Smoothing the ROI Curve for Scientific Data Management Applications

Bill Howe
Portland State University
1825 SW Broadway
Portland OR 97207
howe@cs.pdx.edu

David Maier
Portland State University
1825 SW Broadway
Portland OR 97207
maier@cs.pdx.edu

Laura Bright
Thetus Corporation
34 NW 1st Avenue, Suite 210
Portland, Oregon 97209
bright@cs.pdx.edu

ABSTRACT

Physical scientists increasingly use large, shared data repositories to make discoveries. The technology to manage these repositories tends to be developed ad hoc; database technology has not significantly penetrated this market. Diverse requirements, terabyte and petabyte scale, non-standard data types, and rapid change are the norm for scientific data management applications; each pose challenges for existing technology.

In this paper, we argue that traditional database systems are too monolithic and uncompromising to be generally successful in these extreme environments. We argue that these negative characteristics are reflected in the crooked shape of the Return On Investment curve, and that a smoother ROI can be achieved by adopting a few simple strategies. We then describe our experiences applying these strategies to build data services for an Environmental Observation and Forecasting System.

Categories and Subject Descriptors

H.2.8 [**Database Systems**]: Database Applications – *scientific databases.*

General Terms

Management, Design

Keywords

ROI, metadata, scientific data management, software engineering

1. INTRODUCTION

Scientific data acquisition is no longer a bottleneck to scientific discovery. Physics [39] earth science [12] biology [1], and astronomy [41] have all seen enormous growth in the amount of data available for study. Scientists are "starting to die from data" [11] – they are becoming increasingly inefficient due to time spent on data management tasks. Software supporting these tasks are usually developed ad hoc in response to specific requirements. General-purpose database management systems (GPDBMS) have not been widely adopted in the scientific community, despite some prominent successes [41]. Reasons cited for eschewing GPDBMS include poor performance, lack of support for essential data types (multidimensional arrays, irregular meshes, timeseries, hierarchies), high cost of deployment and maintenance, and poor integration with existing tools [5, 18].

In this paper, we argue that the "all or nothing" nature of database technology is responsible for their lack of acceptance. Scientific applications have extreme requirements: enormous scale [5, 41], novel data types [23, 30], and diverse access patterns [39]. In order to compete with the flexible tools already in use in these extreme environments (the Unix filesystem, FORTRAN, MATLAB, C, Perl), database projects must reward incremental effort with incremental returns. Existing database technology, however, requires enormous up front effort and has a transformational effect on the data landscape. Reliable legacy applications must be rewritten to interact with the database. Users must learn new skills. New employees must be hired to maintain the system, or else programmers and system administrators must shoulder a significant new burden.

We use the shape of the Return on Investment (ROI) curve as a success indicator for scientific data management applications. We offer general strategies for achieving a smooth ROI by building loosely-coupled services, instead of deploying a GPDBMS (though we do identify where a GPDBMS can be used as a component). We will illustrate the strategies using examples from the literature and our own experience with the CORIE Environmental Observation and Forecasting System [2, 23] and its evolution into a part of the Center for Coastal Margin Observation and Prediction (CMOP) [12]. The CORIE system is designed to support research, policy, and commercial interests in the estuarine and ocean waters in the Pacific Northwest and elsewhere.

The bursty return on investment (ROI) produced by GPDBMS projects can be visualized by plotting ROI against time. Return On Investment (ROI) is usually reported as an annual percentage of the initial project cost. A 100% ROI means that every year, the project generates savings (or revenue) equal to its initial cost. A better interpretation of ROI for software projects serving scientists is time saved: How much time must scientists spend on data management relative to the status quo? More precisely, we calculate ROI for a solution X at time t as:

$$ROI_X(t) = \sum_{i=0}^{t} T_{DN}(i) - T_X(i)$$

3rd *Biennial Conference on Innovative Data Systems Research (CIDR)* January 7-10, 2007, Asilomar, California, USA.

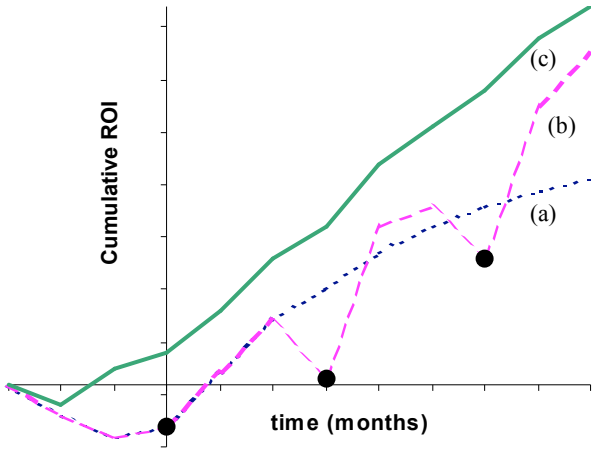

Figure 1. Return on Investment curves with different characteristics. (a) ROI for a single-release project. (b) The crooked ROI of a successful multiple-release project. (c) A smooth ROI curve representing incremental but continuous improvement.

where $T_X(i)$ is the (estimated) time spent on data management tasks under solution X during time period i, and DN is the "Do Nothing" alternative. Negative ROI indicates that, at time t, scientists have spent more time on data management tasks, including pre-deployment activities such as design meetings, than they have saved by using the new software.

In Figure 1, we characterize ROI curves for three methodologies applied to the same hypothetical project. Curve (a) assumes a single release date at $t = 0$. We interpret the "Do Nothing" alternative "do nothing right now, but perhaps do something in the future." That is, we acknowledge that after a few months, new technology might become available or the existing system may be upgraded in some significant but unpredictable way, therefore lowering T_{DN}. Curve (a) is therefore sub-linear to capture the effect of this opportunity cost.

Curve (b) represents a continuously developed system with a three-month release cycle. We assume that the initial release costs are identical to those of Curve (a). Just before each release, the benefits of the system drop commensurate to the additional design, implementation, and testing effort. After each release, benefits improve, hopefully enough to justify the cost of the release. The opportunity costs do not affect Curve (b) as much, since each release offers a chance to incorporate new technologies or otherwise adapt.

Curve (c) represents the potential improvement offered by a smoother, more incremental methodology. Note that the y-axis carries less significance for this curve; there is no pre-defined project completion date. Rather, the system is adapted to handle additional and changing requirements incrementally in each time period.

Note that we have chosen to illustrate successful projects. After the second release on Curve (b), for example, it is possible that the benefit would stay flat or slope downward due to bugs, misunderstood requirements, or other problems. These problems would not be apparent until after spending a considerable amount on the release. A smoother ROI methodology enhances agility: unneeded or buggy features can be identified and rolled back earlier.

In the next section, we will develop the argument that 1) a smooth ROI is required specifically and uniquely in the context of scientific data management applications, and 2) a GPDBMS cannot deliver a smooth ROI. Following that argument, we will recommend a strategy for system designers to provide a smooth ROI, and thereby penetrate the scientific data management market.

2. The Case for a Smooth ROI

We have established an interpretation of the shape of the ROI curve, and reasoned that a smooth ROI is preferred. Our thesis, however, is that the success of scientific data management applications, in particular, is correlated with a smooth ROI, and that this property is specific to this domain.

The reader may be thinking of many general design principles that implicitly advocate a smoother ROI. The debate of modular versus monolithic (e.g., for Operating Systems) is relevant, as are component-based programming paradigms. More recently, Service Oriented Architectures [45] have been proposed as a means of decoupling subsystems by exploiting XML and HTTP. While we support these initiatives for particular application domains (especially scientific data management), we suggest that there are circumstances where a monolithic design and the resulting crooked ROI are desirable.

Large commercial data management applications do not always benefit from a smooth ROI. A smooth ROI is difficult to realize for the transformational projects found in commercial domains. Large-scope software projects have been the norm for at least 15 years in commercial IT departments. Enterprise Resource Planning (ERP), Customer Relationship Management (CRM), Sales Force Automation (SFA), and more recently Business Process Intelligence (BPI) projects all have tremendous scope, affecting the majority of a company's IT infrastructure. Such large projects are necessarily transformational, changing the way a company does business in a fundamental ways. The periods of uncertainty between major releases are necessarily long for these transformational projects, and the ROI curve is necessarily crooked.

We believe that such transformation is not just unavoidable, but desirable. Rarely do companies spend 10s of millions of dollars on software that does not promise to help them "re-engineer" their processes. Monolithic database technology has been central to these projects, perhaps even influencing the trend. These systems offer enormous benefits if one is willing to surrender control to the database and can endure a long-term deployment.

Scientists, however, are generally not looking for fundamental transformations to their procedures, but rather to augment them so that less time is spent finding and manipulating data and more time is spent making discoveries. They want their process streamlined, but not necessarily replaced.

Scientific research projects typically have *science programmers* on staff, domain specialists with significant programming experience. Science programmers tend to naturally borrow from *extreme programming* [4]: do not design for the future, solve problems as they arise, cheerfully redesign if necessary. Computer scientists

may sometimes look dimly on the result, seeing only a tangle of interdependent Perl scripts, C programs, and cron jobs, all processing various ad hoc data formats. However, these systems tend to work, and work rather efficiently.

Our recommendation is to let science programmers be extreme programmers. Offer them services that enhance their ability to implement desired features, rather than replace their trusted facilities with something unfamiliar. The services should reflect the principles and results of the database community, but packaged in a manner palatable to efficacious and confident programmers who have their own methods of software development.

Scientific Data Management Applications demand the flexibility indicated by a smooth ROI. A smooth ROI curve implies near-continuous deployment of new applications and new features and therefore more frequent opportunities to steer the project. This flexibility is important in a scientific research environment for the following reasons:

First, flexibility is required to make the best use out of limited staff. Scientific research staff frequently have multiple roles within the institution, due to requirements for specialized domain knowledge and limited IT budgets. For example, a staff member on the CORIE project writes and maintains domain-specific data analysis algorithms, but also manages the project's website. This situation means that those developing a data management system may also be users of the system. Frequent opportunities to change roles from developer to user maximizes their utility on the project. Research staff and their managers have more flexibility when deciding whether to "fish" (polish and exploit their newly developed technology) or "cut bait" (design and implement additional features).

Second, scientific research environments rarely have a large number of users with identical (or even similar) requirements. For example, a company wishing to upgrade its call center defines a single set of requirements. The benefit of meeting these requirements is multiplied by the large number of "identical" users. Scientific staff tend to each ask different kinds of queries and require different kinds of interfaces. Addressing each scientist's individual requirements incrementally guarantees positive progress. The alternative is to identify the union of everyone's individual requirements and make that a design target. Due to the diversity of tasks, this set is expected to be rather large; a comprehensive system may not be immediately feasible.

Third, requirement changes are even more likely than in commercial environments. A fixed set of requirements suggests a level of certainty about how the system will be used and what the users are trying to accomplish. By definition, however, scientists cannot predict exactly what their research requires. For example, CORIE was initially designed to study the Columbia River estuary. Now CORIE runs 19 forecasts daily for estuaries and bays along the Northwest coast of the United States and around the world. Even with an experienced DBA on staff, a GPDBMS solution would not have saved much effort compared to an application-specific solution, due to the extensive schema changes required to accommodate multiple regions, multiple coordinate systems, additional users, and additional data products.

The RHESSI Experimental Data Center [39] is an example of a system designed explicitly to support diverse and changing usage patterns. Stolte et al. recognized "the need to accommodate constant change," and argue that such "a variety of data views cannot be efficiently supported by any scientific data repository." They instead opt for an extensible design that can be grown incrementally as new requirements general applicability of these techniques remains to be seen, but the work is certainly aligned with our observations.

Current database technology necessarily implies a crooked ROI. Database systems have traditionally been monolithic, offering a thorough set of features but demanding complete control over the data managed. Decoupling features from one another is difficult or impossible. The ability to substitute a storage manager that exploits existing file organizations (e.g., netCDF [28], or HDF [22]) paired with the traditional query language facilities would make a commercial database system much more enticing, for example. The developers of Shore [9] proposed such a mechanism. Previous research on mediators [36] and rich user-defined types [38] for accessing native data outside the database and loading native data into the database, respectively, are relevant here. Also, many scientific file formats are basically storage formats for a single data type: multidimensional arrays. Native database support for multidimensional arrays has been proposed [3, 33], though a monolithic architecture and demand for complete control of the data still plague the design.

3. SMOOTHING STRATEGIES

Based on our experience and a review of the literature, we recommend the following strategies for developing data management solutions for the scientific community, examples of which we provide in Section 4:

Promote "pay as you go" solutions. Recently, *dataspace management* has been proposed as an approach that takes a holistic view of all information in an enterprise, be it structured, semi-structured or unstructured, and whether or not it is supported by an explicit information system [15, 20]. While its goal is to provide services over the entirety of the data, dataspace management takes the pragmatic view that initial limits on time and effort may only permit simple services at first, such as cataloging and keyword search. Additional services or capabilities can come later, little by little, as resources permit.

Let a hundred flowers blossom. Consider deploying two or more redundant solutions rather than forcing a guess. Although fielding multiple parallel solutions flattens the ROI curve, you avoid failures that result in negative ROI. To mitigate the extra effort of fielding multiple solutions, consider limiting features: only one or two capabilities, implemented end-to-end, should be sufficient initially.

Consider that Amazon.com fielded web services with both a RESTful [13] interface and a SOAP [37] interface. They found that the RESTful interface saw 85% more traffic than the SOAP interface. Forcing an up-front decision to use one technology over the other would have obscured this evidence that developers prefer REST.

In the CORIE system, four different web-based mapping tools are in production: Mapserver based on PHP [44], the OpenLayers javascript library (providing rich client side functionality) [35], KaMap (extending a rich client with server-side caching) [29], and the popular Google Maps [16]. Rather than conduct a lengthy

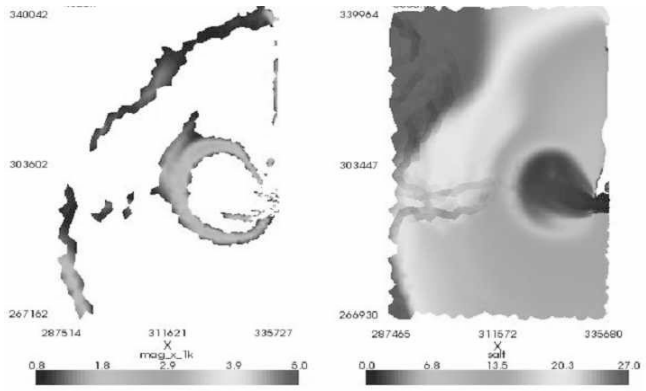

Figure 2. A data product developed with the Gridfield Algebra visualizing the plume of freshwater at the mouth of the Columbia River. The right pane is shaded by salinity. The left pane is restricted to regions of high salinity gradient—one way to define the plume front.

evaluation of these competing technologies, CORIE programmers simply developed one or more potentially redundant mini-apps with each as tool as it was discovered and made them available for use.

Specialize to the current instance. Extreme programmers [4] advise against over-design and attempts at future-proofing. Better, they say, to build software that works in the current environment and be prepared to rebuild it when things change. The analog for a data management might be "Extreme Schema Design." Build structures and indexes that work for the current dataset and query workload, and be prepared to drop and reload data when things change. The "rebuild and reload" procedures can and should be at least partially automated to reduce the cost of their frequent use.

Of course, schema changes break applications, and frequent schema changes could prevent applications from ever working. The solution here is to have a robust API – a level of indirection between the physical structures used to house the data and the applications requesting the data.

We exercise this strategy on the Quarry project [25], complete with a robust API. We put no constraints on the metadata we collect, and we index it automatically using the patterns in the data. If users change the form or content of the metadata they track, our index may no longer be efficient. We respond by re-harvesting, re-loading, and re-indexing the new metadata, in a single operation, as the need arises.

Strive for zero configuration. GPDBMS tend to require more configuration and design effort than systems of similar scope and complexity (e.g, operating systems and web servers). Many areas are ripe for automation, including configuration, physical database design [7], maintenance, and even logical database design [25]. For inspiration, consider that the Google appliance [16] begins to provide search capabilities after being simply plugged into a server rack.

This strategy allows early live access to real data, which can reveal unforeseen problems (we discovered a crucial misspelling, for example). Further, feedback on actual user needs is easier to gather with concrete examples of data idiosyncrasies available.

Operate on In Situ Data. Apply your logical model to their physical representation. It may be possible to re-implement logical operators over in situ data, but this approach can result in a separate implementation for every new application. Converting data en masse to native formats is also a possibility, but this process tends to disrupt legacy systems. Wrapping schemes (e.g., Garlic [36]) avoid these two extremes, but incremental development and deployment of wrappers has received less attention.

The Aqualogic system from BEA [10] provides a Service Oriented development platform for writing wrappers declaratively that produce XML and then composing and transforming the XML with XQuery. Aqualogic can "introspect" relational and XML sources to automatically make them available as services. Introspection considers more than just the structure of the data model; foreign key relationships, for example, are interpreted as nesting in the produced XML. These services are not opaque to the query optimizer. Selection predicates and joins can be pushed down into the relational engine to improve performance.

4. CASE STUDIES

We have been working extensively with environmental scientists who are developing the CORIE Environmental Observation and Forecasting System (EOFS) at the OGI School of Science & Engineering at Oregon Health & Science University. The CORIE system [2, 12, 23], 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. For example, fisheries are interested in the location of the plume front (Figure 2), because salmon are known to congregate there.

The CORIE project has cultivated a dataspace consisting of three loosely-coupled participant subsystems: the Forecast Factory, the Hindcast Archive, and the Observation Pipeline. The Forecast Factory manages an increasing number of forecasts for bodies of water around the world (currently 19). The priorities of the Forecast Factory are daily reliability first, and accuracy second. The Hindcast Archive is a collection of repositories covering various time periods from 1880 to the present using different code versions and configurations. The top priority of the Hindcast Archive is to provide consistent long-term coverage, and to offer a "best available" model for any given time period. The Observation Pipeline provides reliable near-real-time access to the measurements of a sensor network consisting of 18-25 fixed stations around the Pacific Northwest. Data from vessel cruises, underwater autonomous vehicles, and other mechanisms are also available or forthcoming.

In this section, we will describe three tools designed to support the CORIE dataspace: the Quarry metadata engine, the ForeMan Forecast Manager, and a toolkit for retrofitting a data model to ad hoc file formats. First, we briefly introduce each tool.

The Quarry engine was developed to provide browse and search capabilities for the global data inventory [25]. Each of the three subsystems (the Forecast Factory, the Hindcast Archive, and the Observation Pipeline) has a footprint on the shared filesystem consisting of plots, images, documents, configuration files, and

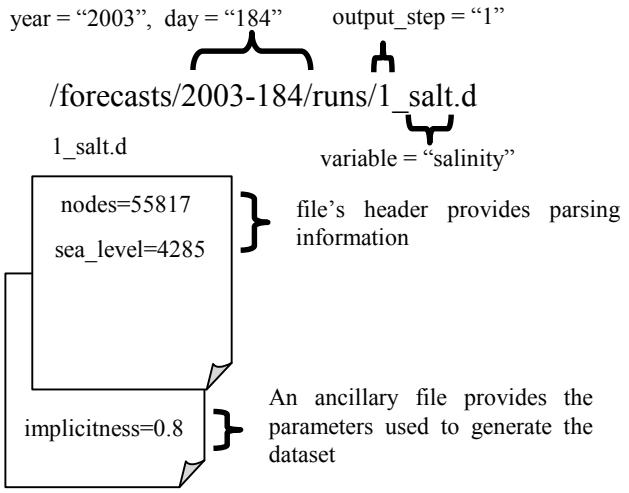

Figure 3. Property-value pairs extracted for a file.

annotations. The Quarry tool allows access to these resources by logical metadata rather than by physical location without requiring an integrated schema design and without retreating to full-text indexing services.

The ForeMan Forecast Manager [6] provides an interface for inspecting, analyzing, and experimenting with different assignments of forecast jobs to compute nodes. ForeMan relies on information extracted from existing developer log files and was deployed with only a small amount of extra work for the developers. We note that log file formats are subject to change, which may require modifications to the existing ForeMan implementation. However, we believe the advantages of rapid deployment and minimal work for developers outweigh this potential overhead.

The third project we describe involves a toolkit for retrofitting a data model onto existing environmental data [24]. The toolkit was developed in conjunction with the Gridfield algebra, a language for manipulating simulation results in the physical sciences—including both forecasts and hindcasts. To allow gridfield-powered programs access to ad hoc file formats, we use simple declarative descriptions of filesystem data to generate access methods. We prefer this approach to the two obvious alternatives: converting the data en masse to a new format or hand-coding individual wrappers for each data type.

4.1 Quarry Metadata Manager

CORIE data produced or consumed by any of the three subsystems, including supplementary data such as derived images, configuration settings, annotations, and logs, are stored as files using naming conventions specific to the subsystem by which or for which they were created. Metadata is encoded in the file names, directory structures, file headers, or in ancillary files (Figure 3). Scientific inquiries involve accessing data from one or more of these subsystems.

Different people know about different datasets and data products. How can we regularize the metadata to let anyone find and retrieve any file? We could try to negotiate a global metadata schema up front, or we could get started by just gathering the facts that are already available. The Quarry system takes the latter approach.

Quarry has three components, which we describe in this section: the Harvester, the Triple Store, and the Quarry Metadata Explorer (QME) web interface.

4.1.1 Harvester

Anyone with files they wish to describe writes a script that, given a file name, produces a set of (property, value)-pairs describing the file. Scripts may be written in any language, and there are no restrictions on the properties or values they may use.

Each script is associated with a regular expression over the file path identifying those files for which it is applicable. The harvester traverses a filesystem and evaluates each regular expression against the full path of each file. On a match, the harvester calls the script with the name of the file provided as an argument. The script accesses whatever information it needs and calls a function Assert(prop, val) to emit a metadata descriptor. The harvester gathers all the descriptors for a particular resource, merges duplicate property assertions into multi-valued descriptors, then writes them to a descriptor file formatted to be compatible with PostgreSQL’s bulk load mechanism. For example, the descriptors produced for the file in Figure 3 would be

{year=2003, day=184, output_step=1, variable=salinity, nodes=55817, sea_level=4285, implicitness=0.8}

4.1.2 Triple Store

Once the descriptors are harvested, they are loaded into a PostgreSQL table [40] with three string attributes: resource, property, and value. Using SQL, we then identify the signature S(r) of each resource r. The signature of a resource is simply the set of properties used to describe it. Using SQL and external scripts, we then create a materialized view for each distinct signature and populate it with one tuple for each resource. For example, the file in Figure 3 has the signature

{year, day, output_step, variable, nodes, sea_level, implicitness}

so we would create a materialized view with seven attributes, one for each property. This materialized view will have at least one tuple in it, the one corresponding to the file in Figure 3:

(2003, 184, 1, salinity, 55817, 4285, 0.8)

Since we create a separate materialized view for each distinct signature, we want the number of distinct signatures to not be too large. Our hypothesis is that although data owners are unrestricted in their choice of properties and values for each resource, the number of unique signatures occurring for a particular system will be small relative to the number of resources. Our hypothesis is supported by some preliminary applications in medical informatics and earth sciences. We find that although there are millions of resources, there are only tens or hundreds of “types” of resources.

We considered (but rejected) an even less restrictive metadata model: keyword tags. Tagging schemes provide a “bare minimum” approach to metadata: users provide arbitrary terms (or phrases) and associate them with resources [14, 26, 42]. Such schemes were designed to encourage metadata attachment by minimizing overhead: users need not struggle with rigid metadata standards or even consider the semantics of their tags. They simply describe their resources using terms from their own mental

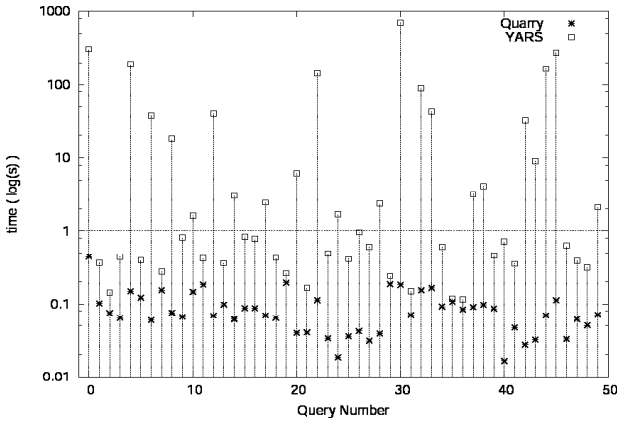

Figure 4. Conjunctive query performance using Quarry and YARS. YARS uses a generic index scheme based on redundant B-Trees, while Quarry clusters resources by their signature. When YARS chooses the wrong join order, the system cannot respond at interactive speeds.

model of their data. This approach works well with very large numbers of users (and hence very large numbers of mental models). In the aggregate, patterns emerge in the tags that enable querying.

However, there is evidence that tags provide too much freedom. Tagging communities are beginning to adopt conventions to emulate a richer data model: the tag "geo:lat=39.234" implicitly represents a namespace ("geo"), a property ("lat"), and a value (" 39.234"). Our source data tends to have more structure than, say, a vacationer's photos, so we adopt a data model of (property, value) pairs rather than keywords alone. We do not use namespaces, however, since shared namespaces are a kind of global schema that we cannot assume exists in the general case.

4.1.3 API

Since the materialized views are generated by the system, users do not know their names and cannot refer to them in the FROM clause of a SQL statement. We provide an API that avoids the need to know.

The query API contains three functions

Properties(conditions)

Values(conditions, property)

Description(resource)

Properties returns unique properties for resources matching conditions. *Values* returns unique values of property for resources matching conditions. *Description* returns the set of (property, value) pairs for a given resource. Property queries are evaluated using global index information. Value queries are evaluated by dispatching a SQL query to each table whose signature subsumes the signature of the query and unioning the results. For example, the query

Values(region=estuary,year=2004, plottype)

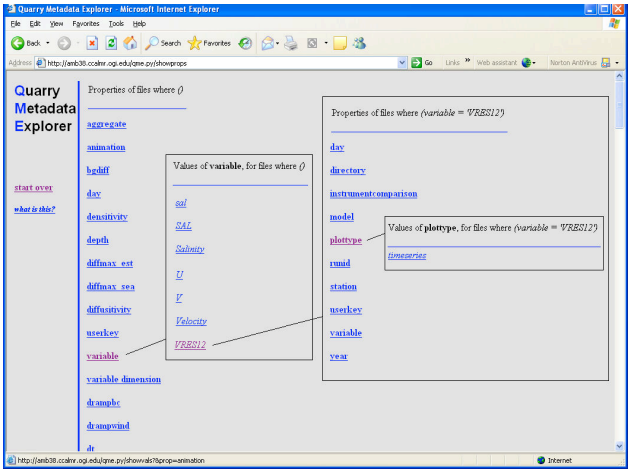

Figure 5. The base interface to Quarry. Several successive steps of a query session are shown superimposed.

returns the values of the property *plottype* associated with resources in any table with attributes "region" and "plottype." The query

Properties(region=estuary,year=2004)

produces a set of properties generated from the columns of any table that (1) has the columns "region" and "year", and (2) has tuples that satisfy the condition. Finally, the query

Description(/forecasts/2003-184/runs/1_salt.d)

returns the complete set of (property, value)-pairs shown in Figure 3.

Harvesting and processing of metadata can be expensive, but is entirely automated. Once the materialized views are created, queries turn out to be quite efficient relative to existing generic triple-store methods. The "Yet Another RDF Store" (YARS) system [21] for storing arbitrary RDF graphs has been shown to outperform other popular triple stores. YARS uses a collection of B-Trees implemented in BerkeleyDB to efficiently answer complex RDF queries with joins. However, as with relational systems, the performance of YARS is sensitive to the choice of join order. In Figure 4, we compare the performance of Quarry and YARS for a set of randomly generated conjunctive queries over a medical informatics dataset. Note that the y-axis is logarithmic. Those queries for which response times are under the dotted line corresponding to one second are not particularly interesting, since the small difference between Quarry and YARS can be attributed to the different technologies used (Java and BerkeleyDB vs. Python and PostgreSQL). However, when YARS selects the wrong join order, the response time is no longer interactive and therefore unsuitable for our purposes. Our API is not as expressive as the RQL query language which YARS supports, however.

4.1.4 QME

Figure 5 shows the base quarry interface, the Quarry Metadata Explorer (QME) for browsing the harvested metadata. QME is a logical hierarchy. The top level is an exhaustive list of all unique properties used to describe any resource from any data source. Selecting a property p returns the list of unique values used for that property for any resource from any source. Subsequently

selecting a value v returns another list of unique properties, this time restricted to those properties asserted for resources satisfying the condition p = v. This alternation between properties and values provides an intuitive way to navigate through the metadata, narrowing the results as conditions are appended.

The resources themselves can be accessed at the leaves of the logical hierarchy. A resource with a known MIME type, such as the gif image data product in Figure 2, can be viewed in the browser. For other files, QME uses the *Description* API call to display all the associated metadata.

To illustrate the use of QME, consider a modeler who wants to find some salinity observations. She can browse the sorted properties for relevant items such as "variable," "quantity" or "units." Selecting variable, she see values such as "SAL," "sal," and "Salinity" in Figure 5, each of which seems to refer to salinity. She also sees a value "VRES12" that she is not familiar with.

Selecting "VRES12" to investigate, she is presented with the set of properties used to describe files for which variable=VRES12. She knows the property "plottype" – it's used to classify data products. Following the plottype link, she finds that every resource with variable=VRES12 also has plottype=timeseries. Going back to select "VRES12" then "userkey" displays the file path of every resource along with an estimate of the number of files matching the conditions. From here, she can view all the timeseries data products for the variable VRES12, and perhaps be able to infer that VRES12 is shorthand for the residual velocity computed over 12 hours.

4.1.5 Quarry Benefits

To keep the ROI curve smooth, we want to (a) let scientists get started quickly, (b) see initial results quickly, and (c) adapt to changing conditions.

Scientists are comfortable writing scripts; the CORIE staff produced a first round of metadata harvesting scripts in one afternoon. After they drop their scripts in the Quarry directory, the harvester picks them up and begins processing files – the scientists' work is done. We are close to the *zero configuration* we sought in Section 2.

Harvesting and processing metadata takes a few hours, mostly due to the need to traverse a large filesystem (around 4 million files). Once the data is processed however, users can access each other's files by metadata rather than by location using QME. The materialized views that power the API are *specialized to the current instance* (Section 2), resulting in interactive performance. Good performance allows the scientists to browse for each other's files; they do not need to fully understand each other's metadata conventions and write queries. Finally, we did not disturb the location or organization of the base data; we are operating on it *in situ* to help keep the ROI smooth.

If and when the scientists want to change the metadata they harvest, they simply modify their scripts. The harvester can be configured to run as a nightly job; the next day, the files are accessible by a new metadata scheme. Scientists are thus empowered to change data organization without, say, a DBA. The more their scripts agree on conventions, the easier it will be to find each other's data through QME; collectively, they can *pay as they go*.

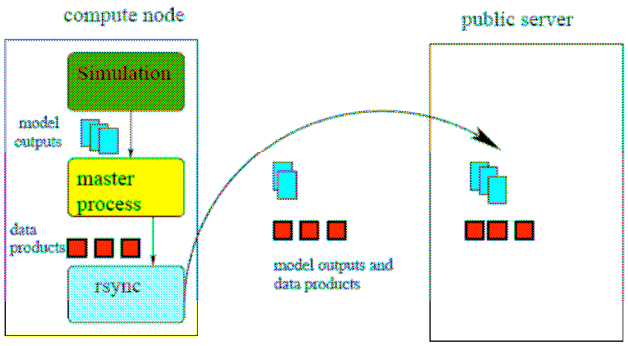

Figure 6. Each forecast is managed by a "master process" script, responsible for kicking off the forecast itself, generating data products, and copying the results back to the public server. Data products are generated incrementally as the forecast results appear then copied (via rsync) back to the data grid for dissemination via the web. Incremental generation of products give researchers the opportunity to detect problems early and adapt quickly.

The Quarry API allows applications (such as QME) to remain robust across these overnight rebuilds.

4.2 Managing the Forecast Factory

Since CORIE's inception, the scope of the operational daily forecast system has grown from three closely related forecasts of just the Columbia River Estuary to 19 forecasts covering regions throughout the Pacific Northwest and around the world. To accommodate the growing scope, the scheduling, execution, and result processing must be carefully optimized.

We have found a factory metaphor useful in modeling the daily forecast workload [6]. Forecasts are allocated to specific compute nodes using scripts that stage in all needed input files, launch the workflow, and stage out data products (Figure 6). When a new forecast is added, existing forecasts must be shifted to different nodes to optimize the completion time of the overall daily workload. Due to the difficulty of estimating forecast running times at different nodes, administrators can use a tool called ForeMan to aid the decision-making process [6].

An excerpt of a screenshot of the ForeMan interface appears in Figure 7. The top half the display is used to monitor forecast execution. Each rectangle represents a forecast run. In this example, all nodes have two CPUs. Narrow solid-colored rectangles denote time periods where only a single forecast is running and a single CPU is in use, and wide multi-colored rectangles denote periods where both CPUs are being used concurrently by two or more forecast runs.

The interface is driven by an open-source relational database. In contrast with a transformative project-wide deployment of a GPDBMS, this database is application-specific, managed by a single developer, and contains only four or five tables (depending on the version). Used in this way, we can take advantage of the query and indexing capabilities of a GPDBMS without bending the ROI curve with attempts to design a comprehensive schema.

Currently there are 12 dedicated forecast nodes, each with two CPUs, and 19 forecast runs, and new nodes will be added as the number of forecasts grows to the expected 50-100 per day. The

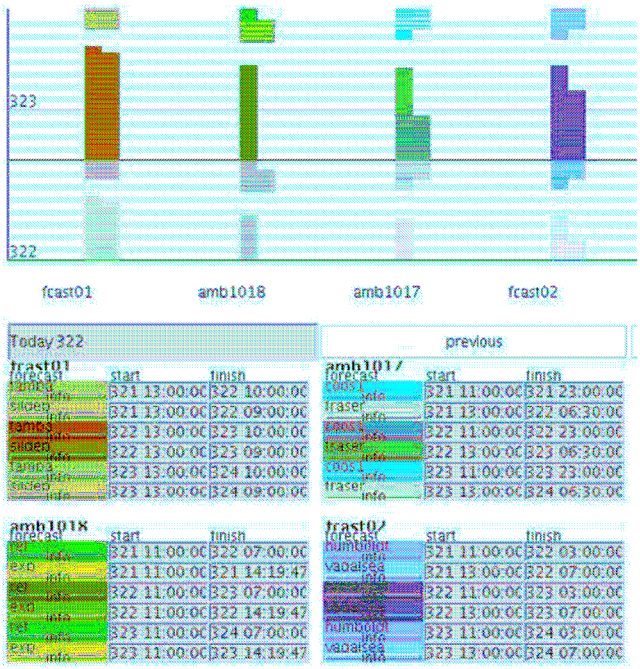

Figure 7. An interface for managing the growing number of forecasts run daily in CORIE. The upper portion displays currently running forecasts (striped bars indicate concurrent execution.) The lower portion displays start and stop times and allows access to additional features such as trend analysis of forecast execution times (cf. Figure 8).

simulation model and data-product generation for a single forecast run concurrently at the same node. Data products are incrementally computed as additional model data is appended to output files, so initial data products are available before an entire run has completed.

Code versions of the simulation models, timestep granularities, and meshes are frequently modified to optimize the forecast accuracy. Data from past forecast executions can aid programmers in estimating the effects of such changes on forecast execution times and improve the ability of programmers to determine a good mapping of forecasts to nodes. This historical data is stored in a relational database and can be retrieved and plotted by the ForeMan interface to reveal trends. In Figure 8, the effect of a particular code change and the effect of a new horizontal unstructured grid can be observed. Around day 150 we observe changes to both mesh size and code version, causing running times to decrease by about 5000 seconds (about 1.5 hours). Around day 160 we observe a significant increase in the running time (about 7 hours) of the forecast due to a major version change in the simulation code. This kind of trend analysis leads to improved resource utilization in the face of evolving forecasts.

4.2.1 Smoothing Strategies for Foreman

There are several possible strategies for populating the ForeMan database. One approach is to ask developers to modify their code to automatically update the database based on runtime events. For example, developers could extend existing code to update the database after each timestep is executed. The advantage of this approach is that the database always contains the most up-to-date

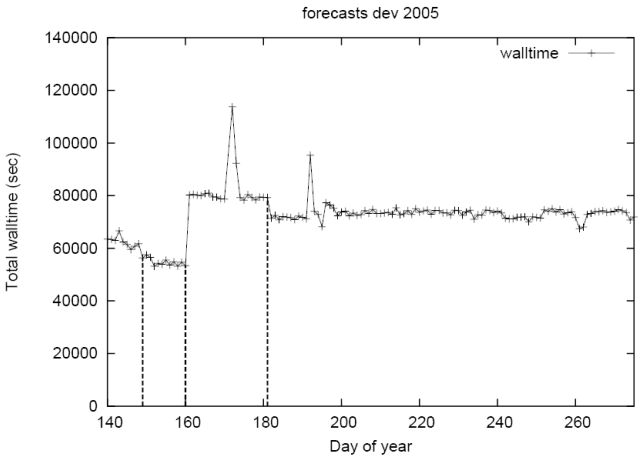

Figure 8. The effect of grid and code changes on the execution time of the CORIE development forecast in Spring 2005. All three dotted lines correspond to a code change; the middle line indicates a grid change as well.

information on the state of the forecast run. However, there are two major limitations to this approach. First, the model must be compiled against database client libraries, reducing portability. Second, the model developers must coordinate with the database developers whenever the log content changes.

A second approach is to have ForeMan periodically parse the existing log files that are generated by each forecast, and populate the database. The advantage of this approach is that it requires minimal work for model developers and does not require changing any existing code (we *specialize to the current instance*). We chose this solution due to the limitations of the first approach (*we operate on in situ data*). While this solution is less robust due to possible log file format changes, we believe the savings in terms of work for the model developers justifies our choice.

The ingest procedures were deployed in the production environment by adding a single line to an existing Perl script. Once deployed, the ForeMan interface detects new Forecasts automatically, without any notification from developers. All forecast data and log files are stored in a filesystem, and data from each forecast resides in its own directory. Whenever a new forecast directory is added, ForeMan's scripts will automatically parse the log files in the new directory and display the forecast data in its graphical interface (we *strive for zero configuration*). In fact, after ForeMan deployment the authors observed several new forecasts using ForeMan without any contact from the model developers.

A final benefit of ForeMan's implementation is that its automatic log file parsing and visual display of forecast runs can alert developers to failures that might otherwise go unnoticed. In one case, the authors observed through the ForeMan interface that several forecasts appeared not to be running. We promptly notified the model developers, who determined that one of the disks had gone into read-only mode. Since no log data was being written for these forecasts, the ForeMan parser did not extract any data for these forecasts and ForeMan did not display them. The problem might have gone undetected for several more days without ForeMan, resulting in even more lost data. Thus, ForeMan has enabled developers to more quickly detect and respond to problems.

4.3 Retrofitting a Data Model

While the daily forecasts are more likely to excite casual visitors (and funding agencies), the Hindcast Archive is a richer source of scientific insight. The goal of the Hindcast Archive is to manage the results of simulation "campaigns" consisting of many runs that share configuration parameters, assumptions, and code versions. These campaigns are used to answer broader scientific questions; e.g., "Is the temperature of the estuary rising over time?"

Although the information to answer these kinds of questions is available, the interface to it is rather impoverished. The model outputs for long sequences of contiguous simulations are stored as packed binary files. Individual runs can be visualized using custom tools, but queries over multiple related runs must be coded by hand.

The results of CORIE simulations can be manipulated and visualized algebraically as *gridfields* [23]. An example of a data product generated using gridfields appears in Figure 2. Both the left and right panes correspond to the region just outside the mouth of the Columbia River Estuary, where the shoreline is just out of view to the east. On the right, the region is shaded by surface salinity. The dark patch near the center is the plume of fresh water jutting out into the ocean. On the left, most of the "grid" on which the solutions are computed has been cut away. The retained portion corresponds to regions where the salinity gradient is high—one definition of the "plume front." The plume front is an important feature for studying ocean dynamics, but also for less academic reasons: salmon like to congregate at the plume front, so fisheries are interested in its location!

To understand aggregate plume behavior, the average gradient over long time periods is a useful quantity to have access to. How can we use gridfields to access the hindcast archive? 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.

Another approach is to convert existing datasets to a special format already equipped with a gridfield interface. Indeed, database vendors frequently assume this approach: Before your data can be manipulated using the relational model, you must surrender control to the 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.

Instead, we derive access methods from lightweight ASCII descriptions of the filesystem [24]. Users write *schema files* that describe the organization of two kinds of data: those embedded in the directory and file names, and those stored within the files themselves. Figure 3 shows an example of data embedded in the file name. This time, we extract the year not as a metadata tag, but as an element of an array. The schema files allow runtime parsing and navigation of these ad hoc file organizations, meanwhile providing three crucial abstractions: First, portions of data files can be accessed by name rather than by address; second, the boundary between file name and file content becomes transparent, and third, the boundary between neighboring files is elided.

The challenge of this approach was to mitigate the cost of abstraction: Can a general access method compete with a hand-coded one?

The answer is "usually." To mitigate the runtime costs of interpreting schema files and data files at runtime, we can pre-analyze schema files and generate compilable access methods. For example, many files in a CORIE campaign tend to use the same header. We can capitalize on this redundancy by reading the header once and generating an efficient access method for all files with identical headers.

We also found it convenient to reason about optimization as an operation on the schema rather than an operation on the query, observing that many schema files can be used to describe the same data files, each favoring particular parts of the file. For example, the header of a file may be unnecessary to answer a particular query, and can therefore be skipped entirely. Rather than derive a query processor that understands how to stride through data, we derive a schema that indicates the header is just "dead space," achieving the same effect.

4.3.1 Smoothing Strategies for Retrofitting

The obvious smoothing strategy used for accessing the hindcast archive was to "work with in situ data." Converting or otherwise reorganizing the entire repository would be expensive, logistically difficult, and probably inconvenient to the scientists (i.e, cause a negative ROI slope).

The gridfield algebra itself is a test of the smooth ROI theme: Can the development of a formal data model ever lead to smooth ROI curve? Database researchers seem to favor a formal, algebraic treatment of unusual data types; witness semi-structured data [8], biological data [43], and free text documents [27].

We consider these efforts to be *data-oriented* examples of Domain Specific Languages (DSL) [31], which we refer to as Domain Specific Data Models (DSDMs). Proponents of DSLs argue that the cost of designing and implementing a new language is justified by the flexibility offered once the language is complete; a similar argument applies to DSDMs Unfortunately, this situation leads precisely to the ROI shape that we have argued is unhealthy in scientific domains.

How can the DSDM approach favored by database researchers be streamlined to exhibit a smooth ROI? Our experience with the gridfield model suggests that there are two relevant techniques.

1. Deploy the model as a reasoning tool prior to implementation.
2. Package and deploy the implementation incrementally as specialized mini-apps.

A DSDM can be used as a tool for reasoning about, explaining, and documenting existing applications even before an implementation is available.

The second recommendation seems contrary to the spirit of DSLs and DSDMs: if the language is hidden inside mini-applications, what is the advantage of using a language-based approach? Since the language designers themselves are exercising the language

rather than the scientists, it is difficult to argue that this kind of approach improves productivity for the scientists themselves. However, it is important to observe that language-powered mini-apps need not be the final deliverable; rather they act as a smoothing agent (an iron?) on the path toward a full data manipulation system.

5. CONCLUSIONS

We have argued that the success of scientific data management projects is reflected in the shape of the ROI curve: a smooth ROI is a correlated with successful projects. We argued that this observation may be a feature peculiar to the scientific domain; commercial projects tend to require the transformative change associated with a crooked ROI. We then suggested several strategies for smoothing the ROI in the scientific domain: *pay-as-you-go*, *let a hundred flowers blossom*, *specialize to the current instance*, *strive for zero configuration*, and *operate on in situ data*. We exercised these strategies in the context of the CORIE Environmental Observation and Forecasting system by developing and deploying three lightweight tools: the Quarry metadata system for browsing and searching unstructured metadata, the ForeMan interface for managing forecast simulations using a factory metaphor, and a toolkit for retrofitting a logical model (*gridfields*) to ad hoc file formats. We conclude that striving for a smooth ROI is crucial to the success of these and related projects.

6. ACKNOWLEDGMENTS

We would like to thank Antonio Baptista and the entire CORIE science team. We would also like to thank Kristin Tufte for her helpful comments and discussion. This work was supported by NSF ITR Award No. ACI-0121475.

7. REFERENCES

[1] Bairoch, A. and Apweiler, R. The Swiss-Prot Protein Sequence Database and Its Supplement TrEMBL In 2000. Nucleic Acids Research, 28:45–48, 2000.

[2] Baptista, A., Wilkin M., Pearson, P., Turner, P., McCandlish, C., and Barrett, P. Coastal and Estuarine Forecast Systems: A Multipurpose Infrastructure For the Columbia River. *Earth System Monitor*, NOAA, 9(3), 1999.

[3] Baumann, P., Dehmel, A., Furtado, P., Ritsch, R., and Widmann, N. Spatio-temporal retrieval with RasDaMan. In VLDB'99: Proceedings of the 25th International Conference onVery Large Databases.

[4] Beck, K. Embracing Change With Extreme Programming. *Computer*, 32(10): 70–77, 1999.

[5] Becla, J. and Wang, D. L. Lessons Learned From Managing a Petabyte. In CIDR '05: 2nd Biennial Conference on Innovative Data Systems Research, 2005.

[6] Bright, L., Maier, D., Howe, B. Managing the Forecast Factory. In Proceedings of the ICDE Workshop on Workflow and Data Flow for Scientific Applications (SciFlow), 2006.

[7] Bruno, N. and Chaudhuri, S.. Automatic Physical Database Tuning: A Relaxation-Based Approach. In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, 2005.

[8] Buneman, P., Fernandez, M., and Suciu, D. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. VLDB Journal 9(1), 2000.

[9] Carey, M. J., Dewitt, D. J., Franklin, M. J., Hall, N. E., Mcauliffe, M. L., Naughton, J. F., Schuh, D. T, Solomon, M. H., Tan, C. K., Tsatalos, O. G., White, S. J., and Zwilling, M. J. Shoring Up Persistent Applications. In Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, 1994.

[10] Carey, M. J. Data Delivery in a Service-Oriented World: The BEA Aqualogic Data Services Platform. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, 2006.

[11] Carlson, S. Lost in a Sea of Science Data. *The Chronicle of Higher Education* 52(42):A35, June 2006.

[12] Center for Coastal Margin Observation and Prediction (CMOP), http://www.stccmop.org/, viewed December 2006.

[13] Fielding, R. Architectural Styles and the Design of Network-Based Software Architectures. Phd Dissertation, University of California, Irvine, US, 2000. Chapter 5: REpresentational State Transfer (Rest).

[14] Flickr, http://www.flickr.com/, viewed July 2006.

[15] Franklin, M., Halevy, A. and Maier, D. From Databases to Dataspaces: A New Abstraction for Information Management. SIGMOD Record 34(4), December 2005.

[16] Google Maps API. http://www.google.com/apis/maps/, viewed December 2006.

[17] Google Search Appliance. http://www.google.com/enterprise/, viewed July 2006.

[18] Gray, J., Liu, D. T., Nieto-Santisteban, M. A., Szalay, A. S., Heber, G., and Dewitt, D. Scientific Data Management In the Coming Decade. Technical Report, Microsoft MSR-TR-2005-10, 2005.

[19] Gray, J. and Szalay, A. S. Where the Rubber Meets the Sky: Bridging the Gap between Databases and Science. IEEE Data Engineering Bulletin 27(4), December 2004

[20] Halevy, A. Y., Franklin, M. J., Maier, D. Principles of dataspace systems. In Proceedings. of the Twenty-Fifth ACM SIGACT/SIGMOD-SIGART Symposium on Principles of Database Systems, Chicago, June 2006.

[21] Harth, A., Decker, S. Optimized Index Structures for Querying RDF from the Web. 3rd Latin American Web Congress, Buenos Aires, 2005.

[22] HDF5: API specification reference manual. National Center for Supercomputing Applications (NCSA), http://hdf.ncsa.uiuc.edu/, 2004.

[23] Howe, B. and Maier, D. Algebraic Manipulation of Scientific Datasets. In Proceedings of the 30th International Conference on Very Large Databases, 2004.

[24] Howe, B. and Maier, D. Retrofitting a Data Model To Existing Environmental Data. In Proceedings of the 17th Interna-

tional Scientific and Statistical Database Management Conference, 2005.

[25] Howe, B., Tanna, K., Turner, P., and Maier, D. Emergent Semantics: Towards Self-Organizing Scientific Metadata. In Proceedings of Semantics for a Networked World: Semantics For Grid Databases, Volume 3226 of *Lecture Notes In Computer Science*. Springer, 2004.

[26] Hsieh, W., Madhavan, J., Pike, R. Data Management Projects at Google. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, 2006.

[27] Ipeirotis, P., Agichtein, E., Jain, P., and Gravano, L. To Search or to Crawl? Towards a Query Optimizer for Text-Centric Tasks, In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, 2006.

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

[29] Ka-Map. http://ka-map.maptools.org/, viewed December 2006.

[30] Lee, B. S., Snapp, R. R., Chen, L., and Song, I.-Y. Modeling and Querying Scientific Simulation Mesh Data. Technical Report, University of Vermont, CS-02-7, 2002.

[31] Leijen, D. and Meijer, E. Domain Specific Embedded Compilers. In Proceedings of the 2nd Conference on Domain-specific Languages, pages 109–122. ACM Press, 1999.

[32] Ma, X., Winslett, M., Norris, J., Jiao, X., and Fiedler, R. Godiva: Lightweight Data Management For Scientific Visualization Applications. In ICDE '04: Proceedings of the 20th International Conference on Data Engineering, pp 732, 2004.

[33] Marathe, A. P., and Salem, K. A Language for Manipulating Arrays. In Proceedings of 23rd International Conference on Very Large Data Bases, pages 46–55, 1997.

[34] O'Mullane, W., Li, N., Nieto-Santisteban, M. A., Szalay, A. S., Thakar, A. R., and Gray, J. Batch is Back: CASJobs, Serving Multi-Terabyte Data on the Web. Technical Report, Microsoft MSR-TR-2005-19, February 2005.

[35] OpenLayers. http://www.openlayers.org/, viewed December 2006.

[36] Roth, M. T., Arya, M., Haas, L., Carey, M., Cody, W., Fagin, R., Schwarz, P., Thomas, J., and Wimmers, E.. The Garlic Project. In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, 1996.

[37] Simple Object Access Protocol (SOAP) 1.1, W3C, www.w3.org/TR/soap/, May 2000.

[38] Seshadri, P. and Paskin, M. Predator: an ORDBMS with enhanced data types. SIGMOD Record, 26(2):568–571, 1997.

[39] Stolte, E., Von Praun, C., Alonso, G., and Gross, T. Scientific Data Repositories: Designing for a Moving Target. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, 2003.

[40] Stonebraker, M., Rowe, L. A, and Hirohama, M. The Implementation of Postgres. IEEE Transactions on Knowledge and Data Engineering, 2(1):125–142, 1990.

[41] Szalay, A. S., Kunszt, P. Z., Thakar, A., Gray, J., Slutz, D., and. Brunner, J. Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, 2000.

[42] TagCloud, http://www.tagcloud.com/, viewed July 2006.

[43] Tata, S., Patel, J. M., Friedman, J. S. and Swaroop, A. Declarative Querying for Biological Sequence Databases, In Proceedings of the 22nd International Conference on Data Engineering, 2006.

[44] Vatsavai1, R. R., Shekhar, S., Burk, T. E., and Lime, S., UMN-MapServer: A High-Performance, Interoperable, and Open Source Web Mapping and Geo-spatial Analysis System. Volume 4197 of *Lecture Notes In Computer Science*. Springer, 2006.

[45] What is Service-Oriented Architecture? http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html.