the cross-product operator.

core cross operator evaluating the “wetgrid” example from Chapter 2, Section 2.5.1. The projecting implementation computes only the nodes and polyhedra of the 3-D cross product, leaving out the 1-cells and 2-cells. Figure 6.6 shows results for the original cross product implementation (“no opt”), the “projecting” implementation (“proj”) and with both improvements (“proj+join”). If C_w and C_r are the cardinalities of the wetgrid and the result gridfield, respectively, then the selectivity (x-axis) is $\frac{C_r}{C_w}$. The selectivity varies according to a user-specified region of interest. In these experiments, we tested four different regions with different sizes.

Times reflect overall execution time. The difference in times highlights the cross-product operator’s significant cost relative to the other operators in the recipe, since the recipes are otherwise identical.

The graph shows that avoiding cell materialization does indeed improve performance. On average, the projecting 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

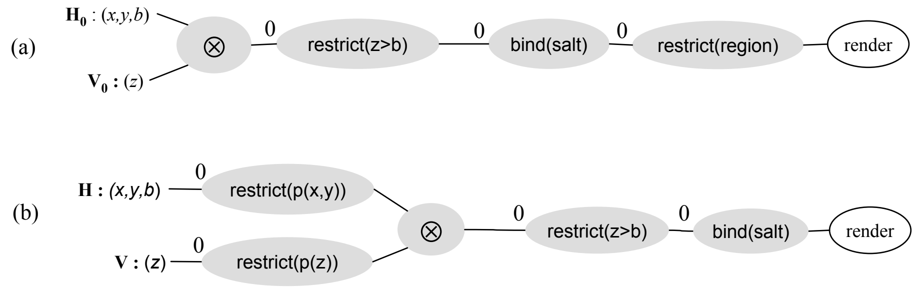


Figure 6.7: (a) Unoptimized recipe for visualizing a 3-D CORIE dataset. (b) Optimized recipe for the same task.

resize attribute arrays in the gridfield when we are wrong. Although the join implementation produces no unnecessary cells, the extra complexity of memory management washes out the performance gain.

6.2 PARTIALLY REGULAR DATASETS

The first experiment involving partially-regular datasets from the CORIE project compares our algebraically optimized recipe in Figure 6.7(b) with the unoptimized recipe in Figure 6.7(a). (These recipes first appeared in Figure 2.17 and Figure 5.7, respectively.) We also tested 1) a relational database extended with spatial data types to represent the cells, and 2) a VTK program extended with custom code to handle those operations inexpressible in VTK.

The CORIE horizontal grid we used 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.

The relational approach implements the recipe using SQL queries. Our test DBMS was PostgreSQL [SRH90], configured appropriately for the large main memory of our experiment platform. We also use the polygon type supplied by the PostGIS geometric extensions [SHRL06] to PostgreSQL. In this schema, the data values are stored separately from the cells to which they are bound, and a join is required to produce the complete gridfield. The 2-D cells are captured as polygons for more efficient spatial searching, at the cost of redundant storage of the coordinates of shared points. That is, two triangles that share a point both must store its coordinates.

The physical design for which results are reported constitutes the best of several configurations. Indices (ordinary B-Trees) are constructed on all join attributes and an R-Tree is constructed on the geometry attributes. We assume that, in practice, the data are partitioned by time such that only a single simulation run need be scanned during query evaluation, so only a single run was loaded into the database.

The times for the relational approach are conservative, 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 did not include the rendering time itself.

The implementation in VTK requires a custom reader for our file formats. Restrictions are implemented using the `VTKThreshold` object. The cross product and bind operators are implemented in the custom reader, since these operations are 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 the work.

Figure 6.8 shows test results for various size regions, which translate to various selectivities of the full wetgrid. Observe that our unoptimized plan is slower than the VTK implementation, even though they implement similar recipes. The

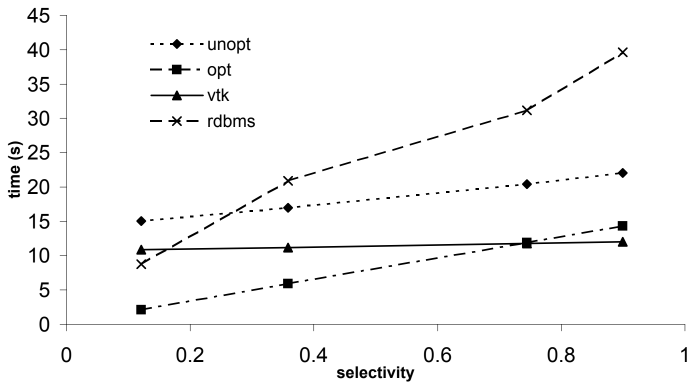


Figure 6.8: The performance of optimized and unoptimized recipes compared with VTK and PostgreSQL implementations.

---

specialized reader for VTK (implementing the cross product, restrict, and bind operations) constitutes only about 15% of the total execution time. In our implementation, these operations constitute about 30% of the total. The specialized reader is indeed more efficient than the generic operators.

The optimized gridfield recipe performs better in all but the highest 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 is expected given that the recipe of Figure 2.17. Regardless of the region being displayed, the entire 3-D grid is generated and traversed. The relational approach is far behind for all but the most selective predicates. Although the optimizer produces query plans that behave like our optimized recipe, the overhead of processing gridded datasets using joins dwarfs the improvement.

The second experiment using partially-regular CORIE datasets involves the vertical slice data product described in Chapter 5, Section 5.2.3. Recall that the optimized vertical slice recipe involved interpolating the 3-D grid onto a user-provided 2-D grid oriented perpendicularly to the surface of the water. We compare the optimized recipe with a VTK program.

The results appear in Figure 6.9. The bar labelled "interp" uses 2-D point-in-cell interpolation by passing an appropriate aggregation function to the regrid operator as described in Chapter 5, Section 5.2.3. The bar labelled "simple" uses a less accurate but also less expensive interpolation function—it simply assigns the value of the first node in the cell rather than interpolating a value based on all the nodes in the cell. The notion of "first" is defined by the order the nodes appear in the data structure and is essentially random. Such a simple interpolation function may or may not be useful in practice, but we use it here to expose the cost of the interpolation method relative to the overall run time. The labels with the '_o' suffix make use of the following observation: Any cell that contains a user-provided point p will intersect the bounding box of the whole user-provided grid. We therefore use the restrict restrict operator to identify the relevant region before searching for cells using the regrid operator. Even our recipes that do *not* exploit this semantic optimization outperform the VTK program.

The selectivity of the bounding box is dependent on the user-provided data. The optimization is more effective when the user data is axially aligned: a very skinny bounding box provides sufficient coverage. The optimization may have been more effective in our experiment than the average case, since the Columbia River flows primarily in the x direction, and user-specified transects tend to follow this flow.

We did not test a relational implementation of this data product, since the GIS extensions do not fully support 3-D objects and functions.

From these experiments, we can conclude that algebraic optimization can have a significant effect on overall evaluation time. Although visualization libraries provide efficient algorithms for specialized tasks, organization of these algorithms into a rigorous logical framework can admit new optimization opportunities by exposing equivalences between grids within complex recipes. We also conclude that GIS extensions for relational databases do not necessarily benefit queries on partially

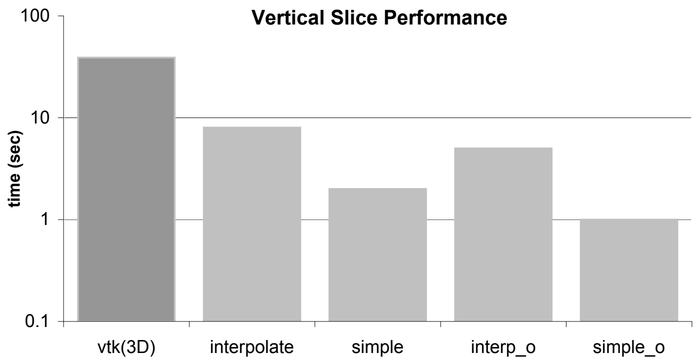


Figure 6.9: Experimental results for the vertical slice data product. Again we find that since the structure of a dataset is opaque to VTK, a simple but powerful optimization is unavailable.

regular datasets. The query time is dominated by the construction of the 3-D dataset rather than the spatial search. This test supports the conclusions drawn by others on the efficacy of using relational databases to store multidimensional numeric arrays [Bau99, MS97, WB98].

In the next section, we will test a commercial database with an aggregation workload, which is more conducive to implementation within a relational database.

6.3 DESIGN AND POPULATION OF A RELATIONAL DATABASE

Aggregation allows apprehension of datasets too large to examine directly or even to visualize effectively. Aggregation is a core feature of relational databases, but is not supported by the Visualization Toolkit. We therefore tested the performance of aggregate queries against the large CORIE datasets using a commercial DBMS, IBM's DB2.

In this experiment, we do not use the spatial features of DB2 since 1) we wish to minimize the storage overhead, and 2) the aggregation queries we test would not benefit from spatial indexing.

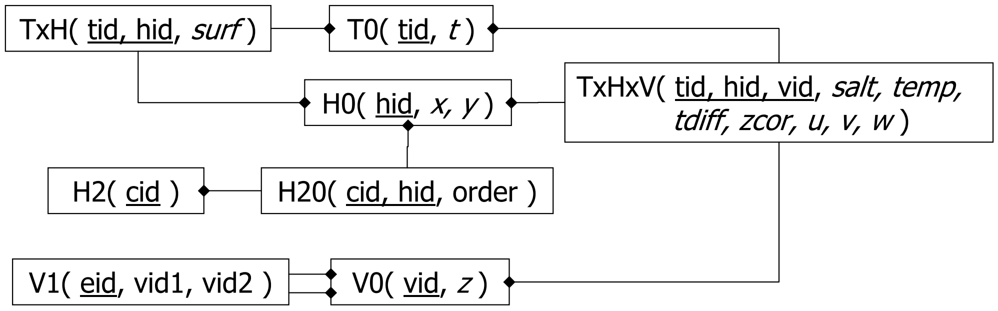


Figure 6.10: A relational schema for storing CORIE simulation results.

The schema we use to store these datasets appears in Figure 6.10. The relationship lines point in the direction of the diamond and should be read as "references," as in "TxH references T0 and H0." Italicized attributes are floating point numbers and non-italicized attributes are unsigned integers. Primary keys are underlined. The relation TxHxV holds the great majority of the data—three unsigned integer identifiers and five floating point numbers for each node in 3-D grid, for each of 96 timesteps. The relation H2 holds identifiers for the 2-cells in the horizontal grid. Attributes dependent only on the cell would be added as attributes to this relation.

We used a CORIE forecast for the Fraser River in the Pacific Northwest for this experiment. This forecast produces one 364MB file for each of 5 variables, a 717MB file for the horizontal velocity data, and several smaller data files for variables defined on the surface. The only surface-dependent variable loaded was elevation.

The data load operation occurs in three phases. The first phase was to parse the CORIE native files and prepare files for use with DB2's load command, a second to load the prepared files (using the efficient LOAD command of DB2), and a third to verify constraints on the loaded data. The horizontal, vertical, and time grids took relatively little time to process for all three phases—about 10 seconds total. The bound attributes took considerably longer: 2 hours and 20 minutes to prepare the files and about 50 minutes to load them. We loaded each timestep separately; each timestep took approximately the same amount of time to prepare (88 seconds) and load (31 seconds). The commit interval during load was set to commit every 10,000 tuples; adjustments to this parameter did not have a significant effect (unless, of course, the number was set high enough to fill the transaction log and throw an exception.)

The load files were prepared in an ASCII format, so the database was forced to convert ASCII numbers to internal binary representations as the data were loaded. The process was consequently CPU bound, suggesting that performance could be improved by 1) preparing a native binary format or 2) using multiple threads to load the data, one for each CPU in the test platform system. The third phase, constraint checking, took 30 minutes.

Once loaded, there are 88,292,064 tuples in the TxHxV relation, the largest. The entire database occupies 2,432,053 4k pages for almost 10GB—over three times the size of the data in its original format. Most of this increase can be attributed to the explicit storage of the i, j, k indices, which were entirely implicit in the original files.

Operationally, bulk loading a relational database with large datasets is not a trivial task. The transaction logs will fill on large inserts, so the user must know or learn to turn off logging or divide the data into smaller chunks manually. DB2's LOAD command accepts a commit-frequency argument, which simplifies this task. When moving data from one table to another within the database, however, there is no analogous option.

After testing some aggregation queries to validate the loaded data, we noticed that the DBMS was reporting spilled sorts, indicating that if more memory were available for sorting, sort-based queries might perform better. After increasing the memory until we no longer received spilled sort messages, we were rewarded with a 17% speedup in a particular aggregation query. We left these settings for the remainder of the tests.

Our recount of the load operation and subsequent configuration is meant to demonstrate that deployment of a relational database in this domain requires some effort. The schema we adopt is appropriate for storing exactly one of the grids used by CORIE. To extend the schema to support another grid, either additional relations must be added (complicating queries and management) or another foreign key must be used to link cells to their grid (exacerbating the space inflation).

Even once this data are loaded, good performance is not guaranteed. Selecting an individual timestep exercises the primary key index, and is quite efficient. The following query computes the average salinity over depth at a particular timestep for each node in the horizontal grid. The query takes only 6.177 seconds with no additional indices and the default configuration settings.

```
  SELECT x,y,avg(salt)
    FROM TxHxV p, H0 h, T0 t
   WHERE p.hid = h.id
     AND p.tid = t.id
     AND timestep = 20
GROUP BY x, y
```

In contrast, the gridfield expression in Figure 6.11 must spend some time constructing the large 3-D grid, and takes 17.8 seconds.

To aggregate over the horizontal area as well as depth, however, the DBMS must stride through the data just as with the non-indexed gridfield recipe. The following query computes the average salinity over all depths and all timesteps for those nodes within a particular bounding box. In this query, we reduce the size over which to aggregate (to a region in which a student was looking for persistent eddies, incidentally).

```
    SELECT timestep, avg(salt)
      FROM TxHxV p, H0 h, time t
     WHERE p.hid = h.id
       AND p.tid = t.id
       AND x BETWEEN 96644 AND 430040
       AND y BETWEEN 5221584 AND 5607198
  GROUP BY timestep
```

This query completes in 293.7 seconds. A gridfield implementation of the query that can exploit the array-like structure of the data (Figure 6.12) takes 83.3 seconds.

Adding an index on the (x, y)-coordinates improves performance on this query, but not by much (3%). The time is not being spent on winnowing the tuples from the small `H0` table, but in simply processing the hash aggregate.

Apart from performance during loading and querying, relational databases demand a line be drawn between static and dynamic information: That which is static becomes the schema and that which is dynamic becomes the instance. They make a further demand that the static information be declared first.

In a scientific-data-processing environment, this line is blurred. The production system may change the grid only infrequently, at a rate that can be accommodated by schema "releases". However, individual scientists may test different grids and data organizations much more frequently during development. Tools to manipulate and integrate results from different experiments are required. The organizational "churn" that characterizes any scientific activity is not amenable to static, fixed structure. Gridfields provide a lightweight alternative, wherein data products can be constructed on the fly from base data, yet common structure is still apparent and can be exploited.

Assuming the relevant simulation programs already emit files, the creation of

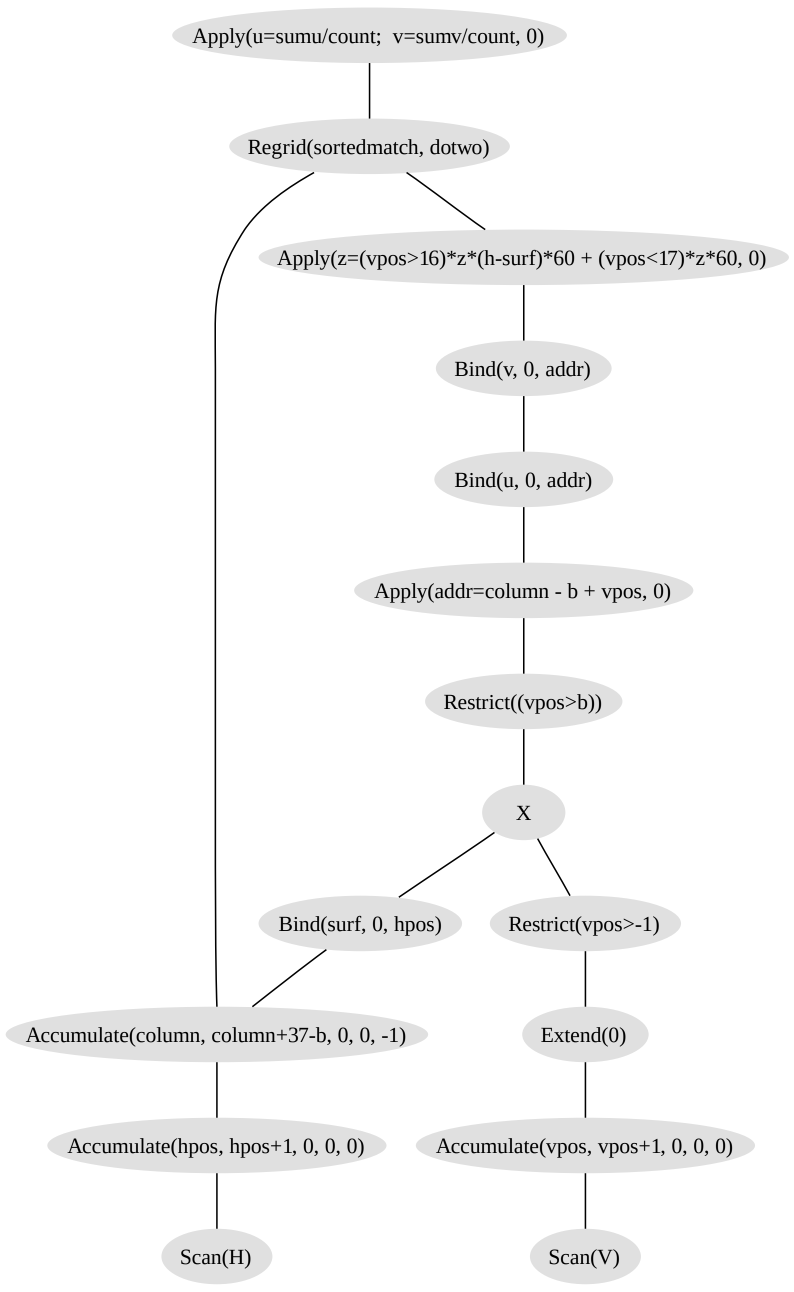

Apply(u=sumu/count; v=sumv/count, 0)
Regrid(sortedmatch, dotwo)
Apply(z=(vpos>16)*z*(h-surf)*60 + (vpos<17)*z*60, 0)
Bind(v, 0, addr)
Bind(u, 0, addr)
Apply(addr=column - b + vpos, 0)
Restrict((vpos>b))
X
Bind(surf, 0, hpos)
Restrict(vpos>-1)
Accumulate(column, column+37-b, 0, 0, -1)
Extend(0)
Accumulate(hpos, hpos+1, 0, 0, 0)
Accumulate(vpos, vpos+1, 0, 0, 0)
Scan(H)
Scan(V)

Figure 6.11: Gridfield recipe implementing aggregation over depth at a particular timestep. The accumulate operator is used to assign positions to the cells; see Section 2.4.6.

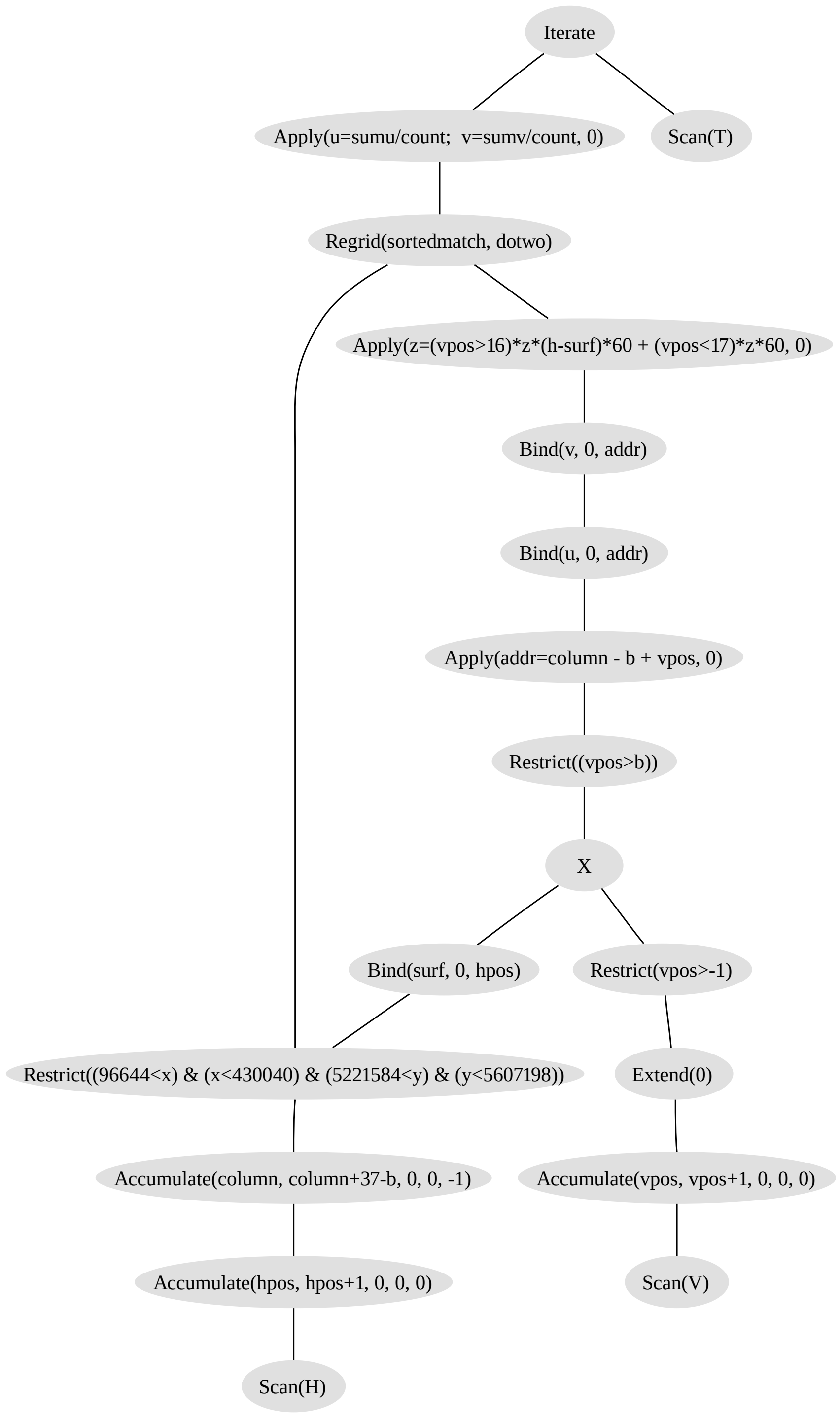

Iterate
Apply(u=sumu/count; v=sumv/count, 0)
Scan(T)
Regrid(sortedmatch, dotwo)
Apply(z=(vpos>16)*z*(h-surf)*60 + (vpos<17)*z*60, 0)
Bind(v, 0, addr)
Bind(u, 0, addr)
Apply(addr=column - b + vpos, 0)
Restrict((vpos>b))
X
Bind(surf, 0, hpos)
Restrict(vpos>-1)
Restrict((96644<x) & (x<430040) & (5221584<y) & (y<5607198))
Extend(0)
Accumulate(column, column+37-b, 0, 0, -1)
Accumulate(vpos, vpos+1, 0, 0, 0)
Accumulate(hpos, hpos+1, 0, 0, 0)
Scan(V)
Scan(H)

Figure 6.12: Gridfield recipe implementing aggregation over depth for a specific bounding box, but for all timesteps. The accumulate operator is used to assign positions to the cells; see Section 2.4.6.

a schema inserts an extra step in the cycle of scientific inquiry via simulation:

DEVISE → DEVELOP → RUN → ANALYZE → DEVISE

Between RUN and ANALYZE, there is a step to design the appropriate storage structures to hold their results. With gridfields, we seek to minimize the time spent translating data from one form to another and maximize the time spent visualizing and analyzing the results. Anecdotally: We once asked the CORIE scientists to relate which aspects of their system they did not expect to change much. After a moment of thought, one staff member replied, "Well, the Earth will probably remain round."

6.4 IRREGULAR DATASETS

Several of the optimizations we have discussed and tested exploit regularity in the dataset indicated by the presence of the cross operator. In this section we consider an experiment involving a fully-irregular dataset.

In Chapter 2, Section 2.5.2, we used the merge operator to combine two related gridfields, one with area and length datasets and one representing the plume. We now explore optimizations available in a similar situation involving blood flow simulation results.

Consider a data product designed to find and visualize regions at risk of causing a stroke in an artery. Regions of negative pressure and high velocity in stenosed arteries (i.e., artificially constricted by plaque) are associated with turbulent flow, which can dislodge plaque and cause a stroke [Sch02]. Assume we have two gridfields **S** and **T** with identical grids but different data. The gridfield **S** holds pressure data, and the gridfield **T** holds velocity (flow) data. To find regions of negative pressure, we use a restriction on **S**. To find regions of high velocity, we use a restriction on **T**. Finally, we will merge these two datasets for visualization.

We need to perform two restrict operations (r) and one merge operation (m) to

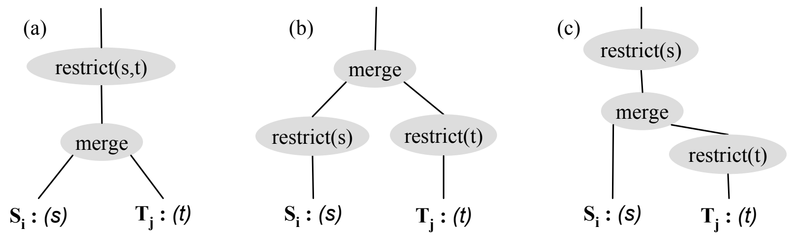


Figure 6.13: Three equivalent sub-recipes. (a) Restrict operations are combined into one. (b) The restrict operators are pushed through a merge, resulting in a *less* efficient plan. (c) Evaluating one restrict at a time might require less memory.

---

obtain the correct output, but in what order should these operations be evaluated? Three equivalent versions of this recipe fragment are shown in Figure 6.13. Figure 6.13(a) shows the two restrictions evaluated *after* the merge. Previously, we improved performance by evaluating restricts early, as in Figure 6.13(b). 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 $\mathbf{S}$ and $\mathbf{T}$, respectively. But observe that in Figure 6.13(a), both arguments to the merge operator are defined over the same grid. This knowledge allows the merge to be evaluated trivially. The recipe in Figure 6.13(c) may also be a good choice. Since $\mathsf{grid}(\mathbf{T})$ is known to be a subgrid of $\mathsf{grid}(\mathbf{S})$, we can still evaluate the merge in linear time. We can also evict the attribute array t from memory right after we have evaluated the first restrict, possibly lowering the memory footprint of the overall recipe.

There are 4M nodes in the Carotid dataset organized into 2M (irregular) hexahedrons. The overall shape of the dataset is illustrated in Figure 6.14. The carotid artery leads to the brain, so the z direction would normally be oriented upwards in the human body; we lay the patient on his or her side for presentation purposes. The dataset contains the artery itself and its interior. The stenosed region is the narrow portion of the upper branch just after the bifurcation. Well after the stenosed region is an area of low pressure, and just before the stenosed region is

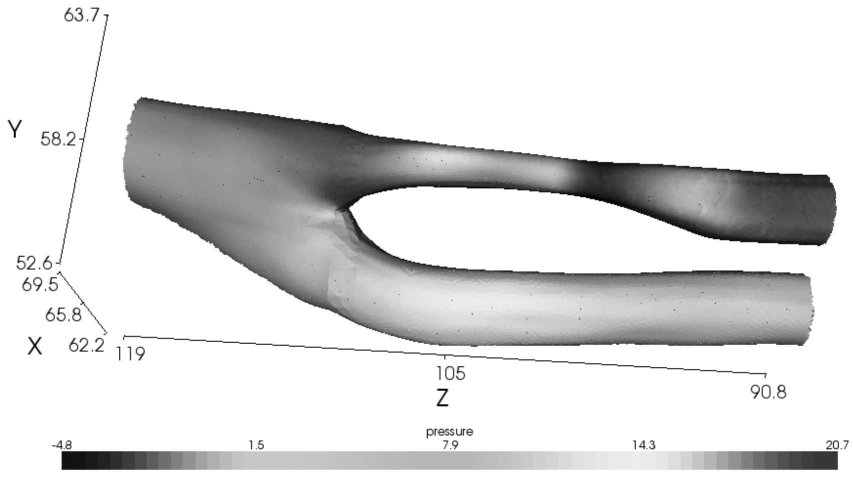


Figure 6.14: A model of a carotid artery equipped with pressure data.

---

an area of high pressure.

The region of low pressure and high velocity is shown in Figure 6.15. The rough surfaces correspond to the cells of the computational grid; this result does not include the smooth outer surface of the artery.

We tested the five recipes for merging the velocity and pressure datasets; the time in seconds to complete each recipe are as follows:

$$\begin{aligned}
\mathsf{merge}(\mathsf{restrict}_{pressure}(\mathbf{S}), \mathsf{restrict}_{velocity}(\mathbf{T})) &\quad : 20.15\text{s} \\
\mathsf{restrict}_{pressure}(\mathsf{merge}(\mathbf{S}, \mathsf{restrict}_{velocity}(\mathbf{T})) &\quad : 13.06\text{s} \\
\mathsf{restrict}_{velocity}(\mathsf{merge}(\mathsf{restrict}_{pressure}(\mathbf{S}), \mathbf{T})) &\quad : 10.53\text{s} \\
\mathsf{restrict}_{velocity}(\mathsf{restrict}_{pressure}(\mathsf{merge}(\mathbf{S}, \mathbf{T}))) &\quad : 12.77\text{s} \\
\mathsf{restrict}_{velocity \wedge pressure}(\mathsf{merge}(\mathbf{S}, \mathbf{T}))) &\quad : 10.75\text{s}
\end{aligned}$$

The effect of the order of the two restrictions is due to a difference in their respective selectivities; the pressure condition is 20% more selective. The conclusion

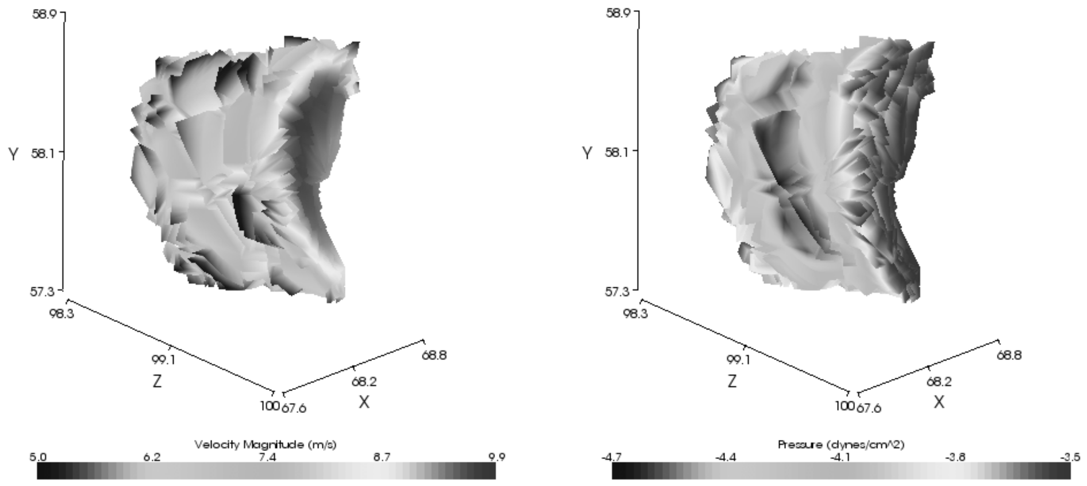


Figure 6.15: A region of a carotid artery associated with increased risk of dislodged plaque and stroke due to high velocity (a) and negative pressure (b).

---

is that evaluating restrictions as early as possible is not always the optimal plan.

This result demonstrates that the optimization rules for working with gridded datasets are not always derived from an analogous rule in the relational algebra. The relational version of this situation would often favor pushing restricts as far down the plan as possible, at least at the logical level. In our case, however, the merge operator can be implemented trivially if the two arguments are known to be defined on the same grid. An analog can be found in column-oriented databases [SAB$^+$05] that have access to a positional join that behaves like a zip merge.

6.5 CONCLUSIONS

Gridfields are designed to be used with a variety of grids. The design is agnostic with respect to the dimension of the grid, the presence or absence of regular patterns in the grid, the types of cells, and the rank at which attributes are bound. Chapter 5 included evidence of the reasoning power of gridfields with respect to the complex grid structures used in the CORIE project. In this chapter, we examine the performance of the gridfield implementation with respect to these CORIE datasets as well as more conventional datasets such as the fully regular datasets and fully irregular datasets we examined.

We began by exploring the performance of gridfields when manipulating highly-regular datasets modeled with a series of cross operators. These datasets admit a fully implicit representation of topology that can be very efficient in certain cases. The fully implicit representation corresponds to a lazy evaluation of the cross operator: No cells are materialized unnecessarily, and costs are significantly reduced. However, if many cells are to be materialized in the result anyway, then the implicit representation offers no benefit. A performance penalty was actually incurred due to cell-by-cell allocation and copying during the transition from an implicit representation to an explicit representation. The cross operator, conversely, knows the global size and structure of its output and can allocate and copy cells in blocks.

The algebraic properties of the gridfield language also mitigate the disadvantage when lazy evaluation is effective and magnify the improvement when it is not. Cells that are not relevant to the task can be filtered out by inserting an accrete operator into the recipe, reducing the size of intermediate results. The accrete can be physically implemented inside the cross operator for additional savings.

For these datasets, we conclude that an implicit representation of product grids (corresponding to a lazy evaluation of the cross operator) does situationally improve performance and is therefore useful in a gridfield implementation. However, this extension must not come at the cost of algebraic reasoning, which offers broader improvements.

For the partially regular datasets found in the CORIE system, we again find that algebraic reasoning leads to significant cost savings. We are able to derive and express recipes that exploit the regularity of the gridfields and even the properties of the interpolation methods to offer savings for the horizontal and vertical slice recipes described in Chapter 5.

We then presented a relational implementation of these recipes in a commercial RDBMS, including an application-specific schema, configuration settings, and data ingest procedures in addition to the relevant SQL queries. We conclude that the "time to insight" is longer using these tools than with gridfields due to the additional design tasks required. Further, use of a relational database incurred significant overhead in space on disk: a factor of 3. Finally, the performance of queries that access a great deal of data is almost a factor of 3 slower than the gridfield implementation, since joins are computed using explicit keys (the hallmark of relational modeling) rather than implicit position.

We also tested the efficacy of a specific algebraic optimization using gridfields with fully irregular datasets. The purpose of this test is primarily to demonstrate that not every optimization rule has a relational analogue. In this case, performance proved worse when restrict operators were pushed towards the leaves of the recipe. This transformation destroyed the equality of two intermediate grids, which in turn had allowed a constant time implementation of the merge operator.

These results provide evidence that gridfields are effective as a "jack-of-all-trades" model capable of handling many kinds of grids without sacrificing performance. Specialized tasks on specialized grids admit more efficient algorithms, but gridfields offer a lightweight, balanced, and flexible solution.

Chapter 7

CONCLUSIONS AND OUTLOOK

Once again, we recall the thesis: Programs written to manipulate simulation results in the sciences exhibit an algebraic structure useful for reasoning and optimization.

We began in Chapter 1 by describing scientific and engineering modeling applications in which these programs frequently appear. Scientists and engineers are modeling increasingly complex domains, such as engine manifolds and jagged coastlines. The differential equations governing these models cannot be solved analytically. To solve the equations numerically, they must be converted to a set of algebraic equations suitable for solution within the computer. The conversion is straightforward for primitive domains such as line segments, triangles, quadrilaterals, tetrahedra, and prisms, so complex domains are decomposed into primitive domains—the cells of a grid. The system of equations for each of these cells is locally constrained to agree with its neighbors on shared values and globally constrained to minimize an error metric. Prepared in this manner, the equations can be solved using conventional matrix manipulation routines augmented with problem-specific methods to improve accuracy.

The datasets produced and consumed by these simulations bind data values to the cells of one or more dimensions. These data are ordinarily stored as arrays; their assembly into a logical grid structure is the responsibility of the programmer. The alternative is to design a model specialized for these datasets, bottom up.

We presented the requirements for such a model in Chapter 2 by describing the grid structures found in practice and analyzing pseudo-code for representative tasks. We found that the common low-level operations in each case included iteration over cells, access to bound data, and traversal of the incidence relationship between cells. Suitable high-level operations include applying arithmetic expressions, selecting a region of interest, mapping one grid onto another, and combining two grids.

Using these requirements, we developed a general combinatorial model of grids that separated the geometry (and other bound data) from the topology. The set-based structures admit set-based operations (e.g., union and intersection) and relationships (e.g., the subgrid relationship). We then defined properties of grids that correspond to intuitive correctness criteria in certain situations. For example, a homogeneous grid of dimension 2 contains no dangling edges or isolated nodes: Every cell is incident to a 2-cell. These properties are not enforced by the model, since the model must accommodate grids found in practice that violate them. However, the ability to reason about and selectively enforce these properties has proven useful, and the grid model provides a succinct language for doing so.

The definition of a grid was extended to allow data values to be bound to the cells of one or more dimensions. Some operators for manipulating grids can be lifted to operate on gridfields, others are particular to the interaction between the grid and the data.

The cross and regrid operators have proven especially useful, allowing us to express topological repetition and map one grid onto another, respectively. The restrict operator provides a means of changing the topology of a gridfield using the bound data—it is unique in this regard. Using restrict, one can only produce a legal subgrid of the input, which is an important correctness criteria. The accrete operator "seeds" a new grid with the cells of a particular dimension, then grows a connected grid by including neighboring cells in the result. Using the fixpoint operator, we can repeat the process to grow connected gridfields whose cells all satisfy some condition on their bound data.

Using several examples, we demonstrated that the gridfield model is sufficiently general to express gridfields and gridfield computations we found in practice.

In Chapter 3, we describe an evaluation of candidate data structures for grids and gridfields, finding that, remarkably, the more flexible data structure also offered better performance. Several data structures were not sufficiently general for our purposes. For example, the quad-edge data structure and its variants can only store 2-D manifold grids. (Informally, a d-dimensional manifold grid can be embedded in a d-dimensional space.) We evaluated three data structures general enough for our purposes: TIGraph, based on incidence graphs; Indexed Cell Set (ICS), a special case of TIGraph; and NGMaps, based on combinatorial involutions. The ICS representation integrates well with existing software, but does not support the full generality of the gridfield model. The enforcement of grid properties is easier with an NGMap, as a TIGraph allows an arbitrary set of cells linked by an arbitrary incidence relation. Since an NGMap can be specialized for grids that obey certain properties, it would seem that we could capitalize on the restricted domain to improve performance. However, we can build an TIGraph involving only those cells and incidence relationships relevant to the user's current task, for considerable savings in such cases. The relevant subgrid cannot be expressed as an NGMap, since it violates the grid correctness criteria that the NGMap was designed to enforce. For example, a task that does not need access to the 1-cells need not include them in an TIGraph representation. However, the 1-cells cannot be excluded in an NGMap representation without violating invariants of the data structure. We decided to adopt an ICS representation where possible, and extend it to the more general TIGraph representation where necessary.

The data structure we use with gridfields includes an explicit incidence relation for each dimension. The incidence relation maps k-cells to 0-cells for each dimension k. The 0-cells are therefore distinguished at the physical level. In practice, we have not encountered many gridfields that do not need access to to the 0-cells, so requiring their presence does not incur a significant additional cost.

The gridfield data structures must be populated from the scientists' simulation results. It is unlikely that the native data formats used by scientists will correspond directly to our chosen representation in memory, so some form of conversion must take place. The formats are generally designed to simplify initial creation rather than simplify access and analysis. In Chapter 4, we develop a simple language for extracting arrays from collections of files. The logical array values may appear embedded in the names of files and directories or in the files themselves. The arrays may span multiple files, and one file may contain multiple arrays. Once extracted, these arrays become the components of the gridfield data structures—attributes, cells, and incidence relations. We develop several techniques for interpreting the structure of a file (described by a schema) to improve performance when extracting the file's content. We tested code generation and partial evaluation techniques and found that these techniques were competitive with hand-coded readers optimized for (and coupled to) a single format.

In the interest of developing an end-to-end system for analyzing gridded datasets, we describe in Chapter 5 how gridfields can be used at all stages of development of a simulation-processing system. First, as a reasoning tool, we demonstrated that gridfields could express the CORIE grids at different stages of evolution, exposing equivalences between them. We then described interfaces for programming with gridfields, including physical operators for animations and distributed computing. Finally, we described a gridfield application for scripting gridfield-powered visualizations.

We investigated the performance of gridfields relative to 1) a popular visualization library, VTK, and 2) a commercial relational database. We found that by using specialized data structures of VTK for specialized tasks, one can outperform the more general gridfield implementation. However, no solution was able to outperform gridfields across the spectrum of gridded datasets we tested—highly regular gridfields, fully irregular gridfields, and partially regular gridfields such as those found in the context of CORIE. We also found that the preliminary tasks required to use a relational database in the scientific domain (schema design, configuration, and bulk-loading) were quite expensive.

The chapters in this work were organized by topic; the chronological list of milestones reached during development of the model and implementation found below provides additional context.

- Chapter 1: Analyze scientific data products for commonalities [HMB03]
- Chapter 2: Develop initial data model of gridfields to express common data characteristics and operations [HM04a]
- Chapter 2: Implement the gridfield model [HM04a]
- Chapter 6: Perform preliminary tests against conventional technologies [HM04b]
- Chapter 4: Develop a language for describing native data sources [HM05]
- Chapter 4: Develop optimization techniques for accessing native data sources [HM05]
- Chapter 2: Develop a library of assignment and aggregation functions for use with regrid.
- Chapter 5: Develop a prototypical gridfield application, DPBuilder
- Chapter 2: Extend the model to allow data bound to multiple ranks simultaneously, simplifying many recipes
- Chapter 2: Extend the algebra with the iterate operator for expressing animations and small-footprint aggregation
- Chapter 2: Extend the algebra with the accrete operator for building connected regions
- Chapter 2: Generalize the iterate operator with the fixpoint operator to accommodate recipes that grow from seed cells
- Chapter 3: Adapt, implement and test alternative data structures derived from the literature for housing gridfields

- Chapter 5: Implement a *catalog* interface through which the scan and bind operators can access native data sources
- Chapter 5: Derive algebraic transformations for operating on partitioned grids
- Chapter 5: Describe and implement an evaluation model for gridfield expressions.
- Chapter 6: Test the gridfield implementation against a popular visualization library
- Chapter 6: Evaluate the use of relational databases to store gridfield data
- Chapter 7: Articulate a vision of how gridfields may be injected in a large-scale e-Science project

Reflecting on the evolution of the gridfield model and implementation, the following lessons stand out as keystones to the success of the project.

Topology vs. Geometry. We reaffirmed Berti's observation that many grid-manipulation programs did not need access to geometry information. We extended this idea by separating topology and geometry completely, modeling the latter as ordinary data. This design resolved issues with conventionally problematic grids—for example, those grids whose geometry changed over time.

Pay-as-you-go. Even before the implementation was available, we deployed the model as a reasoning tool. Early implementations were deployed as part of prepackaged scripts for generating specific data products. Though there was a temptation to postpone deployment until the software was "complete," we made a conscious effort to maintain a smooth return on investment.

Localized Extensibility. We realized early on that computational science as an activity would continue to produce new methods of discretizing a domain, and therefore new types of geometric cells (e.g., curvilinear, concave, hexagonal). Rather than try and anticipate the needs of future scientists, we made the algebra extensible through the regrid operator. Interpolation methods, point-in-region tests and other techniques that rely on special handling of geometry can be encapsulated as custom assignment and aggregation functions. We gathered these specialized routines in one place so as not to interfere with our ability to reason globally about gridfield recipes. Allowing user-defined operators, in contrast, would have obstructed the definition of rewrite rules.

In the next section, we describe some directions of future work.

7.1 FUTURE WORK

Gridfields offer a novel approach to working with scientific simulation results. A logical model guided the development of concrete middleware for transforming native data stored in ad hoc file formats into data products. Although we have striven for a comprehensive approach to manipulating gridded datasets, there are still several avenues for additional research. In this section, we describe the following four areas, anticipating that they will be especially fruitful:

1. Hierarchical grids
2. Adaptive grids
3. Automatic Optimization
4. Proliferating Gridfields

We anticipate that the gridfield model can accommodate these extensions without major surgery. For example, we have experimented with gridfield-valued attributes to model hierarchical grids.

Hierarchical Grids

Figure 7.1 illustrates an example of a hierarchical grid, which is a grid whose cells may be recursively subdivided into other cells, instituting a parent-child relationship between cells. This relationship implies geometric and topological *containment* between parent and child cells, which has at least four applications: (1) for

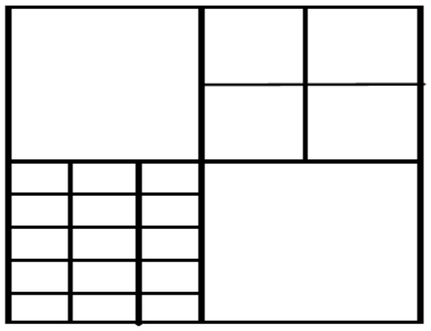


Figure 7.1: A hierarchical grid used for improving the scalability of large grids.

searching for leaf cells efficiently by using the hierarchical structure as an index, (2) for reducing redundancy in the description of the overall grid when reusing the same subdivision for multiple parent cells, or (3) for *multiresolution* processing, where operations that can tolerate approximate answers might not descend the hierarchy below a fixed level, and (4) for parallel processing where separate branches of the hierarchy are handled by separate processes and potentially on separate hardware.

As a first attempt to modeling these structures using the gridfield model, we might express a hierarchical grid as the union of its leaf grids. For example, the hierarchical grid in Figure 7.1 may be expressed as the union of the its four quadrants, where each quadrant is represented by a grid. This approach might be useful for application (2), since the same grid can be reused in multiple locations. However, applications (1) and (3) each require new hierarchy-aware implementations of our operators.

For example, a frequent use of the restrict operator is to locate cells whose geometric realization is contained within a bounding box. This task is expressed by passing in the bounding box as a boolean condition over geometric attributes; e.g.:

$$x > 5 \ \& \ x < 10 \ \& \ y > 15 \ \& \ y < 30$$

To exploit the hierarchy and improve performance for this task, three conditions must be met: (a) the parent cell must be guaranteed to contain (geometrically) the child cells, (b) the parent cell must record information about its own geometry (for example, its minimum bounding rectangle), and (c) the restrict operator must be aware of (a) and (b).

For application (3), the regrid operator must be extended to understand the geometric and topological relationship between a parent cell and its children, and the gridfield representation must record new information with each parent cell. For example, each parent cell could cache the min, max, and average value of all it children, and the regrid operator could sometimes use this information to avoid accessing the children. [RBS02].

This type of grid is important for scalability. Some operations are intrinsically $O(n)$; e.g., multiply salinity by 5 for every cell and visualize the result. Linear time operations will not scale to very large datasets, so parallel or approximate algorithms must be employed. A parallel algorithm must access the same amount of data but divides the work among many processors; an approximate algorithm sacrifices accuracy for performance given fixed resources. In the simple example given in this paragraph, an approximate algorithm might use the pre-calculated average salinity over a large region and multiplying that by 5 rather than physically accessing each and every cell.

Adaptive Grids

Unstructured grids are used to improve precision in modeling highly dynamic processes. For example, many more small cells are used to model the turbulent estuary than are used to model the relatively calm ocean. However, what if the ocean became turbulent and the estuary became calm as the simulation was running? It might be beneficial to increase resolution in the ocean and decrease resolution in the estuary. While this particular scenario involving a turbulent ocean is not very likely, changes in dynamic regions are common for many applications.

Adaptive grids [BDF94, HT04b] change their topology at runtime in response to changing conditions. At each timestep, a cell may be split to create two or more smaller cells, or multiple cells may be merged into one. The gridfield algebra cannot express these modifications of topology. Only the cross operator can generate new cells at runtime, and it does so only in a very predictable manner.

This restriction was intentional: bounds on the size and "shape" of all intermediate results can be traced through a gridfield recipe, a useful step towards cost-based optimization.

To extend the gridfield model with the capability of creating and deleting cells, we have begun investigating a *rule* language for expressing cell-to-cell transformations. For example, a rule of the form of Figure 7.2 could express the splitting of two large cells into four smaller cells.

The rule language could be defined using a generalization of stencils that match involve not only patterns of topology, but patterns of bound attributes. Such a language could likely express more than just adaptive grids; we have informally experimented with such rules to express the grid algorithms in Chapter 2. We anticipate that a suitably defined rule language (acting as a *grid calculus*) would be strictly more expressive than the gridfield algebra.

The ability to change the topology at runtime was intentionally omitted from the features of the gridfield algebra so that we could place static bounds on the sizes of intermediate results. A rules language would require a change to this optimization strategy. A very interesting question is whether the rule language could be defined such that useful properties of intermediate results could be discovered through static analysis.

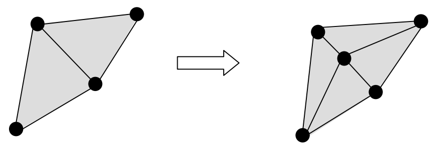


Figure 7.2: Rules describing cell transformations may be suitable for implementing adaptive grids and more.

Automatic Optimization

We have automated a limited number of heuristic optimizations, as described in Chapter 5. The next step is fully automatic and transparent optimization, as is provided by relational databases.

A heuristic optimizer becomes complex and potentially unsound as more and more rules are implemented. Cost-based optimization then becomes an attractive alternative. There are two components to a cost-based optimizer: a cost function and a search strategy. We anticipate that a suitable cost function can be derived from the size of intermediate results. One of the features of the gridfield language is that bounds on these sizes can be traced through every recipe. Also, some operators (e.g., merge) are mainly grid-dependent and not data-dependent. The observed cost on one gridfield is therefore a good indicator if cost on another gridfield with the same grid.

We expect the cost of recipe evaluation to dominate the cost of optimization, so exhaustive search may be a suitable strategy initially. Also, we frequently encounter situations in which the same sub-recipe must be evaluated many times (e.g., during an animation), so the cost of optimization is amortized over multiple executions.

There are other optimization techniques we have considered:

Static Sub-recipes. There are also opportunities to take advantage of static parts of a recipe. For example, a regrid operator repeatedly evaluated may not need to recompute its assignment function over and over, if the cell assignment does not change on each execution. This situation occurs, for example, when the assignment function depends on topology only (e.g., when instantiated as a neighborhood operator). The cell assignment can be computed only once and reused over and over. A We implement this type of optimization by hand currently.

Pipelined Recipes. To evaluate recipes where not all data fits in memory, we need to extend the work on iterator-introduction rules we present in Section 5.3.3. The rules state the conditions that must be met in order to pipeline a recipe, but algorithms to partition a large gridfield with respect to these conditions need to be found and implemented.

Specialized Representations. We showed in Chapter 6 that specialized representations will outperform general representations in some cases. Techniques for transparently switching to these representations when beneficial to do so would make gridfields competitive with even highly focused tools and algorithms. We anticipate that the logical model would not need to be changed to accommodate new representations, though new operator algorithms would be necessary.

Distributed Evaluation. The physical fetch operator described in Chapter 5 ships a recipe to a remote server for evaluation, but algorithms automating when and how to use this operator have not been studied.

Proliferating Gridfields

Integration with existing tools increases proliferation of new technology. There are two likely targets for gridfield technology: relational databases and visualization software libraries.

In the former case, gridfields, grids, and attributes may be cast as user-defined types, and each operator may be implemented as a user-defined function. Algebraic expressions can then be included in SQL statements as usual. The disadvantage of this approach is that gridfields can become very large. Since gridfield expressions are opaque to the optimizer, performance can suffer. A data architect can reduce the size of individual gridfield values by splitting large product gridfields into multiple tuples. For example, we have expressed the CORIE datasets as a gridfield expression of the form $\mathbf{H} \otimes \mathbf{V} \otimes \mathbf{T}$. Instead of one tuple holding the entire dataset, each of $|V_0||T_0|$ tuples could hold one gridfield with grid H. Further, we can abstract out the shared grid and store only the attributes. The relationship between these attributes and the grid H can be maintained by foreign keys.

Gridfield operators can be injected into a visualization library in two ways: First, operators providing novel capabilities can be implemented alongside the native algorithms. For example, the cross operator and the regrid operator are good candidates for inclusion in the VTK library, as there are currently no direct counterparts. Second, many VTK classes effectively implement the same logical gridfield operator. If this relationship could be made explicit, gridfield recipes could be compiled into VTK programs. An optimizing compiler might choose among several VTK algorithms depending on the characteristics of the source data—e.g., structured grid algorithms employed for structured grids. Gridfields may also be used to massage native data into a form usable by VTK. We frequently use VTK in this manner—as a rendering and visualization engine for gridfield recipe results.

In the next section, we consider gridfields as as middleware for a large-scale e-Science project that involves more than just simulation. We then conclude by considering the future potential of gridfields as a general-purpose data model.

7.2 GRIDFIELDS AS SCIENTIFIC MIDDLEWARE

The following describes the Ocean Observatories Initiative (OOI) [ooi], a significant project underway to instrument and study the oceans on an unprecedented global scale.

> Major advances in our understanding of the oceans are currently limited by our ability to make sustained observations over large areas. Continuous, long-term measurements of physical, chemical, geological, and biological variables in the oceans and the sea floor below are required to understand trends and cyclic changes and to capture episodic events such as major earthquakes and harmful algal blooms. Enhanced capabilities for making sustained measurements of the ocean will open up new research opportunities and lead to improved detection and forecasting of environmental changes and their effects on biodiversity, coastal ecosystems, and climate. These advances will provide the tools for improved management of ocean resources such as fisheries, and better-informed decisions on the use of the coastal zone for recreation, development, and commerce. The National Science Foundations (NSF) Ocean Research Interactive Observatory Networks (ORION) Program will capitalize on new technical capabilities provided by Ocean Observatories Initiative (OOI) infrastructure to meet this recent convergence of basic research and societal needs.
>
> The OOI ... will construct networks of ocean observatories using funds from NSF's Major Research Equipment and Facilities Construction account. These observatories will be linked to the Internet via seafloor cables or satellites, allowing global data access. Analysis and visualization tools developed by ORION scientists and made publicly available will permit all users to manipulate the data, thereby gaining a better appreciation of topical issues such as earthquake and tsunami mechanics, fisheries and coastal resource management, and natural and human influences on ocean and climate systems. The ORION Program will increase public awareness of the oceans through a comprehensive education and outreach program.

Although the focus is on observation, modeling of ocean processes is of significant interest to the researchers. Manipulation and visualization of observations and model results, as well as comparisons between them, must be efficient and

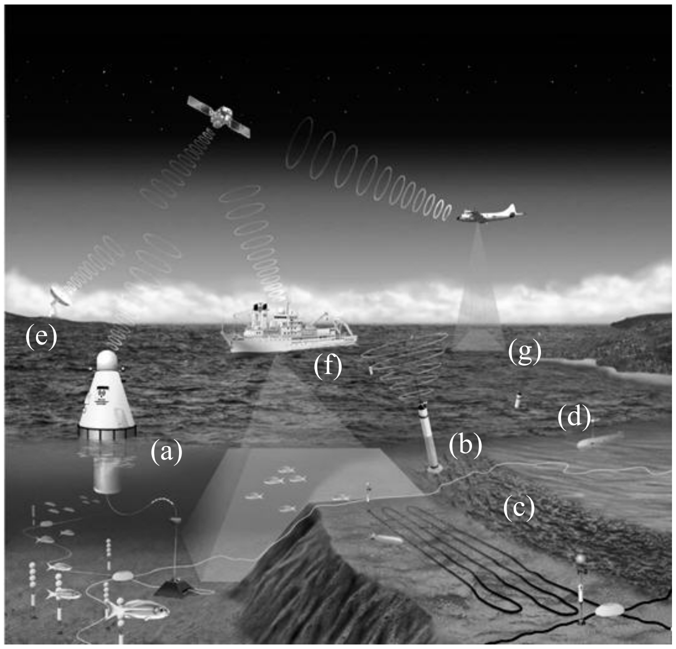


Figure 7.3: Heterogeneous observation methods [DWB]. (a) A buoy equipped with a cable-climbing profiling sensor. (b) A sensor affixed to the ocean floor. (c) Cabled seismographic sensors. (d) Underwater Autonomous Vehicles equipped with sensors. (e) Coastal radar installation for measuring surface elevation. (f) Research vessel with towed sensor arrays. (g) Remote sensing via aircraft.

---

convenient in spite of the extreme heterogeneity. We have argued that gridfields are designed for breadth: performance and convenience in a wide variety of cases. This feature is critical for a data integration and analysis effort on the scale of the OOI.

We have discussed how gridfields may be used for modeling, manipulation, and visualization, but the examples have been drawn from simulation results. In this section, we describe how gridfields can serve as middleware for integrating data from observational platforms with the simulation results we have discussed previously.

Figure 7.3 illustrates a variety of observation capabilities. At (a) and (b), a sensor is attached to the sea floor. The sensor at (a) can climb the cable attached to a buoy to provide a vertical profile, while the sensor at (b) is stationary. The measurements reported by the platform at (b) depends on time, but the horizontal location and the depth are fixed. For these fixed stations, a timeseries of salinity measurements may be described by a gridfield computation:

$$\mathbf{A}_{0:x,y,z,time,salt} = \mathbf{P}_{\mathbf{0:x,y,z}} \otimes \mathsf{bind}(salt, 0, \mathbf{T}_{0:time})$$

where $\mathsf{grid}(\mathbf{P})$ is the unit grid consisting of a single node. The left hand side of this expression (and of those to follow) are variables assigned gridfield values. The vertical profiling platform's observations can be modeled by allowing the z coordinate to depend on time, or by considering each vertical sweep as a single observation, depending on ratio of observation frequency to the speed of the cable crawler. In the latter case, we have:

$$\mathbf{A}_{0:x,y,z,time,salt} = \mathbf{P}_{0:x,y} \otimes \mathsf{bind}(salt, 0, \mathbf{V}_{0:z} \otimes \mathbf{T}_{0:time})$$

The gridfield $\mathbf{V}$ describes the vertical discretization, and $\mathbf{T}$, as before, describe the temporal discretization. If we wish to reason in terms of observation intervals rather than a timeseries, the time gridfield might appear as $\mathbf{T}_{1:\delta t}$ rather than $\mathbf{T}_{0:t}$.

Figure 7.3(c) uses an array of sensors laid on the ocean floor. A grid L describes the topological arrangement of these sensors, and we associate x and y coordinates with each point in this grid. The depth varies over the length of the array, but does not change over time. The authoritative bathymetry reference is delivered on a 2-D unstructured grid, B. To integrate these data, we have:

$$\begin{aligned}\mathbf{A}_{0:x,y,depth,time,salt} = &\mathsf{bind}(salt, 0,\\ &\quad \mathsf{regrid}(\mathbf{L}_{0:x,y}, 0, \mathbf{B}_{0:x,y,depth}, 2, \mathsf{contained\_by}_2, \mathsf{interpolate})\\ &\quad \otimes \mathbf{T}_{0:time}\\ &\quad)\end{aligned}$$

The regrid operator finds a polygon in **B** that contains each point in **L** and interpolates the data onto it. The product of the result and a time grid is the grid to which the salinity measurements may be bound.

Figure 7.3(d) shows an Underwater Autonomous Vehicle (UAV), capable of sampling in four dimensions. One useful topology to overlay on a UAV's observations is to simply link the node-based observations with 1-cells indicating the order in which they were observed, with the overall grid capturing the path taken by the UAV. Sensor measurements are sometimes averaged in situ over a brief time interval to help ensure quality. These averaged data may be more naturally associated with the 1-cells of the path rather than with the nodes. An alternative to this path-grid is to program the UAV to visit the nodes of a prescribed 2-D grid, though it may be difficult to achieve acceptable accuracy with this approach.

Figure 7.3(e) shows a fixed coastal installation. In the figure, the installation appears to be solely a satellite communication uplink. However, there is a type of shore-based radar, similarly deployed, that measures surface elevation over a wide area. These measurements can be modeled as a 2-D gridfield of the water's surface. The topology may be regular (i.e., the grid is constructed as a product of linear grids) or irregular (e.g., the spatial resolution of the device decreases with distance from the installation).

The vessel in Figure 7.3(f), like the UAV, follows a path, but is constrained to the surface of the water. Measurements may be taken at a fixed depth below the changing surface. The surface information may be bound to the same grid as the bathymetry, except that the surface varies with time:

$$\begin{aligned}
\mathbf{S}_{0:surf,x,y,depth,t} &= \mathsf{bind}(surf, 0, \mathbf{B}_{0:x,y,depth} \otimes \mathbf{T}) \\
\mathbf{P}'_{0:x,y,surf,depth,t} &= \mathsf{regrid}(\mathbf{P}_{0:x,y,t}, 0, \mathbf{S}, 3, \mathsf{contained\_by}_3, \mathsf{interpolate}) \\
\mathbf{A}_{0:x,y,surf,depth,time,salt} &= \mathsf{bind}(salt, 0, \mathbf{P}'_{0:x,y,surf,depth,t})
\end{aligned}$$

This time, we use a 3-D version of the contained_by function: two spatial dimensions (represented by the attributes x and y) and the time dimension (represented by the attribute t).

A vessel may also be equipped with an acoustic Doppler profiler (ADP), oriented downwards. An ADP is used to measure velocity of the water using sound waves bouncing off suspended particles. Velocities can be observed at several depths at a time. This instrument produces a gridfield similar to our model of the cable-crawling profiler.

The 2-D and 3-D simulation results described in previous chapters must be validated against these forms of observation. Observations collected through different methods covering the same area are validated against each other. These cross-validation operations require mapping gridfields onto gridfields and interpolating the results. We envision the gridfield model as the logical and physical framework by which this interpolation occurs.

The reader may notice that many examples involve node-bound data. There are a few prominent reasons for the higher frequency of node-bound data relative to data bound at other dimensions. First, the geometry of a region is formalized as a set of points, each equipped with a coordinate vector. A discretization of a continuous region is naturally viewed as a sampling of points; each point naturally retains its coordinate vector. Second, data creators are conditioned to associate data with the nodes due to file formats that do not offer another choice. Finally, programs that manipulate gridded datasets are complicated by the need to move data from one dimension to another. These programs are much simpler if all data are bound to the nodes.

This section predicts a new role for gridfields beyond the manipulation of simulation data: a general data type for manipulating continuous spatio-temporal data discretely within the computer. In this capacity, they serve as a higher-level, more general version of the file formats favored by scientists, but without the bloat associated with relational and object databases.

7.3 INTERPRETING DATA MODELS TOPOLOGICALLY

The design of a data model involves deciding which features of the data will be represented by built-in structures and which features will be represented in the data itself. The relational model uses minimal built-in structures: An entity is described by a record in the database, but relationships between entities are encoded in the data rather than physical pointers. The advantage is simplicity for the modeler, the data creator, and the application writer, at the expense of performance. However, performance has proven ultimately satisfactory due to algebraic optimization augmented with decades of intense research. Interpreted topologically, a record in a table is not structurally close to any other record—we say that the relational model has the *discrete* topology, where each entity is an isolated point with no explicitly encoded neighborhood.

Sometimes a domain may have a dominant relationship that deserves explicit encoding in the model itself. For example, the popularity of XML suggests that many people consider their data to have a dominant hierarchical structure—entities tend to have a natural parent entity. We might characterize the hierarchy as a topology on the entity descriptions. Given an entity, we can traverse a neighborhood of nearby entities, where "nearby" is defined by some composition of the parent, child, and sibling relationships.

Graph-based data models such as RDF operate at the other end of the implicit-explicit spectrum: Each and every relationship between entities is encoded as an explicit structure in the model (an edge). Neighborhoods of data can be explored by traversing the graph edges. Although this model is capable of representing any data one might encounter, the model provides no guidance to the application about how to interpret an instance—an edge in the graph could mean literally anything. With a tree-based model, one or more relationships in the data are distinguished as dominant and encoded structurally in the tree. If properly chosen, this anointed relationship matches the intuition of the application writers, and the data is easier to manipulate. Graph-based models proclaim that all relationships be created equal, leaving application writers to organize the data in a manner convenient for their users. Observing that reorganizing data can be expensive, advocates of this approach attempt to relieve application writers of this burden by not favoring any one dominant organization. The consequence, however, seems to be that by not favoring any particular application, all applications must perform some level of reorganization.

Stream, sequence, and timeseries data models also have a topology derived from the total order imposed on the entities. The neighborhood of an entity is defined by a composition of "next" and "previous" operations. One might imagine that the order need not be encoded explicitly in the data model; a timestamp of some form will indicate order. In practice, however, items are known to arrive out-of-order with respect to their timestamps [SCZ05, TM03].

Each of these data models (tables, trees, graphs, and streams) can be characterized in terms of the topology they superimpose on data. A data model oriented toward a fifth class of topology has been the topic of this dissertation. Scientific datasets may exhibit a very general sort of topology, where neighborhoods are defined by traversing an incidence relation over cells of various dimensions. Although gridfields use a very general notion of topology, the model is not necessarily recommended for expressing relational, tree-based, or graph-based data: The emphasis on numeric values, fixed schema at a particular dimension, and the array-based implementation are all specialized for scientific applications. Gridfields do, however, represent an end-to-end solution for modeling and manipulating datasets with a mesh-based topology, a concept of similar generality, importance, and utility as the discrete topology of sets, the linear, totally-ordered topology of streams, the tree-based topology of XML, and the graph-based topology of RDF.

REFERENCES

[ABB$^+$03] Volkan Akcelik, Jacobo Bielak, George Biros, Ioannis Epanomeritakis, Antonio Fernandez, Omar Ghattas, Eui Joong Kim, Julio Lopez, David O'Hallaron, Tiankai Tu, and John Urbanic. High-resolution forward and inverse earthquake modeling on terasacale computers. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, Phoenix, AZ, 2003.

[ABKL05] William Allcock, John Bresnahan, Rajkumar Kettimuthu, and Michael Link. The Globus striped GridFTP framework and server. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 54–54, Washington, DC, USA, 2005. IEEE Computer Society.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[AMW98] Vedat Akdag, Al Magnuson, and Armin Wulf. Efficient integration of CFD into product design. In *Proceedings of the Ninth Annual Thermal Fluids Analysis Workshop*, Cleveland, OH, 1998.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.

[Aug98] Lennart Augustsson. Cayenne: a language with dependent types. In *ICFP '98: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, New York, NY, USA, 1998. ACM Press.

[Bac02] Godmar Back. Datascript – a specification and scripting language for binary data. In *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 66–77, London, UK, 2002. Springer-Verlag.

[Bat85] Donald S. Batory. Modeling the storage architectures of commercial database systems. *ACM Transactions of Database Systems*, 10(4):463–528, 1985.

[Bau72] Bruce G. Baumgart. Winged-edge polyhedron representation. Technical report, Computer Science Department, Stanford University, Palo Alto, CA, October 1972.

[Bau99] Peter Baumann. A database array algebra for spatio-temporal data and beyond. In R. Y. Pinter and S. Tsur, editors, *NGITS '99: Proceedings of the 4th Workshop on Next Generation Information Technologies and Systems*, pages 76–93, Zikhron-Yaakov, Israel, July 1999.

[BB92] D. M. Butler and S. Bryson. Vector-bundle classes form powerful tool for scientific visualization. *Computers in Physics*, 6(6):576–584, 1992.

[BBA$^+$03] B. Boeckmann, A. Bairoch, R. Apweiler, M. C. Blatter, A. Estreicher, E. Gasteiger, M. J. Martin, K. Michoud, C. O'Donovan, I. Phan, S. Pilbout, and M. Schneider. The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003. *Nucleic Acids Research*, 31(1):365–370, January 2003.

[BDF94] Rupak Biswas, Karen D. Devine, and Joseph E. Flaherty. Parallel, adaptive finite element methods for conservation laws. *Applied Numerical Mathematics: Transactions of IMACS*, 14(1–3):255–283, 1994.

[Ber98] M. Berzins. Mesh quality: A function of geometry, error estimates or both? In *IMR '98: Proceedings of the 7th International Meshing Roundtable*, Dearborn, MI, 1998.

[Ber00] Guntram Berti. *Generic software components for Scientific Computing.* PhD thesis, Faculty of Mathematics, Computer Science, and Natural Science, BTU Cottbus, Germany, 2000.

[BGZ94] R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *Proceedings of the SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Orlando, FL, June 1994. Technical Report 94/9, Department of Computer Science, University of Melbourne.

[BK99] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *VLDB Journal: The International Journal on Very Large Databases*, 8(2):101–119, 1999.

[BL00] Jeanne Behnke and Alla Lake. EOSDIS archive and distribution systems in 2000. In *Proceedings of the IEEE Symposium on Mass Storage Systems*, pages 313–324, 2000.

[Bon93] Anders Bondorf. *Similix 5.0 Manual*, 1993.

[BP89] D. M. Butler and M. H. Pendley. A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, 3(5):45–51, 1989.

[Bri89] E. Brisson. Representing geometric structures in d dimensions: topology and order. In *SCG '89: Proceedings of the 5th Annual Symposium on Computational Geometry*, pages 218–227, New York, NY, USA, 1989. ACM Press.

[BS94a] David Banks and Bart Singer. Vortex tubes in turbulent flows: Identification, representation, reconstruction. Technical report, NASA Langley Research Center, 1994.

[BS94b] A. A. Berlin and R. J. Surati. Partial evaluation for scientific computing: The supercomputer toolkit experience. In *Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne)*, pages 133–141, 1994.

[BW] Jacek Becla and Daniel L. Wang. Lessons learned from managing a petabyte. In *CIDR '05: 2nd Biennial Conference on Innovative Data Systems Research*, Asilomar, CA.

[BW90] Andrew Berlin and Daniel Weise. Compiling scientific code using partial evaluation. *IEEE Computer*, 23(12):25–37, 1990.

[BW93] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Technical Report DIKU 93/22, 1993.

[BWP$^+$99] Antonio M. Baptista, Michael Wilkin, Phillip Pearson, Paul Turner, Cole McCandlish, and Phil Barrett. Coastal and estuarine forecast systems: A multi-purpose infrastructure for the columbia river. *Earth System Monitor, National Oceanic and Atmospheric Administration*, 9(3):1–2, 4–5, 16, 1999.

[CdSP$^+$00] Gilberto Camara, Ricardo Cartaxo Modesto de Souza, Bianca Maria Pedrosa, Lubia Vinhas, Antonio Miguel Vieira Monteiro, Joao Argemiro Paiva, Marcelo Tilio de Carvalho, and Marcelo Gattass. TerraLib: Technology in support of GIS innovation. In *Proceedings of the 2000 Brazilian Symposium on Geoinformatics*, pages 126–133, São Paulo, Brazil, 2000.

[Cod90] E. F. Codd. *The Relational Model for Database Management: Version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[Coq06] Thierry Coquand. Alfa/ agda. In *The Seventeen Provers of the World*, pages 50–54, 2006.

[DG] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th Symposium on Operating System Design and Implementation*. Google, Inc.

[DKL$^+$94] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. Client-Server Paradise. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 558–569, Santiago, Chile, 1994.

[DTP$^+$03] C. G. Degroff, B. L. Thornburg, J. O. Pentecost, K. L. Thornburg, M. Gharib, D. J. Sahn, and A. Baptista. Flow in the early embryonic human heart: A numerical study. *Pediatric cardiology*, 24(4), 2003.

[DWB] Kendra Daly, Stuart Williams, and Steven Bohlen. The joint oceanographic institute. `http://www.joiscience.org/`, viewed September 2006.

[Eas97] J.R. Eastman. *IDRISI User's Guide*. Clark University, Worchester, 1997.

[Ede87] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, Inc., New York, NY, 1987.

[ESR03] ESRI Corporation. ArcGIS: Working with geodatabase topology. Technical report, ESRI, 2003.

[etl05] SAS extraction, transformation and loading (ETL) benchmarking scenario test results on Unisys ES7000 servers. UNISYS, Inc., `http://unisys.com/es7/etl-benchmark`, 2005. Viewed June, 2006.

[FB01] James Frew and Rajendra Bose. An overview of the Earth System Science Workbench. Technical report, Donald Bren School of Environmental Science and Management University of California, Santa Barbara, April 2001.

[FFGM06] Mary Fernandez, Kathleen Fisher, Robert Gruber, and Yitzhak Mandelbaum. PADX: Querying large-scale ad hoc data with XQuery. In *Programming Language Technologies for XML (PLAN-X)*, January 2006.

[FG05] Kathleen Fisher and Robert Gruber. PADS: A domain-specific language for processing ad hoc data. In *PLDI '05: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 295–304, Chicago, IL, 2005.

[FGH04] Leila De Floriani, David Greenfieldboyce, and Annie Hui. A data structure for non-manifold simplicial d-complexes. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pages 83–92, New York, NY, 2004. ACM Press.

[FM95] Leonidas Fegaras and David Maier. Towards an effective calculus for object query languages. *ACM SIGMOD Record*, 24(2):47–58, 1995.

[FP90] Rudolph Fritsch and Renzo Piccinini. *Cellular structures in topology.* Cambridge University Press, 1990.

[FVWZ02] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *SSDBM '02: Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, 2002.

[GLNS+05] Jim Gray, David T. Liu, Maria A. Nieto-Santisteban, Alexander S. Szalay, Gerd Heber, and David DeWitt. Scientific data management in the coming decade. Technical report, Microsoft MSR-TR-2005-10, January 2005.

[GMWH] Perry Greenfield, Todd Miller, Richard L. White, and J. C. Hsu. The numarray library. `http://www.stsci.edu/resources/software_hardware/numarray`, viewed July 2006.

[Gob05] Carole A. Goble. Putting semantics into e-science and grids. In *e-Science*, page 5, 2005.

[GS95] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1995.

[HAC+05] T. Hubbard, D. Andrews, M. Caccamo, G. Cameron, Y. Chen, M. Clamp, L. Clarke, G. Coates, T. Cox, F. Cunningham, V. Curwen, T. Cutts, T. Down, R. Durbin, X. M. Fernandez-Suarez, J. Gilbert, M. Hammond, J. Herrero, H. Hotz, K. Howe, V. Iyer, K. Jekosch, A. Kahari, A. Kasprzyk, D. Keefe, S. Keenan, F. Kokocinsci, D. London, I. Longden, G. McVicker, C. Melsopp, P. Meidl, S. Potter, G. Proctor, M. Rae, D. Rios, M. Schuster, S. Searle, J. Severin,

G. Slater, D. Smedley, J. Smith, W. Spooner, A. Stabenau, J. Stalker, R. Storey, S. Trevanion, A. Ureta-Vidal, J. Vogel, S. White, C. Woodwark, and E. Birney. Ensembl 2005. *Nucleic Acids Research*, 33(Database issue), January 2005.

[HB84] C.M. Hung and P.G. Buning. Simulation of blunt-fin induced shock wave and turbulent boundary layer separation. In *AIAA Aerospace Sciences Conference*, 1984.

[hdf04] HDF5: API specification reference manual. National Center for Supercomputing Applications (NCSA), `http://hdf.ncsa.uiuc.edu/`, 2004. Viewed July 2006.

[HG05] Gerd Heber and Jim Gray. Supporting finite element analysis with a relational database backend; part 1: There is life beyond files. Technical report, Microsoft MSR-TR-2005-49, April 2005.

[HH00] Desmond J. Higham and Nicholas J. Higham. *MATLAB Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[HHO$^+$96] S. Hankin, D. E. Harrison, J. Osborne, J. Davison, and K. O'Brien. A strategy and a tool, Ferret, for closely integrated visualization and analysis. 7(3):149–157, July 1996.

[HLC91] R. Haber, B. Lucas, and N. Collins. A data model for scientific visualization with provision for regular and irregular grids. In *IEEE Visualization*, pages 298–305. IEEE Computer Society Press, 1991.

[HM04a] Bill Howe and David Maier. Algebraic manipulation of scientific datasets. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases*, Toronto, Ontario, CA, 2004.

[HM04b] Bill Howe and David Maier. Algebraic manipulation of scientific datasets. 14(4), 2004.

[HM04c] Bill Howe and David Maier. Logical and physical data independence for file-based scientific applications. *IEEE Data Eng. Bull.*, 27(4):30–37, 2004.

[HM05] Bill Howe and David Maier. Retrofitting a data model to existing environmental data. In *SSDBM '05: Proceedings of the 17th International Conference on Scientific and Statistical Datbase Management*, Santa Barbara, CA, June 2005.

[HMB03] Bill Howe, David Maier, and Antonio Baptista. A language for spatial data manipulation. *Journal of Environmental Informatics*, 2(2), December 2003.

[HMB07] Bill Howe, David Maier, and Laura Bright. Smoothing the ROI curve for scientific data management applications. In *CIDR '07: Proceedings of the Third Biennial Conference on Innovative Data Systems Research*, Ansilomar, CA, 2007.

[HMG94] Hans Hinterberger, Kathrin Anne Meier, and Hans Gilgen. Spatial data reallocation based on multidimensional range queries – a contribution to data management for the earth sciences. In *SSDBM '94: Proceedings of the 7th International Conference on Scientific and Statistical Database Management*, pages 228–239. IEEE Computer Society, 1994.

[HPD$^+$05] Gerd Heber, Chris Pelkie, Andrew Dolgert, Jim Gray, and David Thompson. Supporting finite element analysis with a relational database backend; part 3: Openddx – where the numbers come alive. Technical report, Microsoft MSR-TR-2005-151, November 2005.

[HT04a] K. Hormann and M. Tarini. A quadrilateral rendering primitive. *Workshop on Graphics Hardware*, 2004.

[HT04b] Weicheng Huang and Danesh K. Tafti. A parallel adaptive mesh refinement algorithm for solving nonlinear dynamical systems. *Int. J. High Perform. Comput. Appl.*, 18(2):171–181, 2004.

[IBM93] IBM Corporation. *IBM Visualization Data Explorer User Guide*, 4th edition, 1993.

[Jir94] Gregory A. Jirak. *The Aurora Dataserver Programmer's Guide.* Palo Alto, CA, 1994.

[JJ99] Mark Jones and Simon Peyton Jones. Lightweight extensible records for Haskell. In *Proceedings of the Haskell Workshop*, Paris, France, 1999.

[JS92] Harry L. Jenter and Richard P. Signell. NetCDF: A public-domain-software solution to data-access problems for numerical modelers. Unidata, 1992.

[KaC$^+$01] T. Kurc, U. atalyurek, C. Chang, A. Sussman, and J. Salz. Exploration and visualization of very large datasets with the active data repository. Technical report, CS-TR4208, University of Maryland, 2001.

[Kov89] V. A. Kovalevsky. Finite topology as applied to image analysis. *Comput. Vision Graph. Image Process.*, 46(2):141–161, 1989.

[LCA$^+$03] B. Lee, T. Critchlow, G. Abdulla, C. Baldwin, R. Kamimura, R. Musick, R. Snapp, and N. Tang. The framework for approximate queries on simulation data. *Informatics and Computer Science: An International Journal*, 157(1-2):3–20, December 2003.

[LFA$^+$] Francis Loth, Paul F. Fischer, Nurullah Arslan, Christopher D. Bertram, Seung E. Lee, Thomas J. Royston, Ruo-Hua Song, Wael E. Shaalan, and Hisham S. Bassiouny. Transitional flow at the venous anastomosis of an arteriovenous graft: Potential relationship with activation of the ERK1/2 mechanotransduction pathway. *Jurnal of Biomechanical Engineering*. (submitted).

[LH05] Robert S. Laramee and Helwig Hauser. Interactive 3D flow visualization based on textures and geometric primitives. In *NAFEMS World Congress Conference Proceedings*, St. Juliens Bay, Malta, 2005. The International Association for the Engineering Analysis Community.

[Lie94] Pascal Lienhardt. N-dimensional generalized combinatorial maps and ceellular quasi-manifolds. *International Journal of Computational Geometry and Applications*, 4(3):275–324, 1994.

[LK94] Meng-Sing Liou and Kai-Hsiung Kao. Progress in grid generation: From Chimera to DRAGON grids. Technical Report E-9071, NASA, August 1994.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-specific Languages*, pages 109–122. ACM Press, 1999.

[LMW96] Leonid Libkin, Rona Machlin, and Limsoon Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *SIGMOD '96: Proceedings of the 1996 ACM International Conference on Management of Data*, pages 228–239, 1996.

[Mar58] A. Markov. Insolubility of the problem of homeomorphy. In *Proceedings of the International Congress of Mathematics*, pages 300–306. Cambridge University Press, 1958.

[MC99] Ron Musick and Terence Critchlow. Practical lessons in supporting large-scale computational science. *ACM SIGMOD Record*, 28(4):49–57, 1999.

[MC03] Jim Myers and Alan Chappell. Binary format description language. Technical report, Pacific Northwest National Laboratory, 2003.

[MM04] C. McBride and J. McKinna. The view from the left. *The Journal of Functional Programming*, 14(1):69–111, January 2004.

[MMJ$^+$01] Jim Melton, Jan-Eike Michels, Vanja Josifovski, Krishna Kulkarni, Peter Schwarz, and Kathy Zeidenstein. SQL and management of external data. *ACM SIGMOD Record*, 30(1):70–77, 2001.

[Mor01] Patrick Moran. Field model: An object-oriented data model for fields. Technical report, NASA Ames Research Center, 2001.

[MS97] Arunprasad P. Marathe and Kenneth Salem. A language for manipulating arrays. In *VLDB '97: Proceedings of 23rd International Conference on Very Large Data Bases*, pages 46–55, 1997.

[MSK$^+$] Donna M. Muzny, Steven E. Scherer, Rajinder Kaul, Jing Wang, Jun Yu, Ralf Sudbrak, Christian J. Buhay, Rui Chen, Andrew Cree, Yan Ding, Shannon Dugan-Rocha, Rachel Gill, Preethi Gunaratne, Alan R. Harris, Alicia C. Hawes, Judith Hernandez, Anne V. Hodgson, Jennifer Hume, Andrew Jackson, Ziad M. Khan, Christie Kovar-Smith, Lora R. Lewis, Ryan J. Lozado, Michael L. Metzker, Aleksandar Milosavljevic, George R. Miner, Margaret B. Morgan, Lynne V. Nazareth, Graham Scott, Erica Sodergren, Xing-Zhi Song, David Steffen, Sharon Wei, David A. Wheeler, Mathew W. Wright, Kim C. Worley, Ye Yuan, Zhengdong Zhang, Charles Q. Adams, Ali M. Ansari-Lari, Mulu Ayele, Mary J. Brown, Guan Chen, Zhijian Chen, James Clendenning, Kerstin P. Clerc-Blankenburg, Runsheng Chen, Zhu Chen, Clay Davis, Oliver Delgado, Huyen H. Dinh, Wei Dong, Heather Draper, Stephen Ernst, Gang Fu, Manuel L. Gonzalez-Garay, Dawn K. Garcia, Will Gillett, Jun Gu, Bailin Hao, Eric Haugen, Paul Havlak, Xin He, Steffen Hennig, Songnian Hu, Wei Huang, Laronda R. Jackson, Leni S. Jacob, Susan H. Kelly, Michael Kube, Ruth Levy, Zhangwan Li, Bin Liu, Jing Liu, Wen Liu, Jing Lu, Manjula Maheshwari, Bao-Viet Nguyen, Geoffrey O. Okwuonu, Anthony Palmeiri, Shiran Pasternak, Lesette M. Perez, Karen A. Phelps, Farah J. H. Plopper, Boqin Qiang, Christopher Raymond, Ruben Rodriguez, Channakhone Saenphimmachak, Jireh Santibanez, Hua Shen, Yan Shen, Sandhya Subramanian, Paul E. Tabor, Daniel Verduzco, Lenee Waldron, Jian Wang, Jun Wang, Qiaoyan Wang, Gabrielle A. Williams, Gane K. S. Wong, Zhijian Yao, Jingkun Zhang, Xiuqing Zhang, Guoping Zhao, Jianling Zhou, Yang Zhou, Further Contributors, David Nelson, Hans Lehrach, Richard Reinhardt, Susan L. Naylor, Huanming Yang, Maynard Olson, George Weinstock, and Richard A. Gibbs. The DNA sequence, annotation and analysis of human chromosome 3. *Nature*, 440(7088):1194–1198.

[MtG$^+$99] S. Mackinson, L. Nøttestad, S. Guénette, T. Pitcher, O.A. Misund, and A. Ferno. Cross-scale observations on distribution and behavioural dynamics of ocean feeding norwegian spring-spawning herring. *ICES Journal of Marine Science*, 56:613–626, 1999.

[MWN$^+$04] Xiaosong Ma, Marianne Winslett, John Norris, Xiangmin Jiao, and Robert Fiedler. Godiva: Lightweight data management for scientific visualization applications. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 732, Washington, DC, USA, 2004. IEEE Computer Society.

[Nat06] National Climatic Data Center. NCEP AWIPS eta model data. `http://nomads.ncdc.noaa.gov:9090/dods/NCDC_NOAAPort_ETA`, 2006. viewed April 2006.

[ND05] Jan Newton and Brian Dushaw. Northwest Association of Networked Ocean Observing Systems. `http://www.nanoos.org`, 2005. viewed August 2006.

[Nie] Juha Nieminen. Function parser for C++. `http://warp.povusers.org/FunctionParser/`.

[NS03] A. Nenes and J. H. Seinfeld. Parameterization of cloud droplet formation in global climate models. *Journal of Geophysical Research (Atmospheres)*, 108(14):7–1, July 2003.

[OLNS$^+$05] William O'Mullane, Nolan Li, Maria A. Nieto-Santisteban, Alexander S. Szalay, Aniruddha R. Thakar, and Jim Gray. Batch is back: Casjobs, serving multi-tb data on the web. Technical report, Microsoft MSR-TR-2005-19, February 2005.

[ooi] The Ocean Observing Initiative. `http://www.orionprogram.org/documents/default.html`. viewed August 2006.

[ope05] OPeNDAP: Open-source project for a network data access protocol. `http://opendap.org/`, 2005. viewed August 2006.

[PAL$^+$06] Stratos Papadomanolakis, Anastassia Ailamaki, Julio C. Lopez, Tiankai Tu, David R. O'Hallaron, and Gerd Heber. Efficient query processing on unstructured tetrahedral meshes. In *SIGMOD '06: Proceedings of the 2006 ACM International Conference on Management of Data*, pages 551–562, New York, NY, USA, 2006. ACM Press.

[PCD+03] Line Pouchard, Luca Cinquini, Bob Drach, Don Middleton, David E. Bernholdt, Kasidit Chanchio, Ian T. Foster, Veronika Nefedova, David Brown, Peter Fox, Jose Garcia, Gary Strand, Dean Williams, Ann L. Chervenak, Carl Kesselman, Arie Shoshani, and Alex Sim. An ontology for scientific information in a grid environment: the Earth System Grid. In *CCGrid '03: Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, pages 626–632, Tokyo, Japan, 2003.

[PWN] Mark Papiani, Jasmin Wason, and Denis A. Nicole. An architecture for management of large, distributed, scientific data using SQL/MED and XML.

[RBS02] Philip J. Rhodes, R. Daniel Bergeron, and Ted M. Sparr. Database support for multisource multiresolution scientific data. In *SOFSEM*, pages 94 – 114, 2002.

[SA02] Etzard Stolte and Gustavo Alonso. Efficient exploration of large scientific databases. In *VLDB '02: Proceedings of the 28th International Conference on Very Large Data Bases*, pages 622–633, 2002.

[SAB+05] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 553–564, Trondheim, Norway, 2005. VLDB endowment.

[SBM$^+$00] D.C. Steere, A.M. Baptista, D. McNamee, C. Pu, and J. Walpole. Research challenges in environmental observation and forecasting systems. In *In Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (Mobicom 2000)*, 2000. Boston, Mass.

[SCESL02] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *VIS '02: Proceedings of the 2002 IEEE Conference on Visualization*, 2002. Course Notes for Tutorial 4.

[Sch02] Michael Schneider. Stroke busters in turbulent blood. Projects in Scientific Computing, Pittsburgh Supercomputing Center, `http://www.psc.edu/science/2002/tufo/stroke_busters_in_turbulent_blood.html`, 2002.

[SCZ05] M. Stonebraker, U. Cetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, December 2005.

[SDB83] J. L. Steger, F. C. Dougherty, and J. A. Benek. A Chimera grid scheme: Advances in grid generation. In K. N. Ghia and U. Ghia, editors, *Advances in Grid Generation*, volume 5, pages 59–69, 1983.

[SGT$^+$02] Alexander S. Szalay, Jim Gray, Ani Thakar, Peter Z. Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan van den Berg. The SDSS skyserver: Public access to the Sloan Digital Sky Server data. In *SIGMOD '02: Proceedings of the 2002 ACM International Conference on Management of Data*, pages 570–581, 2002.

[SHRL06] Sandro Santilli, Chris Hodgson, Paul Ramsey, and Jeff Lounsbury. Postgis. http://postgis.refractions.net/, 2006. Viewed June 2006.

[SHT$^+$77] Nan C. Shu, Barron C. Housel, R. W. Taylor, Sakti P. Ghosh, and Vincent Y. Lum. EXPRESS: A data EXtraction, Processing, amd REStructuring System. *ACM Transactions on Database Systems*, 2(2):134–174, 1977.

[SKT$^+$00] Alexander S. Szalay, Peter Z. Kunszt, Ani Thakar, Jim Gray, Don Slutz, and Robert J. Brunner. Designing and mining multi-terabyte astronomy archives: the Sloan Digital Sky Survey. In *SIGMOD '00: Proceedings of the 2000 ACM International Conference on Management of Data*, pages 451–462, 2000.

[SML96] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *VIS '96: Proceedings of the 7th conference on Visualization*, pages 93–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[SRH90] Michael Stonebraker, Lawrence A. Rowe, and Michael Hirohama. The implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, 1990.

[Sri95] R. Srinivasan. XDR: External data representation standard, RFC 1832. Technical report, Sun Microsystems, 1995.

[STH02] Chris Stolte, Diane Tang, and Pat Hanrahan. Query, analysis, and visualization of multidimensional relational databases. In *SIGKDD '02: Proceedings of the 8th ACM International Conference on Knowledge Disovery and Data Mining*, Edmonton, Alberta, Canada, 2002.

[SvPAG] Etzard Stolte, Christoph von Praun, Gustavo Alonso, and Thomas Gross. Scientific data repositories: Designing for a moving target. In

ACM, editor, *SIGMOD '03: Proceedings of the 2003 ACM International Conference on Management of Data*, pages 349–360, New York, NY 10036, USA, June. ACM Press.

[Thi98] P. Thiemann. *The PGG System – User Manual*, 1998. University of Nottingham, Nottingham, England.

[TKSG02] Ani Thakar, Peter Kunszt, Alex Szalay, and Jim Gray. Migrating a multiterabyte archive from object to relational databases. *Computers in Science and Engineering*, 5(5):16–29, 2002.

[TM03] Peter A. Tucker and David Maier. Dealing with disorder. In *Workshop on Management and Processing of Data Streams*, San Diego, CA, June 2003.

[Tre99] L. Treinish. A function-based data model for visualization. In *Proceedings of the 1999 IEEE Conference on Visualization: Late Breaking Hot Topics*, pages 73–76, San Francisco, CA, October 1999.

[TS00] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[unc] Volume visualization datasets. `http://www.siggraph.org/education/materials/vol-viz/volume_visualization_data_sets.htm`, Viewed August 2006.

[vola] Dataset courtesty of General Electric. `http://www.volvis.org`. Viewed September 2006.

[volb] Dataset courtesy of Sonderforschungsbereich 382, German Research Council. `http://www.volvis.org`. Viewed September 2006.

[vtk] The Visualization Toolkit. http://www.vtk.org. Viewed April 2006.

[Wat02] Paul Watson. Topology and ORDBMS technology. Technical report, Laser-Scan, 2002.

[WB97] Norbert Widmann and Peter Baumann. Towards comprehensive database support for geoscientific raster data. In *GIS '97: Proceedings of the 5th International Workshop on Advances in Geographic Information Systems*, pages 54–57, Las Vegas, NV, 1997.

[WB98] Norbert Widmann and Peter Baumann. Efficient execution of operations in a DBMS for multidimensional arrays. In *SSDBM '98: Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pages 155–165, Capri, Italy, July 1998.

[WB03] Martin Westhead and Mark Bull. Representing scientific data on the Grid with BinX - binary XML description language. Technical report, EPCC, University of Edinburgh, 2003.

[Wel00] D. C. Wells. *The FITS Experience: Lessons Learned.* Kluwer Academic Publishers, 2000.

[WG93] Richard H. Wolniewicz and Goetz Graefe. Algebraic optimization of computations over scientific databases. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 13–24, Dublin, Ireland, 1993. Morgan Kaufmann Publishers Inc.

[who] The whole frog project. Lawrence Berkeley National Laboratory, `http://froggy.lbl.gov/`. Viewed August 2006.

[ZB04] Yongjie Zhang and Chandrajit Bajaj. Finite element meshing for cardiac analysis. Technical Report ICES 04-26, University of Texas at Austin, 2004.