Data-Centric Scientific Workflow Management Systems

David T Liu

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-83
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-83.html

June 15, 2007

Data-Centric Scientific Workflow Management Systems

by

David T. Liu

B.S. (University of California, Los Angeles) 2000
M.S. (University of California, Berkeley) 2004

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

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Dr. Michael J. Franklin, Chair
Dr. Katherine Yelick
Dr. Geoffrey Marcy

Spring 2007

The dissertation of David T. Liu is approved.

Chair Date

Date

Date

University of California, Berkeley

Spring 2007

Data-Centric Scientific Workflow Management Systems

by

David T. Liu

Abstract

Data-Centric Scientific Workflow Management Systems

by

David T. Liu

Doctor of Philosophy in Computer Science

University of California, Berkeley

Dr. Michael J. Franklin, Chair

Recent trends in science and technology augur a rapid increase in the number of computations being employed by scientists. Accompanying increased volumes are growing expectations for the tools that scientists use to handle their computations. These increased volumes and expectations present a new set of problems and opportunities in *computation management*. In this thesis, I propose *Data Centric Scientific Workflow Management Systems* (DSWMSs) to address these issues. DSWMSs supersede current approaches by leveraging a deeper understanding of the data manipulated by computations to provide new features and improve usability and performance. Examples of such features include data provenance, work sharing, and interactive computational steering.

In this thesis, I make several contributions towards realizing the concept of a DSWMS. First, in conjunction with scientists from several scientific domains,

I propose a set of services that are not provided by current paradigms, but are made possible in DSWMSs. Second, I define an abstract model, the Functional Data Model with Relational Covers (FDM/RC), for representing scientific workloads and a language for defining and manipulating instances (schemas) of the model. Third, I design and implement GridDB, a prototype DSWMS. GridDB is deployed on a large cluster at Lawrence Livermore National Laboratories where it runs science applications at real-world scales. The deployment uncovers a pair of technical problems involving the provisioning of *data provenance* and *memoization* (computational caching) so I also contribute solutions to these problems.

Dr. Michael J. Franklin
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my research advisor Mike Franklin for helping me finish my thesis. Mike is an unusually charming blend of manager, sage, and buddy. In my time under his advisorship, I have learned skills that I had struggled with all my life: how to write clearly; how to make progress on large, amorphous problems; and how to overcome the fear of failure. Mike often persisted that I could meet his high standards even as I insisted that I could not. More often than not, he was right and those episodes have boosted my confidence.

In the course of my thesis, I have had many collaborators and mentors who have all contributed to my success. Ghaleb Abdulla, my LLNL advisor, offered sound guidance and support during my thesis. Kathy Yelick and Geoff Marcy, members of my thesis committee, have provided profuse encouragement and productive advice on my dissertation. Jim Gray has been a key role model. His work is brilliant, but his most inspirational trait is his commitment to cultivating the next-generation of scientists.

At Berkeley, I met an incredibly talented pool of individuals who were always eager to discuss ideas, and provide feedback and support. I would like to thank Sirish Chandrasekaran, Mark Whitney, Matt Denny, and Shawn Jeffery who helped me regroup during various times of "thesis crisis." I am also grateful towards UC Berkeley Database Group members, including Amol Deshpande, Boon Thau Loo, Daisy Wang, David Chu, Megan Thomas, Mehul Shah, Ryan Huebsch, Sailesh Krishnamurthy, Sam Madden, Shariq Rizvi, Tyson Condie, and Yanlei Diao.

My family showed love, patience, and understanding during my time in graduate school. These include my siblings and their spouses, Connie, Don, Ted and Helen, and my mother, Anna. Their holiday schedules were always arranged to accommodate my deadlines and their love and support always energized me. I would also like to thank my late father, Allen, for inspiring me to pursue a Ph.D. To me, a doctoral thesis is a long journey of independent thinking. Though I did not realize it at the time, my decision to pursue a doctorate was a reflection of my father, who spent his life as an independent thinker and philosopher.

Finally, I would like to thank my life-partner, Tina, who has shared my fortunes in graduate school. I often joke that most of my precious twenties were donated towards the pursuit of science. Without much consultation, I committed her to the same cause. Her encouragement, forbearance, and unique ability to make me laugh has sustained me through my most challenging times. Without Tina, I could not have completed my work, and I dedicate this thesis to her.

Chapter 1

Introduction

In recent years, we have witnessed a trio of trends that have tightened the relationship between computation and science. The first trend is an embrace of computation by the scientific community. Scientists from many disciplines — including astrophysics, high energy physics, chemistry, proteomics, genomics and climatology — have proclaimed computation as an indispensable tool for scientific discovery. The suitability of computation for science has been buttressed by several high profile successes; for example, the detection of hundreds of extra-solar planets (2; 3), simulation-based predictions of global warming, (4; 5), and grassroots efforts to find extra-terrestrial life (6) and cures for elusive diseases (7; 8; 9).

The second trend is the proliferation of clustered computer architectures. A *cluster computer* is a large computer constructed from many small processing components connected by high-speed interconnects. Especially successful in recent years is the "beowulf" or "commodity cluster," which is constructed from

inexpensive, mass-market components (10). While the clustered approach has been used in other domains (**?**), it has dominated the domain of scientific computation. Cluster computing has many advantags. At a basic level, it drives continued improvement in the latency and throughput of scientific supercomputers. More importantly, however, cluster computers have *democratized* the realm of High Performance Computing (HPC). Today, even a modest band of scientists with a small budget has access to HPC resources.

The third trend drawing scientists closer to computation is the increased production of large data sets in digital format. Modern data collection apparatuses, in the form of telescopes, particle colliders, gene arrays and sensor networks, log large volumes of data directly onto magnetic disks (11). At the same time, scientists augment their already-massive data archives with considerable amounts of simulated data. Large nquantities of computational resources are required this scientific data.

The end-result of the three trends described — the legitimization of computation in science, the democratization of HPC and the production of large scientific datasets — is a broad and firm symbiosis between the scientist and her computing resources. Today's scientist relies on computing resources to sense, generate, transform, analyze and visualize scientific data. As the price of constructing and maintaining clusters continues to plummet, the computing capacity available to each scientist skyrockets.

1.1 The Need for Computation Management

While the affinity between scientist and computer marks progress, it also creates problems. One key ramification of this relationship is a dramatic increase in the number of computations being executed by scientists. This increase is the result of a multidimensional expansion: not only are more scientists employing computation, but each scientist writes more programs, composes them in more combinations, shares them more easily, and runs them more often.

A second ramification is increased cooperation. Digitization creates many possibilities for people to share effort, resources and results. In many domains, scientists have formed large-scale, multi-disciplinary collaborations with unprecedented complexity, scope, and longevity. Examples of current-day collaborations include the Grid Physics Network (GriPhyN) (12), the International Virtual Data Grid Observatory (13), the Particle Physics Data Grid (14), and the European Union DataGrid (15). These collaborations, with their improved economies-of-scale, are actively pursuing a new generation of capabilities to make their scientists more productive, their resources more efficient, and their discoveries more reliable.

To address an increase in computation volume and the demand for new capabilities, a new kind of infrastructure focused on *computation management* is required. As I argue in the next section, the current *process-centric* model that most scientists use to execute their computational jobs will be insufficient. Rather, a new *data-centric* infrastructure that assigns first-class importance to *data-awareness* is required.

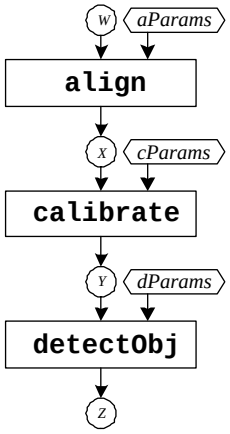


Figure 1.1. Example workflow.

1.2 Example: Astrophysics Image Processing

To illustrate the problems of the status quo, this section introduces an example. The example is a set of dependent programs, or a *workflow*, that is used to perform image processing in astrophysics. A workflow such as this is required, for example, to convert telescopic images into catalogs of objects amenable to quantitative analysis. Similar workflows are found in other domains that also employ image processing; for example, military agencies and meteorologists may use a similar workflow to process satellite images to track military vehicles and weather patterns.

A graphical representation of the workflow is shown in Figure 1.1. The workflow consists of three programs, `align`, `calibrate` and `detectObj`, which are constrained by data dependencies. `calibrate` requires data created by `align` and `detectObj` requires data created by `calibrate`. Collectively, the three programs convert images to structured data (performed by `detectObj`) after making

various adjustments to a picture's coordinates (performed by `align`) and color (performed by `calibrate`). Each of the programs takes a set of files (denoted by circles) and command-line parameters (denoted by hexagons) as input and creates a set of files as output. The input and output files are located in a scratch directory that is passed to the program. An astronomer's typical job request will execute the workflow on thousands of telescopic images. Each image may also be processed a few times using different parameters for each workflow program. Since there can be many images, with many different processing parameters and the programs can be long-running, it is not uncommon for jobs to require thousands of CPU hours. As such, jobs are typically executed on a cluster to exploit data parallelism. Individual processes are often also parallelized across multiple nodes using parallel programming (e.g., MPI or UPC) if they are too long-running.

In the current state-of-the-art, scientists express their computation jobs using a *process-centric* approach, where a script (e.g. in perl) constructs command-line strings that invoke the processes that make up the job. These command-lines are submitted to a *process-centric* middleware system that then dispatches the process onto a cluster node (or set of nodes in parallel computations) for execution. Archetypal examples of such middleware include Condor (16) and Globus(17). The process-submission interface in these systems is identical to that provided by the unix operating system and therefore has a low learning barrier.

An example script for the workflow of Figure 1.1 is provided in Listing 1.1. Execution of the program `align` is carried out in lines 10-17 while the execution of the programs `calibrate` and `detectObj` are carried out in lines 23-28 and 34-38,

respectively. Barriers at lines 20 and 31 ensure that instances of a program do not execute prior to the completion of all executions of a program that provides input data. For pedagogical purposes, the script shown only executes the workflow on two images and the parameters to each program are not varied. In reality, these scripts are usually more complicated.

1.3 From Process-Centric to Data-Centric

The fundamental shortcoming of the process-centric model is that the middleware used by scientists is not cognizant of the data read or written by the processes that it executes. In this model, processes are specified by encoding a program and its arguments in a string. The middleware extracts a program name from the string (by convention, the first token in the string), instantiates a process based on the program, and passes on the rest of the string to the process. Since the string is not accompanied by metadata, the middleware does not have the means to parse or understand it. Additionally, programs read and write files to the file system. The locations of these files are unknown to and therefore not recognized by the middleware.

Line 14 of Listing 1.1 contains an example of the process centric interface. In it, the script calls the `dispatch` procedure, and passes it a string with the program name and arguments. After receiving the string, the middleware system will extract the program name `align` from it, but will not attempt to parse the rest of the string, which includes runtime parameters (`${aParm}` and `${i}`) and a path to a working directory (`${IMGS_DIR}`). By ignoring the data used and created by

```
#!/usr/bin/perl

my $IMGS_DIR = "/imgs";
my $NUM_IMGS = 2;

# integrate modifications from [24]
my(@alignDoneFlags, @calibrateDoneFlags);

# execute the align program over all images
for(my$i=1; $i<$NUM_IMGS; $i++){
 foreach my $aParm("loose"){
  # executes an align process on a cluster node and returns a flag variable
  # that is set upon completion
  my $doneFlag = dispatch("align -param ${aParm} -workDir ${IMGS_DIR} -imgNum ${
      i}");
  push($doneFlag,@alignDoneFlags);
 }
}

# wait for all executions of align to complete
barrier(@alignDoneFlags);

# execute the calibrate  program over all images
for(my$i=1; $i<$NUM_IMGS; $i++){
 foreach my $cParm("violet"){
  my $doneFlag = dispatch("calibrate -param ${cParm} -workDir ${IMGS_DIR} -
      imgNum ${i}");
  push($doneFlag,@calibrateDoneFlags);
 }
}

# wait for all executions of calibrate to complete
barrier(@calibrateDoneFlags);

# execute the detectObj program over all images
for(my$i=1; $i<$NUM_IMGS; $i++){
 foreach my $dParm("flat"){
  dispatch("detectObj -param ${dParm} -workDir ${IMGS_DIR} -imgNum ${i}");
 }
}
```


Listing 1.1. Perl script for the image processing workflow of Figure 1.1 over 2 images.

processes, process-centric middleware is severely limited in its potential. Before enumerating these limitations, I introduce the alternative proposed in this thesis.

A *Data-Centric Scientific Workflow Management System* (DSWMS), like a process-centric middleware system, mediates access to cluster resources. Unlike process-centric middleware, however, the DSWMS *possesses an awareness of the data manipulated by processes*. This awareness applies to data that is imported into the system, as well as data created by programs that are executed by the DSWMS. The awareness provides extra information on the location and structure of data, which enables the DSWMS to provide features that are otherwise difficult or impossible to provide in process-centric middleware. Examples of such features are data provenance, work-sharing, and type-checking. Later in the section, I will describe these features in detail. First, however, I will describe the environmental context surrounding a DSWMS.

Figure 1.2 illustrates a DSWMS within its operating environment. A DSWMS provides a veneer, or *overlay*, on top of existing process-centric middleware. The programming model of a DSWMS is partitioned into two phases (which I term a *2-phase* programming model). The first phase is *workflow definition*, where a user provides information that empowers a DSWMS with data-centricity. This information includes descriptions of program command-line formats and knowledge that helps the DSWMS identify file system data created and used by programs. Workflow definition produces *functions*, which are similar to programs in that they transform data, but different in that their interfaces are explicitly described. Functions can also be composed together to create composite functions, which correspond to workflows.

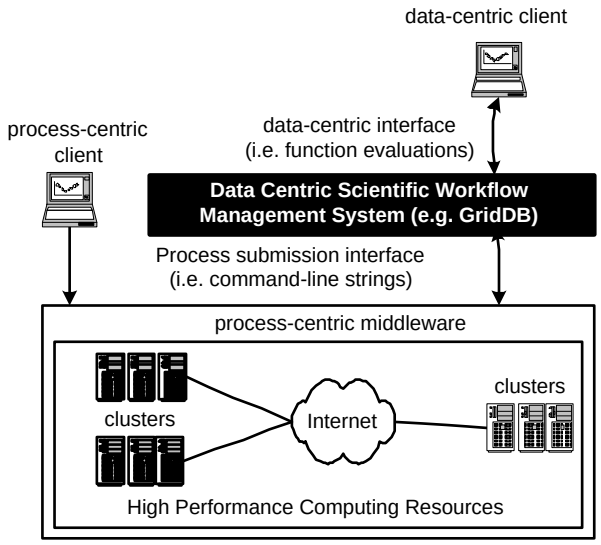


Figure 1.2. The Role of a Data-Centric Scientific Workflow Management System (DSWMS) in High Performance Computing. GridDB is a prototype DSWMS.

After the workflow definition phase, a user engages in the *workflow execution* phase, where previously defined functions are *evaluated*. Workflow execution can be carried out by a different user, resulting in an efficient division-of-labor. Workflow execution is expressed in a notation that explicitly distinguishes data from program, allowing the DSWMS to catalog the evaluation's input data. Evaluations are translated into processes, which are executed on clusters managed by process-centric middleware. Each process is executed in a sandboxed space in the file system, allowing the DSWMS to distill and identify output files created by the process. As such, the DSWMS is able to associate each function evaluation with its input and output data.

To illustrate the data-centric interface and explain its advantages, we compare the data-centric programming model with the process-centric programming model

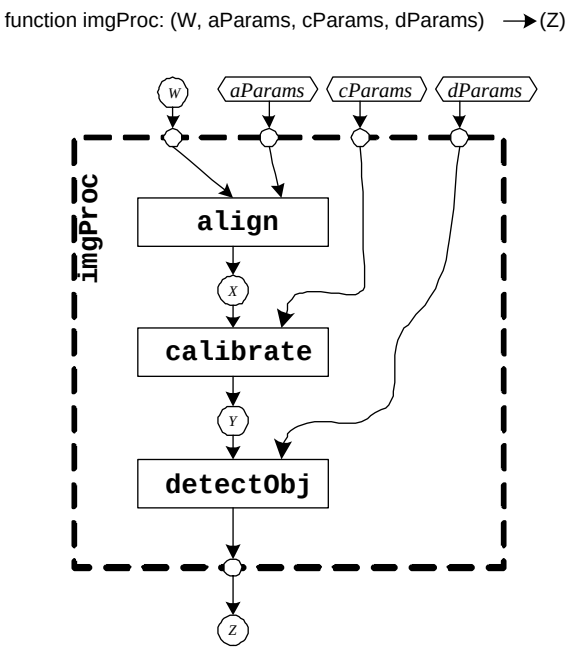


Figure 1.3. A function for the workflow of Figure 1.1. Specified during the workflow definition phase.

using the example of Listing 1.1. Figure 1.3 illustrates a composite function defined during workflow definition. The function `imgProc` transforms a 4-tuple of data $(W, aParams, cParams, dParams)$ into a singleton (Z) data item. Users of the function are abstracted away from the internal functions (which correspond to the image processing programs) and dataflow connections that implement the function.

To execute workflows, a scientist simply submits the function evaluations of Listing 1.2. With this notation, the DSWMS can easily parse out the function (i.e., `imgProc`) and data (i.e., (`'img1'`, `'loose'`, `'blue'`, `'flat'`) and (`'img2'`, `'loose'`, `'blue'`, `'flat'`)). Each of the two function evaluations are translated into three processes, resulting in a total of six processes identical to those created

```
1 compute imgProc('img1', 'loose', 'blue', 'flat');
2 compute imgProc('img2', 'loose', 'blue', 'flat');
```

Listing 1.2. GridDB workflow execution statements for executing the job represented in Listing 1.1

by the script of Listing 1.1. The DSWMS provides users with an interface that is much more compact (compare Listing 1.2 with Listing 1.1). More importantly, however, the DSWMS is also able to provide a number of features that are difficult or impossible when using the process-centric approach. These include:

- **Data Provenance**: The *provenance*, or *lineage*, of a data item is the history of its existence and evolution. Though other definitions are reasonable, we define a datum's provenance as the program that created the data, the input data used by that program, and recursively, the provenance of the input data. As a base-case, some data is imported into the system and not created by a program. Data provenance is becoming increasingly important as more scientists collaborate by using data that they themselves did not generate. In such cases, a scientist must be able to peruse a datum's provenance in order to understand its meaning or evaluate its reliability. Lacking data-awareness, process-centric middleware cannot properly catalog provenance or make it available for perusal.

- **Interactive Monitoring and Steering**: Scientific computation jobs are often long-running, requiring days or weeks. The ability to interactively monitor and steer these jobs can unveil important data faster (18), reducing the time to create meaningful results. A DSWMS provides users with

facilities to monitor partial results as they complete and prioritize other partial results in the job.

- **Modularity**: The ability to create and reuse modular components will be key to building and managing the complex software of future science. By clearly separating workflow definition from evaluation, the 2-phase approach stipulates the creation of modules that can be reused in future workflow compositions. Because the modules' interfaces are clearly described, they can also be cataloged and queried. In contrast, the process-centric approach intermingles composition with execution, complicating efforts to reuse job specification scripts.

- **Type Checking**: Type-checking is an established technique for detecting errors quickly. Equipped with an awareness of data, the DSWMS can identify bugs both during workflow definition (i.e. if two incompatible programs are composed together) and execution (i.e. if a workflow is executed with improper arguments). This is especially important because jobs are long-running and can fail long after they have started but well before they have completed, leading to frustration and a loss of productivity.

- **Work Sharing**: As computation volumes and user populations increase, the ability to share redundant work amongst users becomes crucial. Luckily, with increased standardization of scientific programs in collaborations, such sharing opportunities will become increasingly common. By exploiting data-awareness, the DSWMS is able to encode computations in a representation that allows equivalence testing. When two computations are

equivalent, the work performed to satisfy one may also be used to satisfy the other. Such overlapping computations may occur both in concurrent and non-concurrent jobs (where result caching is used). This thesis gives careful consideration to the latter case, also known as *memoization*, where function evaluation results are cached so that future evaluation requests may reuse them.

As stated earlier, a DSWMS derives its power from data-awareness, which is provided by the 2-phase programming model. The 2-phase programming model emulates traditional Database Management Systems (DBMSs), where users first specify *data schemas* in a *Data Definition Language* (DDL) before issuing queries in a *Data Manipulation Language* (DML). The schemas are put to good use, as DBMSs can then provide a declarative interface, offer transactional semantics, handle concurrency, perform error checking, and automatically optimize for performance. Commercial DBMSs based on the 2-phase programming model have been wildly successful, driving a multi-billion dollar industry. This thesis seeks to apply the same model to the domain of computation management.

The 2-phase programming model is somewhat controversial in the scientific computing domain because it has been rejected by scientists in the past (19). One may anticipate that scientists would be reluctant to relinquish the lighter-weight 1-phase programming model represented by the process-centric approach. To encourage adoption then, we incorporate three important evolutionary features into our approach. First, the proposal integrates a "pay-as-you-go" approach to the definition phase. Rather then forcing modeling of all workflow and data to be done at the finest granularity, the system supports coarse-grained schemas that

incur less human (and system) overhead at the cost of providing fewer features. The features made available by coarse-grained models are made clear to users, who can subsequently refine schemas to incrementally procure more benefits as the need becomes apparent.

Second, our approach accommodates the use of legacy science codes. Scientists have made considerable investment in legacy codes and our evolutionary approach encourages these assets to be leveraged within the new paradigm. Perhaps more importantly, users can write new science codes in the same programming models that they have written their legacy codes in. By co-opting the programming models of the process-centric interface, we also inherit their benefits.

Finally, we have gone to great lengths to ensure that our DSWMS system can work on existing platforms. Infrastructural investments in operating systems, file systems and batch schedulers are massive and platform standards are slow to evolve. As a result, it is essential that our new paradigm coexists with existing infrastructure. As an example, Chapter 5 shows an instance where we encounter a deficiency in current file systems, but nevertheless propose an evolutionary solution that works well with current file systems rather than relying on the adoption of new ones.

In summary, while a 2-phase programming model is likely to encounter resistance on the part of some scientists, our approach minimizes adoption costs by supporting a pay-as-you-go migration path, accommodating legacy science codes and programming models and ensuring efficient operation on currently deployed software infrastructures.

1.4 Contributions

In this thesis, I make the following contributions:

- I propose the concept of a Data-Centric Scientific Workflow Management System (DSWMS). My proposal includes a Data Definition Language (DDL) and Data Manipulation Language(DML) for defining schemas and interacting with the DWSMS in a declarative manner. I also propose a host of DSWMS services that automate scientists' tasks, provide new features, and increase the efficiency of resource usage. My proposal has been developed jointly with partners in the scientific community and is the result of iterative design.

- I make a pair of architectural contributions. First, I design a modular software architecture for a DSWMS. The architecture integrates seamlessly with incumbent software platforms (operating systems, file systems, batch schedulers and databases) and scientific software packages. Second, I integrate the DSWMS within various computing environments.

- To test my ideas, I implement a java-based prototype called GridDB. Furthermore, I evaluate GridDB in a production environment at a world-class supercomputing facility (Lawrence Livermore National Labs) featuring a cluster with more than 1000 nodes. During this evaluation, I execute real workloads drawn from the fields of high-energy physics, astrophysics and biochemistry. The exercise reveals hidden challenges in efficiently implementing two important services, data provenance and memoization.

- One problem that I encounter involves *preserving* data during workflow execution. Preservation of file system data is a prerequisite for data provenance and memoization. Unfortunately, there is an intricate trade-off between correctness and scalability with existing HPC file systems. Here, I shape a solution that achieves scalability and correctness on current platforms by employing a combination of user-hints and parallelization.

- A further problem that I encounter involves the provisioning of memoization. I show that when one takes into account the overhead of indexing and retrieval, memoization can result in a net performance loss in middleware-scarce (the DSWMS is the middleware in this case) environments. In other environments (i.e., middleware-rich), memoization may achieve a significant performance improvement. I provide a solution that trades off the benefits of memoization against its overheads in an incremental manner, achieving superior performance across a wide range of environmental settings.

- Finally, I identify issues that have not been addressed by this thesis or other research and suggest future directions to improve our understanding of computation management.

1.5 Roadmap

The remainder of this thesis is organized as follows. Chapter 2 provides background information on how scientists currently carry out scientific processing as well as how related systems and frameworks support scientific processing. The

main contributions described above are contained in Chapters 3 through 6. Chapter 3 proposes the concept of a DSWMS, and describes the design and implementation of GridDB, our DSWMS prototype. Chapter 4 describes the deployment of GridDB at Lawrence Livermore National Laboratories. Chapter 5 presents a comprehensive solution for *Data Preservation*, a building block required for data provenance and memoization. Chapter 6 addresses the issue of middleware overhead when providing memoization. Finally, Chapter 7 offers closing remarks and directions for future research.

Chapter 2

Background

This chapter presents background information that is pertinent to this thesis. The first section introduces a high-level model of the scientific knowledge discovery process and argues that it is being transformed by computation. The next section surveys recent trends in technology that are driving adoption by scientists. The third section provides an overview of the *workflow*, which is an important concept in computation management. I defer discussions of work related to specific techniques proposed in this thesis to subsequent chapters, where they are juxtaposed against individual contributions.

2.1 The Scientific "Knowledge Supply Chain"

We start by describing the *Knowledge Supply Chain* (KSC), a high-level process model for the creation and dissemination of scientific knowledge. The KSC is illustrated in Figure 2.1 and consists of four stages:

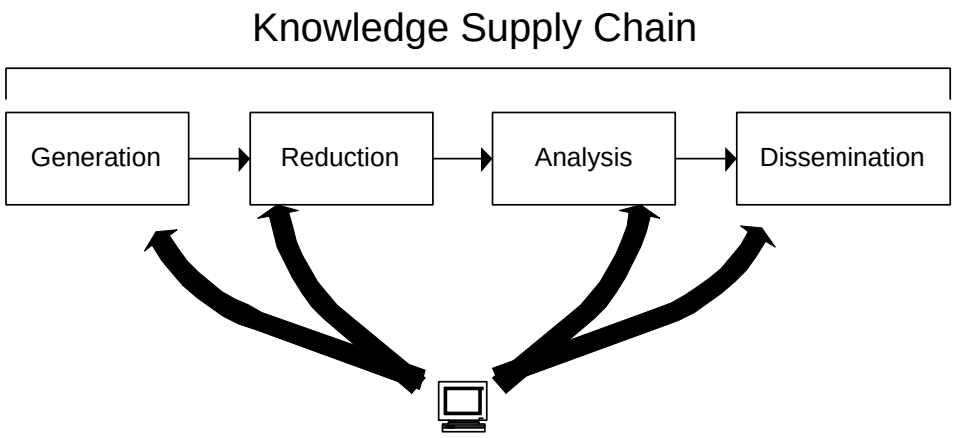


Figure 2.1. Computation is infiltrating every stage of the Scientific "Knowledge Supply Chain."

1. **Generation**: where data is collected or created.

2. **Reduction**: where data is transformed from low-level measurements into higher-level, real-world object representations.

3. **Analysis**: where data is perused, mined, and visualized to extract meaningful facts and validate hypotheses.

4. **Dissemination**: where conclusions and data are communicated across the scientific community.

The linear representation shown in Figure 2.1 is an approximation. In actuality, there is feedback amongst the stages. Separating the activities into these four stages, however, is useful for analyzing the involved in knowledge-discovery. Along these lines, we will now use the KSC to support the claim that scientists have incorporated computer technology into every stage of knowledge discovery.

2.1.1 Generation

In the *generation* stage, scientists have adopted computation for two purposes. First, there is widespread use of computer simulation as a means for collecting experimental data. Simulations allow scientists to "act out" a phenomenon virtually rather than recreate the physical conditions of its occurrence. Simulations offer many advantages over traditional experiments. They enable a faster, cheaper, safer and/or a safer vehicle for data collection. In some cases, simulation has expanded the scope of collectable data. For example, simulations of the global climate or galaxy or in the design of drugs, proteins, semiconductors, airplanes, and nuclear weapons all exhibit more than one of the mentioned benefits. Simulations in fields such as these have increased both in number and detail, resulting in an explosion in simulated data volumes.

There has also been a shift towards electronic data collection in *traditional* experiments. Today, scientific apparatuses from many domains stream their data directly onto digital storage devices. These devices include DNA arrays, particle colliders, telescopes, and environmental sensor networks. Some apparatuses create terabytes to petabytes of data per year. For example, the LSST telescope, which will come online in 2012, will produce about 10 petabytes per year (20; 21).

We are also seeing an increase in the number of instruments in operation. This is known as the "scale-out" effect: as experimental instruments become commoditized and drop in cost, the number of instruments deployed increases, often exponentially (22). The result is a second driving force behind the explo-

sion in experimental data. This trend mimics the scale-out effect of computer hardware, which is embodied in the concept of a beowulf cluster(10).

2.1.2 Reduction

As the data produced during generation has turned digital, the ensuing reduction and analysis stages have followed suit. Scientific data created during generation usually comes in the form of low-level readings or signals organized in space and time. During reduction, these low-level readings are combined into higher-level objects of interest. For example, a telescope counts the number of electrons passing through a fixed area per wavelength and time-interval. An "image" is essentially an array of such readings for a given time-interval. This data, in its low-level form, is difficult to reason about. It must be transformed into higher-level object representations; such as stars and galaxies. By further aggregating data about stars and galaxies over time, one can then synthesize even higher-level data objects such as representations of supernovae and gravitational micro-lensing events [1]. The process of transforming low-level measurements into higher-level objects and events is *reduction*. As more scientific data becomes represented digitally, scientists are using computer algorithms to reduce them. Typical workflows are composed of programs for generating and reducing scientific data. Section 2.3 further elaborates on these workflows.

[1] A gravitational microlensing event occurs when a massive object passes in front of a radiating object and the gravitational field of the massive object magnifies the intensity of the radiating object. One use of these events is in detecting planets that are otherwise undetectable.

2.1.3 Analysis

The reduction of scientific data into high-level objects leads to the third stage, analysis. During this stage, scientists pose hypotheses and validate them over their data (collections of high-level objects). As one example, an astronomer may pose the hypothesis that "bright galaxies tend to have a low `local extinction`." To validate this hypothesis, he will examine his collection of galaxy objects, correlating their brightness against their `local extinction`. The collection of galaxy objects may be large and slow to examine manually. Often, a computer is the only feasible means of performing such analyses. Therefore, similar to the generation and reduction stages, scientists are relying heavily on computers during the analysis stage. Simple analyses may take place on spreadsheets. More complicated analyses may involve writing programs in languages such as Python, C++ or Matlab.

In recent years, scientists have increasingly turned to databases for managing and mining their datasets (19). Under the database paradigm, scientists transform and retrieve datasets using Structured Query Language (SQL), a specialized language for data manipulation. The scope of SQL is broad enough to answer many scientists' questions while the simplicity of the language allows questions to be posed with little effort. In addition, databases provide scientists with automatic resource optimization, sparing the scientist of yet another chore. With the advent of shared scientific archives available through the internet, many scientists have disengaged themselves from tasks in the generation and reduction stages, solely focusing their efforts on analysis (22). The Sloan Digital Sky Survey (SDSS) is a notable example of a large, publicly available scientific database.

SDSS offers large catalogs of astronomical objects to scientists over the world wide web. The site has received over 170 million page hits and answered over 20 million queries over the last 5 years (23).

2.1.4 Dissemination

The fourth stage, *dissemination*, has also been revolutionized by digitization. During the dissemination stage, scientists communicate their results to peers. Results are then debated and evaluated and then reconciled and synthesized with other results. Traditionally, dissemination has occurred in-person at meetings and conferences, or in-print, in journals and conference proceedings. With the advent of the world wide web, however, much dissemination now occurs online. Online dissemination has many benefits: lower latency, searchability, automatic citation-counting and reduced shelf-space requirements. Notable examples of scientific publication archives include Citeseer (24), the ACM Digital Library (25) and Google Scholar (26).

2.1.5 Summary and Implications

In this section, we have introduced the Knowledge Supply Chain (KSC), a high-level model for the creation and dissemination of scientific knowledge. We have also described how scientists have adopted computer technology in every stage of the KSC. There exists a global complementarity across the 4 stages. As the degree of technological adoption increases in one stage, adoption is encouraged or forced in other stages. To list a few examples: if data created during

generation is in digital format, a scientist will be compelled to reduce it with a computer. Increased data volumes from the generation and reduction stages call for computer-aided processing during analysis. In the opposing direction, the availability of digital tools during analysis encourages the creation of digital data in the generation and reduction stages. The existence of positive feedback loops explains breadth of technology adoption and suggests sustained adoption in the future. A ramification of this trend is an increase in computation volumes, engendering a need for more and better computation management, as can be provided by DSWMSs.

2.2 Technological Trends

Having described how computation is being incorporated into the scientific process, I now turn to the individual technological trends that have spurred scientists to adopt technology. I cluster the trends into three areas:

1. **HPC Infrastructure**: Trends in this area make more computations *possible* by improving the availability of required resources.

2. **Parallel Programming Models**: Trends in this area catalyze the creation of scientific computations by making them *easier to specify*.

3. **Analysis, Visualization, and Publication**: Trends in this area increase the *yield* of computations by facilitating the extraction and dissemination of useful knowledge from computation-created data.

Each of the areas is addressed in turn.

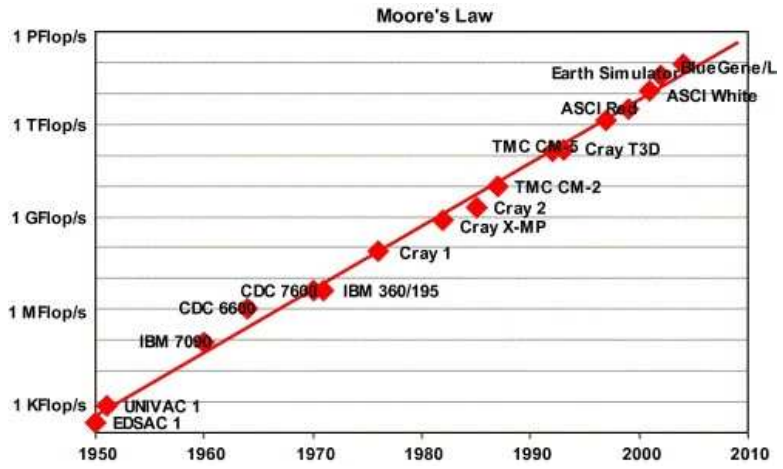


Figure 2.2. Exponential growth in computing power has been sustained over 6 decades, most recently by cluster computers. BlueGene/L, ASCI White and ASCI Red are all cluster computers. From (1).

2.2.1 HPC Infrastructure

We start with improvements in the performance cost of HPC infrastructure. These improvements have inflated the potential for running large collections of intensive computations. Infrastructural improvements are driven by the commoditization of hardware and software components. The end result of this commoditization is a move towards clustered architectures. Clustered architectures have sustained exponential growth in computing into its sixth decade (Figure 2.2), increased the average throughput of individual clusters, and reduced the cost of clusters.

A fundamental enabler of the clustered architecture is improved pricing of hardware components. Prices for processing, storage and networking elements have all fallen exponentially in the past few decades(27). As one example, the price of storage plummets at the rate of 100 times per decade(28). Today, one

can purchase 1 terabyte of hard disk space for only $400 (29), or 1 petabyte for only $400,000 (vs. about $40M 10 years ago).

A second enabler for clusters is a commoditization of software infrastructure. Such commoditization is largely due to the proliferation of the open-source software (OSS) development model. The OSS paradigm allows software to be installed for free, rather than for a licensing fee per machine. It has also allowed scientists modify their software when they feel it is necessary and worthwhile. In particular, the Linux operating system and associated utilities have been a cornerstone of the cluster revolution. It is now the most widely used operating system in scientific computing (30) and now runs more than 75% of the world's fastest 500 machines(1).

A new, but key, class of software is the *batch scheduler* (16; 31; 32; 33; 34). A batch scheduler allows users to submit cluster jobs through a single node. The software gives the illusion that the user has at her disposal a single, powerful machine, while in actuality, the machine is composed of many small computing systems. The seminal system in this category is Condor (16). Statistics show that the number of hosts managed by Condor has doubled in each of the last three years and now stands at over 100,000 (35; 36).

Even while the power of individual clusters has been growing quickly, computer scientists have increased available computing even further through *grid computing*. Grid computing middleware allows cluster owners to share their computing resources with each other. Globus (17) is the archetypal instance in this category. Globus allows users to enroll their clusters into *computational grids* which allow seamless sharing of clusters across geographies and administrative

domains. For example, the Grid3 (37) project has federated 2800 CPUs from six science projects over 22 sites into a massive grid.

Grid computing would not be possible without the deployment of long-haul fiber optic networks. These deployments have tripled aggregate wide-area bandwidth each year for the past ten years (38). There is about 60,000 times more aggregate bandwidth now than a decade ago and the growth will continue for the next few years. Improvements in bandwidth are responsible for nullifying the barriers of geography.

Similar to grid computing, *public computing* has increased the amount of HPC available to scientists. Public computing projects allow individual users to donate spare computing cycles to run compute-intensive science applications. The first and most prominent example of public computing is seti@home (6) while follow-on projects such as folding@home (9) have also achieved impressive success. Most recently, the software infrastructure behind seti@home has been generalized to allow any scientific project to build a similar peer-to-peer network through the Berkeley Open Infrastructure for Network Computing (BOINC) project. As of January 2007, 37 science projects have adopted BOINC, harnessing 3000 CPU years per day from 1.5 million hosts and 900,000 users (39).

2.2.2 Parallel Programming Models

The previous section discussed improvements in computing infrastructure, the underlying resources of scientific computation. This section continues by describing improvements in programming models, which scientists use to express

programs that *utilize* those resources. Computer scientists have been working to improve the productivity of scientific programmers by designing efficient programming abstractions. The commissioning of the High Productivity Computing Systems (HPCS) project by DARPA, NSF and DOE (40) is a strong indication of a community focus on improving programmer productivity.

Several advancements in parallel programming models are making parallel programming easier. First, standardization on the Message Passing Interface (MPI) has streamlined the development of supporting tools and accumulation of knowledge (41). MPI is a language standard that allows a scientific programmer to control precisely when communication is incurred between processors. By controlling processor communication at a fine-grain, programmers are able to achieve high levels of performance from their HPC resources.

Unfortunately, MPI is based on primitive technologies that are two decades old. The abstractions provided by MPI are rather low-level, involving the marshaling of objects and transportation of bits and conceptually reflect hardware components rather than logical concepts. The result is that MPI programs can be difficult to program and make portable. To correct this shortcoming, there has been a recent push to develop programming models that provide the benefit of MPI — precise control over communication — but also provide higher-level programming abstractions. The result of these efforts is a family of languages, the Global Address Space (GAS) languages, which allow users to directly reference memory objects without having to marshall them or micro-manage their transportation. Such languages merge the best features of message passing and *shared memory* programming models. The shared memory model is an earlier

alternative to message-passing that is easy to use but is less scalable because it does not allow fine-grained control of communication costs. The GAS programming model has been integrated into several popular programming languages, java (Titanium (42), C (Unified Parallel C (43)) and Fortran (Co-Array Fortran (44)). While researchers are still optimizing implementations of these languages, progress is steady and promising. For readers who are unfamiliar with these languages, sample programs can be found in (41).

A third important development in parallel programming is the emergence of domain-specific parallel languages that accelerate the expression of a restricted set of programs. A recent example is the map-reduce scheme (45), which allows a user to parallelize his program onto many machines, given that he can model his program as `map` and `reduce` functions. Each function, `map` and `reduce`, processes a set of (key,value) 2-tuples and produces another set of (key,value) 2-tuples. The map function allows a user to emit one or more tuples from its input tuple-set while the reduce function performs aggregation on all input tuples sharing the same key. To obtain simpler programming, users of this paradigm give up fine-grained communication and the ability to iterate over their datasets an indeterminate number of times, which are otherwise available in languages such as MPI.

The seminal instance of the map-reduce paradigm is the MapReduce system at Google, Inc. (46). Google has used MapReduce to parallelize many of the most intensive and important web engineering tasks, such as constructing graphs of the web, calculating page rank, and measuring the popularity of different pages. Google cites thousands of programs being written in the paradigm and

asserts that users with no parallel programming experience are able to write and execute programs on thousands of machines within hours of being introduced to the model. Since Google's publication of MapReduce, other versions of the paradigm have surfaced, including an open-source package called Hadoop (47).

Decades before the rise of the map reduce paradigm, a similar value proposition — large-scale parallelization through a simple programming interface — was provided through parallel databases (48). In fact, the expressibility of the map-reduce model is equivalent to that provided in relational databases [2]. Map-reduce, however, has provided an alternative that allows processing over file system data. In contrast, parallel databases required that data be loaded into database-specific storage managers. Loading the data has proven to be a significant impediment to many scientific programmers (19). Beside the overheads of defining schemas the data becomes inaccessible to widely used text-processing utilities such as grep. By avoiding this loading, map-reduce has made simplified parallel programming available to more users.

While there has been solid advancement in parallel programming models, these developments will not substitute for the computation management infrastructure proposed in this thesis. The parallel programming models described here aim to split long-running computations into a set of *closely coupled* computations that are distributed over many CPUs but still collectively operate as one computation. Features in these programming models focus on the efficient exchange of low-level in-memory data structures. Computation management is a higher level

[2]While map reduce is more expressive than the relational algebra, databases are almost universally equipped with user-defined functions and aggregates, which allow them to match map reduce in expressibility.

activity involving the specification and manipulation of many instances of the programs, including those specified through parallel programming languages. As such, the capabilities provided by the two different technologies are *orthogonal* to one another. In fact, the adoption of better parallel programming models will encourage the creation of more computations and *exacerbate* need for computation management, not relieve it.

2.2.3 Analysis, Visualization and Publication

A third category of trends concerns an increase the *informational yield* of computational data, further encouraging the use of computation and heightening the need for computation management. Here, we focus on improvements in complementary technologies that make the data created by HPC *useful*. Each of these are trends in "downstream" technologies that are used in the *analysis* or *dissemination* stages of the KSC.

The first trend in this category is the adoption of large, web-based scientific databases. The seminal example of important scientific databases is the Sloan Digital Sky Survey (SDSS) (49), a digital archive that has been made available to scientists all around the world. The archive stores over 40 terabytes of astronomical data and makes it available to the public. The site features the ability to query data by clicking on images, writing ad hoc SQL, or accessing through a web services interface. The site has shown tremendous success over the last five years (23) and has even exhibited futuristic features such as "cooperative" querying, where mistakes in a user's queries can be automatically identified by

matching the query against similar, valid queries. It does not appear that there are any trends that will prevent these databases from scaling into the petabyte range within the next few years (50).

There has also been advancement in visualization technology(51; 52; 53; 54). Recently cited as one of the 10 most important advances in scientific computing over the last two decades (55), visualization enables scientists to create "big picture" views that are unavailable otherwise. Visual aids have evolved from two-dimensional, black-and-white drawings to three-dimensional, multi-colored, navigable renderings. In the process, they have increased the speed at which people can extract and communicate discoveries from digital data. Visualization has impacted a broad range of scientific disciplines, including aeropsace engineering(56), bio- and chem-informatics (57; 58) and climatology (59; 60).

A third important area of progress is in the domain of Internet-based publication. First used simply as a means of making traditional publications available on-demand, Internet-based publication is now enabling new features. For example, the Signaling Gateway Portal from the Nature Publishing Group (61; 62) is providing several features beyond searching and downloading. Through the portal, scientists can drill down into the datasets behind a publication instead of simply accepting a graph that represents one view of the available data. The ability to peruse supporting data allows scientists to repeat published experiments and validate hypotheses more thoroughly. Other features include the automatic clustering of related articles, and even automated interpretation of publication contents (e.g. "Show me all published statements about the *hedgehog* gene and also find opposing statements to each original statement"). Finally, these publica-

tions assimilate many of the positive attributes of traditional print publications; for example, they are peer-reviewed, citable, and maintained indefinitely.

2.2.4 Summary

This section has examined a compilation of trends that collectively enable, encourage, and/or force scientists to generate and use more computation in their daily lives. First, there has been increased deployment of standardized HPC infrastructure. These improvements increase the availability of resources required for running computations. Second, improvements in parallel programming have eased the specification of high-performance parallel programs. Finally, advancements in *downstream* technologies are helping scientists extract useful information from the results of their computations. As I argued in Section 2.1, scientists have demonstrated their satisfaction with the efforts of technologists by incorporating computing technologies into every aspect of their daily work. The two macro-trends described in this and the previous section — increased adoption of technology in the scientific process and sustained technological improvement — conjointly explain the explosion in computation volumes and justify the creation of a computation management infrastructure.

2.3 Workflows

In this section, we discuss the *workflow*, a concept that many emerging computation management systems are based on. We define a workflow as a composition

of dependent programs or processes (program executions). A workflow can be represented with a directed acyclic graph (DAG). Nodes in the DAG represent programs or processes while edges represent data dependencies. An edge from node A to node B indicates that A produces data that is used by B.

It is useful to distinguish between *concrete workflows* and *abstract workflows*. A concrete workflow is a composition of *processes*. A process is an instance of a program, and can therefore be defined by a program and a set of inputs on which the program is applied. The process can be submitted to an operating system for execution using a system call equivalent to UNIX's `exec`. Processes may execute on one or multiple nodes, using a parallel execution framework such as MPI or map-reduce. Inputs to the process consist of filesets (sets of files) that are accessible to the program and command-line arguments that are passed to the program.

In contrast to a concrete workflow, an abstract workflow is a set of dependent *programs* (rather than processes). Abstract workflows are used to define concrete workflows. By combining an abstract workflow with a set of inputs, one can define a concrete workflow. Typically, an abstract workflow is applied to multiple input instances to create a collection of concrete workflows. The concrete workflows are then executed in a data-parallel fashion on a cluster. For example, both the script of Listing 1.1 and the commands of Listing 1.2 conceptually create collections of concrete workflows. Besides being used to instantiate concrete workflows, abstract workflows can also be reused as components in other abstract workflows. GridDB's framework for module definition maximizes the potential for such reuse.

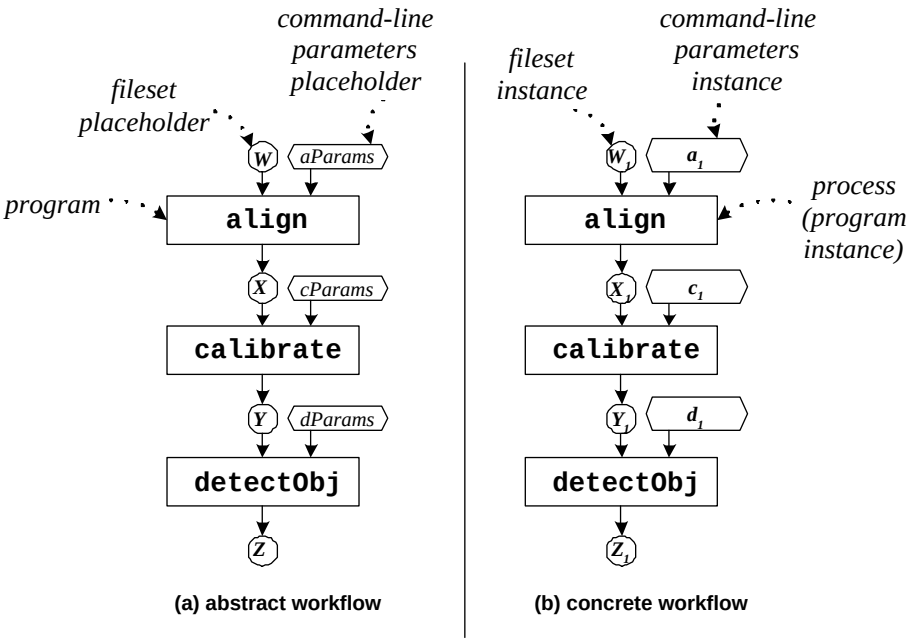


Figure 2.3. (a) An abstract workflow for the image processing workflow of Chapter 1 (b) A concrete workflow created by instantiating the abstract workflow of (a) with input set (X_1, a_1, c_1, d_1).

Figure 2.3(a) depicts the abstract workflow of the image processing application of Chapter 1. The abstract workflow uses as input one fileset (W)and 3 command-line inputs (a, c, and d) and produces as output three filesets (X, Y, and Z). Placeholders for fileset data are denoted by circles while placeholders for command-line inputs are denoted by hexagons. Figure 2.3(b) provides a concrete workflow that is based on the abstract workflow of Figure 2.3(a). This concrete workflow, when executed, applies the programs, `align`, `calibrate`, and `detectObj` to input instances W_1, a_1, c_1, and d_1. After execution, filesets X_1, Y_1, and Z_1 are created.

Workflows are useful constructs because they enable modularity, and scientists

use modularity to cope with the complexity of scientific computation. To solve their large problems, scientists divide and conquer their programs into smaller modules that can be designed, developed, debugged and executed in a piecewise fashion. Modularity has many other benefits. First, different modules may be developed concurrently by different programmers who have specialized skill sets. Modular designs are also amenable to incremental development, which is often more efficient or convenient. Finally, modular designs are easier to extend, since new components can be incorporated in a plug-and-play manner.

In contrast to the linear topologies of Figure 2.3, workflows often exhibit *fan-out* and *fan-in*. Fan-out occurs when the output of a program is consumed by more than one program or when a user applies an individual program over a data object multiple times (by varying another input to the program). Fan-in occurs when a workflow aggregates the data created by multiple processes. The fan-in may occur in both abstract and concrete workflows.

Examples of fan-out and fan-in are shown in the DAGs of Figure 2.4. Figure 2.4(a) demonstrates fan-out and fan-in in an abstract workflow. This workflow compares the results from two different versions of `detectObj`, `detectObj1` and `detectObj2` . Fan-out occurs because two different programs `detectObj1` and `detectObj2` are applied to fileset Y. Fan-in occurs because the program `cmpObjs` reads both filesets $Z1$ and $Z2$.

Figure 2.4(b) demonstrates fan-out and fan-in in a concrete workflow. This workflow executes the abstract workflow of Figure 2.3(a) on two sets of inputs (X_1, g_1, c_1, d_1) and (X_1, g_1, c_1, d_2) and then concatenates them into the same fileset. Since the two workflows' executions have identical executions of programs `align`

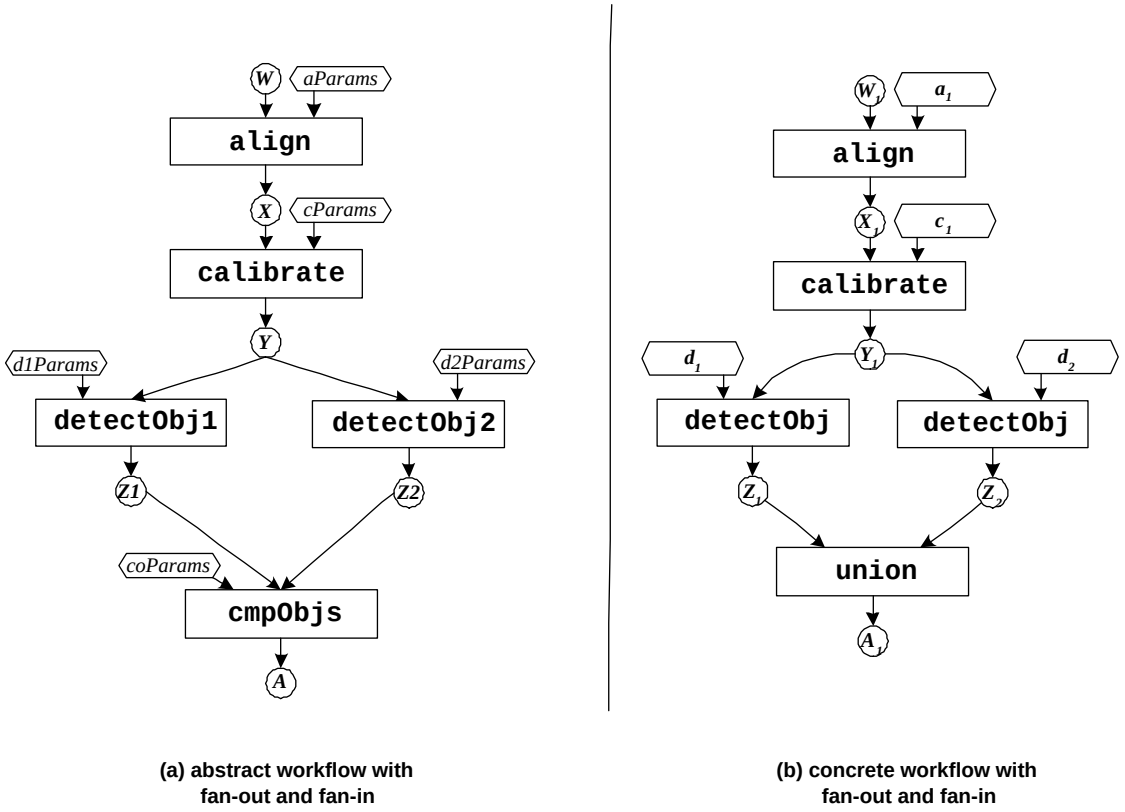


Figure 2.4. Workflows that exhibit fan-out and fan-in. (a) an abstract workflow. (b) a concrete workflow.

and `calibrate`, those processes are shared between the two workflows. Where they differ is in the execution of `detectObj`, where one uses parameter set d_1 and the other uses parameter set d_2. Fan-out occurs because the two processes `detectObj1` and `detectObj2` are applied to the same fileset. Fan-in occurs in this fileset because the `union` program takes both filesets as input to create the final fileset.

The workflow model described in this section is similar to that of other work-

flow systems described in the literature. These other systems are described as related works in Section 3.8.

2.4 Chapter Summary

This chapter has presented background information that is pertinent to the rest of this thesis. This chapter was composed of three main sections. The first section introduced the Knowledge Supply Chain (KSC), a high-level model of the scientific knowledge discovery process. It also showed how every step in the KSC has been transformed by computation. The second section continued with a review of recent trends in technology. These trends are increasing the capacity, ease-of-use, and informational value of computation. The confluence of these trends has resulted in an increase in the volume of computations being used, creating a need for computation management. The third section of this chapter offered an overview the workflow, a fundamental abstraction for specifying and representing computations.

Chapter 3

GridDB: A Prototype DSWMS

Having motivated the case for Data-Centric Scientific Workflow Management Systems (DSWMSs), this chapter presents the design and implementation of a DSWMS called GridDB. GridDB is based on two core principles: First, scientific programs can be abstracted as typed functions, and program invocations as typed function evaluations. Second, that while most scientific data is not relational in nature, a key subset, including the inputs and outputs of scientific workflows, have relational characteristics. This data can be manipulated with SQL and can serve as an interface to the full data set. Using this principle, users can be provided with: (1) a declarative, SQL-like interface to computation and (2) the benefits of data-centric processing as outlined in Section 1.

Following these two principles, I developed a scientific workflow data model, the Functional Data Model with Relational Covers (FDM/RC), and a data definition language for creating FDM/RC schemas. I then developed a set of software services that implement the data-centric GridDB model on top of existing

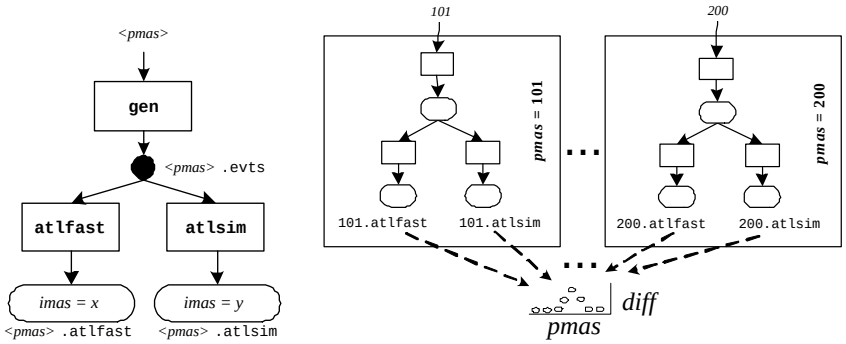
process-centric middleware. This chapter describes the FDM/RC model and its implementation and also demonstrates its usefulness with two example workflows taken from High Energy Physics and Astronomy. It also reports on experiments that validate the model and implementation.

The remainder of this chapter is structured as follows. Section 3.1 introduces a running example for the chapter. Section 3.2 describes the GridDB job management interface. Section 3.3 covers the FDM/RC data model. Section 3.4 presents the design and implementation of the GridDB prototype. Section 3.5 demonstrates GridDB's modeling of a complex workflow. Section 3.6 describes advanced performance-enhancing features. Sections 3.8 discusses Related Work and Section 3.9 summarizes the chapter.

3.1 High-Energy Physics Example

In this section we introduce a workflow obtained from the ATLAS High-Energy Physics experiment (63; 64). We refer to this workflow as `HepEx` (High Energy Physics Example) and use it as a running example throughout the chapter.

The ATLAS team wants to supplement a slow, but trusted detector simulation with a faster, less-precise, one. To guarantee the soundness of the fast simulation, however, the team must compare the responses of the new and old simulations to various physics events. A workflow achieving these comparisons is shown in Figure 3.1(a). It consists of three programs: an event generator, `gen`; the fast simulation, `atlfast`; and the original, slower simulation, `atlsim`. `gen` is



(a) (b)

Figure 3.1. (a) `HepEx` abstract workflow (b) `HepEx` grid job.

called with an integer parameter, *pmas*, and creates a file, ⟨*pmas*⟩`.evts` that digitally describes a particle's decay into subparticles. ⟨*pmas*⟩`.evts` is then fed into both `atlfast` and `atlsim`, each simulating a detector's reaction to the event, and creating a file which contains a value, *imas*. For `atlfast` to be sound, the difference between *pmas* and *imas* must be roughly the same in both simulations across a range of *pmas* values [1]. All three programs are long-running and compute-bound, and thus require parallelization on a cluster.

3.1.1 Roles of Scientists

Workflow execution often involves multiple users specializing in different roles. We identify three roles that occur in both the process- and data-centric paradigms. An individual scientist may play one or more of the roles within a

[1]The physics can be described as follows: *pmas* is the mass of a particle, while *imas* is the sum of subparticles after the particle's decay. $pmas - imas$ is a loss of mass after decay, which should be the same between the two simulations.

role	activity
coder	writes programs
modeler	composes programs
analyst	executes and analyzes results of programs

Figure 3.2. Roles in Workflow Execution.

specific application. The three roles are as follows (summarized in Figure 3.2) : *coders*, who write programs; *modelers*, who compose these programs into workflows; and *analysts*, who execute the workflows and analyze their results.

To deploy `HepEx` in the process-centric paradigm, *coders* write the three programs `gen`, `atlfast`, and `atlsim`, in an imperative language, and publish them on the web. A *modeler* then composes the programs into an *abstract workflow*, or AWF. Logically, the AWF , is a DAG of programs to be executed in a partial order. Physically, the AWF is encoded as a script(65), in perl or some other procedural language. Each program to be executed is represented with a *process specification* (proc-spec) file, which contains a program, a command-line to execute the program, and a set of input files (66; 67). The AWF script creates these proc-spec files along with a *precendence specification* (prec-spec) file that encodes the dependencies among the programs.

The *analyst* carries out the third and final step: *data procurement*. Existing middleware systems are sufficient for providing a single-machine interface to a cluster or grid(65). Thus, the *analyst* works as if he/she is submitting jobs on a single (very powerful) machine and the middleware handles the execution and

management of the jobs across the distributed resources. The *analyst* creates a *job* by executing another script that invokes the AWF script multiple times. For example, to run `HepEx` for all *pmas* values from 101 to 200, the AWF script would be invoked 100 times. Each invocation results in three processes being submitted and scheduled. Figure 3.1(b) shows the `HepEx` job consisting of these invocations.

3.2 Job Processing With GridDB

In the previous section, we identified three roles involved in the deployment of scientific processing applications. With GridDB, the job of the *coder* is not changed significantly; rather than publishing to the web, the coder publishes programs into a GridDB code repository available to other scientists. In contrast, the modeler sees a major change: instead of encoding the AWF in a procedural script, he expresses it in the GridDB data definition language (DDL), thereby conveying workload information to GridDB, and allowing it to provide data-centric services and interfaces. We describe the data model and DDL used by the *modeler* in detail in Section 3.3. The analyst's experience is also changed dramatically. Here, we focus on the *analyst*'s interactions using the GridDB data manipulation language (DML).

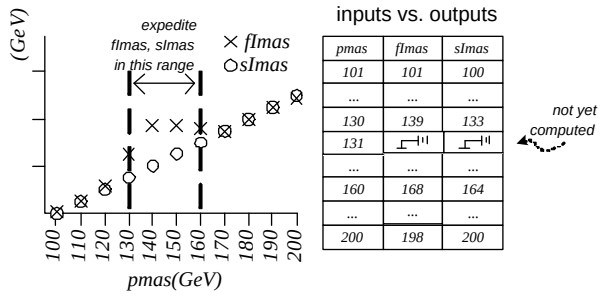


Figure 3.3. Example of Computational Steering.

3.2.1 Motivating Example: Computational Steering

To illustrate the benefits of data-centric workflow execution, we describe how Computational Steering can an analyst with provide increased flexibility and power to the job execution process.

Recall that GridDB provides a relational interface to workflow execution; for example, consider the table on the right-side of Figure 3.3. This table shows, for each value of an input *pmas*, the output *imas* values from two simulations (*fImas* and *sImas*). On the left-side of of the figure, we show a snapshot of streaming partial results in the table's corresponding scatter plot, which, at the time shown, has been populated with 20 out of 200 data points.

The scatter plot indicates discrepancies between *fImas* and *sImas* in the range where *pmas* is between 130 and 160, a phenomenon that needs investigation. Using the steering service, an analyst can prioritize data in this range simply by selecting the data in this range (e.g., by dragging a mouse pointer between the dashed lines) and prioritizing it with a GUI command. GridDB is capable of expediting the computations that create the points of interest. Through this

graphical, data-centric interface, the user drives workflow execution. In contrast, users of process-centric middleware usually run these jobs in batch, missing the opportunity direct limited resources towards the most important resources.

By understanding data and workflow structure, GridDB is able to provide many services that are unavailable in process-centric middleware. In the next section, we explain GridDB's modeling principles.

3.2.2 Functions and The Relational Cover

The data-centric GridDB model rests on two important principles: (1) the use of *functions* to model programs and the existence of the *relational cover*. As mentioned in Chapter 1, GridDB represents programs as functions in order to explicitly document the data that programs manipulate. Abstract workflows, which are logically compositions of programs,are then represented as *composite functions*. A functional representation does not require coders or modelers to change to programs themselves, but rather, consists of wrapping programs with functional definitions (as described in the following section). Functional representations allow GridDB to transcend the process-centric approach, where middleware is unaware of the data being manipulated by programs.

The *relational cover* is the subset of a workflow's data that can be represented as relations. As such, the relational cover can be described, and then manipulated using the Structured Query Language (SQL). GridDB then exploits this fact to provide users with a relational interface.

While most scientific data is not relational in nature, the inputs and outputs

to workflows can typically be represented as tables. For *input* data, consider data procurement, as described in Section 3.1: a scientist typically uses nested-loops in a script to enumerate a set of points within a multidimensional parameter space and invoke an AWF for each point (as shown in Listing 1.1 of Chapter 1). Each point in the input set can be represented as a tuple whose attributes are the point's dimensional values, a well-known technique in OLAP systems (68); therefore, an input set can be represented as a tuple set, or relation.

The relational nature of *outputs* is observed through a different line of reasoning: scientists commonly manipulate workflow output with interactive analysis tools such as *fv* in astronomy(69) and *root* or *paw* in high energy physics(70; 71). Within these tools, scientists manipulate (with operations like *projection* and *selection*) workflow data to generate multidimensional, multivariate graphs (72). Such graphs, including scatter plots, time-series plots, histograms, and bar-charts, are fundamentally visualizations of relations.

Figure 3.4 shows a `HepEx` model using these two principles[2]. In the figure, the `HepEx` workflow is a represented as a function, `simCompareMap` , which is a composition including three functions representing the workflow programs: `genF`, `atlfastF`, and `atlsimF`. The input data is represented as a relation of tuples containing *pmas* values, and the outputs are represented similarly.

[2]This model is created by DDL commands written by the modeler, as we describe in Section 3.3.

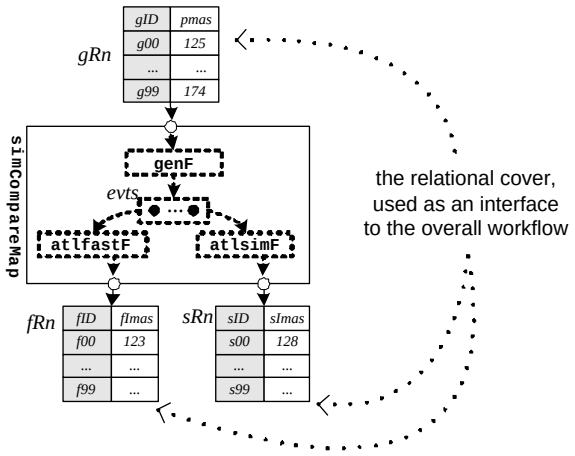


Figure 3.4. GridDB's `simCompareMap` corresponds to the process-centric workflows of Figures 3.1(a) and 3.1(b)

3.2.3 Data Manipulation Language

Having discussed the main modeling principles of GridDB, we now describe the data manipulation language provided for analyst interaction. With GridDB, *analysts* first create their own computational "sandbox" during a *workflow setup* phase. This sandbox consists of private copies of relations to be populated and manipulated by GridDB. Next, the *analyst* specifies the workflow he/she wishes to execute by connecting sandboxed relations to the inputs and outputs of a GridDB-specified workflow function.

The DML required for setting up `HepEx` is shown in Listing 3.1. Line 1 creates the sandbox relations. For example, `gRn` is declared with a type of *set of* `g`. The next statement (Line 2) assigns the two-relation output of `simCompareMap`, applied to `gRn`, to the output relations `fRn` and `sRn`. The modeler, at this point,

```
1 gRn:set(g); fRn:set(f); sRn:set(s);
2 (fRn,sRn) = simCompareMap(gRn);
3 INSERT INTO gRn VALUES pmas = @ {101,...,200} @;
4 SELECT * FROM autoview(gRn,fRn,sRn);
```

Listing 3.1. Analyst DML in HepEx

has already declared simCompareMap, with the signature: set(g) $\rightarrow$ set(f) $\times$ set(s), an action we show in Section 3.3.3.

With a sandbox and workflow established, *data procurement* proceeds as a simple INSERT statement into the workflow's input relation, gRn, as shown in Line 3. Insertion of values into the input relation triggers the issuance of proc-specs for process execution. The work submitted is conceptually similar to the job depicted in Figure 3.1(b). A readily apparent benefit of GridDB's data procurement is that INSERT's are type-checked; for example, inserting a non-integral value, such as 110.5, would result in an immediate exception.

Scientific jobs commonly seek to discover relationships between different physical quantities. To support this task, GridDB automatically creates relational views that map between inputs and outputs of functions. We call such views *automatic views*. For example, GridDB can show the relationships between tuples of gRn, fRn, and sRn in a view called autoview(gRn, fRn, sRn) (Line 4). Using this view, the *analyst* can see, for each value of *pmas*, what values of *fImas* resulted. The implementation of autoviews is described in Section 3.4.1. The autoview mechanism also plays in an important role in the provisioning of Computational Steering, as is discussed further in Section 3.6.1.

3.2.4 DML Summary

To summarize, GridDB provides a SQL-like DML that allows users to initiate workflow execution by creating private copies of relations, mapping some of those relations to workflow inputs and then inserting tuples containing input parameters into those relations. The outputs of the workflow can also be manipulated using the relational DML. The system can maintain *automatic views* that record the mappings between inputs and outputs; these "autoviews" are useful for analysis in their own right, and also play an important role in supporting Computational Steering. By understanding the semantics of workflow and data, GridDB is able to provide data-centric services unavailable in process-centric middleware.

3.3 Data Model: FDM/RC

The previous section illustrated the analyst's interface for job submission and steering, given that a modeler has defined a schema. In this section, we describe a data model for the schema and its the data model's definition language. The data model is called the Functional Data Model with Relational Covers(FDM/RC) and has two main constructs, *entities*, and *functions* which map entities to other entities. To take advantage of the relational cover, a subset of the entities are modeled as relations.

3.3.1 Core Design Concepts

Before defining the data model, we discuss two concepts that shape the FDM/RC: (1) the inclusion of *transparent* and *opaque* data and (2) the need for *fold/unfold* operations.

Opaque and Transparent Data

The FDM/RC models *entities* in three ways: *transparent*, *opaque* and *transparent-opaque*. One major distinction of GridDB, compared with process-centric middleware, is that it understands detailed semantics for some data, which it treats as relations. Within the data model, these are *transparent* entities, since GridDB can interact with their contents. By providing more information about data, transparent modeling enables GridDB to provide richer services. For example, the input, *pmas*, to program `gen` in Figure 3.1(a) is modeled as a transparent entity: for example, as a tuple with one integer attribute, *pmas*. Knowing that the input is a tuple with an integer attribute, GridDB can perform type-checking on `gen`'s inputs.

On the other hand, there are times when a modeler wants GridDB to catalog a file, but has neither the need for enhanced services. In these cases, GridDB allows lighter weight modeling of these entities as *opaque* objects. As an example, consider the output file of `x.evt`. The file is used to execute programs `atlfast` and `atlsim`. GridDB must catalog and retrieve the file for later use, but analysts do not need extra services. Therefore, the file is best modeled as an opaque entity.

Finally, there is *opaque-transparent* data, file data that is generated by pro-

grams, and therefore needs to be stored in its opaque form for later usage, but also needs to be understood by GridDB to gain data-centric services. An example is the output of `atlfast`; it is a file that needs to be stored, possibly for later use (although not in this workflow), but it can also be represented as a single-attribute tuple (attribute *fImas* of type `int`). As part of data analysis, the user may want to execute SQL over a set of these entities.

Unfold/Fold

The second concept deals with GridDB's other core abstraction: programs behind functions, i.e. a user makes a function call instead of program invocation. The *unfold* and *fold* operations define the "glue" between a program and its dual function.

For this abstraction to be sound, function evaluations must be defined by a program execution, a matter of two translations: (1) The function input arguments must map to program inputs and (2) the program's output files, upon termination, must map to the function's return values. These two mappings are defined by the *fold* and *unfold* operations, respectively. In Section 3.3.3, we describe *atomic* functions, which encapsulate imperative programs and employ fold and unfold operations.

3.3.2 Definition

Having described the core concepts of our data model, we now define it more carefully. The FDM/RC has two constructs: *entities* and *functions*. An FDM

schema consists of a set *entity*-sets, T, and a set of functions, F, such that each function, $F_i \in F$, is a mapping from entities to entities: $F_i : X_1 \times \ldots \times X_m \rightarrow Y_1 \times \ldots \times Y_n$, $X_i, Y_i \in T$, and can be the composition of other functions. Each *non-set* type, $\tau = [\tau_t, \tau_o] \in T$, can have a *transparent* component, τ_t, an *opaque* component τ_o, or both[3]. τ_t is a tuple of scalar entities. Set-types, $set(\tau)$, can be constructed from any type τ. The *relational cover* is the subset, R, of types, T, that are of type $set(\tau)$, where τ has a transparent component. An FDM/RC schema (T, R, F) consists of a type set, T; a relational cover, R; and a function set, F.

3.3.3 Data Definition Language

An FDM/RC schema, as we have just described, is defined by a data definition language (DDL). In this section, we describe the main constructs and illustrate them with HepEx's data definition, when possible. A complete grammar can be found in appendix A. The DDL for HepEx is shown in Listing 3.2.

Types

As suggested in Section 3.3.1, modelers can define three *kinds* of types: *transparent*, *opaque* and *transparent-opaque*. All types, regardless of their kind, are defined with the `type` keyword. We show declarations for all three `HepEx` types below.

Transparent type declarations include a set of typed attributes and are pre-

[3]One of τ_t and τ_o can be null, but not both.

fixed with the keyword `transparent`. As an example, the following statement defines a transparent type `g`, with an integer attribute *pmas*:

```
transparent type g = (pmas:int);
```

Opaque types do not specify attributes and therefore are easier to declare. *Opaque* type declarations are prefixed with the keyword `opaque`. For example, the following statement declares an opaque type *evt*:

```
opaque type evt;
```

Suppose an entity `e` is of an opaque type. It's opaque component is accessed as `e.opq`.

transparent-opaque type declarations are *not* prefixed, *and* contain a list of attributes; for example, this statement declares a transparent-opaque type `f`, which has one integer attribute *imas*:

```
type f = (imas:int);
```

A variable of type `f` also has an opaque component.

Finally, users can construct set types from any type. For example, this statement creates a set of `g` entities:

```
type setG = set(g);
```

Because `setG` has type `set(g)`, and `g` has a transparent component, `setG` belongs in the relational cover.

Functions

There are four kinds of functions that can be defined in DDL: *atomic*, *composite*, *map* and *SQL*.

Function interfaces, regardless of kind, are defined as typed lists of input and output entities. The definition header of atomic function `genF` is:

```
atomic fun genF (params:g):(out:evt)
```

This declares `genF` as a function with an input `params`, output `out`, and type signature `g` $\rightarrow$ `evt`. We proceed by describing body definitions for each kind of function.

As mentioned in Section 3.3.1, *atomic functions* embody programs, and therefore determine GridDB's interaction with process-centric middleware. The body of atomic function definitions describe these interactions. Three items need to be specified: (1) the program (using a unique program ID) that defines this function. (2) The *unfold* operation for transforming GridDB entities into program inputs and (3) the *fold* operation for transforming program outputs to function output entities. Three examples of atomic functions are `genF`, `atlfastF`, and `atlsimF`(see their headers in Listing 3.2, Lines 12-14). Because the body of an atomic function definition can be quite involved, we defer its discussion to Section 3.3.3.

Composite functions are used to express complex workflows, and then abstract it — analogous to the encoding of abstract workflows in scripts. As an example, a composite function `simCompare` composes the three atomic functions we have just described. It is defined with:

```
fun simCompare(in:g):(fOut:f,sOut:s) = (atlfastF(genF(in)), atlsimF(genF(in));
```

This statement says that the first output, `fOut`, is the result of function `atlfastF` applied to the result of function `genF` applied to input `in`. `sOutxo`, the second return-value, is defined similarly. The larger function, `simCompare`, now represents a workflow of programs. The composition is type-checked and can be reused in other compositions.

Map functions, or *maps*, provide a declarative form of finite iteration. Given a set of inputs, the map function repeatedly applies a particular function to each input, creating a set of outputs. For a function, F, with a signature $X_1 \times \ldots \times X_m \rightarrow Y_1 \times \ldots \times Y_n$, a map, $FMap$, with a signature: $set(X_1) \times \ldots \times set(X_m) \rightarrow set(Y_1) \times \ldots \times set(Y_n)$, can be created, which executes F for each combination of its inputs, creating a combination of outputs.

As an example, the following statement creates a map function, `simCompareMap` with the type signature $set(g) \rightarrow set(f) \times set(s)$, given that `SimCompare` has a signature $g \rightarrow f \times s$:

```
fun simCompareMap = map(simCompare);
```

We call `SimCompare` the *body* of `simCompareMap`. Maps serve as the front-line for data procurement — analysts submit their input sets to a map, and receive their output sets, being completely abstracted from the underlying hardware.

The benefit of transparent, relational data is that GridDB can now support *SQL* functions within workflows. As an example, a workflow function which joins two relations, holding transparent entities of r and s, with attributes a and b, and returns only r tuples, can be defined as:

```
   sql fun (R:set(r), S:set(s)):(ROut:set(r)) = sql(SELECT R.* FROM R,S WHERE
R.A = S.B);
```

In Section 3.5, we will show an SQL workflow function that simplifies a spatial computation with a spatial "overlaps" query, as used in an actual astrophysics application.

Fold/Unfold Revisited

Finally, we return to defining atomic functions and their *fold* and *unfold* operations. Recall from Section 3.3.1 that the fold and unfold operations define how data moves between GridDB and process-centric middleware.

Consider the body of function `atlfastF`, which translates into the execution of the program `atlfast`:

```
   atomic fun atlfastF(inEvt:evt):(outTuple:f) =
exec(
‘‘atlfast’’,
[(‘‘events’’,inEvt)],
[(/.atlfast$/, outTuple, ‘‘adapterX’’)]
)
```

The body is a call to a system-defined `exec` function, which submits a process execution to process-centric middleware. `exec` has three arguments, the first of which specifies the program (with a unique program ID) which this function

```
1 //opaque-only type definitions
2 opaque type evt;
3
4 //transparent-only type declarations
5 transparent type g = (pmas:int);
6
7 //both opaque and transparent types
8 type f = (fImas:int);
9 type s = (sImas:int);
10
11 //headers of atomic function definitions for genF, atlfastF, atlsimF
12 atomic fun genF(params:g):(out:evt) = ...;
13 atomic fun atlsim(evtsIn:evt):(outTuple:s) = ...;
14 atomic fun atlfastF(inEvt:evt):(outTuple:f) =
15  exec(``atlfast'',
16       [(``events'',inEvt)],
17       [(/.atlfast\$/, outTuple, ``adapterX'')]);
18
19 //composite function simCompare definition
20 fun simCompare(in:g):(fOut:f,sOut:s) =
21  ( atlfast(gen(in)), atlsim(gen(in)) );
22
23 //a map function for simCompare
24 fun simCompareMap = map(simCompare);
```

Listing 3.2. Abridged HepEx DDL

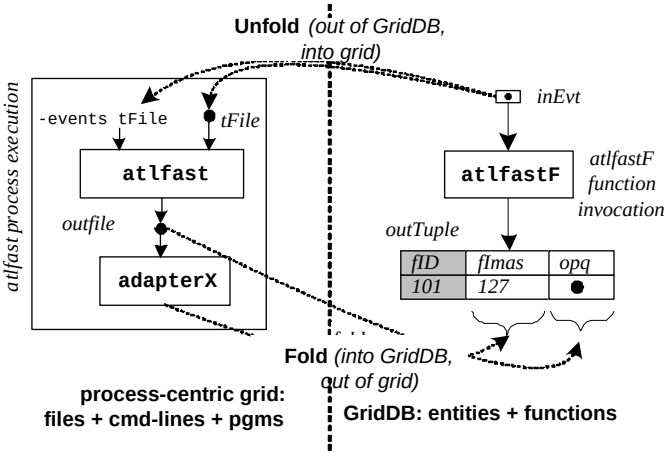


Figure 3.5. Unfold/Fold in *atomic* functions

maps to. The second and third arguments are lists that specify the *unfold* and *fold* operations for each input or output entity.

The second argument is a list of pairs (only one here) which specifies how arguments are unfolded. In this case, because inEvt is an opaque entity, the file it represents is copied into the process' working directory before execution, and the name of the newly created file is appended to the command-line, with the tag `events`. For example, `atlfast` would be called with the command-line `atlfast -events tFile`, where `tFile` is the name of the temporary file (top of Figure 3.5).

The last argument to `exec` is also a list, this time of triples (only one here), which specify fold operations for each output entity (bottom of Figure 3.5). In this case, the first list item instructs GridDB to look in the working directory after process termination, for a file that ends with `.atlfast` (or matches the regular expression `/.atlfast$/`). The second item says that the `opq` component

of the output, `outTuple`, resolves to the newly created file. The third item specifies an adapter program — a program that extracts the attributes of `outTuple`'s transparent component into a format understandable by GridDB; for example, comma-separated-value format. GridDB ingests the contents (in this case, *fImas*) into the transparent component. The adapter program is also registered by the *coder* and assigned a unique program ID.

3.4 GridDB Design

Having discussed GridDB's programming interface, we now turn our discussion to the design of GridDB, focusing on the query processing of analyst actions, as embodied in DML statements. GridDB's software architecture is shown in Figure 3.6. The GridDB overlay mediates interaction between a GridDB Client and process-centric middleware. Four main modules implement GridDB logic: the Request Manager receives and initializes queries; the Query Processor manages query execution; the Scheduler dispatches processes to process-centric middleware; and an RDBMS (we use *PostgreSQL*) stores and manipulates data and the system catalog.

In the rest of this section, we describe how GridDB processes DML statements. We do not discuss the processing of DDL statements, as they are straightforward updates to the system catalog.

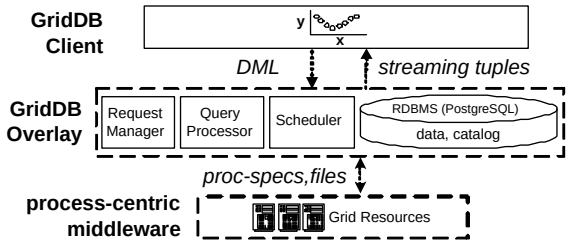


Figure 3.6. GridDB's Architecture

3.4.1 Query Processing

Our implementation strategy is to translate GridDB DML to SQL, enabling the use of an existing relational query processor for most processing. One consequence of this strategy is that the main data structures must be stored in tables. In this section, we take a bottom-up approach, first describing the tabular data structures, and then describing the query translation process.

Tabular Data Structures

GridDB uses three kinds of tables; the first two store entities and functions. The last stores processes, which are later outsourced to process-centric middleware for execution. We describe these three in turn.

Entity Tables: Recall from Section 3.3.1 that non-set entities may have two components: a transparent component, τ_t, which is a tuple of scalar values; and an opaque component, τ_o, which is an opaque set of bits. Each entity also has a unique system-assigned ID. Thus, an entity of type τ having an m-attribute transparent component (τ_t) and an opaque component (τ_o) is represented as the

following tuple: $(\tau ID, \tau_t.attr_1, \ldots, \tau_t.attr_m, \tau_o)$. Entity-sets are represented as tables of these tuples.

Function Memo Tables: Given an instance of its input entities, a function call returns an instance of output entities. Function evaluations establish these mappings, and can be remembered in function *memo tables* (73). A function, F, with a signature $X \rightarrow Y$, has an associated memo table, $FMemo$, with the schema (FID, XID, YID). Each mapping tuple has an ID, FID, which is used for process book-keeping (see below); and pointers to its domain and range entities (XID and YID, respectively). Each domain entity can only map to one range entity, stipulating that XID is a candidate key. This definition is easily extended to functions with multiple inputs or outputs.

Process Table: Function evaluations are resolved through process executions. Process executions are stored in a table with the following attributes: $(PID, FID, funcName, priority, status)$. PID is a unique process ID; FID points to the function evaluation this process resolves; *priority* is used for execution order; and *status* is one of *done*, *running*, *ready*, or *pending*, where a pending process cannot execute because another process that creates its inputs is not *done*.

Query Processing: Translation to SQL

Having represented entities, functions and processes as tables in the RDBMS, query processing proceeds predominantly as SQL execution.

In this section, we describe how analyst actions are processed and show, as

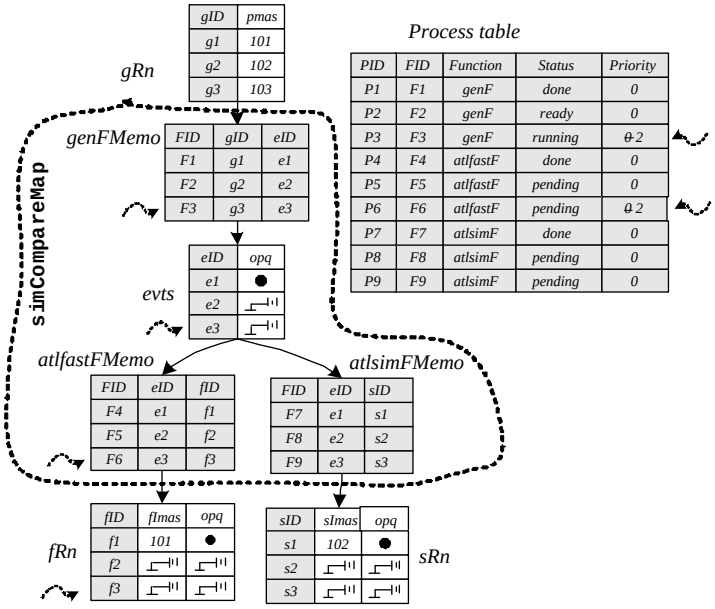


Figure 3.7. Internal data structures representing `HepEx` functions, entities, and processes. Shaded fields are system-managed. Dashed arrows indicate interesting tuples in our discussion of Computational Steering (Sec. 3.6.1)

an example, query processing for the `HepEx` use-case. Internal data structures for `HepEx` are shown in Figure 3.7. The diagram is an enhanced version of the analyst's view (Figure 3.4).

Recall from Section 3.2, the three basic analyst actions: *workflow setup* creates sandbox entity-sets and connects them as inputs and outputs of a map; *data procurement* submits inputs to the workflow, triggering a function evaluation to create outputs. Finally, streaming partial results can be perused with *automatic views*. We repeat the contents of listing 3.1 for convenience:

```
1:  gRn:set(g); fRn:set(f); sRn:set(s);

2:  (fRn,sRn) = simCompareMap(gRn);
```

```
3:  INSERT INTO gRn VALUES pmas = {101,...,200};
4:  SELECT * FROM autoview(gRn,fRn);
```

Workflow Setup During workflow setup (Lines 1-2), tables are created for the entity-sets and workflow functions. Workflow setup creates a table for each of the four entity-sets (`gRn`, `fRn`, `sRn`, `evts`), as well as each of the three functions ($genFMemo, atlfastFMemo, atlsimMemo$). `evts`is a table for storing intermediate results, which are created by `genF`. It is not created by directly by the user, but by GridDB while resolving the definition of `simCompareMap`. At this step, GridDB also stores a *workflow graph* (represented by the solid arrows in the figure) for the job.

Data procurement and Process Execution Data procurement is performed with an `INSERT` statement (Line 3) into a map's input entity-set variables. In GridDB, `INSERT`s into entity-tables trigger function evaluations, if a workflow graph indicates that the entity is input to a function. Function outputs are appended to output entity tables. If these tables feed into another function, function calls are recursively triggered. Calls can be resolved in two ways: a function can be evaluated, or a memoized result can be retrieved. Evaluation requires *process execution*.

Process execution is a three step procedure that uses the fold and unfold operations described in Section 3.3.3. To summarize: first, the function's input entities are converted to files and a command-line string using the *unfold* op-

eration; second, the process (defined by program, input files and command-line string) is executed on computational resources; and third, the *fold* operations ingest the process' output files into GridDB entities.

In the example of Figure 3.7, a data procurement `INSERT` into `gRn` has cascaded into 9 function calls ($F1$-$F9$ in the three function tables) and the insert of tuple stubs (placeholders for results) for the purpose of partial results. We assume an absence of memoized results, so each function call requires evaluation through a process ($P1$-$P9$ in the process table).

The process table snapshot of Figure 3.7 indicates the completion of three processes ($P1$, $P4$, $P7$), whose results have been folded back into entity tables (entities $e1$, $f1$, $r1$, respectively).

Automatic Views (Autoviews) A user may peruse data by querying an autoview. Because each edge in a workflow graph is always associated with a foreign key-primary key relationship, autoviews can be constructed from workflow graphs. As long as a path exists between two entity-sets, an automatic view between can be created by joining all function- and entity-tables on the path.

In Figure 3.8, we show `autoview(gRn, fRn)`, which is automatically constructed by joining all tables on the path from `gRn` to `fRn` and projecting out non-system attributes.

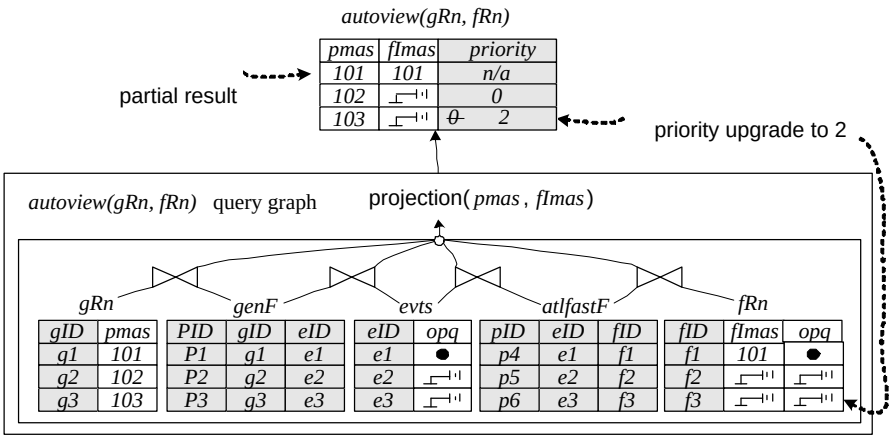


Figure 3.8. `autoview(gRn, fRn)`

3.5 `ClustFind`: A Complex Example

Up until this point, we have demonstrated GridDB concepts using `HepEx`, a rather simple use-case. In this section, we describe how GridDB handles a complex astronomy application. First, we describe the application science and general workflow. Next, we describe how the workflow can be modeled in the FDM/RC. Finally, we show how the example benefits from memoization and interactive query processing, advanced features that we describe in the following section.

3.5.1 Finding Clusters of Galaxies

The Sloan Digital Sky Survey (SDSS) (74) is a 12 TB digital imaging survey mapping 250,000,000 celestial objects with two orders of magnitudes greater sensitivity than previous large sky surveys. `ClustFind` is a computationally-intense

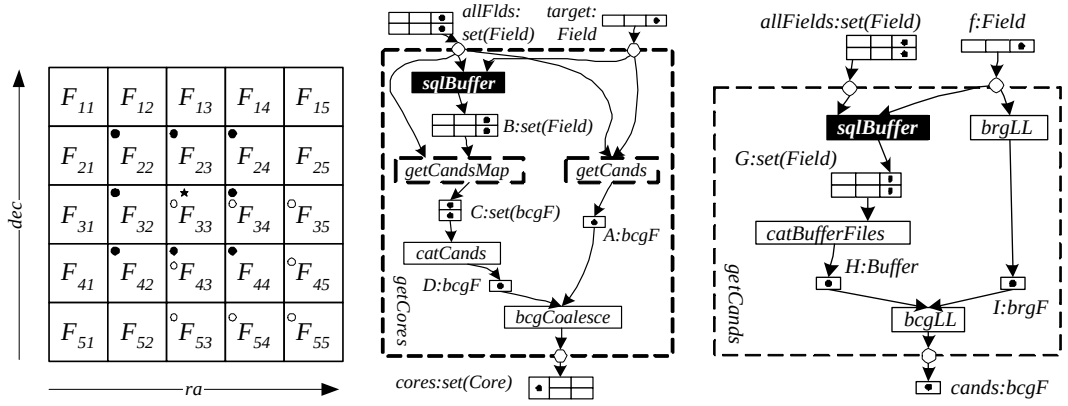
SDSS application that detects galaxy clusters, the largest gravitation-bound objects in the universe. The application employs the MaxBCG cluster finding algorithm (75), requiring 7000 CPU hours on a 500 MHz computer (76).

In this analysis, all survey objects are characterized by two coordinates, *ra* and *dec*. All objects fit within a two-dimensional mesh of fields such that each field holds objects in a particular square (Figure 3.9(a)). The goal is to find, in each field, all cluster *cores*, each of which is the center-of-gravitation for a cluster. To find the cores in a *target* field (e.g., F_{33}, annotated with a $\star$ in Figure 3.9(a)), the algorithm first finds all core *candidates* in the target, and all candidates in the target's "buffer," or set of neighboring fields (in Figure 3.9(a), each field in the buffer of F_{33} is annotated with a $\bullet$). It then applies a core selection algorithm, which selects cores from the target candidates based on interactions with buffer candidates and other core candidates.

3.5.2 An FDM/RC Model for `ClustFind`

In this section, we describe the FDM/RC function, `getCores`, which, given a target field entity, returns the target's set of cores. `getCores` is shown as the outermost function of Figure 3.9(b). The analysis would actually build a map function using `getCores` as its body, in order to find cores for *many* targets.

`getCores` is a composite of five functions: `getCands`, on the right-side of the diagram, creates A, a file of target candidates. The three left-most functions — `sqlBuffer`, `getCandsMap`, and `catCands`— create D, a file of buffer candidates. Finally, `bcgCoalesce` is the core selection algorithm; it takes in both buffer



(a) (b) (c)

Figure 3.9. (a) `ClustFind` divides sky objects into a square mesh of buckets (b) `getCores`, the body of the top-level `ClustFind` map function. (c) `getCands`, a composite subfunction used in `getCores`.

candidates, D, and target candidates, A, returning a file of target cores, `cores`. During the fold operation, `cores` is ingested as a set of `Core` entities (shown at the bottom of Figure 3.9(b)). `ClustFind` analysis is carried out with a map based on the `getCores` function we have just described, mapping each target field to a set of cores.

This use-case illustrates three notable features not encountered in `HepEx`: (1) it uses an SQL function, `sqlBuffer`. Given a target field (`target`) and the set of all fields (`allFields`), `sqlBuffer` uses a spatial overlap query to compute the target's buffer fields, B. (2) it uses a nested map, `getCandsMap`, which iterates over a dynamically created set of entities. This means that materialization of B will create a new set of processes, each executing the contents of `getCands` to map an element of B to an element of C. (3) `getCores`, as a whole, creates a

set of `Core` objects from one target, having a signature of the form $\alpha \rightarrow set(\beta)$. This pattern, where one entity maps to a *set* of entities, is actually quite common and suggests the use of a *nested relational model and query language*(77). This use-case suggests that extending the FDM/RC with nested types is an interesting avenue for future exploration.

In Figure 3.9(c), we show `getCands`, a function used in `getCores`, and also the basis of `getCandsMap`, which is also used in `getCores`. Given a target field, f, `getCands` returns a set of core candidates, `cands`. It is interesting to note that, like `getCores`, a buffer calculation is needed to compute a field's candidates — resulting in the reuse of `sqlBuffer` in `getCands`. As an example, in computing the candidates for F_{44}, we compute its buffer, or the set of fields annotated with a $\circ$ in Figure 3.9(a). Note that the `bcgLL` function within `getCands` is the most expensive function to evaluate (76), making `getCands` the bottleneck in `getCores`.

The use-case demonstrates two kinds of entity types (examples in parentheses): opaque (A), and transparent-opaque (*target*); set types (C); and all four kinds of functions: atomic (`bcgCoalesce`) , composite (`getCands`), sql (`sqlBuffer`) , map (`getCandsMap`). The atomic functions, which cause process executions, are the solid boxes.

3.5.3 Memoization & Steering In `ClustFind`

Embedded in our description is this fact: `getCands` is called ten times per field: twice in computing the field's cores and once for computing the cores for

each of its eight neighbors. By modeling this workflow in a GridDB schema, astronomers automatically gain the performance of memoization, without needing to implement it.

Secondly, it is common for astronomers to point to a spot on an image map — for instance, the using SkyServer interface(74) — and query for results from those coordinates. As these requests translate to (ra, dec) coordinates, GridDB's data-driven steering service accomodates selective prioritization of interesting fields. We describe the implementation of both of these advanced features in the next section.

3.6 Performance Enhancements

GridDB's model serves as a foundation for two other performance-enhancing features: computational steering and memoization. We describe these two services and how they apply to `HepEx` and `ClustFind`.

3.6.1 Computational Steering

Due to the conflict between the long-running nature of computation jobs and the iterative nature of knowledge discovery, scientists have expressed a need for computational steering (78), the ability to view and prioritize parts of a job while it is running (63; 79).

In this section, we describe the steering of jobs through a relational interface. We introduce steering with an example. Consider the autoview that was shown

at the top of Figure 3.8. The view presents the relation between *pmas* and *fImas* values. The user has received one partial result, where pmas= 101. At this point, the user may upgrade the priority of a particular tuple (with pmas= 103) with an SQL `UPDATE` statement:

```
A: UPDATE autoview(gRn, fRn) SET PRIORITY = 2 WHERE pmas = 103
```

By defining a relational cover, GridDB allows prioritization of data, rather than processes. The GridDB `UPDATE` statement is enhanced; one can update the `PRIORITY` attribute of any view. This scheme is expressive: a set of views can express, and therefore one may prioritize, any combination of cells in a relational schema (the relational cover).

Next, we turn to how such a request affects query processing and process scheduling, where GridDB borrows an technique from functional languages, that of lazy evaluation (73). Any view tuple can always be traced back to entities of the relational cover, using basic data lineage techniques (80). Each entity also has a functional expression, which encodes all necessary and sufficient function evaluations. Since function evaluations are associated with process execution, GridDB can prioritize only the most relevant process executions, delaying other, less-relevant computations.

As an example, consider the processing of the prioritization request in Figure 3.8. The only missing uncomputed attribute is *fImas*, which is derived from from relational cover tuple $f3$. Figure 3.7 (see dashed arrows) shows that $f3$ is a result of function evaluation $F6$, which depends on the result function of evaluation $F3$. The two processes for these evaluations are $P3$ and $P6$, which

are prioritized. Such lineage allows lazy evaluation of other irrelevant, possibly function evaluation, such as any involving `atlsimF`.

In summary, the FDM/RC, with its functional representation of workflows and relational cover, provides a data-centric, tabular interface for process prioritization.

3.6.2 Memoization

Recall from Section 3.4.1 that function evaluations are stored in memo tables. Using these tables, memoization is simple: if a function call with the same entities has been previously evaluated and memoized, we can return the memoized entities, rather than re-evaluating. This is possible if function calls, and the programs which implement them, are deterministic. Scientific programs are often deterministic, as repeatability is paramount to experimental science (81). However, if required, our our modeling language could be extended to allow the declaration of non-deterministic functions, which may not be memoized, as is done with the `VARIANT` function modifier of PostgreSQL (82).

3.7 Validation

To demonstrate the effectiveness of steering and memoization, we conducted validation experiments with our prototype GridDB implementation and the small cluster testbed of Figure 3.10. We implemented a java-based prototype of GridDB, consisting of almost 19K lines of code. Modular line counts are in Table.

Module(s)	LOC	Module(s)	LOC
Rqst Mgr. & Q.P.	1495	Catalog Routines	756
Scheduler	529	Data Structures	7207
Client	7471	Utility Routines	1400
Total	18858		

Table 3.1. LOCs for a java-based GridDB prototype.

3.1. The large size of the client is explained by its graphical interface, which we implemented for a demonstration of the system (83). The system was built using Condor (16) as its process-centric middleware; therefore, it allows access to a cluster of machines. An alternative would be to use Globus, which would enable it to leverage distributively-owned computing resources. The change should not be conceptually different, as both are process-submission interfaces.

Measurements were conducted on a small cluster consisting of six nodes (Figure 3.10). The GridDB client issued requests from a laptop while the GridDB overlay, a "Condor Master" batch scheduler (16) and 4 worker nodes each resided on one of 6 cluster nodes. All machines, with the exception of the client, were Pentium 4, 1.3 GHz machines with 512 MB RAM, running Redhat Linux 7.3. The client was run on an IBM Thinkpad Mobile Pentium 4, 1.7 GHz with 512 MB RAM. The machines were connected by a 100 Mbit network. This chapter presents simple validation on a small cluster. In the next chapter, we extend these results with a more realistic validation of GridDB on a large, industrial cluster.

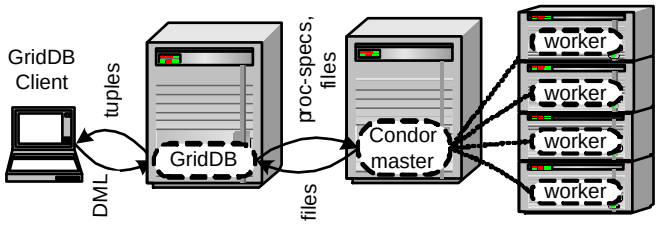


Figure 3.10. Experimental setup.

Validation 1: Computational Steering

In the first validation experiment, we show the benefits of steering by comparing GridDB's *dynamic* scheduler, which modifies its scheduling decisions based on interactive data prioritizations, against two static schedulers: *batch* and *pipelined*. In this experiment, an analyst performs the data procurement of Section 3.2, inserting 100 values of *pmas* into `simCompareMap`. 200 hundred seconds after submission, we inject a steering request, prioritizing 25 as yet uncomputed `f` tuples:

`UPDATE autoview(gRn, fRn) SET PRIORITY = 2 WHERE` $131 \leq$`pmas`≤ 150

The *batch* scheduler evaluates all instances of each function consecutively, applying `genF` to all *pmas* inputs, and then to `atlsimF`, and then `atlfastF`. The *pipelined* scheduler processes one input at a time, starting with *pmas*=1, and applying all three functions to it. Neither changes its schedule based on priority updates. In contrast, the GridDB *dynamic* scheduler does change its computation order as a user updates preferences.

In Figure 3.11, we plot the *Number of Prioritized Data Points* returned vs. *time*. As can be seen in the plot, GridDB(dynamic) delivers all 20 interesting

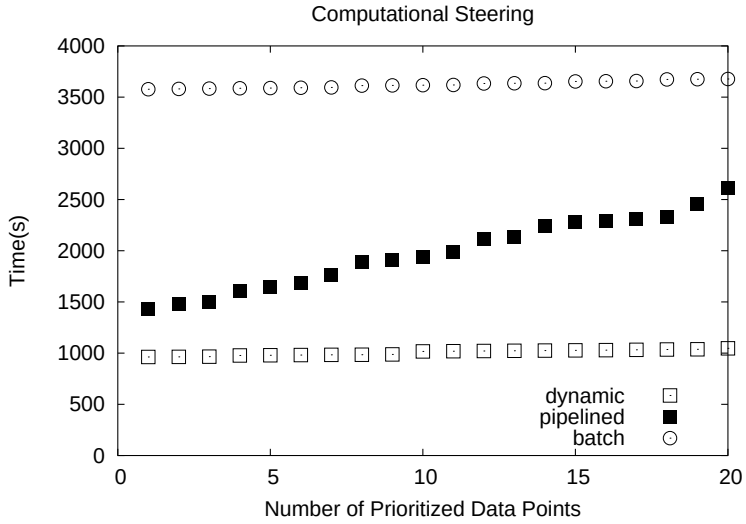


Figure 3.11. Validation 1, Computational Steering for `HepEx`.

results within 1000s. In contrast, the static pipelined and batch schedulers require 2608s and 3677s, respectively. In this instance, GridDB cut "time-to-interesting-result" by 60% and 72%, respectively.

The performance gains are due to the *lazy evaluation* of the expensive function, `atlsimF`, as well as the prioritization of interesting input points, two effects explained in Section 3.6.1.

Validation 2: Memoization Speedup

We validated the GridDB memoization implementation by testing how well it exploits `ClustFind` memoization opportunities (from Section 3.5). We observed that when memoization is used, system throughput speeds up by 6.13 relative

to when it is absent. Note that process-centric middleware typically does not provide a memoization service.

In these experiments, we used GridDB to drive cluster core search for square meshes of varying size. The smallest, of size 5, is shown in Fig. 3.9(a). Each field was of length 0.1×0.1 degrees. Recall from Section 3.5.3 that the GridDB modeling of `ClustFind` analysis presents a prime opportunity for memoization, as the most expensive functions are also repeated many times.

As shown in Fig. 3.12, an analysis using memoization (memo) out-performs an analysis without memoization (noMemo) for meshes from sizes 6 to 19. Meshes of size 5 have no memoization opportunities; we can only calculate one target (each target requires a 5 by 5 buffer around it for computation). At mesh size 19 (361 fields), a memoized analysis requires 5041 seconds while one without memoization requires 30894 seconds — a speedup of 6.13.

3.8 Related Work

In this section, we survey alternatives to GridDB's data-centric approach to supporting scientific workflows. The major alternatives fall into three categories: process-centric middleware, *non-data-centric* workflow systems and traditional database systems. As we will argue in this section, GridDB is the first system to support a combination of three features. First, GridDB employs the use of schemas and a 2-phase programming model (see Chapter 3) for representing fundamental characteristics of an incoming workload. Second, unlike most other workflow systems, GridDB's schema paradigm allows the modeling of data struc-

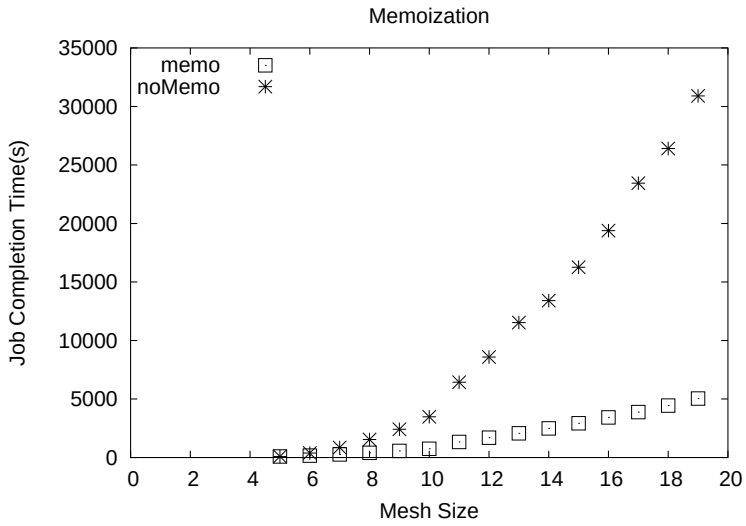


Figure 3.12. Validation 2, Memoization in `ClustFind`

tures *in addition* to workflow structures. GridDB exploits this added information to improve user experience. Finally, GridDB is designed and implemented to support massively parallel computer architectures, as opposed to small-scale desktop architectures.

The first category we discuss is process-centric middleware. This category of software includes batch schedulers such as Condor (16), PBS (84), SLURM (31), Torque (34) and cluster federation software such as Globus (17). These systems provide a streamlined process submission interface, essentially allowing users to submit processes to one machine, even though they will ultimately be dispatched to any of a large number of nodes within a massively parallel computer (such as a cluster). These systems often also provide traditional operating systems services for massively parallel computers; for instance, the matching of work to resources, or queuing services that help to avoid thrashing. Process-centric middleware

systems are a boon to scientists working on large computers. Without them, users would have to manage each job and node separately. As the practice of submitting large batches of computation becomes commonplace, however, the shortcomings of process-centric middleware become apparent. In particular, their abstractions — centered around "processes," or program executions — are probably too low-level for a scientist who is focused synthesizing large collections of processes.

This shortcoming has been partially remedied through the advent of *workflow systems*, which are often built on-top of process-centric middleware. Most fundamentally, workflow systems allow scientists to structure collections of computations by representing dependencies amongst computations. A collection of computations and their dependencies can be represented through directed acyclic graphs, which are synonomous with the term "workflow." Equipped with these representations, workflow systems can automatically enforce ordered execution of processes. They can also provide many of the value-added services that we mention in this chapter, including the tracking of lineage.

Workflow management systems for "grid computing" represent a vibrant sector of computer science research. For example, (85) offers a survey of more than 30 workflow systems, languages and tools. Some of the more well-known systems include Chimera (76; 86), DAGMan (87), Ptolemy (88) and MyGrid (89). Relative to these systems, GridDB was the first system to advocate data-centricity, an understanding of data structure within workflows in addition with workflow structure. As shown earlier in this chapter, data-centricity gives GridDB additional information over workflow structure and allows it to further improve user experience.

The final category of alternatives is the traditional *Database Management System* (DBMS). Traditional DBMS's pioneered the use of a 2-phase programming model for data management and therefore provide good features for querying large collections of structured data. In fact, the techniques of traditional DBMS's have seen much recent success. On a small-scale, scientists have implemented domain-specific lightweight database utilities that allow users to quickly analyze small datasets using simple relational operations such as select and project. Examples of these systems include Fitsviewer (69) in Astronomy and Root (70) in Physics. On a large-scale, the Sloan Digital Sky Survey (SDSS) has proven the scalability and advantages of using DBMS's to store, manage and serve large scientific archives. SDSS has revolutionized the scientific process itself, enabling scientists to "outsource" data collection and cleaning to another party and focus solely on data analysis. It has inspired many scientists to learn SQL, an undertaking that many scientists were previously reluctant to commit to.

While traditionally DBMSs are well-suited for supporting the analysis task of scientists, their ability to support earlier phases of processing: data generation and transformation (recall the KSC of Chapter 2), are limited. In particular, there is minimal support in existing implementations for running large collections of processes expressed in traditional procedural programming languages (Fortran/C/C++/Java) on massively parallel computers. In addition, DBMSs lack abstractions for representing workflows and therefore providing services surrounding workflows.

One system, Zoo (90; 91), from the University of Wisconsin, is worthy of special mention. Zoo, which is a *Desktop Experiment Management Environment*

is the only other system to our knowledge to incorporate both workflow and database structure in its model for schemas. GridDB and Zoo bear many similarities; for example, Zoo also provides a data model (Moose) and query language (Fox) for representing and manipulating scientific worfklows and data (92). Also, the notions of *folding* and *unfolding* file data are addressed through the Frog and Turtle object-to-file mapping framework (93).

Unlike GridDB, however, Zoo was designed before the proliferation of commodity clusters and therefore was focused towards the desktop, rather than massively parallel architectures. Additionally, GridDB uses the simpler, relational data model, as opposed to an object-oriented data model (92), as is used in Zoo. We believe this simplicity to be an advantage and have argued in Section 3.2.2 why the model is sufficient for modeling an important subset of data, the inputs and outputs. Finally, GridDB leverages its knowledge of both workflow and data to provide computational steering services, which are unavailable in Zoo.

3.9 Chapter Summary

In this Chapter, I have presented GridDB, a DSWMS that provides a relational interface, and data-centric services for job submission and management. GridDB exploits two key principles: first, imperative programs can be modeled as typed-functions and second, that a key subset of data, the relational cover, can be modeled as relations, and used as a window to the full data set. As such, I have developed a data model (FDM/RC) and query language (a DDL and a DML) for representing workflows and accessing their data through a relational

interface. I have demonstrated the use of GridDB in modeling High Energy Physics and Astronomy use cases and have validated these ideas by measuring a prototype implementation in a small cluster environment. The next chapter experiments with the use of GridDB on a large cluster with hundreds of nodes.

Chapter 4

GridDB in a Realistic Environment

The previous chapter described the design and implementation of GridDB, a Data-Centric Scientific Workflow Management System (DSWMS). As part of the GridDB project, I conducted a "test deployment" at Lawrence Livermore National Laboratories (LLNL). This deployment was done in conjunction with several astrophysicists at the lab. I had several goals in mind: to further verify assumptions about science applications, to uncover gaps in language expressibility, and to identify performance and scalability barriers encountered by GridDB. The effort involved porting GridDB and several scientific codes onto a large cluster and then executing workloads using GridDB. In this chapter, I focus on one representative investigation within the test deployment. This particular investigation reveals a fundamental conflict between correctness and scalability.

The conflict arises because scientific programs routinely behave in a way that

undermines data provenance and memoization. A naive solution to this problem involves issuing a large number of metadata operations upon a cluster's underlying file system. Such a solution encounters scalability barriers because conventional file systems do not offer adequate metadata throughput. In the second half of this chapter, I examine an alternative solution that reduces metadata load. These reductions can be achieved with the help of user-provided hints. I run a set of scalability experiments that indicate improvements when using the alternative approach. The results motivate the ensuing chapter, which proposes a comprehensive scheme to improve GridDB's scalability.

The remainder of this chapter is organized as follows. Section 4.1 introduces the environment and scientific application that we use, including two alternative workflow models for the application. The next two sections, 4.2 and 4.3, recount our efforts in and lessons from executing the two workflows. Section 4.4 summarizes the chapter.

4.1 Deployment Environment and Application

We begin with a description of the science application and deployment environment. While our overall evaluation involved applications from different domains (including biology, biochemistry and national security), we focus here on the SuperMACHO astrophysics application to illustrate our findings. The SuperMACHO application analyzes data captured by a telescope at the Cerro Tololo Inter-American Observatory(CTIO) in the mountains of Chile. The SuperMACHO application had two characteristics that made it appealing as a testbed

application. First, collaboration scientists felt that data-centric services such as data provenance, memoization, and computational steering were promising productivity enhancements and so were interested in evaluating GridDB's viability. Secondly, the application has high resource demands, providing an opportunity to test GridDB's scalability. Processing a single night's worth of telescope images takes as much as 100 CPU days of compute power (using current CPU's).

The SuperMACHO project seeks to discover and catalog Massive Compact Halo Objects (MACHOs). MACHOs can be found by observing *gravitational microlensing* events, where the MACHO under observation passes in front of a luminous object, brightening it for a brief moment in time. These events can be detected by characterizing the brightness of sky objects over time. To measure objects' brightness, telescope images containing the objects are continually captured and analyzed through an image processing workflow.

A graphical representation of the application is shown in Figure 4.1. Raw images from the CTIO telescope are passed into the main SuperMACHO image processing codes, shown within the shaded box. These image processing codes analyze bitmapped images and extract objects, which are then loaded into a database for further analysis. The image processing codes were written in a combination of C and perl, spread across 5 separate packages. Each package has, on average, 10,000 lines of code. Each image requires an average of 18,905 seconds to process and as many as 500 images are processed each night.

To obtain the necessary computational power, we executed these pipelines on the Multiprogrammatic Capability Cluster (MCR), a large (11.2 Teraflop) linux cluster with 1152 nodes, each with two 2.4 GHz Pentium 4 Xeon processors

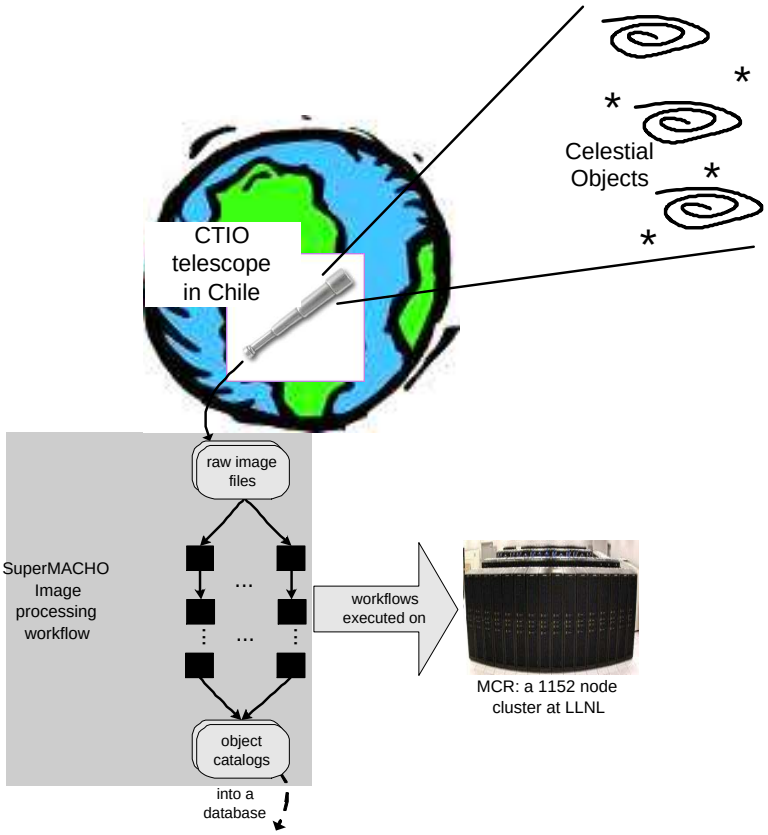


Figure 4.1. Deployment Scenario.

and 4 GB of memory. When we started our work in 2004, MCR was ranked as the 11th fastest computer in the world. Since MCR was shared by many users simultaneously, we were unable to run our pipelines on the entire cluster. The maximum number of nodes that we could acquire at once was 296. Fortunately, this allocation size was large enough for us to start identifying the scalability issues that DSWMSs face.

4.1.1 Application Modeling

There are many options for modeling the workflow of a scientific application. A key question that a modeler[1] faces concerns the granularity at which a workflow should be modeled. The advantages of finer-grained models include increased availability of intermediate data products for memoization, finer-grained provenance tracking and finer-grained control during computational steering. The advantages of coarser-grained models include decreases in modeling burden and middleware overhead. In this section, we describe two workflow models within the SuperMACHO image processing pipeline. The first is a coarse-grained model spanning the entire computation from images to objects. The second is a finer-grained model covering the first half of the computation at a fine granularity.

The coarse-grained workflow is shown in Figure 4.2. The workflow consists of two kinds of programs. First, the MSC program analyzes an input image and creates 16 intermediate filesets (RedImg_1,...,RedImg_{16}) and one Calibration Images fileset. In addition to the telescope image, MSC requires a set of Reference Images and command-line parameters (params) as input. Each RedImg_n fileset from MSC is then fed into a separate PHOT_i (with $i \in 1, ..., 16$) program, resulting in fan-out in the abstract workflow. Each PHOT_i program also uses the Calibration Images and outputs another fileset (SkyObjects_i) storing objects that may be ingested into a database. In terms of computational requirements, each MSC process requires an average of 569 seconds to execute while each PHOT_i process requires an average of 1146 seconds to execute. The average time to exe-

[1]Recall from the previous Chapter that the modeler is responsible for defining workflows.

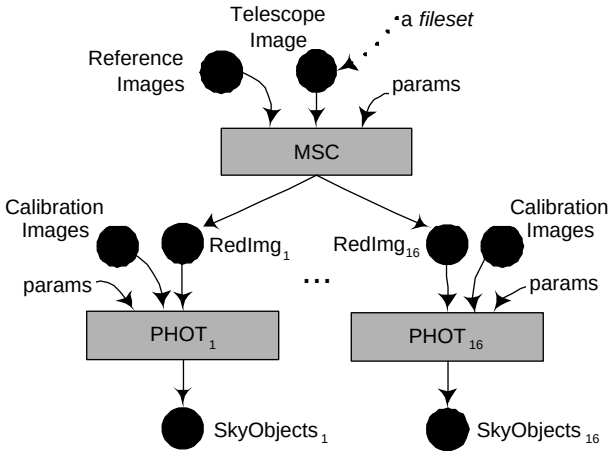


Figure 4.2. Coarse-grained SuperMACHO Image Processing Workflow

cute one workflow process is 1112 seconds (a weighted average over the 16 PHOT processes and 1 MSC process).

Figure 4.3 depicts the fine-grained workflow, which is a refinement of the MSC program in the coarse-grained workflow. As such, the workflow transforms a telescope image into 16 RedImg$_n$ filesets, each suitable as an input into the PHOT program. For experimental efficacy, the workflow does not include the PHOT program. The workflow's topology is a linear pipeline with 3 programs, MSC$_1$, MSC$_2$ and MSC$_3$. Each program takes in a set of reference images, a set of parameters, and a "primary input." The primary input is a fileset either externally provided by the user or created by an upstream workflow program. MSC$_1$, MSC$_2$ and MSC$_3$ require 369, 164 and 36 seconds to execute on a single image, respectively (the sum of the three equals the runtime of MSC, cumulatively).

In contrast, relative to the coarse-grained workflow, the fine-grained workflow has two characteristics that make it more challenging for GridDB to execute.

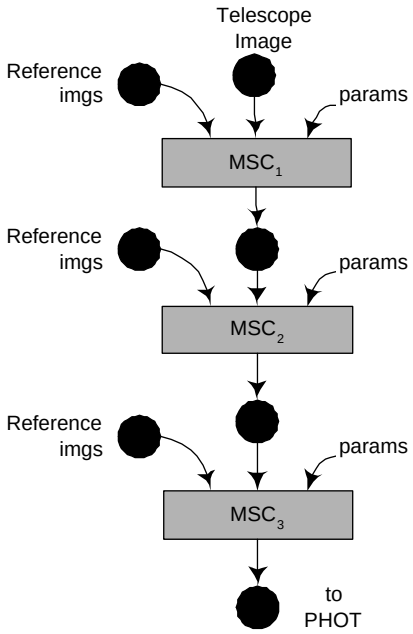


Figure 4.3. Fine-grained workflow. A refinement of MSC.

First, the average runtime of a process within this workflow is only 190 seconds, about 6 times shorter than the average runtime of a process in the coarse-grained workflow. The shorter runtimes mean that a given set of compute-nodes can execute processes at a higher throughput. Second, because workflow processing involves the alternating use of GridDB and compute-node resources, faster execution by compute-nodes results in higher demands on GridDB, increasing the likelihood that GridDB will become a bottleneck.

4.2 Coarse-Grained Model Execution

Having described two sample workflow models within the SuperMACHO application, we continue now with our experiences executing the coarse-grained workflow. This section starts by illustrating a fundamental problem to providing data provenance and memoization: a program's routine activity of writing output files may change the directory structures that hold its input files. Changes to these structures disrupt GridDB's ability to provide data provenance and memoization. A simple scheme to circumvent this problem is to use a *deep linking* file transmission scheme, which *preserves* file data in the face of a modifying workflow program. Unfortunately, the deep linking scheme incurs high performance penalties due to a strong reliance on file system metadata operations. Hence, we propose a second transmission scheme, *selective deep linking*, which alleviates the key performance bottleneck by reducing file system traffic. Experiments show that selective deep linking improves performance significantly. The key result of the section, that file system traffic management is key to providing scalable data preservation, drives the work of Chapter 5.

4.2.1 Fileset Preservation

We start by motivating the need for preservation and a deep linking file transmission scheme. In the GridDB implementation described in Chapter 3, file transmission was performed through *root-linking*, which leaves file-based input data vulnerable to modification. We illustrate the mechanism for these modifications through an example (shown in Figure 4.4(a)).

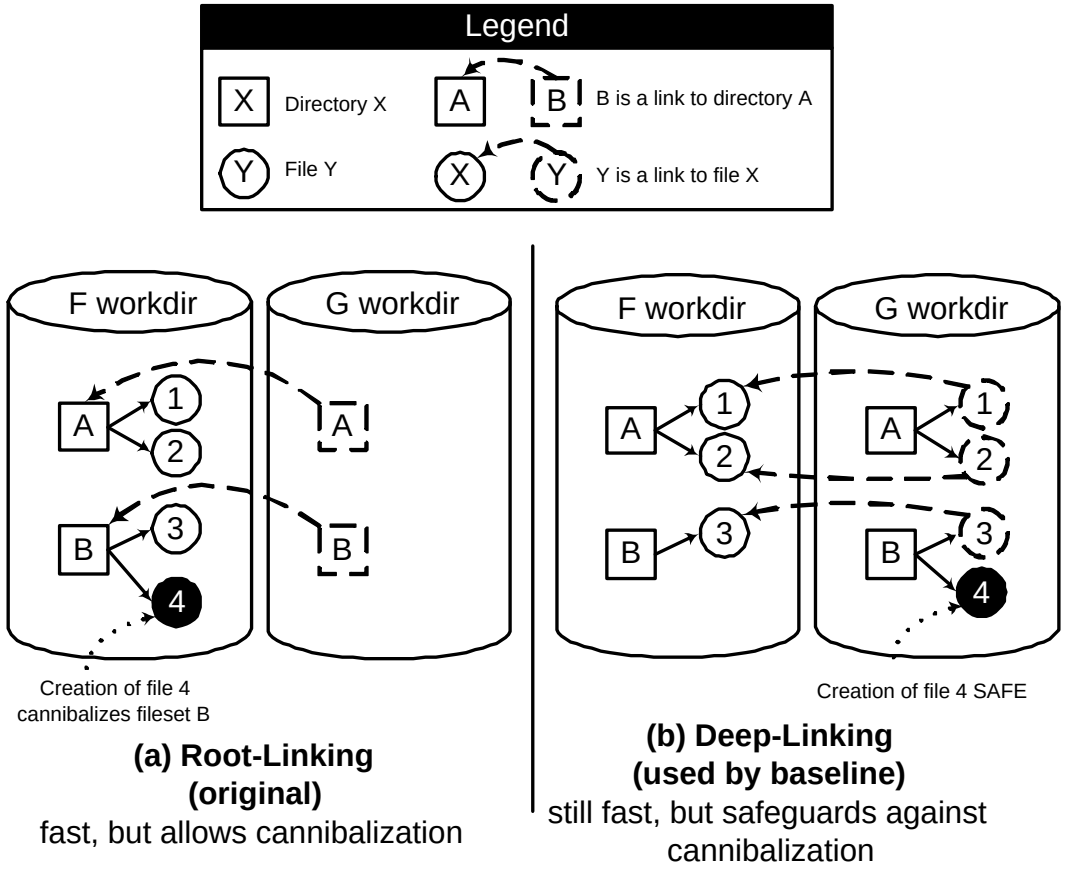


Figure 4.4. Linking Schemes in the previous and current implementation.

Consider the file system interaction between two workflow processes `F` and `G`, where `G` uses file data that is generated by `F`. After `F` executes, `G` must be given access to some of `F`'s output files. After `F`'s execution, its working directory contains 2 directory trees, `A` and `B`. The roots of these trees are represented as white boxes in the left cylinder of Figure 4.4(a). Directory `A` contains two files, `1` and `2` and directory `B` contains one file, `3`. The 3 files are represented by clear circles in the left cylinder. After the execution of `F` and before the execution of `G`, the two filesets, embodied by the directory trees rooted at `A` and `B`, are

transmitted into the working directory of `G`. Using the root-linking transmission scheme, each of the filesets are transmitted through a single link to the root of the fileset. These links are represented by the dotted boxes in the cylinder on the right, which feature arrows connecting the links to their targets. Using these links, `G` may read all files under `A` and `B`.

A problem arises, however, if `G` needs to add a file to a path that is "covered by" one of the input filesets; for example, a file with the path `B/4`. By adding this file, `G` will change the composition of the fileset rooted at `B`. Before execution, it contains only the file `3`, but after execution, it will contain the files `3` and `4`. The file `4` is represented with a black circle. Unlike ClustFind and HepEx (the examples of the previous chapter), the workflow programs of SuperMACHO modify their input filesets in this manner.

Cannibalizing an input fileset by adding files to it causes two problems. First, as an input to process `G`, `B` is part of its provenance. By changing `B`, the provenance of `G` has lost its fidelity. Second, a change in `B` undermines GridDB's ability to reuse it during memoization. If another process `H` reads the fileset under `B` before `G`'s execution, it will be different than if `H` reads it *after* `G`'s execution. Therefore, such modifications would render `H`'s execution non-deterministic.

To simultaneously preserve file data and accommodate this sort of modifying behavior with workflow programs, one must make copies of modified objects. In our example, if a copy of fileset `B` is made prior to `G`'s execution, `F`'s copy of `B` remains unmodified after `G`'s execution. The simplest scheme for providing this *data preservation* is just to copy the directory structures, in their entirety, from the working directory of `F` to that of `G`. While such a scheme achieves correctness,

the performance and storage overheads would be too high due to the large size of scientific filesets. For example, the filesets passed from MSC to PHOT can contain gigabytes of data and intermediate filesets of this size are not uncommon amongst scientific applications. Additionally, a typical job request may execute a workflow hundreds of times, amplifying these overheads.

An alternative scheme called *deep linking* can avoid the overheads of copying while still preserving input filesets. Deep linking transmits directories through copies, but transmits files through links. As a result, file additions can be carried out without modifying the contents of input filesets because they are made to a separate copy of a directory. Files, while shared, are marked as read-only (using `chmod`) to protect them against accidental modification by workflow programs. Overheads are vastly reduced since files, which account for most of the bytes in a fileset, are shared between working directories.

Figure 4.4(b) shows the application of deep linking to the previous example. Directories `A` and `B` are copied from the working directory of `F` to `G` while files `1`, `2` and `3` are transmitted through links and then marked as read-only. Now, as process `G` adds the file `4` to directory `B`, the original copy of directory `B` (in `F`'s working directory) remains unaltered. In the next section, we measure the scalability of a GridDB server that executes workflows using a deep linking transmission scheme.

4.2.2 Deep Linking Experiments

To understand the scalability of GridDB, we conduct several trial runs on MCR. In each run, GridDB (which runs on a single node) executes the coarse-

grained workflow on a set of cluster nodes ranging in size from 4 to 256. This section shows representative results from one of the 128 node runs. In these experiments, we observe that GridDB, using deep linking, could be a bottleneck to execution. In particular, the cost of deep linking is prohibitive and impedes GridDB's ability to dispatch processes onto compute nodes quickly. As a result, a scientist's ability to speed up a computation by adding computation nodes to her cluster is limited by GridDB. A profile of the run in this section suggests avenues for improvement, which we pursue in later sections.

Setup

Because MCR is a shared facility, dedicated use of its nodes must be acquired through a reservation system. To emulate a cluster containing n compute-nodes, a block of $n + 1$ nodes is reserved. Within such a block, one node is designated as a GridDB node and the remaining n as compute nodes. After securing a reservation, a request to process $numImages$ through the coarse-grained workflow is submitted. Each image is processed independently and in parallel with the other images. Each process in each workflow goes through fold and unfold stages, which are handled by GridDB, and also an execution stage, which is handled by a compute-node. To exploit concurrency, GridDB processes as many as 16 folds and unfolds at a time. Because reservations are short (10 hours) and difficult to come by, and multiple experiments were conducted in each allocation to fully-utilize them, expirations sometimes occurred before runs completed. Even so, these truncated experiments offered enough concrete data to help characterize system performance.

Throughout each run, the number of compute-nodes utilized is monitored. Compute-node utilization is used as a gauge of GridDB's performance. As described in the previous chapter, each process within the workflow must be processed by both GridDB (fold and unfold) and by a cluster node (exec). High compute-node utilization is an indication that GridDB is performing well since it is performing its processing fast enough to keep the compute nodes busy. Alternatively, if compute nodes are scarcely utilized, GridDB is causing a bottleneck in workflow execution.

File system data is handled through a Panasus storage appliance with 20 shelves, 3 directors and 8 storage blades. Dedicated access to the file system is not possible since it is shared by all of MCR's nodes and cannot be reserved. To minimize contention for storage resources, all of our experiments were carried out between 6pm and 6am, when these resources were the least loaded. Because the file system turns out to be the main bottleneck, our work in the next chapter focuses on taking measurements in an environment where network and file system resources are isolated from external interference.

Results

We continue by describing our sample run of GridDB executing the coarse-grained workflow using deep linking. As previously mentioned, this run is executed on a cluster with 128 compute-nodes. The run's job request consists of 256 images to be processed through the coarse-grained workflow. Recall that the coarse-grained workflow contains MSC and PHOT processes and that each MSC process requires an average of 569 seconds to execute while each PHOT process

requires an average of 1146 seconds. If the workload is carried out to completion, GridDB would execute a total 256 MSC processes and 4096 (16∗256) PHOT processes. In this run, GridDB in fact executes all 256 MSC processes but because the cluster allocation expires before completion, it only executes 1200 PHOT processes. Despite early termination, the results allow us to characterize system performance while running both MSC and PHOT processes.

Compute-node utilization for this run is shown in Figure 4.5. The GridDB scheduler runs all MSC processes before running any PHOT processes. As a result, the utilization profile can be split into two separate stages based on what type of process is executing at the time. During stage 1, between 0 and 5713 seconds, the compute nodes are mostly executing MSC processes. During stage 2, between 5714 and 14554 seconds, the compute nodes are mostly executing PHOT processes. During stage 2, an average of 50.2 (39%) nodes are utilized whereas during stage 1, only an average of 24.2 (18.9%) nodes are utilized. Over the entire run, an average of only 40 nodes are utilized at any given time while a maximum of 66 nodes are utilized.

These utilization rates are far from ideal. They suggest that if a scientist executes this workflow through a single-node implementation of GridDB, it would not be beneficial for her to employ a cluster of more 66 nodes because GridDB cannot do its work fast enough to keep more than 66 nodes busy.

Through profiling this and other executions, the bottleneck was shown to be GridDB's dependence on the underlying file system. This was surprising, considering that we only used industrial grade storage appliances (the only ones available on MCR). Like most file systems, unfortunately, industrial appliances

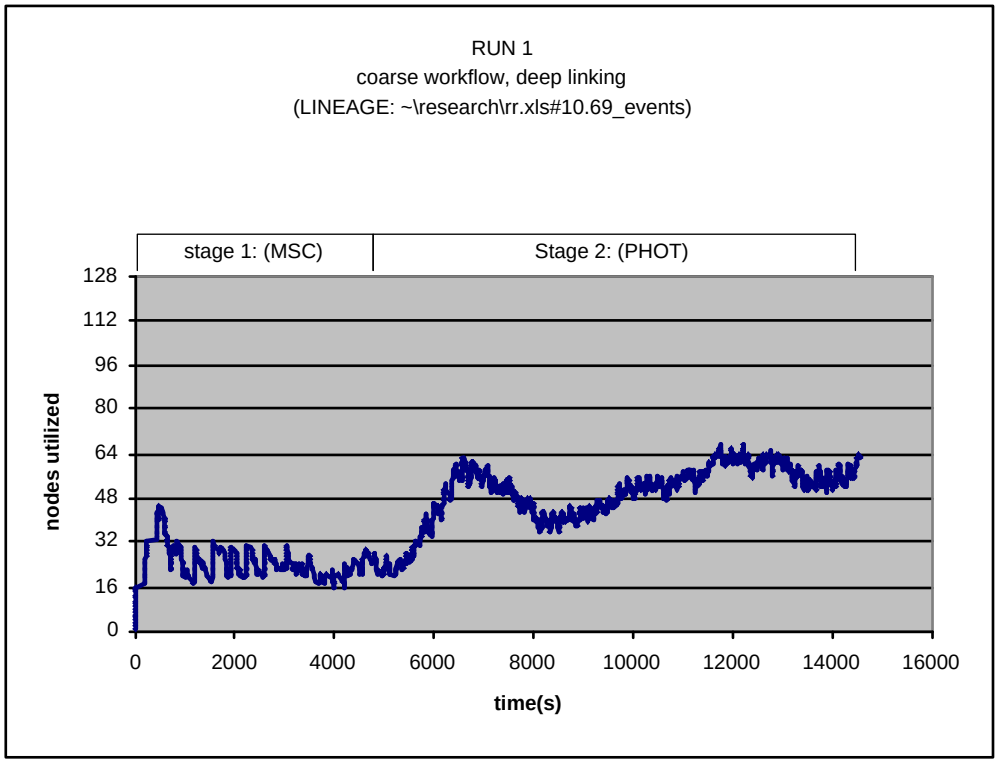


Figure 4.5. Run 1: Coarse-Grained Workflow with Deep Linking (128 compute nodes)

are optimized for streaming IO performance. Little attention is paid to increasing metadata operation throughput, and metadata operation throughput is key to performing deep links quickly.

Deep links require a large number of operations such as `ln`, `stat`, `readdir`, `mkdir` and `chmod`. Profiling reveals that 99% of the time spent in the fold and unfold stages are spent in these calls. The volume of system calls is large, 2715 for each MSC process and 1484 for each PHOT process. Given that as many as 16 processes were being handled concurrently, the storage appliances could not perform adequately. The difference in call counts also explains why utilization

is lower in stage 1 than in stage 2. GridDB handles MSC processes in a slower manner because of the larger number of metadata operations.

While this experiment shows results for one particular file system, experiments on MCR's other file systems (including a Lustre file system and a NetApp 960) yield similar results. The next chapter reports on additional experiments run in an isolated environment. The results of those experiments further corroborate the file system as the primary bottleneck.

Given that GridDB's poor performance can be traced to its strong dependence on file system metadata calls, we turn our attention to reducing this dependence. In the next section, we introduce an alternative transmission protocol that manages to reduce metadata operation usage while still preserving filesets.

4.2.3 Efficient File Preservation

Our alternative approach, called Selective Deep Linking, is a more efficient transmission scheme that is based on the simple observation that a fileset that never has files added to it does not need to be deep linked. Rather, it is sufficient to transmit these filesets with a single link to the root. Using this optimization, metadata operations may be reduced substantially. A read-only directory tree does not need to be traversed and its files do not need to be linked individually or marked as read-only. To understand which filesets have files added to them and which do not, GridDB may accept hints from the modeler. This chapter confines itself to illustrating the basic concept of selective deep linking, deferring a full specification of the modeler interface to the next chapter.

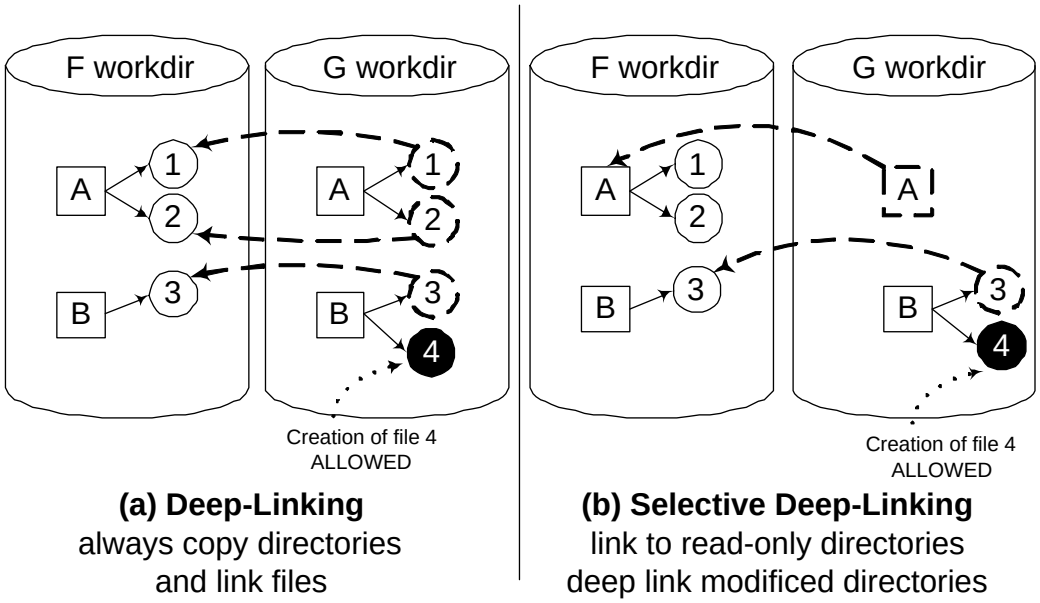


Figure 4.6. Examples of (a) Deep Linking and (b) Selective Deep Linking

Figure 4.6 juxtaposes the use of selective deep linking in our running example (b) against deep linking (a). During workflow definition, a modeler states in the function definition of program `G` that its input fileset, represented by the directory tree under `A`, will not be modified while the fileset under directory `B` *will* have files added to it. As such, the fileset under `A` is transmitted with a single link to the root while the fileset under `B` is transmitted through a deep link. During the execution of `G`, file `B/4` can still be created without cannibalizing any inputs, but the number of file system object creations is reduced from 3 links and 2 directory creations to 2 links and 1 directory creation. Likewise, the number of metadata operations used to perform the transmission is also reduced.

Selective deep linking achieves substantial metadata operation reductions in the SuperMACHO coarse-grained workflow. File system operations for the two

Stage	Deep Linking	Selective Deep Linking
MSC	2715	173
PHOT	1484	142

Table 4.1. Comparison of Algorithms with respect to linking requirements.

workflow programs are shown in Table 4.1. Selective deep linking reduces file system operations by an order of magnitude for both workflow programs. File system operations counts are reduced from 2715 to 173 in MSC and from 1484 to 142 in PHOT.

4.2.4 Selective Deep Linking Experiments

We continue in this section by showing a sample run with GridDB executing the coarse-grained workflow using selective deep linking. This run illustrates that with selective deep linking, GridDB spares much of the file system overhead incurred by deep linking and is able to support at least 4 times as many compute-nodes as it can with ordinary deep linking.

The run presented here was performed on a cluster of 256 compute-nodes. The workload request consists of 512 images to be processed through the coarse-grained workflow. Note that this is twice as many nodes and twice as many images as in the previous run. The workload request consists of 512 MSC processes and 8192 PHOT processes. In this case, the run was able to complete all 512 MSC and 4396 PHOT processes before termination.

The compute-node utilization for this run is shown in Figure 4.7. Recall that Stage 1 consists of MSC executions while Stage 2 consists of PHOT executions.

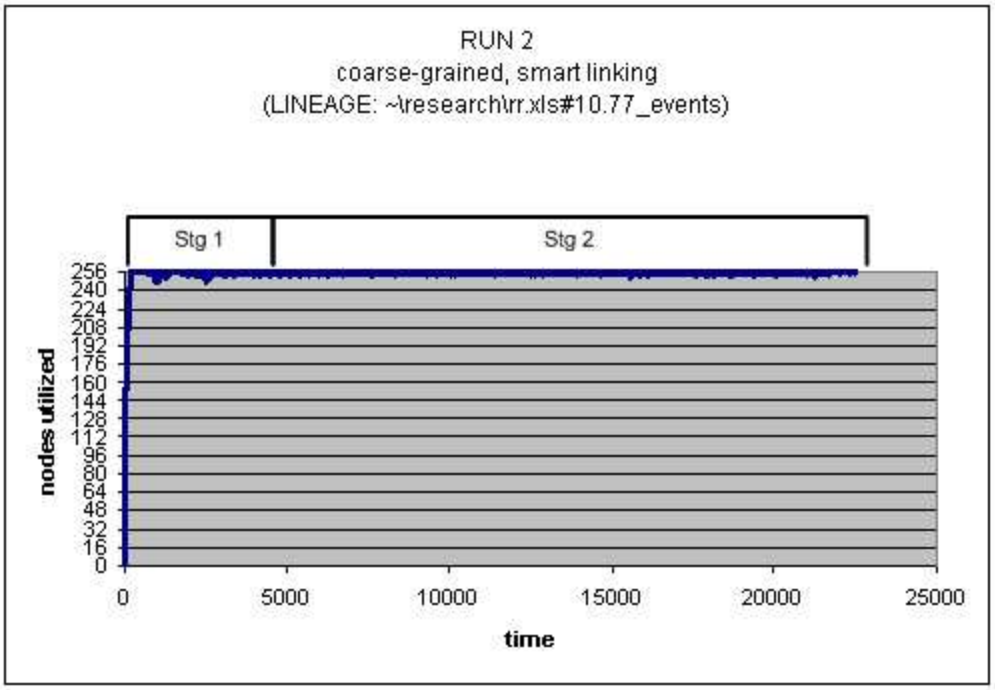


Figure 4.7. Run 2: Coarse-Grained Workflow with Selective Deep Linking (256 nodes)

During stage 1, between 0 and 2304 seconds, the compute nodes are mostly executing MSC processes. During stage 2, between 2305 and 20510 seconds, the compute nodes are mostly executing PHOT processes. Node utilization is near-perfect in both stages. During stage 1, an average of 252.4 nodes (98.5%) are utilized while in stage 2, an average of 254.2 nodes (99.3%) are utilized.

These utilization rates are near-optimal. Essentially, the experiments indicate that a single-node implementation of GridDB, when operating with selective linking, does not become a bottleneck with a cluster size of 256 nodes for this application. Since compute nodes are nearly always utilized in this case, it is likely that GridDB also performs well with even larger cluster sizes. We were

unable to test this hypothesis, however, because larger blocks of nodes were too difficult to reserve.

4.2.5 Summary

In this section, we made several observations while executing a realistic science application with the coarse-grained model on large clusters. First, we identified the need for fileset preservation in order to support data provenance and memoization. Second, we observed that a naive deep linking transmission scheme caused a single-node implementation of GridDB to become a scalability bottleneck in large cluster settings. The bottleneck was traced to an over-zealous use of file system metadata operations. Finally, we illustrated that a selective deep linking transmission scheme, which uses metadata operations in a more judicious manner, is able to substantially improve GridDB's scalability. These results indicate that the key to providing scalable fileset preservation is careful use of file system metadata operations. In the next chapter, we act further on this observation by proposing a comprehensive framework that allows modelers to specify hints.

4.3 Fine-Grained Model Execution

In the previous section, we showed that by using selective deep linking on a workload based on the coarse-grained model, we are able to achieve near-perfect utilization on a 256 node cluster. In this section, we "stress test" GridDB by

imposing a more challenging workload based on the fine-grained model. The stress test shows that even with the benefits of selective deep linking, GridDB may still be a bottleneck. Again, scalability issues are traced back to insufficient file system metadata performance. The work in this section motivates our efforts to parallelize file system metadata in the next chapter. We begin the discussion by highlighting differences between the fine-grained and coarse-grained workflow. We then present experimental results from a run of the fine-grained workflow on a large cluster.

Figure 4.8 summarizes the characteristics of the fine-grained model. The fine-grained model is more challenging to GridDB when compared to the coarse-grained model. The increased challenge presents itself in two ways. First, the component processes run for a shorter amount of time (369, 164 and 36 seconds), an average of 190 seconds. This average is roughly 6 times less than the average of 1110 seconds using the coarse-grained model. These shorter runtimes translate into more frequent demands upon GridDB and increase its likelihood of being a bottleneck. Second, the fine-grained model requires roughly three times as many file system operations as the coarse-grained model per process during selective deep linking transmission. The fine-grained modeled requires an average of 450 metadata operations relative to an average of 151 in the coarse-grained case.

4.3.1 Fine-Grained Model Experiment

As in previous sections, we present one representative experiment to illustrate GridDB's performance on the fine-grained workflow. The run presented executes

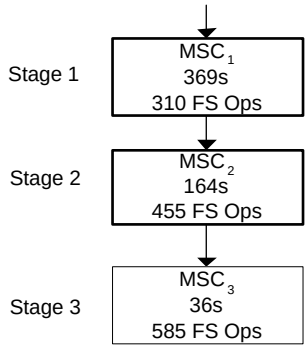


Figure 4.8. Fine-grained model characteristics.

on a cluster with 256 compute-nodes. The workload request consists of 512 images to be processed through the fine-grained workflow. Complete execution consists of 512 processes on each of the three programs, MSC_1, MSC_2, and MSC_3. Unlike the previous runs, all 512 processes for each program are completed.

Compute-node utilization for this run is shown in Figure 4.9. Similar to previous runs, this run can be split into stages; in this case, three (i.e., one for each program). As shown in the Figure, node utilization rates depend heavily on the processes being executed. Generally, the longer the time to execute the program of a particular stage, the more nodes are utilized. In stage 1, an average of 226 nodes are utilized because of the long 369 second runtime of MSC_1 processes. In contrast, stage 3 node utilization averages just 22 nodes due to the short 36 second runtime of MSC_3 processes. In stage 2, where processes run for a moderate 164 seconds, we observe an intermediate utilization rate of 128 nodes. These results again show that GridDB's scalability could be a concern on more challenging, but realistic, workloads.

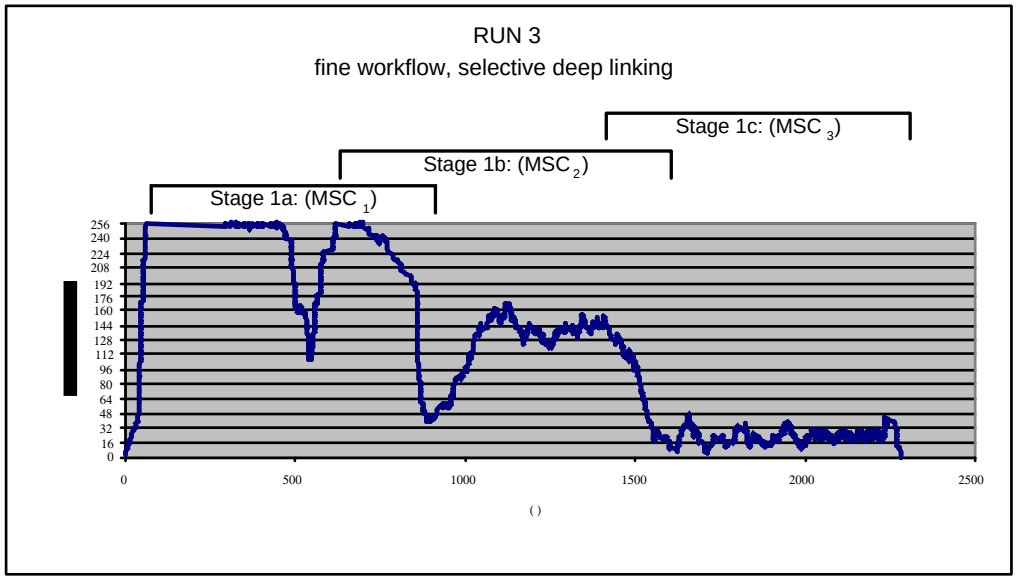


Figure 4.9. Run 3: Fine-Grained Model with Selective Deep Linking

As with previous runs, we profile GridDB to determine where most of its time is spent. Like in those cases, the majority of execution time is spent performing metadata system calls (85-99%, depending on the stage). The results in this section demonstrate two points. First, GridDB's performance must be managed carefully if one is to fully exploit large clusters. Second, even with selective deep linking, file system metadata performance continues to be a bottleneck. We address these issues in the next section by parallelizing metadata requests across multiple file systems to increase metadata throughput and ultimately, GridDB's throughput.

4.3.2 Behaviors of Other Scientific Workflows

In light of the observation that file creation behavior would complicate preservation and therefore impact the scalability of GridDB, we profiled several other workflows to affirm the finding. These included another workflow from Super-MACHO, and two from proteomics (94) and biochemistry (95) projects. In these applications, we observed the same pattern: the workflows add files to a common workspace, and therefore destroy prior images of the workspace. We also discovered, however, that programs sometimes also modify the *files* that they use as input. For example, sometimes a program changes a file in place or overwrites an unrelated file that happens to use the same name as an existing one. As a result, a solution to providing preservation must copy *files* that face modification in addition to directories.

4.4 Chapter Summary

In this chapter, I described experimental results from an evaluation of GridDB on a testbed at Lawrence Livermore National Labs. The results were based on runs of an astrophysics workflow over telescopic data using a large cluster. During these runs, I identified a scalability bottleneck in GridDB that arises while trying to preserve file data in the face of workflow program modifications. File data preservation is essential to support data provenance and memoization. The scalability bottleneck was traced back to limitations in the metadata capabilities of underlying file systems. In the next chapter, I build upon these findings by proposing a robust framework for providing preservation in a scalable manner.

Chapter 5

Data-Preservation

In the previous chapter, I explained why *data-preservation* is required to support both Data Provenance and Memoization and how program behavior, in the form of file creations and modifications, can undermine the preservation of data. In order to compensate, file system objects that face modification must be selectively copied. Also, for more challenging workloads, even a system that selectively copies modified objects may be insufficiently scalable. This chapter first presents a framework for a DSWMS (such as GridDB) to receive and process hints about program behavior from the modeler. Second, it examines various methods of parallelizing data-preservation to boost scalability in the face of challenging workloads. Finally, it includes a comprehensive performance study conducted on a wide range of workloads and file systems.

5.1 Introduction

A DSWMS supporting data-preservation *preserves*, or makes recoverable at a later time, data written and read during workflow execution. Data-preservation, is required for supporting data provenance and memoization. The main challenge in supporting data-preservation is that workflow programs often alter the files and directories that they use as input. In some cases, a workflow program may explicitly modify files. More often, a program will add files to a directory structure, implicitly modifying the directory's composition. In the face of such program behavior, a DSWMS must make a snapshot of files or directories that face modification. A second requirement is that the DSWMS must do so with good performance, and scale well as the number of workflows it is executing grows.

To address these issues, we present a solution that is based on a combination of user-provided hints to reduce the frequency of copying and parallelization to increase copying throughput. The solution can be deployed on top of current file systems and is able to scale almost arbitrarily. We also evaluate our approach on real use-cases from astrophysics and run experiments on a cluster using four different file systems.

As discussed in the previous chapter, a key requirement for preservation of file-based data is the copying of file system objects that face modification from a process that reads or has access to them. Therefore, before a program runs, a *snapshot* of the disk space that it has access to must be taken. The naive approach for a DSWMS to perform snapshotting is to make a copy of *all* of a program's

inputs prior to execution. This solution, however, is generally too expensive for two reasons. First, many scientific programs are data-intensive, reading or writing many files, some of which can be very large (96). Second, scientists often work in a pattern that generates "bursty" workloads, where a large volume of work is requested over a short period of time. For instance, "parameter sweeps" apply an analysis many times with different input conditions. As another example, large datasets to be processed by workflows are often submitted in a batch. In such scenarios, this "copy all" approach may overwhelm a DSWMS and the underlying file system, slowing down workflow execution, and ultimately bogging down the scientific discovery process.

One sensible approach to handling this problem is to employ a *versioning*, or *copy-on-write*, file system (97; 98; 99; 100) for automatically detecting modification events, selectively copying file system objects on-demand and eliminating gratuitous copying. Unfortunately, systems currently in use at scientific-computing facilities typically do not support this capability. Building a system upon it goes against our core design philosophy of making DSWMS's easy to adopt. Versioning file systems suitable for scientific computation are not well supported by vendors and it is unlikely that they will be made available to scientific users in the near-term. Even if vendors began to produce them, underlying infrastructure such as file systems are typically slow to evolve. Therefore, in this chapter, we focus on providing users with a solution that is compatible with current infrastructure, which consists of *eager copy* file systems.

Our approach involves implementing data-preservation in a way that portable, but also efficient. It is two-pronged: first, we design a copy-reducing scheme that

employs user-hints to determine which files or directories face modification, and therefore which files or directories require copying. We show that on an astrophysics workflow, this mechanism can reduce preservation-induced overheads by more than two orders of magnitude.

We also show, however, that the hinting mechanism alone is insufficient to overcome the challenges of large jobs consisting of many simultaneous workflow executions. To handle such jobs, we further apply several basic parallelization techniques to the DSWMS. In conjunction with the hinting mechanism, our parallel DSWMS is able to achieve near-perfect scale-up, allowing overheads to be reduced by adding more hardware. Together, the two techniques provide an effective framework for supporting data-preservation in a reasonably efficient manner.

Our conclusions are supported by real use-cases from astrophysics and experiments on a variety of file systems. We use GridDB as a platform for our work. While our evaluation is based on one particular system, we believe that our solutions are relevant for *any* DSWMS that supports data-preservation.

While data provenance has recently gained momentum as an important feature in scientific workflow (89; 86; 101) and database (102; 103; 104) systems, we are unaware of any work that has addressed the performance implications of preserving file-based data. There are, however, a number of sensible alternatives to performing data-preservation.

One can envision alternative approaches using either (a) versioning file systems (97; 98; 99; 100) (as mentioned earlier), (b) custom compilation, (c) I/O library substitution (105) or (d) a version control system (106; 107). These

alternatives are disadvantageous because they are rarely supported by current infrastructure (technique a), not generally applicable (techniques b and c), too complicated (technique c) or inefficient (technique d). In contrast, our technique of combining a hinting mechanism with parallelism provides a practical, efficient and portable solution for supporting data-preservation on currently available infrastructure.

The remainder of this chapter is organized as follows: Section 5.2 reviews relevant parts of our system and workflow model. Section 5.3 discusses the interface for our hinting mechanism, as well as algorithms for processing user hints. Section 5.4 contains experimental results demonstrating the effectiveness of hinting. Section 5.5 describes our approach to parallelization. The chapter is concluded in Section 5.6.

5.2 System Model

We start by reviewing background that is relevant to the work of this chapter. This includes the software architecture, workflow model and DSWMS execution flow.

5.2.1 Software Architecture

Figure 5.1 shows the DSWMS software components relevant to workflow execution and how these software components are mapped onto hardware. Science programs are executed in parallel on a large number of cluster nodes (represented

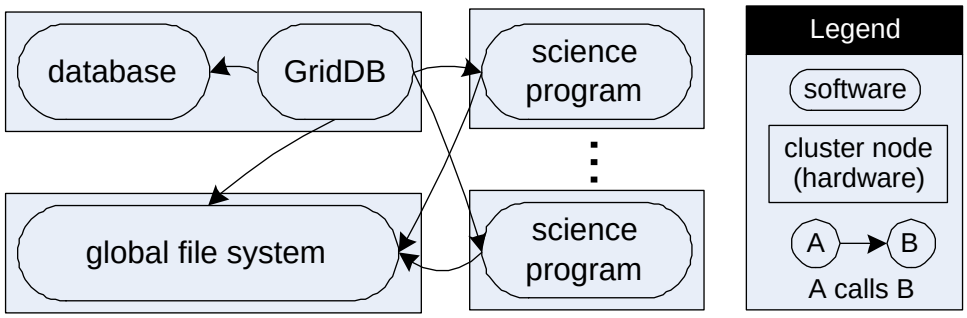


Figure 5.1. Software components involved in cluster-based workflow execution, as well as their mapping onto nodes.

as rectangles in the figure). These science programs read and write data from and to a global file system. The programs are launched by GridDB. GridDB makes use of a database system to store and retrieve execution parameters, provenance information and metadata about the science programs it is running. GridDB also needs to interact directly with the file system in order to make file data available to and ensure that it is not changed by science programs.

5.2.2 Workflows

The workflows that we focus on fit a well-defined, but generic, model. Workflows are partially ordered sets of *programs*, each of which transforms input data into output data. Programs are written in a variety of programming models, including shell/perl/python scripts and compiled languages. Program input is a combination of files pre-staged in a program's *workspace* directory and command-line parameters submitted during invocation. Program output is the set of files in a program's workspace after execution. A *process* is an invocation of a program.

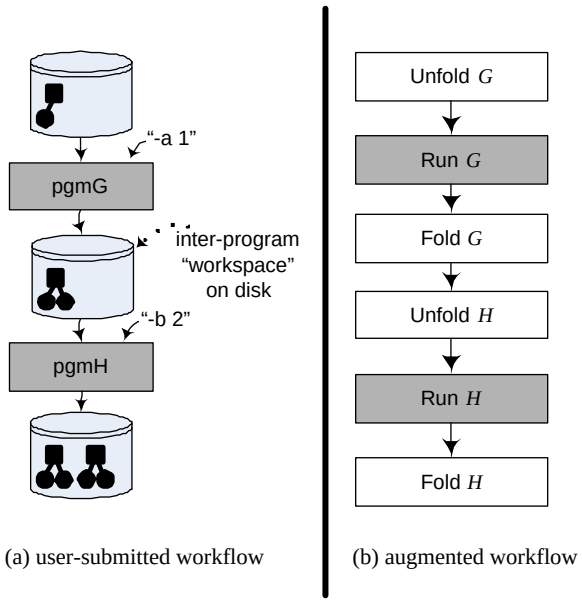


Figure 5.2. (a) A two-process workflow and (b) it's translation to a GridDB augmented workflow during execution.

It is defined by a program and concrete instances of each input. It creates a concrete instance of outputs. Figure 5.2(a) shows an example two-process workflow.

Intuitively, a set of workflow programs can be seen as a sequence of actors that modify the state of a disk workspace. Each program in the workflow starts with the workspace left by its predecessors and then adds to and modifies and deletes files from it. The initial state of the workspace is the process' input, and acts as part of the provenance of the process' output, the final state of the workspace. If the initial workspace is modified, it loses its viability as provenance. Likewise, it loses its ability to be reused as input to a future process, as required by smart recomputation. As such, a DSWMS must find a way to *preserve* workspaces in the face of process-induced changes.

5.2.3 Data-Preserving Transmission and Cataloging

Data-preserving transmission is the act of making input data available to a process while simultaneously making sure it is not modified. It applies to *both* command-line parameter data and file system data. Fundamentally, a DSWMS can transmit data either *by reference*, by creating a pointer to the original, or *by value*, by creating a copy of it and making it available. While transmitting by reference is the faster and cheaper method, transmitting by value preserves data in the face of change.

In GridDB, because parameter data is stored in the database, it must be serialized into strings that are passed on the command-line. Serialization implicitly creates a copy independent of the originals. Though this is a form of extraneous copying (it is unclear that the program will modify the parameters) command-line parameters are typically small. For instance, a flexibly parameterized program with 1000 double precision parameters requires as little as 8 KB of copying; a figure that is dwarfed by the size of file data, which is commonly in the gigabyte range.

On the other hand, because file-based data is large, the naive approach of indiscriminate copying (i.e. the "copy all" approach) will likely have a negative impact on performance. More likely, a system needs a way of determining which parts of a workspace must be copied, and which parts can be transmitted by reference. In Section 5.3, we describe a mechanism for making this decision based on user-provided hints.

In addition to data-preserving transmission, a second activity that GridDB

must carry out to make preserved data retrievable is the *cataloging* of provenance information. GridDB uses a relational database to keep mappings of processes to their programs, inputs and outputs. These mappings enable the answering of provenance queries such as "what process created data item X?" and smart recomputation queries such as "was program P run with input Y before?"

5.2.4 Workflow Execution Path

Having described data-preserving transmission and cataloging, we now explain how they fit within GridDB's broader workflow execution framework. Data-preserving transmission and cataloging are carried out in stages before and after each process. In GridDB, the "before" stage is called *unfold* and the "after" stage is called *fold*. Workflow execution in GridDB can be depicted by an "augmented" workflow that includes the overhead stages along with program execution stages. Figure 5.2(b) shows an augmented workflow for the user-submitted workflow of Figure 5.2(a). As suggested by the figure, the overhead stages extend the critical path of workflow execution. As such, care must be taken to minimize their performance impact.

The unfold stage carries out two activities. First, input data is pulled from the database and command-line strings are constructed. Second, input files are transmitted into the process' workspace. After unfold, execution of the program occurs on a cluster node. Upon completion, the fold stage catalogs files created by a process, which are sitting in its workspace, and marks them read-only. The read-only marking ensures that the file data in the workspace is preserved. A

future process attempting to modify the data will be halted. As such, processes must have personal copies of objects they want to modify.

Both the unfold and fold stages perform operations against a database and a file system. These interactions involve the storage and retrieval of disk-based information, and are likely bottlenecks to program execution. We extended the performance studies of the previous chapter by repeating our experiments on a small cluster on which we had exclusive access (described in Section 5.4 and 5.5). These experiments confirm that if one uses naive "copy all" transmission, the transmission of file data (during the unfold stage) consumes the lion's share of execution time[1]. As such, the next section outlines a mechanism to carry out file transmission in a more efficient manner.

5.3 Efficient File Transmission Using Hints

In this section, we provide an alternative to naive transmission. Our mechanism, based on user-hints, exploits the fact that in many scientific workflows, programs tend to modify only a portion of their initial workspace. Those parts of the workspace that are unchanged can be shared between a producing program and a consuming program through the use of linking. In this section, we first describe a interface for specifying hints about which parts of a workspace change. Afterwards, we describe how GridDB processes these hints to reduce copying.

[1]In our tests, file transmission accounted for more than 99% of total time spent in the fold and unfold stages.

5.3.1 Hinting Interface

Hinting is performed by a *modeler*. As described in Chapter 3, a modeler is a scientist who conveys the behavior of the program to GridDB through a schema. In this chapter, we extend the definition of a schema, by incorporating a *file access model* (or just *model*) to it. A model is a description of how files and directories within a workspace are accessed by a program. A model consists of a set of *hints*. A hint H is a pair (P, M) where P is a path that identifies a directory or file relative to the workspace root and M is an *access mode* describing how the consuming program uses objects (directories or files) under the tree.

We use a graph representation for directory trees, where files and directories are nodes, and directory membership is represented by a directed edge. In such a representation, a hint H is said to *cover,* or apply to, the node n represented by path P. We also say that n is *annotated* with H. In cases where a node is not annotated, it is covered by the hint of the nearest annotated ancestor. We call the set of nodes covered by a hint its *region*.

Our hinting interface provides four access modes with varying degrees of permissiveness. As we discuss in the next section, the less permissive modes involve less copying and thus are less expensive to support. The modes, in decreasing order of permissiveness, are:

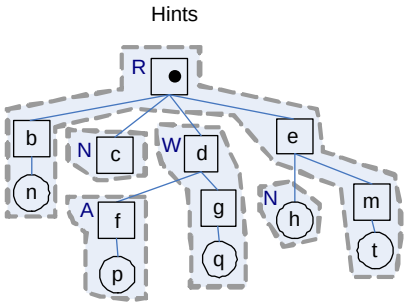
- Write (W): a program will modify or delete files and directories under write regions.
- Add-only (A): a program will add files to directories, but may not modify

existing files in add-only regions. Essentially, the program is allowed modify directories, since adding a file to a directory modifies it.

- Read-only (R): a program will only read files and directories in a read-only region.
- No-Read(N): a program will not read files or directories in a no-read region.

Hints empower GridDB to perform data-preserving transmission with fewer copies. As a simple example, suppose that a workspace consists of two directory trees that are used by a program. One directory tree contains files that are modified by the program while the other directory tree only contains files that are read. While the first tree must be copied in order for the original to be preserved, the second tree can be transmitted through a link without endangering its preservation. By submitting hints that indicate W and R access modes for the first and second trees, we can avoid copying the second directory.

Figure 5.3(a) shows an example workspace annotated with hints of each type. We use this workspace as a running example of hint specification and processing. Boxes denote directories, circles denote files and capital letters next to nodes denote hints. The workspace is partitioned into five regions, each governed by a particular access mode. A dotted line surrounds each region. As one example, the R hint presides over the region containing $\bullet$, b, n, e, m and t. The access modes dictate what a consuming program can do in each region. For example, the program may add files to directory f, but cannot modify file p, because these nodes are contained in an add-only region. As a second example, the program cannot read or write to directory c, because it is in a no-read region.



(a)

```
Program P{
 hints = {
  . : R;
  c : N;
  d : W;
  d/f : A;
  e/h : N;
 }
}
```

(b)

Figure 5.3. (a) Hints on an inter-program workspace represented. The hints divide the workspace into regions, each associated with an access mode. (b) The code for specifying the hints of (a).

One disadvantage of this hinting mechanism is the burden that it places upon the modeler. Though we maintain that the efforts of the modeler will frequently be justified by sufficient performance gains (see Section 5.4), we recognize that modeling is a burden that requires precious human time. To mitigate this impact, we have developed *multi-granularity modeling*, which allows a modeler to reduce the amount of effort she commits by sacrificing performance gains. The trade-off lies in deciding on the granularity of her models. Finer-grained models provide more program information, leading to more efficient transmission of workspaces while coarser-grained models provide less information, but require less effort to construct. The trade-off can be made by coalescing regions and annotating them with the most permissive access mode required by any node in the region.

As an example, a modeler could omit the A hint on node f in the example of Figure 5.3(a). Without this hint, all nodes under d — including f and p — are covered by the W hint. As a consequence, nodes f and p are transmitted to support write access, which is more permissive than the add-only access. Because transmission permits more privileges than necessary, some performance optimization opportunities will be lost. The advantage, however, is that the modeler submits fewer hints. Besides needing to know less about the program, the modeler also maintains a smaller model in the face of program changes.

As an extreme example of coarse-grained modeling, the modeler could construct a model consisting of a single W hint on the workspace root. Such a model would require the copying of *all* nodes, which is identical to the naive "copy all" approach. This approach requires minimal effort on the part of the modeler, but also results in the worst performance. In the absence of any hints, GridDB automatically generates this model.

5.3.2 Hint Processing

Having discussed the hinting interface, we now discuss the procedure for processing hints. During processing, our goal is to create a consumer workspace that supports the access modes of a consumer program while minimizing the number of file system objects copied and linked. Hint processing proceeds through two phases. In the *Resolution* phase, a *transmission mode* — copy, link or omit — is assigned to each node. In the *Transmission* phase, each node is transmitted according to its transmission mode.

Resolution

The goal of Resolution is to determine the transmission mode of each node. Resolution achieves this with a traversal of all nodes in the workspace. As we will explain, a node's transmission mode is determined by its access mode and the transmission modes of its children. As such, determining the transmission mode of a node is a three step process: (1) determine a node's access mode, (2) determine the transmission modes of its children and (3) resolve the node's transmission mode based on the results of (1) and (2). Figure 5.4(a) lists the resolve procedure, which executes the three steps for a node n using a depth-first traversal of its descendants. Step (1) is carried out by lines 2-3, step (2) by lines 5-7 and step (3) by line 9. The transmission modes of all nodes determined by applying resolve to the workspace root. If the root node is not annotated with a hint, GridDB automatically assigns it with an access mode of W. To institute this default access mode, the algorithm is initialized with the following call resolve("." , W).

The logic for determining a node's transmission mode is encapsulated in the getTMode subroutine of (b). The logic is rule-based — transmission modes are assigned based on the values of a node's access mode and its children's transmission modes. The three transmission modes, omit, link and copy, are denoted by the letters, o, l and c, respectively. Intuitively, when n and all of its children are not accessed, transmission is omitted. If n and its children are only accessed in read-only mode, n is transmitted through linking. Otherwise, n must be transmitted through copying.

```
1   RESOLVE(n, parentAccMode)
2     if(n.accessMode = null)
3       n.accessMode ← parentAccMode
4
5     foreach c ∈ n.children do
6       RESOLVE(c, n.accessMode)
7     end
8
9     n.tMode ← GETTMODE(n);
10  end
```

(a)

```
1   GETTMODE (n)
2     if(n.aMode = N and ∀ c ∈ n.children c.tMode = o)
3       n.tMode ← o
4     else if((n.aMode ∈ {N, R} or
5       n is a file and n.aMode = A) and
6       ∀ c ∈ n.children c.tMode ∈ {o,l})
7       n.tMode ← l
8     else
9       // (n is a file and n.aMode = W) or
10      // (n is a directory and
11      // (n.aMode ∈ {A,W} or
12      // ∃ c ∈ n.children c.tMode = c))
13      n.tMode ← c
14  end
```

(b)

Figure 5.4. (a) The RESOLVE procedure used during Resolution. (b) The GETTMODE subroutine used by RESOLVE to determine a node's transmission mode.

The rules that determine when each of the three cases apply make up the body of the getTMode subroutine. Transmission may be omitted when the access mode of n is N and the transmission modes of all children are o (line 2). Transmission may proceed through linking (lines 4-6) if the access mode of n implies that the node will not be modified(N and R for directories and N, R or A for files) and descendants are either not transmitted or transmitted through a link (the children of n only have transmission modes of either o or l). In all of the remaining cases (lines 9-12), copying is required. In these cases, n is being modified or some child

of n needs to be copied. If n is a file, it must have a transmission mode of W. If n is a directory, it must have an access mode of A or W or one of its children must have a transmission mode of c.

In some circumstances, copying of n will be required even if its access mode is no-read or read-only. This occurs when at least one of n's children needs to be copied. While n itself will not face change, n must be copied because the original version of n in the producer's workspace does not provide an access path to the copied children in the consumer's workspace. The new copy of n provides this access path.

Figure 5.5 shows the result of performing resolution on our running example. Capital letters to the left of each node denote access modes. Those access modes originally specified by user hints are underlined. Lower-case letters to the right of each node denote transmission modes. Consider the processing of the root node ($\bullet$). First, its access mode is propagated to b and e, its two unannotated children. Then, the transmission modes of each of its four children are determined by applying resolve to each child. Nodes b, c, d and e resolve to transmission modes of l, o, c and l, respectively. With the transmission modes of all children determined, the transmission mode of the root node is then determined. Because one of its children has a transmission mode of c, it is also assigned a transmission mode of c. As an example of omission, node c receives a transmission mode of o because it has an N access mode and does not have any children. As an example of linking, node e has a transmission mode of l because it has an R access mode and all of its children have either o or l access modes.

As the reader may have noticed, linking may indirectly transmit nodes with a

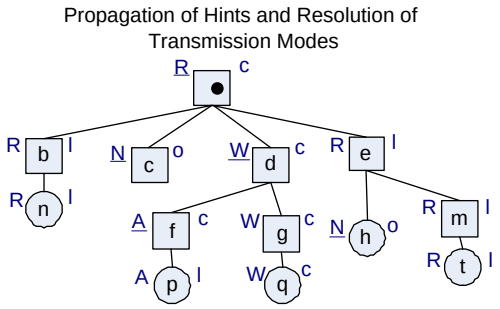


Figure 5.5. Resolution determines whether and how each node is transmitted. The lower-case letter on the right of each node represents the transmission mode: "o" if the node's transmission is omitted, "l" if it is transmitted by a link, and "c" if it is transmitted by copying.

"no-read" access mode. For example, while the leaf node h of Figure presv/6Fig has an access mode of N, it is effectively transmitted by a link to its ancestor e. That is, h's access mode is implicitly escalated from N to R. GridDB automatically performs this escalation in the interest of performance. It is cheaper to transmit h along with all other nodes in the tree under e through a link than it is to decompose e into subtrees that are transmitted differently. In the latter case, we would need to create a copy of e, and transmit m through a link. By transmitting the entire tree with one link, we save a directory copy. The disadvantage of such an optimization is that we lose the ability to *prevent* programs from reading parts of a workspace.

Transmission

Having determined the transmission mode of each node, we can now transmit the directory tree from producer to consumer. Transmission proceeds as another

```
1    TRANSMIT(n)
2        if(n.tMode = o)
3            do nothing
4        else if(n.tMode = l)
5            create n in the consumer's workspace as a link to
             n in the producer's workspace
6        else // n.tMode = c
7            create n in the consumer's workspace as a copy
             of n in the producer's workspace
8            foreach c ∈ n.children do
9                TRANSMIT(c)
10           end
11   end
```

Figure 5.6. The TRANSMIT procedure used during Transmission.

traversal from the root of the workspace. Transmission uses the transmit procedure of Figure 5.6 and is carried out by executing it on the root of a workspace.

The body of the transmit procedure contains one rule to handle each transmission mode. If n has an o transmission mode, we do nothing (lines 2-3). If n has an l transmission mode, we create a link from n the consumer's workspace to n in the producer's workspace (lines 4-5). If n has a c transmission mode, we create a copy of n in the consumer's workspace and then recursively apply transmit to each child of n (lines 6-10). Recursively applying transmit is not required when we omit transmission of a node or transmit through linking. In the case of omission, *none* of the descendants of n are accessed, so none of n's children need to be transmitted. In the case of the linking, *all* of n's children

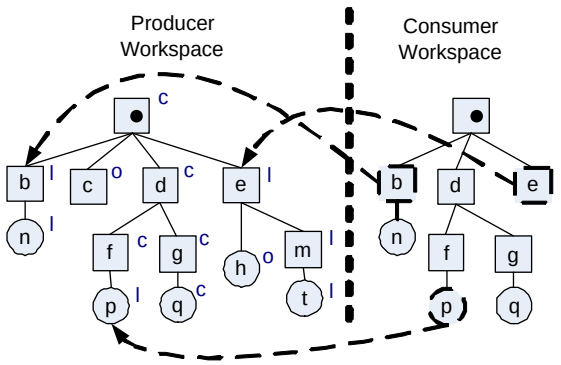


Figure 5.7. The transmission of a workspace from producer to consumer.

are automatically transmitted through a link to n, so none of them need to be further processed.

Figure 5.7 illustrates the transmission of our running example. The left side shows the producer's workspace annotated with transmission modes while the right side shows the consumer's workspace. Transmission of all nodes proceeds with a depth-first traversal from the root (•). The root, with a c transmission mode, is copied into the consumer workspace. The "transmit" procedure is recursively applied to each of its children. Of the root's children, both b and e are linked into the consumer workspace (dotted arrows denote links) while the transmission of node c is omitted. Like the root, node d is transmitted with copying, which results in the recursive application of transmit on its children.

5.4 Evaluation of the Hinting Mechanism

In this section, we present experiments that demonstrate the efficacy of our hinting mechanism. First, we evaluate the hinting mechanism through a combination of analysis and benchmarking. Then, we measure the overall speed of a GridDB server equipped with the mechanism.

5.4.1 Transmission Microbenchmark

Our first set of experiments illustrates the utility of the hint-based model through analysis and measurement. In this experiment, we construct three models for the transmission of a typical program from a real astrophysics pipeline. We analyze the number of file system objects copied if transmission were to proceed by each model and also measure the performance of transmission on a variety of file systems. From our experiments, we draw the following conclusions:

- Hints can dramatically reduce the number of file system objects and bytes copied, effectively curtailing the overhead of data-preserving transmission.
- Highly effective models can be constructed with only a small number of hints.
- Our hinting framework allows scientists to effectively make a trade-off between modeling effort and performance.

Platform and Use-Case

Our experiments were conducted on a 15-node cluster, which we will refer to as *Request*. Each node in request contains 2 1.7 GHz Xeon processors and 2 GB of memory. Unlike the MCR cluster of the previous chapter, Request allows us exclusive access to network and file system resources. The cluster contains 4 network accessible storage devices: a BlueArc Titan Server, a Firewire SCSI storage brick, a Nexsan ATABoy SCSI RAID device with 11 spindles and a Seagate IDE drive managed by an ext3 file system. All file systems were accessed through NFS. No other jobs were running on the cluster during our experiments. Each data point reported is the average of 5 runs.

Our validation is based on the MSC image-processing pipeline of the SuperMacho astrophysics survey (108). The MSC pipeline consists of 8 programs that help reduce telescopic images into catalogs of celestial objects. The mean execution time for a program is 60 seconds on a Request cluster node and inter-program workspaces, on average, consist of 830 files and directories and 315 MB. To illustrate our benchmarks, we chose an average workspace, between two of the programs, called wc and fn. At the end of the section, we show that similar results were obtained for the other inter-program workspaces.

Hint Effectiveness on One Program

Our first results demonstrate the effectiveness of using hints to reduce transmission costs. We compare the differences amongst three hinting models: the default model (copy all) and two models that reduce copying by annotating

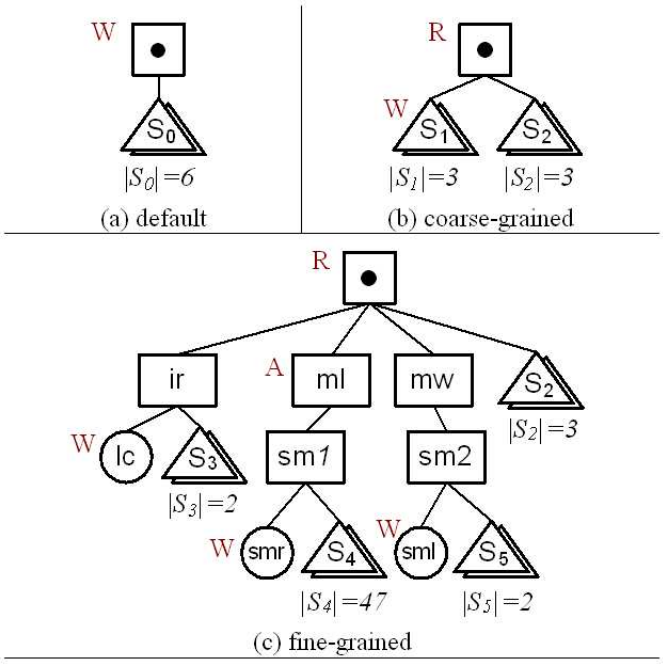


Figure 5.8. Three models for transmitting the workspace between the wc and fn programs. From (a) to (c), they are at increasingly finer levels of granularity. Absolute values denote the cardinality of a collection of trees.

workspace nodes with hints. To represent two different levels of effort by a modeler, the two models differ in the granularity of their hints. The coarse-grained model limits its annotations to nodes directly below the workspace root while the fine-grained model does not limit locations where annotations are placed. The three models, default, coarse and fine, are depicted schematically in Figure 5.8 They require only 0, 4 (three hints to annotate the three trees under S_1 and 5 hints, respectively. The default model requires 0, rather than 1, hint because in the absence of hints, GridDB assumes that the entire tree is writable (the most general model).

Figure 5.8(a) shows the number of file system objects and bytes copied vs. the

number of objects linked for each model . As the models become more detailed (from default to coarse to fine), the numbers of files, directories and bytes copied decreases dramatically, while the number of links increases. Using the fine model (vs. the default model), GridDB is able to eliminate 79% of directory copies and reduce file copies from 812 to 2. The number of bytes copied falls accordingly, from 294 MB down to 20 KB.

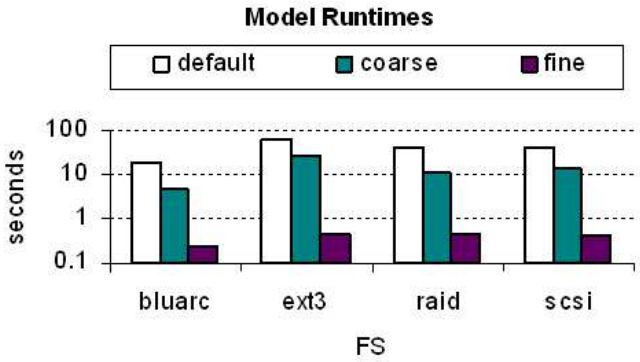
To understand how these reductions are achieved, we explain some details of this use-case. In using the coarse model rather than the default model, GridDB circumvents the copying of a large directory tree containing more than 700 files and almost 300 MB of read-only data by marking it read-only (a member of the objects in S_2 in Figure 5.8(b)). The fine model makes its primary gains over the coarse model by avoiding the copying of most of the remaining files, including a large read-only image that consumes about 285 MB (a member of the objects in S_5 of Figure 5.8(c).

As the reader may have noticed, the exchange of links for object copies is not one-to-one. This behavior is caused by the fact that changing an access mode from write or add-only to read-only will replace the copying of *all* files and directories under the tree with a single link. Figure 5.9(a) also shows that hinting need not be a labor-intensive process. The coarse and fine models, each of which greatly reduce objects copied, can be represented with only 4 and 5 overlapping hints, respectively.

While our results show, quite decisively, that hinting reduces the number of objects copied, we wanted to confirm that these reductions translated into real performance gains. As such, we micro-benchmarked the time to perform

model	Copied dirs	files	MBytes	Links created	Hints
default	24	812	294	0	0
coarse	12	95	150	3	4
fine	5	2	.020	53	5

(a)



(b)

Figure 5.9. (a) Number of files, directories and bytes copied and links created to transmit the inter-program workspace between two programs, wc and fn, for three different hint models. The table also shows the number hints required to specify the model. (b) Performance of transmission using each model across four file systems. The fine model achieves 2 orders of magnitude speedup over the default model irrespective of file system.

a transmission on each of Request's four file systems. These micro-benchmarks measure the speed of transmitting a workspace using a particular model within a particular file system. The results are shown in Figure 5.9(b). The graph indicates a clear correlation between model granularity and performance, irrespective of file system. Transmitting with the default model requires from 19-67 seconds while transmitting with the fine model requires less than 1/2 second on all file systems. In each case, transmission is sped up by two orders of magnitude.

Hint Effectiveness on Other Programs

To generalize our results, we applied the same analysis to the transmission of the other 6 snapshots inside the MSC pipeline. For each of the other pipeline programs, we obtained results that were similar to those obtained for the wc-fn snapshot. The table in Figure 5.10 lists the number of objects (files and directories) and bytes copied using default and fine models. As shown, there is a 96-99% reduction in number of objects and a 99+% decrease in bytes. In each case, directory trees within an inter-program snapshot are transmitted through linking, rather than copying. Fine models for each of the programs were expressed in 4-6 hints. We did not run disk performance benchmarks for the other programs, but we would expect similar performance characteristics based on the similarities in numbers of files, directories and bytes copied.

One pattern that we observed across the workflow programs was a pattern of "workspace augmentation" Each program tends to add files to the workspace it uses as input. Transmitting a workflow with such a pattern results in a large number of links for two reasons: (1) If files are being added to a workspace, some path within the workspace is best annotated with an add-only (A) hint. Such a hint means that all files under the path are transmitted with links. (2) The number of files under the path may become large as the pipeline progresses, particularly if many programs add files under the same path.

This pattern can be observed within our example. As indicated in Figure 5.10, the number of links required to transmit a workspace increases as we traverse deeper into the pipeline. All programs are adding traces to a particular

snapshot	files + dirs dflt	files + dirs fine	bytes (KB) dflt	bytes (KB) fine	links
fx-xt	799	5	293980	16	26
xt-ma	806	5	294032	16	33
ma-mk	812	5	294076	16	39
mk-wc	826	6	294184	16	46
wc-fn	836	7	294260	20	53
fn-cc	844	7	294316	20	60
cc-ph	886	23	438728	20	86

Figure 5.10. Reductions in numbers of files and directories (objects) copied and bytes reduced by transmitting with a fine-grained model (vs. the default model). Object reduction is between 96% and 99% and byte reduction is more than 99% in all cases. Also, the number of links used in transmitting the snapshot using the fine-grained model.

sub-directory, which is annotated with an add-only access mode in each program's hint model. We looked at one other astrophysics pipeline (109) and a protein clustering pipeline (110) and observed a similar pattern of workspace augmentation. In section 5.4.2 we will show that due to the large number of link creations caused by this pattern, GridDB's bottleneck during the unfold stage is still the file system despite successful efforts to reduce file system copying.

5.4.2 Overall Server Performance

In this section, we widen our investigation by assessing the overall performance of a centralized GridDB server as it executes workflow processes in a data-preserving manner. Our interest is to gauge the "overhead" latency induced

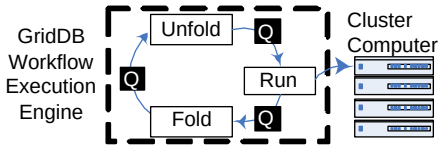


Figure 5.11. GridDB's workflow execution engine consists of three event processing stages, Unfold, Run and Fold, connected by message queues.

by GridDB when it is asked to execute N process requests. We do this by measuring the performance of the unfold (including transmission) and fold stages. We continue to use the workspace between wc and fn as our use case. We can summarize our results with these two points:

- With the assistance of hints, a centralized server can handle small jobs with very low overheads.
- Our centralized server will be overwhelmed by large jobs (of 1000 requests).

Experimental Setup

GridDB is programmed as an event-driven server whose logic is divided into stages connected by event queues. Stages are serviced by threads that dequeue requests from a queue, execute a task over the request, and possibly enqueue a request to an output queue. GridDB's workflow execution engine consists of 3 stages: Unfold, Run and Fold (shown in Figure 5.11). Each stage is serviced by a tunable number of threads, which we adjust to maximize throughput.

GridDB's implementation is written in a combination of java and C. While most of the code is java-based, many file system operations (e.g. symlink())

are not directly accessible through the java API. In these instances, we wrote wrappers to make C system calls through the Java Native Interface (111). Both java and C codes were compiled with optimizations enabled. Our experiments were executed with Sun's Java 1.4.2 JVM.

Our experimental setup is identical to that shown in Figure 5.11. A GridDB server and associated database run on a cluster node while the global file system runs on a separate node. Since our experiments were focused on characterizing DSWMS performance, we did not need to run the science programs in every experiment. Instead, we executed them once, stored images of their working directories, and reused these images in our fold and unfold experiments. We used PostgreSQL 7.4.3 as our database and varied the file system across different experiments.

Metrics

We seek to measure the "performance penalty" of data-preservation, or the time to run the fold and unfold overhead stages divided by the time to run a process (the "run" stage). To obtain this metric, we benchmark our server's ability to process requests through the Unfold (U) and Fold (F) stages. The metrics we use are:

- **Maximum Stage Throughput** ($TP_{max}^{[stage]}$): The maximum number of requests per unit of time a server can process requests through a particular stage. The maximum is empirically determined by varying the level of multiprogramming within a server and measuring throughput.

FileSys	TP_{max}^{U}	TP_{max}^{F}	$Ovhd_{20}$(s)	$Ovhd_{1000}$(s)
bluearc	6.9	10.2	4.86	242.97
ext3	2.3	8.9	10.94	547.14
firewire	6.1	9.2	5.45	272.63
raid	5.3	8.7	6.07	303.62
median	5.7	9.0	5.7	286

Figure 5.12. GridDB's unfold (TP_{max}^{U}) and fold (TP_{max}^{F}) throughput on various file systems. Units are in unfolds/sec and folds/sec, respectively. $Ovhd_N$ is the number of seconds required to process a job of N processes.

- **Job Overhead** ($Ovhd_N$): The time required execute N workflow processes through unfold, and then through fold: $Ovhd_N = N/TP_{max}^{F} + N/TP_{max}^{U}$

Results

Figure 5.12 shows the results of running GridDB on top of a variety of file systems. Multiprogramming levels, or the number of requests processed simultaneously, were 6 when run on top of the bluearc and ext3 file systems and 10 when run on top of the firewire and raid file systems. Unfold throughput (N/TP_{max}^{U}) ranges from 2.3 to 6.9 unfolds/second with a median of 5.7 while Fold throughput (N/TP_{max}^{F}) ranges from 8.7 to 10.2 with a median of 9.0. These throughput rates are quite sufficient for the small jobs created by simple, or exploratory, analysis.

For example, a scientist may create a job of size 20 by sweeping through 10 values in one parameter dimension and 2 values in a second dimension. Based on median throughputs of 5.7 for unfold and 9.0 for fold, unfolding all processes in the job requires 3.5 seconds and folding all processes requires 2.2 seconds. Total

overhead would be 5.7 seconds. The fn program (the consumer program) runs for 60 seconds, so the percentage overhead would be 9.5%, a reasonable penalty for procuring data provenance and smart recomputation.

Unfortunately, because performance is sensitive to job size, the performance will be much worse for users executing large jobs. For example, the LSST (112) is a next-generation astrophysics survey that plans on processing *thousands* of telescope images at a time using an image processing pipeline similar to MSC. Part of their science mission is to alert other terrestrial telescopes of transient celestial events so their time constraints are quite intensive. Suppose that we wanted to satisfy the requirements of such collaboration. As the table in Figure 5.12 shows, processing a job of just 1000 requests with median unfold and fold rates would require 286 seconds. Overhead for a one-minute program is almost 500%, which is not suitable for this application.

5.5 DSWMS Parallelization

To bridge the gap between the capabilities of a centralized system and the demanding requirements of large jobs, we turn to parallelization schemes that essentially trade-off additional hardware for additional performance. To better understand how parallelism may help, we start by examining a profile of where time is spent during fold and unfold in the centralized server.

5.5.1 Profiling

A profile of the fold and unfold stages is shown in Figure 5.13. The figures suggest that, for the most part[2], most of the time spent during both unfold (a median of 62%) and fold (a median of 53%) were spent in the file system. For unfold, this is somewhat surprising, considering that the database issues 11 queries per request to store and retrieve metadata and provenance information, and fine-grained modeling had reduced file system time by two orders of magnitude in our micro-benchmark. Closer scrutiny of the profile revealed that most file system time (80-90%) is spent in creating links using the symlink() call. Most of these links are caused by the fact that the program adds files to a directory, which then needs to be transmitted in add-only mode. Transmitting a directory tree in add-only mode requires one link for each file in the tree. This accounts for 47 out of the 53 links required to transmit the wc-fn snapshot. Unfortunately, while links are cheaper than copies, they are still a bottleneck at the frequency with which they are requested. In the wc-fn workspace, 53 are requested (near the average of 49 in the table of Figure 5.10). Similar to the unfold stage, the file system is the bottleneck in the fold stage. In the fold stage, file system time is spent marking directory trees read-only with the chmod() system call.

Unfortunately, shifting from copying to linking does not relieve as much pressure from the file system as we had hoped. The shift causes a disproportionate reliance on metadata operations, which have traditionally been prioritized below other features, such as fast I/O or seamless recovery.

[2]The exception is the bluearc device, which is a fast file system partially implemented with hardware (113).

FileSys	Unfold			Fold		
	DB	FS	Other	DB	FS	Other
Bluearc	11%	34%	55%	83%	11%	6%
ext3	2%	82%	16%	38%	54%	8%
firewire	5%	58%	36%	40%	52%	8%
raid	4%	65%	31%	34%	57%	9%
median	5%	62%	34%	39%	53%	8%

Figure 5.13. A profile of where GridDB spends its time in fold and unfold.

5.5.2 Parallelization Tests

The strong reliance of workflow execution on file system performance implies that any parallelization scheme would not be useful without increasing the amount of hardware carrying out file system functionality. In this section, we test this hypothesis by measuring several schemes for parallelizing the DSWMS. The configurations straddle trade-offs between centralizing or parallelizing the three software components important to workflow execution: GridDB, the database and the file system. The three configurations we examine are described below:

Shared-fs (sh-fs): In the shared-fs architecture, GridDB and the database are parallelized across multiple cluster nodes. File system requests are centralized on a single node. This scheme is congruent with the fact that many clusters provide a globally accessible central file system. A centralized file system offers benefits from economies of scale, such as decreased fragmentation and support for sophisticated features such as disk striping. Additionally, a centralized file system

offers location transparency and simplified recovery. The principle disadvantage is that the file system may become a bottleneck.

Shared-db (sh-db): The shared-db configuration is analogous to the shared-fs scheme, except that the database is shared, and the file system is scaled along with GridDB.

Shared-nothing (sh-no): The shared-nothing scheme bundles and scales up all three components. While such a scheme should maximize scalability, file system and database data are dispersed amongst the clusters nodes, complicating system design and maintenance.

5.5.3 Experimental Description

In these experiments, we investigate the performance characteristics of the various configurations. In each case, we scale-up the number of GridDB nodes, along with the number of files systems (sh-db), the number of database systems (sh-fs), or both (shared-nothing). The sh-fs approach was executed with all four file systems. A sh-fs configuration with file system x is denoted sh-fs(x). A job consisting of 128 * n requests was sent to the DSWMS, where n is the number of GridDB servers process results. We measured the aggregate fold and unfold throughput that each configuration could achieve.

5.5.4 Results

Figure 5.14 shows the results of our parallel experiments. The figure shows the total time required to unfold and fold 1000 requests. The sh-fs configuration using the ext3 (sh-fs (ext3)) file system is not shown because it performs significantly worse than any of the shown configurations. The figure shows that two shared-fs configuration, sh-fs(firewire) and sh-fs(raid), fail to scale beyond 4 nodes. Profiling the performance of these configurations shows failure to scale during both unfold and fold. During unfold, file system linking is the bottleneck. During fold, permission modifications (chmod()) are the bottleneck. The only sh-fs configuration that achieves scalability is the sh-fs(bluarc) configuration. The bluarc is a hardware-assisted file system implementation. It is able to speed up linking and permission modification fast enough so that the file system is no longer the bottleneck. The sh-db configuration also scales well to 8 nodes.

While the sh-fs (bluarc) and sh-db configurations perform well, their scalability tapers at 8 nodes. The sh-no configuration, on the other hand, scales perfectly. With 1 shared nothing node, latency is 190 seconds. With 8 nodes, this time is reduced to 23.6 seconds, or 1/8th of the latency with only 1 node. For a "typical" 120 second[3] computation, the percentage overhead is reduced from 158% down to 19.7% (a perfect 8 to 1 ratio). A comparison of the performance of the shared nothing configuration against ideal scale-up Figure 5.15 shows perfection and suggests that scale-up should continue as more nodes are added. The logical end of such a progression is to process each request on its own DSWMS node. In

[3]This number is the result of analyzing traces of batch jobs submitted to LLNL's MCR cluster between 4/1/05 and 4/7/05.

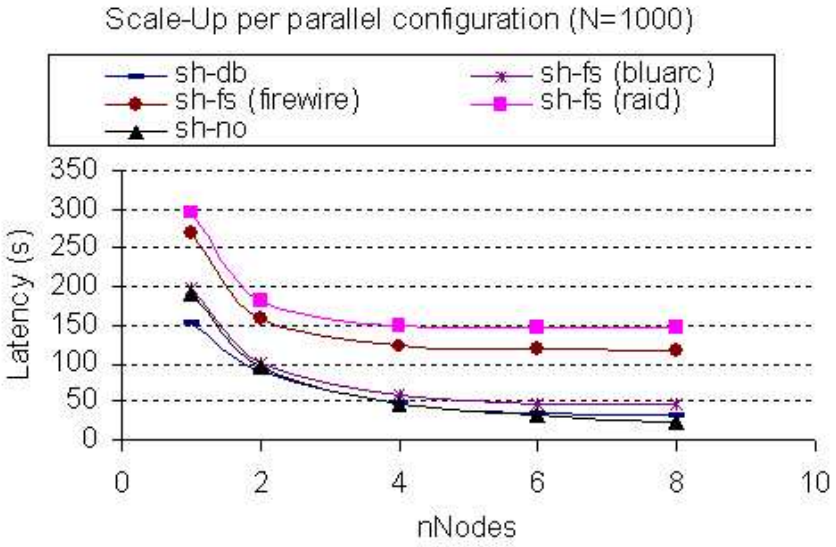


Figure 5.14. Parallelization experiments. Reduction in overall latency vs. number of nodes per configuration.

such a case, the total overhead would be equal to a small (δ), the time required to run the unfold and fold stages for one request in isolation on a cluster node.

5.5.5 Summary of Results

In this section, we validated our approach using a typical workflow program from a real astrophysics pipeline application. We demonstrated that our hinting mechanism can dramatically improve the efficiency of data-preserving transmission while incurring only small and tunable overheads upon a modeler. We also showed that a centralized GridDB server equipped with fine-grained hints can execute jobs consisting of a small number of workflows with low overheads. Larger

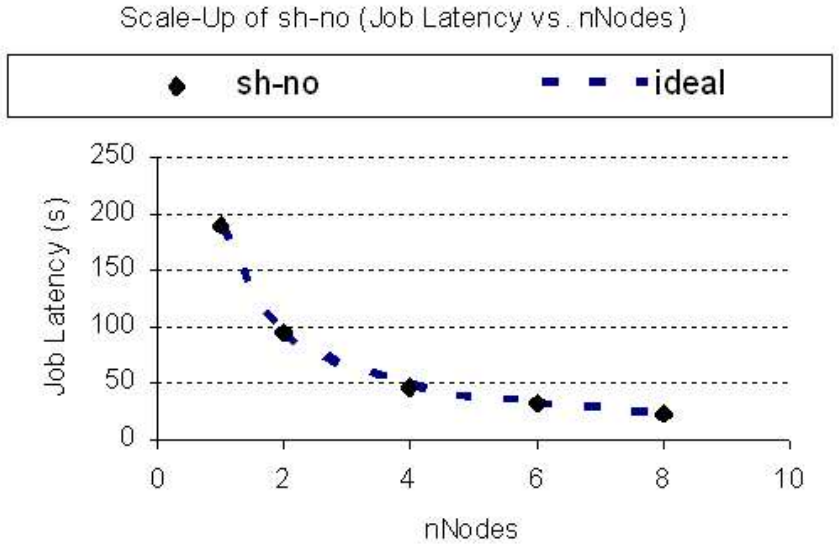


Figure 5.15. The Shared nothing configuration achieves ideal scale-up.

jobs are handled with a combination of the hinting mechanism and GridDB parallelism.

5.6 Chapter Summary

In this chapter, I identified *data-preservation* as an important feature of DSWMSs that can cause performance liabilities if naively implemented. I devised a DSWMS-level solution for performing preservation efficiently through the help of user hints. To efficiently handle large jobs, I introduced the use of parallelism in the DSWMS. I showed that file system operations must be parallelized in order to achieve speed-up. In particular, the shared nothing parallel implementation achieved ideal scale-up. Ideal scale-up allows system operators to reduce preser-

vation overhead to almost arbitrarily low levels by allocating additional cluster nodes to DSWMS tasks. The provided solution is a practical and portable means of providing data-preservation on current cluster platforms.

Chapter 6

Flexible Memoization

In the previous chapters, I described the core architecture of GridDB and argued that preservation was a core component needed for data provenance and memoization. In this chapter, I shift the focus onto the the provisioning of a memoization feature. As described in Chapter 1, memoization is the caching of computation results so that future requests may be fulfilled by retrieving old results rather than issuing new computations. Future scientific experiments, with their large sizes and increased collaboration, will need computation sharing to deal with their increased loads. Fortunately, because DSWMS's are data-centric, they are able to identify these sharing opportunities. In this chapter, I investigate how a DSWMS may most effectively provide this memoization. In particular, I observe that the memoization operator must be *flexible* to perform well in a diverse array of environments and provide techniques to achieve this flexibility.

As I will show, in situations where compute resources (i.e., nodes that run science code) are plentiful relative to DSWMS resources, it may be prudent to

forego memoization in order to avoid its overheads. There also exist circumstances where it is better to operate memoization in a tempered form, where it is only partially active. In this work, I propose a flexible memoization framework that can operate along a spectrum of modes that trade-off overhead and retrieval efficiency. Simulations show that such a framework is key to creating a robust DSWMS that can behave well in the face of different environmental scenarios.

6.1 Background and Motivation

Figure 6.1 shows the overall system architecture. As usual, a user submits his job requests to a cluster and receives results back in the form of filesets and database objects. The cluster is partitioned into DSWMS and compute nodes, which run the middleware and execute scientific programs, respectively. The DSWMS itself can be characterized as two components, which handle the needs of memoization and execution. The memoization logic determines whether each evaluation may be satisfied through result retrieval or must be resolved through a computation. If the evaluation must be computed, the evaluation is translated into a process and dispatched onto a compute node using the execution logic. This includes the unfold and fold steps described in the previous chapters.

Figure 6.2 contains a flow chart of the memoization logic, which is encapsulated in an operator. The operator first decomposes each job into a set of evaluations, as depicted in the left side of the diagram. Then, the operator processes each evaluation by searching the repository for an "equivalent" [1] evaluation

[1] We discuss different notions of equivalence in the next section.

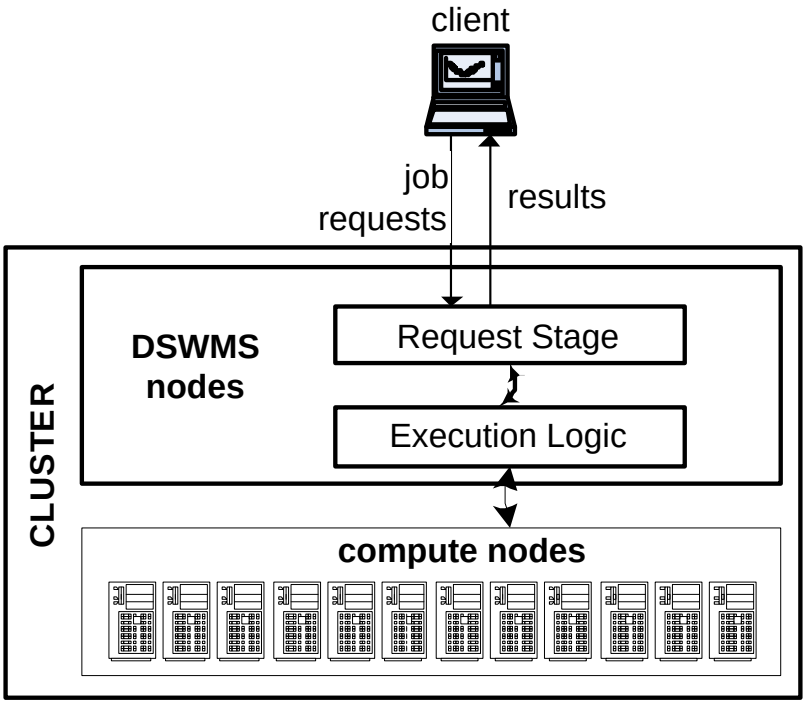


Figure 6.1. Submission of job requests to a cluster.

(step S). If the memoization operation finds a match, the stored result is returned to the user. If not, the evaluation is sent to the execution logic for execution on a compute node (step X). After computation, the operator indexes newly created results in a repository, where they can be used to satisfy future requests (step I). At each step, the evaluation may wait in a queue until an appropriate resource (DSWMS or compute node) becomes available.

Because computations are often long-running, the use of memoization to avoid them may significantly accelerate job fulfillment. Unconditional use of memoization, however, may actually be harmful. This is true for two reasons. First, the cost of search is non-negligible. The search process may require the examination

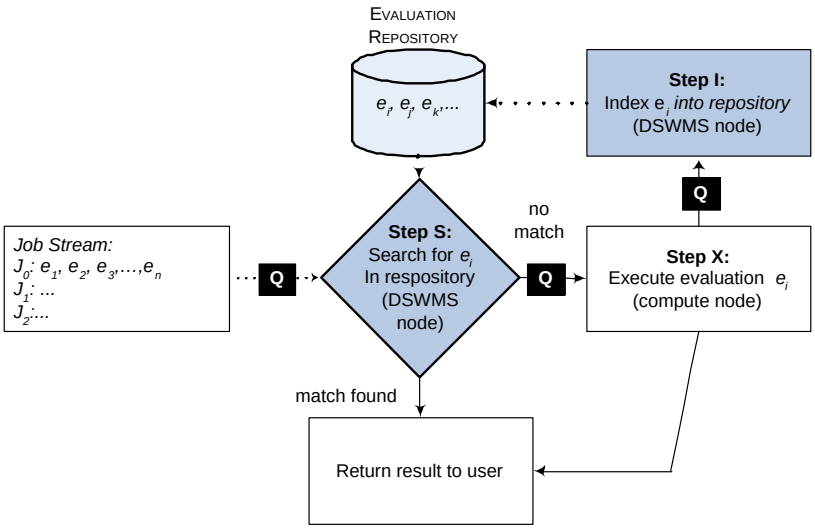


Figure 6.2. Memoization logic.

of very large datasets to determine whether or not evaluations are equivalent. Second, because scientific jobs are bursty by nature — a job submission instantly requests many evaluations — evaluations may be stalled in a DSWMS queue while others are being processed. In a situation where compute node resources are relatively plentiful, an evaluation search may create a bottleneck by preventing jobs from running on compute nodes.

We illustrate such an occurrence with a simple example. Consider a job of n independent evaluations. The job runs on a cluster consisting of one DSWMS node and enough compute nodes to run all evaluations in parallel. Assume that the repository contains 50% of the evaluations requested by the job and that a repository search requires t_s time units while executing an evaluation requires t_x time units. The DSWMS processes one search request at a time.

Despite the fact that half of the job can be fulfilled through result retrieval,

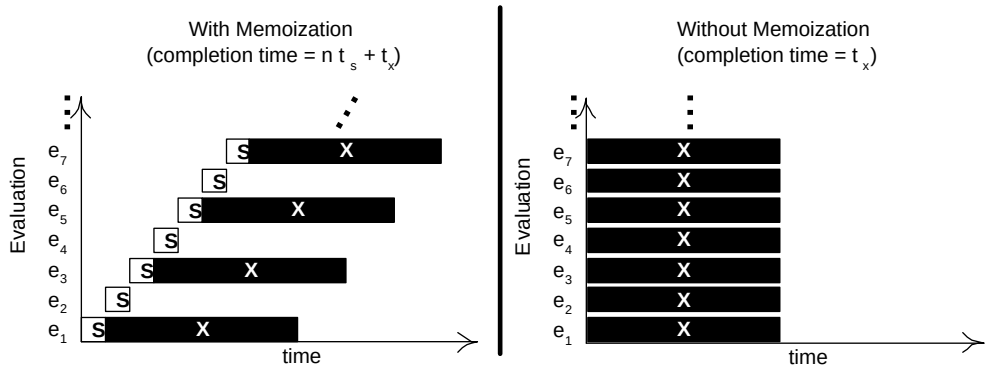


Figure 6.3. Gantt charts showing job processing with and without memoization.

this is a scenario where the use of memoization will adversely affect the cluster's performance. Figure 6.3 displays two gantt charts representing job progress with (on left) and without (on right) memoization. Each row represents the life-cycle of one evaluation as it goes through the S and X steps. I steps are omitted from this example without loss of generality.

In the presence of memoization, each evaluation engages in a search phase where the DSWMS looks for a suitable repository match. These efforts are apparently fruitful, as executions for 50% of the evaluations are averted. Because all of the evaluation requests arrive at once and the DSWMS cannot process them simultaneously, however, most of the evaluations spend time in the DSWMS queue. The result is that the last job completes at time $n * t_s + t_x$. Contrast this with the cluster's performance in the absence of memoization as shown on the right hand side. Under this configuration, all n evaluations may be executed simultaneously, so the job is completed by time t_x, which is optimal. Despite being based on an appealing philosophy of conservation, memoization actually produces negative consequences. Consider the negative impact of memoization when assigned

plausible values of $n = 1000$, $t_s = .25s$ and $t_x = 120s$ (based on Chapter 5). In this case, operating without memoization requires 120 seconds while operating with memoization requires 370 seconds — a tripling of completion time!

While this example shows potential pitfalls, the ultimate impact of memoization depends on a combination of independent factors. One set of factors can be neatly classified as the "supply and demand" of DSWMS and compute resources. As the number of available compute nodes increases (i.e., compute supply) or runtime of programs decreases (i.e., compute demand), memoization becomes less favorable. As the availability of DSWMS resources (i.e., DSWMS supply) increases — for instance, through DSWMS parallelization — or the degree of data-centricity (i.e., DSWMS demand) decreases, memoization becomes more favorable. Beyond these factors, a separate factor is the affinity between in-flight jobs and the evaluation repository as reflected by the hit-rate of repository lookups for a particular job. For this factor, higher affinity favors memoization.

With such a complex array of factors determining the favorability of memoization, it is not surprising that there is a need for intermediate forms of memoization that do not completely eliminate, but reduce, overheads, while still managing to find some repository matches. Section 6.3 demonstrates that these intermediate forms of memoization are key to performing well across a variety of environments.

Before presenting our memoization tuning mechanisms, we describe a view of the memoization operator as a *load-conversion* utility. This view will simplify the decision-making process by establishing the connection between memoization policies and their impact on resources. In order to tune the operator successfully,

one must observe that reducing overhead (and reducing recall[2]) relieves load from DSWMS resources, but increases load on the compute nodes. This is because the overheads of memoization are incurred upon DSWMS resources while the benefits of it are passed on to compute-node resources. Likewise, by increasing memoization overheads, one is essentially increasing load on DSWMS resources and reducing load on cluster nodes. Policy designers must acknowledge that tuning the memoization operator impacts different resources, or risk making poor choices. The danger is that one may unconditionally choose a memoization policy that aims to maximize the recall/overhead ratio. Such a policy, however, ignores the fact that the value of high recall diminishes in a DSWMS-bound environment and increases in a compute-bound environment.

6.2 Mechanisms for Tunable Memoization

This section addresses the need for flexible memoization by proposing a battery of mechanisms for shifting load between DSWMS and compute resources. Our mechanism design space consists of two dimensions, as shown in Figure 6.4. Each dimension represents a spectrum between low overhead (DSWMS-relieving) and high recall (compute-relieving). The first dimension balances overhead and recall by choosing from one of three basic memoization approaches: *no memoization*, *light memoization*, or *thorough memoization* (NEMO, LEMO, or TEMO). If either of LEMO and TEMO are chosen, a second dimension concerning the relative prioritization of search and indexing (EIDX and DIDX) becomes relevant.

[2]Recall is the percentage of eligible repository matches that a memoization operator finds.

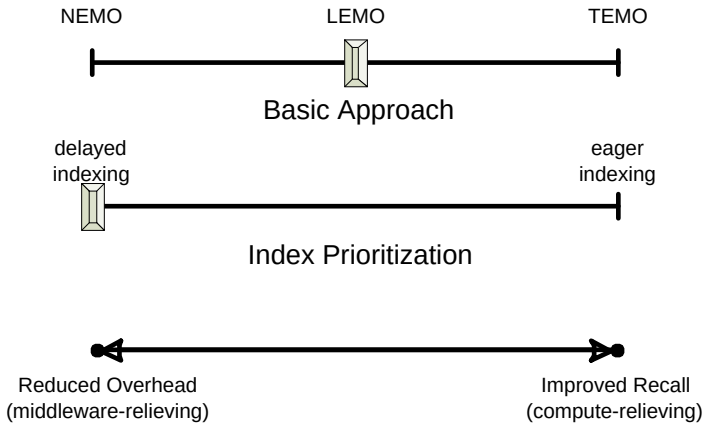


Figure 6.4. A two dimensional space of mechanisms for trading off overhead and recall (or DSWMS and compute-node load).

6.2.1 Basic Approaches: NEMO, LEMO and TEMO

We start by exploring the trade-offs amongst the three basic approaches, which differ in whether and how they identify matching evaluations during the S step. Using NEMO, memoization is foregone altogether and all overheads are spared. In contrast, LEMO and TEMO both employ memoization to exploit retrieval opportunities, but extend the critical path of job fulfillment with search and indexing overheads. TEMO distinguishes itself from LEMO by identifying a larger set of memoization opportunities, though it also incurs a higher overheard. The differences between TEMO and LEMO can be traced back to their method of determining whether two entities or evaluations are *equivalent*.

Equivalence is core to finding reusable evaluations. Notions of equivalence can be applied to both entities and evaluations. Loosely defined, two entities are equivalent ($x_i = x_j$) if one may "safely" replace the other as a result to a user or as an input into an evaluation. More specifically, two evaluations are

equivalent if they apply the *same* function to equivalent input entities. That is, all users will consider entities x_i and x_j "the same" and all evaluations using x_i as input will produce equivalent output entities if x_i were replaced by x_j. Detecting equivalence between entities presents a challenge because what may be considered "the same" by one user could be consider different by another.

To address this challenge, we define *physical equivalence*, a restrictive form of equivalence that is machine verifiable. Two entities are physically equivalent if they have the same bit representation. Physical equivalence may be directly tested, either by running `diff` between two filesets (in the case of opaque entities) or by byte-wise comparing two tuples (in the case of transparent entities).

The notion of physical equivalence is sound[3] because two entities will not be deemed equivalent if they cannot be safely replaced with one-another. It is not complete[4], however, because two entities that may be safely interchanged may not be bitwise equal. For example, if one entity is the compressed format of another entity, a user may consider them to be the same but they will be physically inequivalent. Because physical equivalence is machine verifiable, however, we use it as our fundamental form of equivalence. We defer the adaptation of less restrictive forms of equivalence to future researchers. With a well-defined notion of physical equivalence between entities, we can also define equivalence between evaluations. Two evaluations are physically equivalent if they apply the exact same function to physically equivalent inputs. The results of these evaluations will also be physically equivalent.

[3]Notion A is sound with respect to notion B if $A \rightarrow B$. Here, notion A is $physicalEquiv(e_1, e_2)$ and notion B is $sameToUsers(e_1, e_2)$

[4]Notion A is complete with respect to notion B if $B \rightarrow A$.

TEMO uses physical equivalence to determine whether a requested evaluations may be fulfilled with an indexed evaluation. Testing for physical equivalence may be expensive, as filesets used as input may require megabytes or gigabytes of storage. As a result, comparing these evaluations may require seconds of compute time. Requests may contain thousands of evaluations and repositories may contain millions. Thus, to reduce the cost of physical equivalence, TEMO uses checksums. When a newly computed evaluation is indexed, a checksum is calculated over each of its inputs. Checksum comparison is then used as a cheap method of dismissing negative matches during the search phase. If two checksums match, the direct comparison proceeds to verify that two objects are really equivalent. While checksumming may reduce the cost of physical equivalence testing, it still requires that the entities be read at least once to compute the checksum and again to do a direct match in the case of matching checksums. Input entities may be very large and so an alternative, cheaper, method of equivalence testing is desired.

LEMO addresses this need by using a cheaper, though less sensitive form of equivalence called *lineage equivalence*. Intuitively, two entities are lineage equivalent if they result from the same set of function evaluations applied to the same set of imported entities (entities that were not created by a function evaluation). Lineage equivalence is recursively defined: we say that x_i and x_j are lineage equivalent ($x_i =_l x_j$) if they refer the same entity or are the products of lineage equivalent evaluations. Two evaluations are lineage equivalent if they apply the same function to input functions that are lineage equivalent.

Checking the equivalence of two lineages is strictly less costly than checking

that two entities are physically equivalent. It is possible, however, that two entities are physically equivalent, but were derived through distinct lineages. In these cases, physical equivalence returns true whereas lineage equivalence returns false. One common example of this scenario is when two separate versions of a program yield the same result when applied to the same input. A second example is when two evaluations of the same function differ only in an input that does not impact the result. Because of such cases, lineage equivalence is a sufficient, but not necessary, condition for physical equivalence ($x_i =_l x_j \Rightarrow x_i = x_j$ but $x_i = x_j \neg \Rightarrow x_i =_l x_j$).

We contrast the sensitivities of the two equivalences through an example. We consider two scenarios where two related evaluations are requested in sequence. In both scenarios, a DSWMS first receives and fulfills a request to process an evaluation $g(f(x_i))$. After fulfillment, the DSWMS's repository stores an entity Y, with lineage $f(x_i)$ and an entity Z, with lineage $g(f(x_i))$.

In the first scenario, a follow-on request R is for evaluation $g(f(x_i))$. In this case, a DSWMS employing either physical or lineage equivalence is able to establish equivalence between R and Z and thus, can reuse Z to satisfy R. Note that R and Z are lineage equivalent because they both apply the function g to entities with lineage $f(x_i)$.

In the second scenario, the follow-on request R' is for $g(f'(x_i))$ where f' and f are different functions that return physically equivalent results on input x_i. In other words, $f'(x_i)$ and $f(x_i)$ are physically equivalent. Because they do not apply the same function, however, they are not lineage equivalent. In such a scenario, a DSWMS employing lineage equivalence will fail to find a repository

match for R'. Using physical equivalence, however, the DSWMS can establish equivalence between the two functions.

Environmental Factors Up until now, we have described three basic approaches for providing memoization: NEMO, LEMO and TEMO. Here, we use an example to illustrate how one must consider environmental factors when choosing an approach. As we show, NEMO is favorable in compute node-rich environments, TEMO is favorable in compute node-poor environments and LEMO is favorable in intermediate environments.

Consider a scenario where a job of four independent evaluations $(E_1, ..., E_4)$ is submitted to a cluster consisting of one DSWMS node and a variable number of compute nodes. The DSWMS's repository contains two of the job's four evaluations (E_1 and E_3). We assume that if the DSWMS uses the TEMO approach to memoization, it will discover both matches, while the LEMO approach will only find one match. We range the number of compute nodes from two (compute-scarce) to four (DSWMS-scarce) and contrast the performance of the three approaches in these three environments.

A matrix matching each approach to each scenario is provided in Figure 6.5. Each cell in the matrix contains a gantt chart profiling the execution of the four jobs. Each row represents execution with a particular algorithm while each column represents execution within a particular environment. The winning approach for each environment is contained in a shaded cell. As shown, the best choice depends on the availability of compute resources. With four compute nodes (col-

umn one), all executions could be run in parallel so performing lookups will not reduce latency. In this case, NEMO will be the best approach.

At the other end of the spectrum, when there are only two compute nodes (column three), compute resources are a bottleneck and the conservation of compute resources by LEMO and TEMO is now profitable. In this case, the environment is so compute-scarce that the additional computations spared by TEMO (vs LEMO) more than offset the additional search overhead.

Finally, in the intermediate case with three compute nodes (column two), LEMO is the winner. While both LEMO and TEMO manage to reduce the number of executions in a useful manner, the additional recall achieved by TEMO is unnecessary while the additional overhead delays job completion.

As stated earlier, a second factor affecting the favorability of the different approaches is the affinity between a job and the repository. In the presence of poor affinity, we favor NEMO, as searches are likely to be wild-goose chases. If a job and the repository have strong affinity, LEMO or TEMO are favorable. If a large portion of repository matches are only detectable with TEMO, the favorability will shift towards TEMO.

6.2.2 Index Prioritization: EIDX and DIDX

A second dimension along which one may configure a memoization operator is to adjust the relative priorities of search and indexing tasks. As previously discussed, DSWMS resources are responsible for both of these functions: searching to determine whether or not an evaluation may be retrieved, and indexing

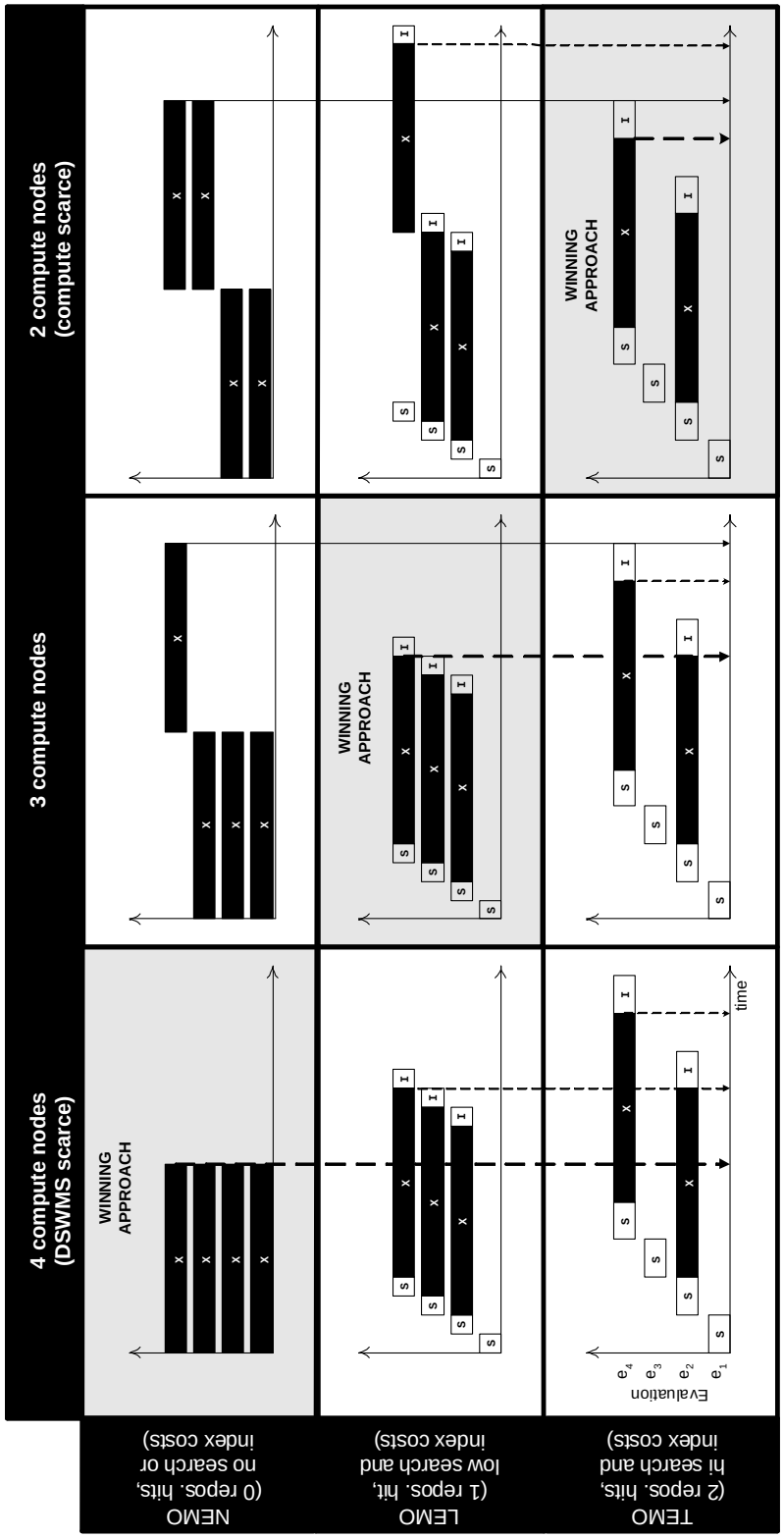


Figure 6.5. A matrix contrasting the performance of the three basic approaches (NEMO, LEMO and TEMO) in three environments.

to make results available to future searches. Under heavy load, there may be contention between the two functions and a DSWMS system will need to choose which has priority. As we show, the comparative priority of index and search tasks acts as a load shifting mechanism. In particular, processing search events before indexing events (*Delayed Indexing* or DIDX) is a DSWMS-relieving mechanism while processing indexing events over search events (*Eager Indexing* or EIDX) is a compute node-relieving mechanism.

We start by considering the positive impact of Eager Indexing on recall. The fundamental role of indexing is to make the result of an evaluation retrievable in the future. If a second request for the evaluation occurs before indexing of the first request can be completed, the search will fail even though in principle the results are available. Eager Indexing reduces the probability of these false negatives by indexing evaluations earlier in time. In contrast, the use of delayed indexing sacrifices timely indexing of evaluation in favor of timely search. Because eager indexing improves recall, it is compute-relieving. Because delayed indexing relieves a DSWMS of its responsibility to index promptly (at the expense of the compute-nodes), it is DSWMS-relieving.

For example, consider a cluster that receives requests for two jobs J_1 and J_2 where J_1 requests evaluations A and B and J_2 requests evaluations A and C. The performance of the cluster is examined under two environmental circumstances, where the cluster has a large compute pool (2 nodes) and a small compute pool (1 node). Gantt charts for the four combinations of the schemes and cluster sizes are shown in Figure 6.6.

As shown in the left column, EIDX is able to exploit the fact that J_2's requests

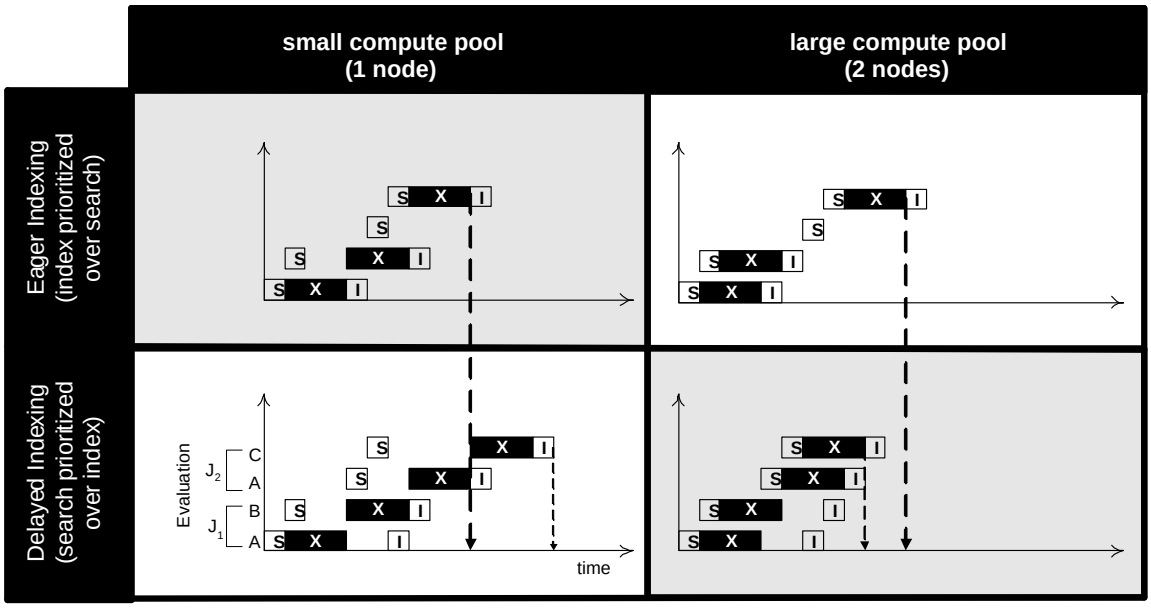


Figure 6.6. A matrix contrasting the performance of EIDX and DIDX in small and large compute pools.

for A can be fulfilled with the result created to satisfy J_1's request for A. It is able to identify this opportunity because it indexes A before searching for it during the processing of J_2. In contrast, EIDX is not successful in the large cluster (right column). While it still reduces load on the compute nodes, compute nodes are no longer a scarce resource. By extending the amount of time required by the DSWMS to dispatch its jobs (it interjects two indexing tasks between the search tasks), EIDX has extended the critical path. On the other hand, DIDX fares better with its strategy of dispatching executions before indexing results. The additional compute resources more than offset the extraneous execution.

6.3 Performance Studies

In this section, we present simulations that demonstrate the importance of a flexible memoization operator and provide guidance on configuring the operator. Our simulations are run over a set of prototypical environmental and workload scenarios. We present two sets of experiments, one investigating the trade-offs of our three basic approaches, NEMO, LEMO and TEMO, and a second investigating the relative merits of eager and delayed indexing (EIDX and DIDX).

6.3.1 General Setup

We start by describing the setup used in both studies, as depicted in Figure 6.8. In each experiment, we submit 1 or 2 jobs to a cluster consisting of 1 DSWMS node and n compute nodes. Each job consists of 1024 independent evaluations. For simplicity (and without loss of generality), we assume that each evaluation requires a fixed amount of time for search, execution and indexing. The job sequence submitted to the cluster and the initial contents of the repository are varied from experiment to experiment.

In each experiment, we initialize the repository with evaluations that match anywhere from none to all of the evaluations requested by the jobs. We assume that each evaluation requires 120 seconds, the median program execution time of scientific programs in a survey we discussed in Chapter 5. We vary search and index costs for LEMO- and TEMO-based operators from 0.125 to 2 seconds per request. In each experiment, we set the search and index costs to be equal.

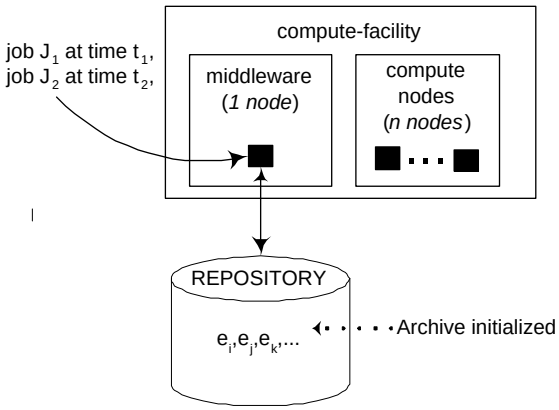


Figure 6.7. General simulation setup of studies I and II.

6.3.2 Study I: NEMO, LEMO and TEMO

Our first study compares the performance of the three basic approaches as the degree of compute-scarcity changes.

Experimental Setup

Our experimental setup for this study is a refinement of the general setup described in Section 6.3.1. We simulate clusters of three different sizes: a small cluster with 64 compute nodes, a medium cluster with 128 compute nodes and a large cluster with 1024 compute nodes. We set search and index times equal to 0.25 sec/request when using LEMO and 1 sec/request when using TEMO. Hit Rates for LEMO and TEMO range from 0 to 100% in 10% increments.

Results

The performance of the three approaches for large, medium and small clusters is compared in Figures 6.8, 6.9, and 6.10, respectively. The three graphs indicate that in large clusters, TEMO and LEMO under-perform NEMO, in small clusters, TEMO and LEMO outperform NEMO, and results are mixed on medium clusters. On the large cluster (Figure 6.8), memoization-based methods are an impediment to speedy processing, even in the presence of high hit rates. In this case, the additional step of searching the archive for past computations only serves to delay the processing of uncached evaluations while the plenitude of compute nodes obviates the benefits of retrieval. In this case, the use of LEMO increases runtime by more than 3-fold while the use of TEMO increases runtimes by almost 10-fold, regardless of hit rate. Fortunes are reversed in the small cluster (Figure 6.10). In this case, either of LEMO and TEMO achieve significant speedups. For example, at a 50% hit rate, both approaches cut runtimes by roughly half. At a hit rate of 100%, LEMO is even able to cut runtimes down to nearly one-tenth when compared to NEMO. In the case of the medium cluster (Figure 6.9), LEMO supersedes NEMO only at hit rates of 30% or more while TEMO under-performs NEMO regardless of hit rates. Altogether, the three scenarios demonstrate the need for a versatile operator that can make a context-sensitive decision on the use of memoization.

A second observation concerns the relative merits of TEMO and LEMO. As shown in Figure 6.10, the improved hit rates achieved by TEMO can render it superior to LEMO. As one example, suppose that TEMO can achieve a cache hit rate 50% while LEMO can only achieve a hit rate of 10%. In this case, the use of

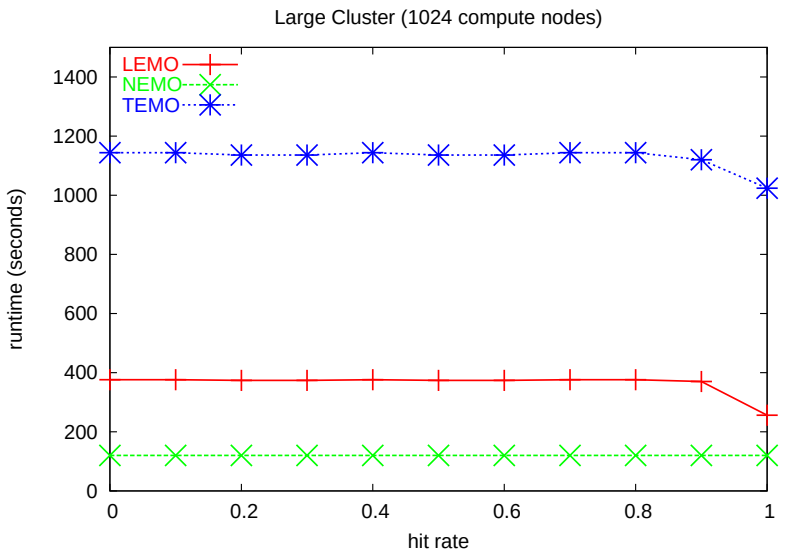


Figure 6.8. Performance of basic approaches on a large cluster.

TEMO improves performance over LEMO by 37% (LEMO requires 1806s while TEMO requires 1136s).

As the reader may have noticed, there exists a "tipping point" in TEMO's performance curve at a hit rate of 0.5. This is the point at which the DSWMS becomes the bottleneck. Even though hit rates higher than 0.5 continue to decrease the amount of work being performed by the cluster, the work reductions occur on the compute nodes, which are not the bottleneck after the tipping point. As hit rates increase, fewer evaluations require computation and compute nodes begin going idle. The DSWMS does not benefit from the same workload reduction. As hit rates increase, its workload remains the same. With a DSWMS processing rate of 1 second per request, the cluster requires at least 1024 seconds to send all requests through the DSWMS. As similar tipping point exists with LEMO-based

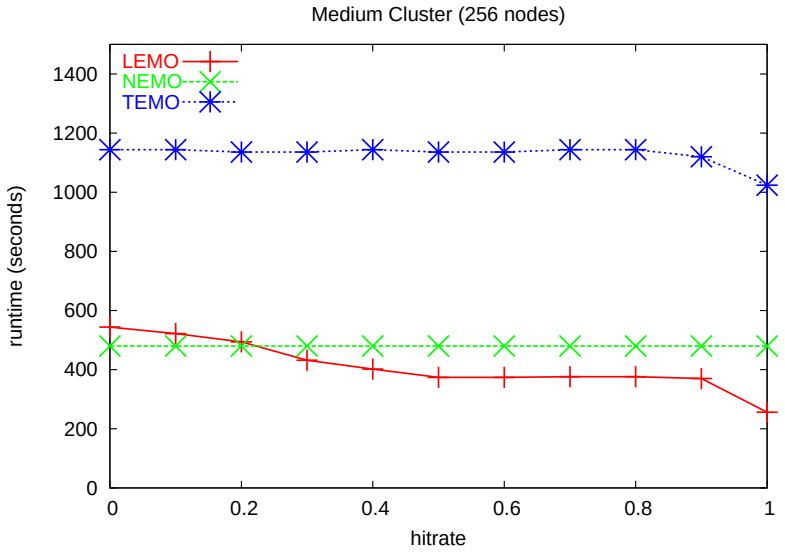


Figure 6.9. Performance of basic approaches on a medium cluster.

memoization. It is not until a hit rate of 0.9, however, that the tipping point is encountered.

The fact that LEMO's plateau comes at a higher hit rate than that of TEMO has practical implications. In particular, when DSWMS resources are constrained, it will be imprudent to opt for the higher hit rates of TEMO rather than the lower overheads of LEMO. Consider our example with 64 nodes (Figure D25.alpha). Suppose we had a choice of using LEMO to achieve a 80% hit rate verses using TEMO to achieve a 100% hit rate. In this case, we would be wise to choose LEMO in spite of its inferior hit rate. The runtime of LEMO at an 80% hit rate (496s) is less than half the runtime of TEMO at a 100% hit rate (1024s). Like other results in the section, this result shows that thoughtful consideration of resource availability is key to maximizing performance.

In this study, we showed wide variations in the favorabilities of NEMO, LEMO

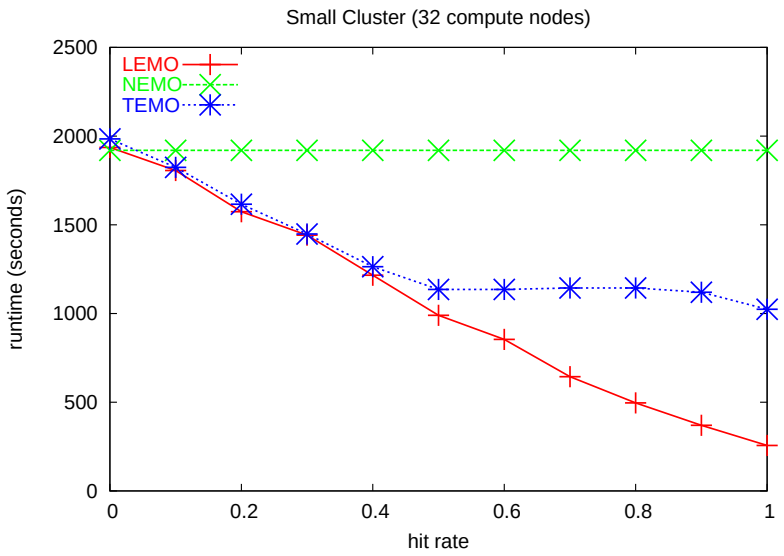


Figure 6.10. Performance of basic approaches on a small cluster.

and TEMO in small, medium and large clusters. Cluster size is but one example of a broader class of environmental factors. Similar trends can be observed as one varies any factor that affects the relative scarcities of DSWMS and compute nodes. Recall from Section 6.1 that these other factors are: the availability of DSWMS resources, the time and resource demands of evaluations and the degree of data-centricity of evaluations.

6.3.3 Study II: EIDX and DIDX

Suppose that a policy designer has decided to use either of the two memoization-employing approaches. She is then faced with the second decision of whether to use *Eager* or *Delayed Indexing* (EIDX or DIDX). Our second study provides guidance on how this choice should be made. Like the choice of a basic

approach of the previous section, this choice is also dependent on environmental factors.

Fundamentally, EIDX will only be favorable over DIDX in situations where there are two or more overlapping, concurrent jobs. First, in the absence of overlapping jobs, there will be no advantage to indexing (or memoization). The lack of locality would render cache searches futile. In this case, DIDX will incur fewer disruptions to a DSWMS's job of dispatching evaluations onto compute nodes. Second, if overlapping jobs do exist, but they are not concurrent, delayed indexing will be sufficient for later jobs to take advantage of the work performed by earlier jobs. Indexing does not need to occur with immediacy because overlapping jobs that exploit the cache will not be issued until the current job is completed.

Though these conditions, that jobs must be both overlapping and concurrent, seem rather stringent, they should be frequently satisfied in future scientific collaborations. First, because collaborations will involve thousands of scientists executing workflows on shared clusters, concurrent jobs will be common. Second, because scientists will often share computations and their parameters — in order to conduct controlled experiments where most variables are fixed — their aggregate workloads will exhibit a great deal of locality.

In the presence of concurrent, overlapping jobs, the relative favorability between EIDX and DIDX depends on the magnitude of DSWMS overheads. The relationship is in fact non-trivial. As DSWMS overhead increases, it generates two secondary effects that are diametrically opposed. One of these effects favors eager indexing while the other favors delayed indexing. Ultimately, favorability is decided by the reconciliation of these two effects.

The first effect concerns the occurrence of EIDX-favoring events. For EIDX to supersede DIDX, there must exist instances where tasks for the search and indexing of the same evaluation reside in the DSWMS queue simultaneously. In these circumstances, EIDX will choose to index the evaluation before searching for it, successfully resolving the evaluation through a cache hit. DIDX, on the other hand, will prioritize the search over the index task, sacrificing the retrieval opportunity. With all other factors held constant, higher DSWMS costs will slow down the processing of search and indexing tasks (for both DIDX and EIDX), increase the number of queued tasks, increase the potential for search/index "collisions," and increase the viability of EIDX. As DSWMS overheads decrease, on the other hand, queue sizes diminish along with the possibility of such collisions and the advantage of EIDX. By increasing the potential for search/index collisions, the first effect increases the favorability of EIDX.

The second effect opposes the first: as DSWMS overheads increase, DSWMS resource become scarcer. This second effect has a larger impact on EIDX than on DIDX. Because EIDX processes indexing tasks before search tasks, its DSWMS load is higher relative to delayed indexing. As DSWMS overhead increases, it affects EIDX more than DIDX. At some point, DSWMS load increases so much that DSWMS resources become a bottleneck and increases in DSWMS overhead result in a commensurate increase in overall runtime. Because increases in DSWMS overhead affect EIDX more than DIDX, this point will occur for EIDX first. When it *does* occur, EIDX's performance will degrade rapidly. Eventually, DIDX will also succumb to the same problems, but it occurs at a higher level of DSWMS overhead. By increasing DSWMS-scarcity for EIDX faster than

for DIDX, the second effect increases the favorability of DIDX. The next set of experiments illustrate the bipolar influence of DSWMS overhead.

Experimental Setup

In these experiments, we examine the performance of a cluster in fulfilling two consecutive jobs that request the same set of evaluations. Other than the fact that two jobs, rather one job is requested, our simulation parameters are identical to the "small cluster" in the first experiment from our first study. To summarize, the cluster consists contains 64 compute nodes, each job consists of 1024 evaluations and each evaluation requires 120 seconds to execute. The arrival time of the second job occurs after the first job, but early enough so that DIDX cannot index any of the first job's evaluations in time. Because EIDX prioritizes indexing, it is able to index some or all of the evaluations before the DSWMS begins processing the second job. Search and index overheads are equal to one another and are ranged from 0.125 to 2 seconds.

Results

We illustrate the two effects of increased DSWMS overhead in Figures 6.11 and 6.12. Figure 6.13 shows the confluence of the two effects by comparing the overall runtimes of EIDX and DIDX in a ratio. Figure 6.11 shows that as DSWMS overheads increase, EIDX is able to achieve higher hit rates even though the second job comes closely behind the first job. Figure 6.12 shows the amount of time required to process the first job, and therefore the adverse effects

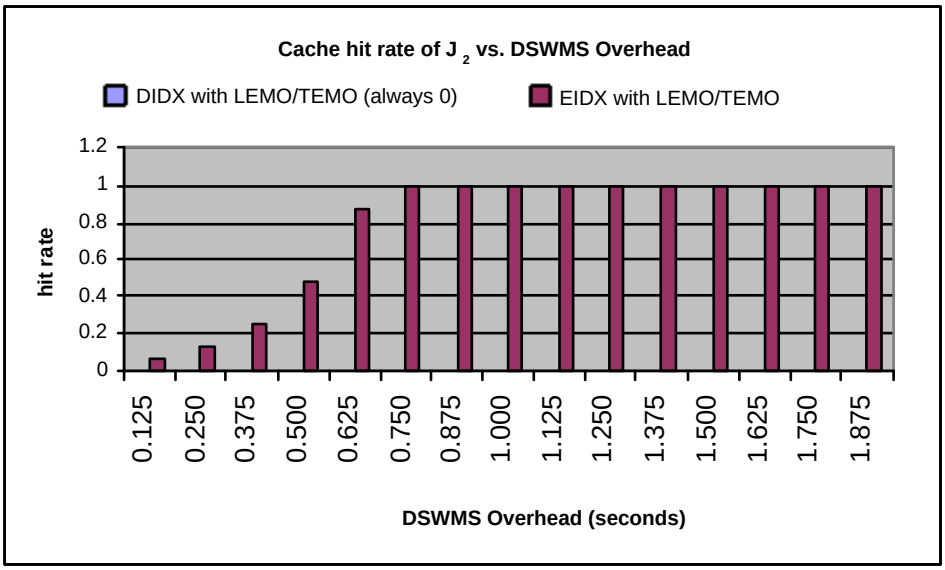


Figure 6.11. Cache hit rates when processing job the second job (J_2) using DIDX (always 0) and EIDX vs. DSWMS overhead

of increased DSWMS overhead. Up until 0.875 seconds, both EIDX and DIDX show only slight increases in runtime. When overhead reaches 1 second, however, there is a marked increase in runtime. At this point, the DSWMS becomes a bottleneck, so increases in DSWMS overhead result in proportional increases in overall runtime. While this measurement was taken on the first job only, the processing of the second job exhibits similar characteristics.

The overall impact on DSWMS overheads on performance is illustrated in Figure 6.13, which shows that the relative performance of EIDX initially increases, peaks out with a DSWMS overhead of 0.625 seconds and then drops. The initial increase is attributed to the higher hit rates as shown in Figure 1. The descent is attributed to the second effect as shown in Figure 6.12.

While in this scenario, increased hit rates make EIDX the better choice up to

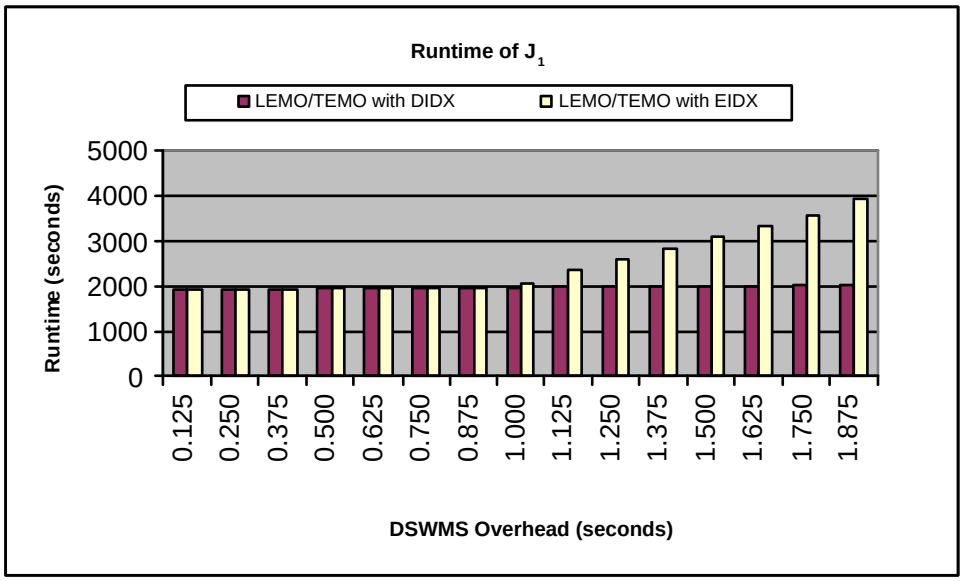


Figure 6.12. Runtimes for the first job (J_1) for DIDX and EIDX as DSWMS overhead is varied.

a DSWMS overhead of 10 seconds, this will not always be the case. The magnitudes of the two effects are dependent on environmental factors. As previously mentioned, the hit rates of EIDX and DIDX depend on the timing of the second job as well as affinity between the two jobs. The magnitude of the second effect depends on compute-scarcity within the environment. In particular, environments with higher computational scarcity will have a delayed onset of the second effect. For example, if we had only 32 nodes instead of the 64 in our experiment, the tipping point where the EIDX-based DSWMS becomes a bottleneck would be at 2 seconds instead of at 1 second.

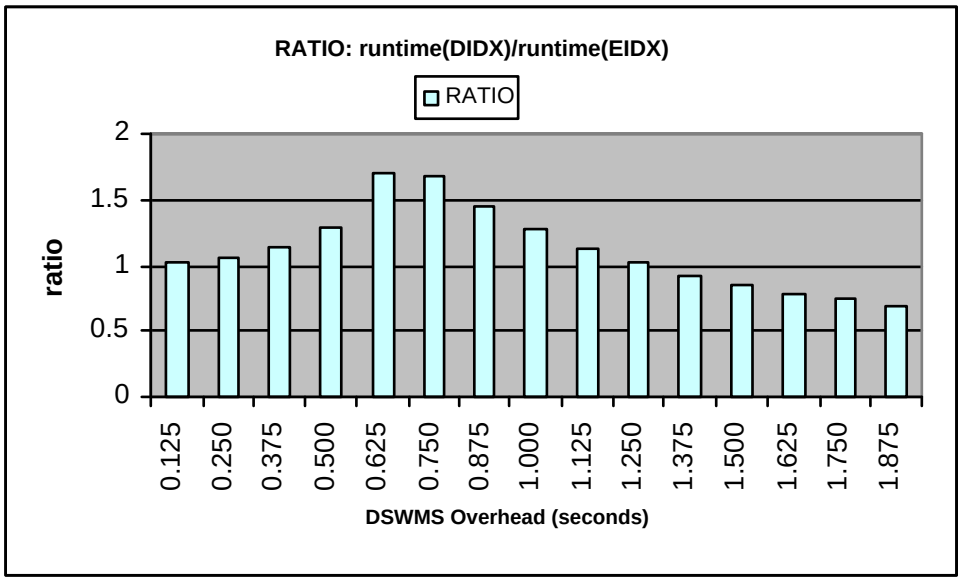


Figure 6.13. Ratio of the overall runtimes of DIDX and EIDX vs. DSWMS overhead.

Summary of Experiments

In this section, we have presented results from two studies that demonstrate the need for a flexible memoization operator and provide guidance on how the memoization operator should be configured. In our first study, we showed that in the presence of a large cluster, it will be best to use the NEMO approach. In smaller clusters, one of the LEMO or TEMO approaches will be better. The relative favorability between LEMO and TEMO will depend on the relative hit rates of the two approaches, or the type of overlap between incoming jobs and the repository. It will also depend upon the level of compute node scarcity in the cluster.

In our second study, we compared the favorability of eager and delayed in-

dexing (EIDX and DIDX). We determined that EIDX would only be superior to DIDX in the presence of co-existing, overlapping jobs. In addition, the DSWMS overhead has a large impact on the relative favorabilities of EIDX and DIDX. As DSWMS overhead increases, it generates two independent and diametrically opposed effects whose summation determines favorability. We described the two effects and the environmental factors that determine their magnitude.

6.4 Related Work

Having proposed and evaluated various mechanisms for flexible memoization, I now describe related work. Several techniques proposed in the database management literature (114; 115; 116; 117) reduce the volume or priority of index updates in order to mitigate load. Our delayed indexing (DIDX) mechanism is based on similar foundations. These works, however, decide whether to delay indexing based purely on the level of load the system is experiencing. In contrast, our flexible memoization operator decides whether indexing should be delayed based on many additional factors, including workload attributes and the relative availability of different resource classes.

A number of frameworks have also been proposed for caching scientific data in "computational grids" (118; 119; 120; 121; 122). These systems cache data in order to conserve computation and communication resources. Unlike our proposal, however, these systems do not allow cached entities to be retrieved by functional expression. Rather, these systems index their data entities by unique identifiers, which do not match the format of user requests. For example, our

memoization operator permits retrieval requests in the form of $g(f(x_i))$ whereas other scientific systems typically only allow the retrieval of x_i.

The issue of data equivalence has been mentioned in the work of Cavanaugh and Graham in (123). Like us, the authors identify notions for physical and lineage equivalence. In their work, however, the authors are focused on collaboration: equivalence is used to to reconcile different sets of data after they have been created and analyzed. In contrast, we focus on runtime efficiency, using equivalence to identify redundant work in real-time.

6.5 Chapter Summary and Future Work

In this chapter, I identified the need for a flexible memoization operator that modulates its retrieval efforts in accordance with environmental conditions. The chapter proposes a series of mechanisms that use varying degrees of DSWMS and compute node resources depending on their supply and demand. In a pair of simulation studies, I demonstrate a dire need for a flexible memoization operator and show that each of the two design dimensions proposed is in fact helpful in optimizing a cluster's performance under different environmental conditions.

While this chapter provides comprehensive mechanisms for modulating memoization, it does not to address the issue of policy; that is, when and how does an administrator decide how the memoization operator should be configured? One can foresee a full-range of approaches that vary in administration overhead, ability to adapt, and system complexity. At one end, the memoization operator may be configured statically when the DSWMS is initiated. Such an approach is

simple to implement and administer, but will not dynamically adapt to changing conditions. A more flexible approach may be to configure each function individually, as each function may have different overheads and benefits with respect to memoization. Also, we may allow changes to memoization policy during runtime to confront environmental changes. To push further along these lines, we may implement a fully adaptive approach that monitors and predicts environmental conditions during runtime and automatically picks policies based on predictions. This approach would be appealing to administrators, but would result in greater implementation complexity. Because of these issues, memoization policy is an interesting avenue for future exploration.

Chapter 7

Concluding Remarks

In recent years, we have seen an increased coupling between scientists and computer technology. This couple has created an explosion in computation volumes and lead to increased user needs and expectations for their tools. These trends give rise to a new set of problems and opportunities in computation management. In this dissertation I have proposed the concept of a Data-Centric Scientific Workflow Management System (DSWMS) to address these needs. The DSWMS is an intermediary software layer that provides a declarative interface, improved performance and a host of features that will make next-generation scientists more productive. The DSWMS exploits"data-awareness" to supersede current-generation "process-centric" middleware.

I have defined the vision of a DSWMS and accomplished several milestones towards its realization. My contributions include the selection of a feature set, the definition of a data model and language, and the implementation of a prototype called GridDB. In addition, I have uncovered several technical problems

by evaluating GridDB in a large-scale environment. These problems have been addressed in the chapters of this thesis. Finally, I have verified the expressibility of the FDM/RC model on tens of scientific workflows, 6 of which have been described in detail as use-cases in this thesis.

This thesis represents an initial approach to realizing DSWMSs, there are also areas where future improvement can be made. With data-centricity, DSWMSs are empowered with additional workload information, which is only partially exploited in this work. This additional information can be used to provide even more power to users. As one example, a DSWMS can use data-centricity for enhanced cost estimation and resource allocation. In particular, the DSWMS can, through repeated observation, associate parameter values of computations with execution characteristics such as runtime or memory usage. These estimates will be better than those obtainable through process-centric middleware and can be reported to users to help them plan their schedules or can be used by the DSWMS to utilize resources more effectively.

In the future, the data-centric approach to computation management will be required if science projects are to achieve their goals. As the core idea behind the wildly successful multi-billion dollar database management industry, the 2-phase programming model I advocate has a promising track record. Additionally, the evolutionary philosophies that we advocate — incremental adoption, interoperability with current programming models, and compatibility with existing infrastructure — should facilitate adoption. With their increased potential for automation and efficiency, DSWMSs will be a core tool in the future search for world-changing discoveries.

Bibliography

[1] Top 500 Supercomputers. `http://top500.org/`.

[2] Marcy G, Butler P, Fischer D, Vogt S, Henry G, et al. California & Carnegie Planet Search. http://exoplanets.org/.

[3] Marcy GW, Butler RP (1998) Detection of extrasolar giant planets. Annual Review of Astronomy and Astrophysics 36:57–97. doi:10.1146/annurev.astro.36.1.57.

[4] Cox P, Betts R, Jones C, Spall S, Totterdall I (2000) Acceleration of global warming due to carbon-cycle feedbacks in a coupled climate model. Nature 408:184–187.

[5] Climate Simulation at Sandia National Laboratories. http://www.cs.sandia.gov/capabilities/ClimateSimulation.

[6] Anderson DP, Cobb J, Korpela E, Lebofsky M, Werthimer D (2002) Seti@home: An experiment in public-resource computing. Commun ACM 45:56–61. doi:http://doi.acm.org/10.1145/581571.581573.

[7] (2007). FightAIDS@Home Project: A joint effort of Entropia and Scripps Research Institute. http://www.fightAIDSatHome.org.

[8] (2007). Rosetta@Home: Protein Folding, Design and Docking. http://boinc.bakerlab.org/rosetta/.

[9] (2007). Folding@Home: Distributed Computing. http://folding.stanford.edu/.

[10] Sterling T, Savarese D, Becker DJ, Dorband JE, Ranawake UA, et al. (1995) BEOWULF: A Parallel Workstation for Scientific Computation. In: Proceedings of the 24th International Conference on Parallel Processing.

[11] Gray J, Compton M (2005) A call to arms. ACM Queue 3:30.

[12] (2003). Grid physics network (griphyn) white paper.

[13] Paul Avery ea (2001). ivdgl itr proposal: An international virtual-data grid laboratory for data intensive science. `http://www.phys.ufl.edu/~avery/ivdgl/itr2001/proposal_all.pdf`. "Proposal 0122557".

[14] Livny M, Mount R, Newman H, Pordes R (2001). Particle physics data grid collaboratory pilot. `http://www.ppdg.net/docs/SciDAC/PPDG_overview.pdf`.

[15] (2000) Data Management in an International Data Grid Project.

[16] Litzkow M, Livny M, Mutka M (1988) Condor - a hunter of idle workstations. In: Proceedings of the 8th International Conference of Distributed Computing Systems.

[17] Foster I, Kesselman C (1997) Globus: A metacomputing infrastructure toolkit. The International Journal of Supercomputer Applications and High Performance Computing 11:115–128.

[18] Hellerstein JM, Avnur R, Raman V (2000) Informix under control: Online query processing. In: Data Mining and Knowledge Discovery 4(4). pp. 281–314.

[19] Gray J, Liu DT, Nieto-Santisteban M, Szalay A, DeWitt DJ, et al. (2005) Scientific Data Management in the Coming Decade. In: SIGMOD Record.

[20] (2007). Lsst press release: Google joins large synoptic survey telescope project. `http://www.lsst.org/News/docs/google.pdf`.

[21] (2007). Large synoptic survey telescope (lsst) home page. `http://www.lsst.org/`.

[22] Gray J, Szalay A (2006) 2020 computing: Science in an exponential world. Nature doi:10.1038/440413a.

[23] Vik Singh and Jim Gray and Ani R Thakar and Alexander S Szalay and Jordan Raddick and Bill Boroski and Svetlana Lebedeva and Brian Yanny (2006) MSR-TR-2006-190: SkyServer Traffic Report The First Five Years. Technical report, Microsoft Research.

[24] Bollacker K, Lawrence S, Giles CL (1998) CiteSeer: An autonomous web agent for automatic retrieval and identification of interesting publications. In: Sycara KP, Wooldridge M, editors, Proceedings of the Second Inter-

national Conference on Autonomous Agents. New York: ACM Press, pp. 116–123. URL citeseer.ist.psu.edu/bollacker98citeseer.html.

[25] (2007). ACM Digital Library. http://portal.acm.org.

[26] (2007). Google Scholar. http://scholar.google.com.

[27] Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, et al. (2006) The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183%.html.

[28] Gray J, Shenoy P (2000) Rules of Thumb in Data Engineering. In: ICDE.

[29] Gray J (2007) Presentation: Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King. In: CIDR. http://research.microsoft.com/~Gray/talks/Flash_is_Good.ppt.

[30] Erich Strohmaier, Jack J Dongarra, Hans W Meuer, Horst D Simon (2005) Recent Trends in the Marketplace of High Performance Computing. Cybertechnology Watch Quarterly 1.

[31] Jette, M, Grondona, M (2002) SLURM: Simple Linux Utility for Resource Management, User's Manual. http://www.llnl.gov/LCdocs/slurm/slurm.pdf.

[32] Capit N, Costa GD, Georgiou Y, Huard G, n CM, et al. (2005) A batch scheduler with high level components. In: Cluster computing and Grid 2005 (CCGrid05). URL http://oar.imag.fr/papers/oar_ccgrid05.pdf.

[33] GNU Free Software Foundation GNU Queue Manual. http://www.gnu.org/software/queue/.

[34] TORQUE Resource Manager. http://www.clusterresources.com/pages/products/torque-resource-manager%.php.

[35] Thain D, Tannenbaum T, Livny M (2006) How to measure a large open-source distributed system: Research articles. Concurr Comput : Pract Exper 18:1989–2019. doi:http://dx.doi.org/10.1002/cpe.v18:15.

[36] Condor World Map. http://www.cs.wisc.edu/condor/map/.

[37] Foster I, Gieraltowski J, Gose S, Maltsev N, May E, et al. (2004) The grid2003 production grid: Principles and practice. In: HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04). Washington, DC, USA: IEEE Computer Society, pp. 236–245. doi:http://dx.doi.org/10.1109/HPDC.2004.36.

[38] George Gilder (1997) Telecosm. Simon & Schuster.

[39] BOINC Statistics on 1/16/07. http://boinc.netsoft-online.com/.

[40] (2007). High productivity computing systems project. http://www.highproductivity.org/.

[41] Yelick K, Demmel J, Simon H, Culler D, Bailey D, et al. (2004). Computer Science 267: Parallel Processing Lecture Notes, Lecture 1: Overview of Parallel Computing. http://www.cs.berkeley.edu/ yelick/cs267-sp04/lectures/01/lect01-intro.pdf.

[42] Yelick K, Semenzato L, Pike G, Miyamoto C, Liblit B, et al. (1998) Titanium: A high-performance Java dialect. In: ACM, editor, ACM 1998 Workshop on Java for High-Performance Network Computing. New York, NY 10036, USA: ACM Press.

[43] El-Ghazawi T, Carlson W, Sterling T, Yelick K (2003) UPC: Distributed Shared-Memory Programming. Wiley-Interscience.

[44] Numrich R, Reid J (1998). Co-array fortran for parallel programming.

[45] Introduction to Parallel Programming and MapReduce. `http://code.google.com/edu/parallel/mapreduce-tutorial.html`.

[46] Dean J, Ghemawat S Mapreduce: Simplified data processing on large clusters. pp. 137–150.

[47] Hadoop: A distributed computing platform. `http://lucene.apache.org/hadoop/`.

[48] DeWitt D, Gray J (1992) Parallel Database Systems: The Future of High Performance Database Systems. Commun ACM 35:85–98. doi:http://doi.acm.org/10.1145/129888.129894.

[49] Szalay AS, Kunszt PZ, Thakar A, Gray J, Slutz D, et al. (2000) Designing and mining multi-terabyte astronomy archives: the Sloan Digital Sky Survey. pp. 451–462.

[50] Szalay AS, Gray J, Vandenberg J. Petabyte scale data mining: Dream or reality?

[51] Upson C, Thomas Faulhaber J, Kamins D, Laidlaw DH, Schlegel D, et al. (1989) The application visualization system: A computational environment for scientific visualization. IEEE Comput Graph Appl 9:30–42. doi:http://dx.doi.org/10.1109/38.31462.

[52] Nielson GM, Hagen H, Müller H, editors (1997) Scientific Visualization, Overviews, Methodologies, and Techniques, Dagstuhl, Germany, May 1994. IEEE Computer Society.

[53] Wong PC, Bergeron RD (1994) 30 years of multidimensional multivariate visualization. In: Nielson et al. (52), pp. 3–33.

[54] McCormick BH (1988) Visualization in scientific computing. SIGBIO Newsl 10:15–21. doi:http://doi.acm.org/10.1145/43965.43966.

[55] Simon H (2006). Presentation: Progress in Supercomputing: The Top Three Breakthroughs of the Last 20 Years and the Top Three Challenges for the Next 20 Years. `http://computing.ornl.gov/presentations/simon.pdf`.

[56] Bryson S, Levit C (1992) The virtual wind tunnel. IEEE Comput Graph Appl 12:25–34. doi:http://dx.doi.org/10.1109/38.144824.

[57] Lee JP, Ahlberg C, Carr D, Grinstein G, Kinney J, et al. (2001) Visualization for bio- and chem-informatics: are you being served? In: VIS '01: Proceedings of the conference on Visualization '01. Washington, DC, USA: IEEE Computer Society, pp. 515–518.

[58] Ingrid Remy and Stephen W Michnick (1997) Visualization of Biochemical Networks in Living Cells. In: Proceedings of the National Academy of Sciences.

[59] Max N, Crawfis R, Williams D (1993) Visualization for climate modeling. IEEE Comput Graph Appl 13:34–40. doi:http://dx.doi.org/10.1109/38.219448.

[60] Gordin DN, Edelson DC, Gomez LM (1996) Scientific visualization as an interpretive and expressive medium. In: ICLS '96: Proceedings of the 1996 international conference on Learning sciences. International Society of the Learning Sciences, pp. 409–414.

[61] The Signaling Gateway. `http://www.signaling-gateway.org/`.

[62] Timo Hannay. Berkeley Database Seminar: Database Publishing at Nature. `http://db.cs.berkeley.edu/dblunch-fa2005/timo.ppt`.

[63] Carminati, F, et al (2003) Hepcal ii: Common use cases for a hep common application layer for analysis. Technical report, LHC Grid Computing Project.

[64] (2003). Personal communication with Craig Tull.

[65] Condor-G and DAGMan Hands-On Lab. `http://www.cs.wisc.edu/condor/tutorials/miron-condor-g-dagman-tutorial.%html`.

[66] Condor Manual. Chapter 2.6: Submitting a Job to Condor.

[67] `globus-job-submit` man page. `http://www.globus.org/v1.1/programs/globus-job-submit.html`. Accessed 11/19/03.

[68] Zhao Y, Deshpande PM, Naughton JF (1997) An array-based algorithm for simultaneous multidimensional aggregates. In: Proceedings of the 1997 ACM SIGMOD international conference on Management of data. ACM Press, pp. 159–170. doi:http://doi.acm.org/10.1145/253260.253288.

[69] fv: The Interactive FITS File Editor. `http://heasarc.gsfc.nasa.gov/docs/software/ftools/fv/`. Accessed 10/28/03.

[70] Brun R, Buncic N, Fine V, Goto M, Rademakers F, et al. (1997) ROOT - An Interactive Object Oriented Framework and its application to NA49 data analysis. In: Proceedings of Computing in High Energy Physics.

[71] PAW: Physics Analysis Workstation. `http://wwwasd.web.cern.ch/wwwasd/paw/`. Accessed 10/28/03.

[72] Wong P, Bergeron R (1997). 30 years of multidimensional multivariate visualization.

[73] John Hughes (1985) Lazy memo-functions. Functional Programming Languages and Computer Architecture :129–146.

[74] Sloan digital sky survey. `http://www.sdss.org/`.

[75] Annis, J, Kent S, Castender F, Eisenstein D, et al. (2000) MaxBCG Technique for Finding Galaxy Clusters in SDSS Data . In: AAS 195th Meeting.

[76] Annis J, Zhao Y, Voeckler J, Wilde M, Kent S, et al. (2002) Applying chimera virtual data concepts to cluster finding in the sloan sky survey. In: Supercomputing.

[77] Serge Abiteboul and Richard Hull and Victor Vianu (1995) Foundations of Databases: The Logical Level, Addison-Wesley Longman Publishing Co., Inc., chapter Chapter 20: Complex Values.

[78] Raman V, Raman B, Hellerstein JM (1999) Online dynamic reordering for interactive data processing. In: The VLDB Journal. pp. 709–720.

[79] Olson D, Perl J. PPDG-19: Grid Service Requirements for Interactive Analysis. `http://www.ppdg.net/pa/ppdg-pa/idat/papers/analysis_use-cases-grid-reqs%.pdf`. Access 11/21/03.

[80] Cui Y, Widom J, Wiener JL (2000) Tracing the lineage of view data in a warehousing environment. ACM Transactions on Database Systems 25:179–227.

[81] (1998) Handbook of Mathematics and Computational Science. Springer Verlag.

[82] Wensel S (1988) Postgres reference manual. Technical Report UCB/ERL M88/20, EECS Department, University of California, Berkeley. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/1988/1022.html`.

[83] David T Liu and Michael J Franklin and Devesh Parekh (2003) Demo. GridDB: A Relational Interface to the Grid. In: SIGMOD.

[84] A Bayucash and R L Henderson and C Lesiak and B Mashn and T Proerr and D Tweten (1999) Portable batch system: External reference specification. Technical report, MRJ Technology Solutions.

[85] Scientific Workflows Survey. http://www.extreme.indiana.edu/swf-survey/.

[86] Zhao Y, Voeckler J, Wilde M, Foster I (2002) Chimera: A virtual data system for representing, querying, and automating data derivation. In: 14th Conference on Scientific and Statistical Data Management.

[87] Dagman home page. http://www.cs.wisc.edu/condor/dagman/. Accessed 10/25/03.

[88] Buck J, Ha S, Lee EA, Messerschmitt DG (1994) Ptolemy: A framework for simulating and prototyping heterogenous systems. Int Journal in Computer Simulation 4.

[89] (2005) Contextualised Workflow Execution in MyGrid. In: EGC. pp. 444–453.

[90] Ioannidis YE, Livny M, Ailamaki A, Narayanan A, Therber A (1997) Zoo: a desktop experiment management environment. In: Proceedings of the 22 nd Conference on Very Large Data Bases (VLDB), 1996. pp. 580–583.

[91] Ailamaki A, Ioannidis YE, Livny M (1998) Scientific workflow management by database management. In: Statistical and Scientific Database Management. pp. 190–199.

[92] Wiener JL, Ioannidis YE (1993) A moose and a fox can aid scientists with data management problems. In: Workshop on Database Programming Languages. pp. 376–398.

[93] Anjur V, Ioannidis YE, Livny M (1996) FROG and TURTLE: Visual bridges between files and object-oriented data. In: Proceedings of the Eighth International Conference on Scientific and Statistical Database Management. Stockholm, Sweden: IEEE, pp. 76–85.

[94] (2005). Personal communication with Bonnie Fitzpatrick.

[95] (2005). Personal communication with Xiao Wen.

[96] May JM (2001) Parallel I/O for high performance computing. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[97] Muniswamy-Reddy KK, Wright CP, Himmer A, Zadok E (2004) A versatile and user-oriented versioning file system. In: FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies. Berkeley, CA, USA: USENIX Association, pp. 115–128.

[98] Muniswamy-Reddy K (2003) Versionfs: A Versatile and User-Oriented Versioning File System. Master's thesis, Stony Brook University. Technical Report FSL-03-03, `www.fsl.cs.sunysb.edu/docs/versionfs-msthesis/versionfs.pdf`.

[99] Peterson Z, Burns R (2005) Ext3cow: a time-shifting file system for regulatory compliance. Trans Storage 1:190–212.

[100] CORNELL B, DINDA P, AND F (2004). Wayback: A user-level versioning file system for linux.

[101] Liu D, Franklin M (2004). Griddb: A data-centric overlay for scientific grids.

[102] Buneman P, Khanna S, Tan WC (2001) Why and where: A characterization of data provenance. In: ICDT. pp. 316–330.

[103] Cui Y (2002) Lineage tracing in data warehouses. Ph.D. thesis. Adviser-Jennifer Widom.

[104] Widom J (2005) Trio: A system for integrated management of data, accuracy and lineage. In: CIDR. pp. 262–276.

[105] Program Library HOWTO. `http://www.tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html`.

[106] CVS Home Page. `http://www.nongnu.org/cvs/`.

[107] Michael Pilato (2004) Version Control With Subversion. Sebastopol, CA, USA: O'Reilly & Associates, Inc.

[108] SuperMACHO, A Next Generation Microlensing Survey of the LMC. `http://www.ctio.noao.edu/supermacho`.

[109] Nikolaev S, Cook KH, Stubbs CW, Smith RC, Rest A, et al. (2003) Object-Based Photometry Pipeline for SuperMACHO Project. In: Bulletin of the American Astronomical Society. pp. 1389–+.

[110] Zemla A (2003) Lga: a method for finding 3d similarities in protein structures. Nucleic Acids Research 31:3370–3374.

[111] Liang S (1999) Java(TM) Native Interface: Programmer's Guide and Specification. Prentice Hall PTR.

[112] (Accessed 2/2005). LSST Project Site. http://www.lsst.org/.

[113] BlueArc's Titan Architecture White Paper. http://www.bluearc.com/html/library/downloads/ba_arch_wp.pdf.

[114] Colby LS, Griffin T, Libkin L, Mumick IS, Trickey H (1996) Algorithms for deferred view maintenance. pp. 469–480. URL citeseer.ist.psu.edu/18364.html.

[115] Chandrasekaran S (2005). Query processing over live and archived data streams.

[116] Stonebraker M (1989) The case for partial indexes. SIGMOD Rec 18:4–11. doi:http://doi.acm.org/10.1145/74120.74121.

[117] Seshadri P, Swami AN (1995) Generalized partial indexes. In: ICDE. pp. 420–427. URL citeseer.ist.psu.edu/seshadri95generalized.html.

[118] Baru C, Moore R, Rajasekar A, Wan M (1998). The sdsc storage resource broker. URL citeseer.ist.psu.edu/baru98sdsc.html.

[119] Chervenak AL, Palavalli N, Bharathi S, Kesselman C, Schwartzkopf R (2004) Performance and scalability of a replica location service. hpdc

00:182–191. doi:http://doi.ieeecomputersociety.org/10.1109/HPDC.2004.27.

[120] Singh G, Bharathi S, Chervenak A, Deelman E, Kesselman C, et al. (2003) A metadata catalog service for data intensive applications. sc 00:33. doi:http://doi.ieeecomputersociety.org/10.1109/SC.2003.10020.

[121] d'Orazio L, Jouanot F, Labbé C, Roncancio C (2005) Building adaptable cache services. In: MGC '05: Proceedings of the 3rd international workshop on Middleware for grid computing. New York, NY, USA: ACM Press, pp. 1–6. doi:http://doi.acm.org/10.1145/1101499.1101502.

[122] Cardenas Y, Pierson JM, Brunie L (2005) Uniform distributed cache service for grid computing. In: IEEE, editor, In 16th DEXA: In 2th International Workshop on Grid and Peer-to-Peer Computing Impacts on Large Scale Hereogeneous Distributed Database Systems. IEEE Computer Society, pp. 351–355. URL `http://liris.cnrs.fr/publis/?id=1958`.

[123] Cavanaugh R, Graham G Apples and apple-shaped oranges: Equivalence of data returned on subsequent queries with provenance information. In: Workshop on Data Provenance, 2002.

[124] Liu DT, Franklin MJ (2004) The design of griddb: A data-centric overlay for the scientific grid. In: VLDB. pp. 600–611.

Appendix A

GridDB Language Specification

In this section, we describe a context-free grammar that defines GridDB's declarative language. The grammar is listed in full in Section A.1 and is decomposed into four sections. Section 0 provides a top-level description of the language, specifying that statements can either be Data Definition Language (DDL) statements or Data Manipulation Language (DML) statements (`<ddlStmt>` and `<dmlStmt>` in the `<stmt>` rule). Section 1 defines DDL statements while Section 2 defines DML statements. Section 3 provides auxiliary rules to describe tokens and lists of tokens. Next, we elaborate on Sections 1 and 2, which describe the key ways in which a user interactis with the language.

The top-level rule in section 1 is that which defines `<ddlStmt>`. This rule states that DDL statements can either be type declarations (`<typeDeclaration>`) or function declarations (`<functionDeclaration>`). The syntax for specifying types — opaque, transparent and opaque-transparent — is defined in section 1A, starting with the `<typeDeclaration>` rule. The syntax for

specifycing functions — atomic, composite, and maps — is described in section 1B.

The top-level rule in section 2 is that which defines `<dmlStmt>`. This expands into one of 6 rules, defining statements for each of the 6 types of DML statements: container variable declarations (`<containerDecl>`), container variable bindings (`<containerBindingStmt>`), data procurement statements (`<dataProcurementStmt>`), data viewing statements (`<dataViewingStmt>`), data provenance queries (`<dataProvenanceStmt>`) and computational steering statements (`<compSteeringStmt>`). These statements are further defined in sections 2A through 2F.

Finally, we note the use a shorthand notation for representing "list production" rules. We use this shorthand because list productions occur frequently. The shorthand rule is:

```
<Y> ::= <<X> <DELIM>>+
```

and represents:

```
<Y> ::= <X> | <X> <DELIM> <Y>
```

For example, the following rule:

```
<stmtList> ::= <<stmt> ;>+
```

is equivalent to:

```
<stmtList> ::= <stmt> | <stmt>; <stmtList>
```

A.1 GridDB Declarative Language Grammar

```
//////////////////////////////////////
// SXN 0: Top-Level
<GridDBStmt> ::= <statementList>
<stmtList> ::= <<stmt> ;>+
<stmt> ::= <ddlStmt> | <dmlStmt>

//////////////////////////////////////
// SXN 1: Data Definition Language
<ddlStmt> ::= <typeDeclaration> | <functionDeclaration>;

//////////////////////////////////////
// SXN 1A: Types
<typeDeclaration> ::= <opaqueType> | <transaparentType> | <opaqueTransparentType>
<opaqueType> ::= opaque type <typeName>;
<typeName> ::= IDENTIFIER
<transaparentType> ::= transparent type <typeName> = ( <typeBody> )
<typeBody> ::= <attrList>
<attrList> ::= <<attrDefn> ,>+
<attrDefn> ::= <attrName> : <attrType>
<opaqueTransparentType> ::= type <typeName> = ( <typeBody> )

//////////////////////////////////////
// SXN 1B: FUNCTIONS
<functionDeclaration> ::= <atomicFunction> | <compositeFunction> |
 <mapFunction>
<atomicFunction> ::= atomic <functionHeader> = <atomicFunctionBody>
<functionHeader> ::= fun <functionName>(<formalInputArgList>)
: (<formalOutputArgList>)
<functionName> ::= <IDENTIFIER>
<formalInputList> ::= <<formalArg>,>+
<formalOutputList> ::= <<formalArg>,>+
<formalArg> ::= <formalArgName> : <type>
<formalArgName> ::= <name>
<name> ::= <IDENTIFIER>
<type> ::= <IDENTIFIER>
<atomicFunctionBody> ::=
exec( programSpec = <programSpec> ,
      unfoldSpec = <unfoldSpec> ,
```

```
      foldSpec = <foldSpec> );
<programSpec> ::= ( <codeModule> , <driverProgram> )
<codeModule> ::= <STRING_LITERAL>
<driverProgram> ::= <STRING_LITERAL>
<unfoldSpec> ::= ( <tagSpecList> )
<tagSpecList> ::= <<tagSpec> ,>+
<tagSpec> ::= ( STRING_LITERAL , <qualifiedAttr> )
<qualifiedName> ::= <<name> . >? <name>
<foldSpec> ::= ( <outSpecList> )
<outSpecList> ::= <<outSpec>,>+
<outSpec> ::= ( <transparentFoldSpec> | <opaqueFoldSpec> )
<transparentFoldSpec> ::= <formalArgName> , <fileSetPath>, <adapter>
<fileSetPath> ::= <REGEX>
<adapter> ::= <STRING_LITERAL>
<opaqueFoldSpec> :: = <formalArgName> , <fileSetPath>
<compositeFunction> ::= <functionHeader> = <compositeBody>
<compositeBody> ::= ( < <compStatementList> ; >+ )
<compStatementList> ::= <compStatement>
<compStatement> ::= <nameDeclaration> | <varBinding>
<nameDeclaration> ::= <intermediateVar> : <type>
<intermediateVar> ::= <name>
<varBinding> ::= ( <varList> ) = <expr>
<varList> ::= <<name>,>+
<expr> ::= <qualifiedName> | <functionName>(<exprList>)
<exprList> ::= <<name>,>*
<mapFunction> ::= map(<functionName>,<attrPosnList>)
<attrPosnList> ::= { <numList> }
<numList> ::= <<NUMBER>,>*

////////////////////////////////////////
// SXN 2: DATA MANIPULATION LANGUAGE
<dmlStmt> ::= <containerDecl> |
 <containerBindingStmt> |
 <dataProcurementStmt> |
 <dataViewingStmt> |
 <dataProvenanceStmt> |
 <compSteeringStmt>

////////////////////////////////////////
// SXN 2A: CONTAINER DECLARATIONS
```

```
<containerDecl> ::= <containerName> : <type>
<containerName> ::= <name>

////////////////////////////////////////
// SXN 2B: CONTAINER BINDING
<containerBindingStmt> ::= <outputContainers> = <functionName>
 (<inputContainers>)
<outputContainers> ::= <containerList>
<inputContainers> ::= <containerList>
<containerList> ::= <<containerName>,>+

////////////////////////////////////////
// SXN 2C: Data Procurement
<dataProcurementStmt> ::= INSERT INTO <container> VALUES <tupleSet> |
 <valueSet>
<valueSet> ::= <containerName> = { <valueList> }
<valueList> ::= <numberList> | <stringLiteralList>

////////////////////////////////////////
// SXN 2D: Viewing Data
<dataViewingStmt> ::= SELECT <columnList>
 FROM <fromList>
 WHERE <predicate>
<fromList> ::= <<fromAtom>,>+
<fromAtom> ::= <container> | autoview(<containerList>)
<columnList> ::= <<attrHandle>,>+
<attrHandle> ::= <containerName>.<attrName>
<predicate> ::= <predicateAtom> | <predicate> <binaryLogicalOp> |
 NOT <predicate>
<binaryLogicalOp> ::= AND | OR
<predicateAtom> ::= <attrHandle> <OP> <VALUE>
<OP> ::= < | > | = | <= | >= | <>

////////////////////////////////////////
// SXN 2E: Data Provenance
<dataProvenanceStmt> ::= provenance <entID>
<entID> ::= <NUMBER>

////////////////////////////////////////
// SXN 2F: Computation Steering
```

```
<compSteeringStmt> ::= UPDATE <fromAtom> SET PRIORITY = NUMBER WHERE
<predicate>

////////////////////////////////////
// SXN 3: AUXILIARY
<IDENTIFIER> ::= [a-zA-Z][a-zA-Z0-9]*
<STRING_LITERAL> ::= " [a-zA-Z0-9]* "
<REGEX> ::= / [a-zA-Z0-9/+*?()]+ /
<NUMBER> ::= [0-9]+
<numberList> ::= <<NUMBER>, >+
<stringLiteralList> ::= <<STRING_LITERAL>, >+
<VALUE> ::= <STRING_LITERAL> | <NUMBER>
```