GridDB: A Data-Centric Overlay for Scientific Grids

David T. Liu Michael J. Franklin

UC Berkeley, EECS Dept.
Berkeley, CA 94720, USA
{dtliu,franklin}@cs.berkeley.edu

Abstract

We present GridDB, a data-centric overlay for scientific grid data analysis. In contrast to currently deployed process-centric middleware, GridDB manages data entities rather than processes. GridDB provides a suite of services important to data analysis: a declarative interface, type-checking, interactive query processing, and memoization. We discuss several elements of GridDB: workflow/data model, query language, software architecture and query processing; and a prototype implementation. We validate GridDB by showing its modeling of real-world physics and astronomy analyses, and measurements on our prototype.

1 Introduction

Scientists in fields including high-energy physics, astronomy, and biology continue to push the envelope in terms of computational demands and data creation. For example, the ATLAS and CMS high-energy physics experiments are both collecting and analyzing one petabyte (10^{15} bytes) of particle collision data per year [13]. Furthermore, continuing advances in such "big science" fields increasingly requires long-term, globe-spanning collaborations involving hundreds or even thousands of researchers. Given such large-scale demands, these fields have embraced *grid computing* [25] as the platform for the creation, processing, and management of their expermental data.

1.1 From Process-Centric to Data-Centric

Grid computing derives primarily from two research domains: cluster-based metacomputing [31, 15] and distributed cluster federation [22]. As such, it has inherited a *process-centric* approach, where the software infrastructure is focused on the management of program invocations (or *processes*). Process-centric grid middleware enables users to submit and monitor jobs (i.e., processes). Modern grid software (e.g., Globus[24] and Condor [31]) also provides additional services such as batching, resource allocation, process migration, etc. These systems, however, provide a fairly low-level, OS-like interface involving imperative programs and files.

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

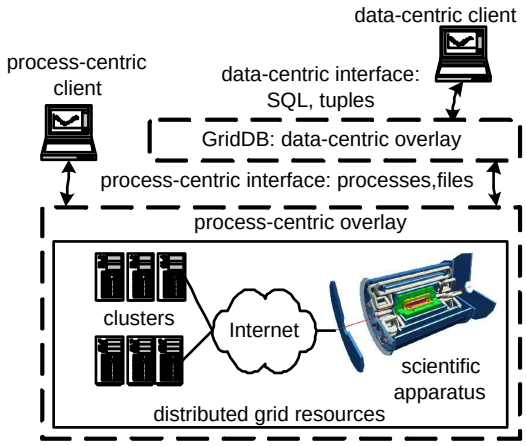


Figure 1: Grid Access Overview

The process-centric approach is a direct extension of the techniques used by scientists in the past. But, with the increasing complexity, scope, and longevity of collaborations and the continuing growth in the scale of the scientific endeavour, it has become apparent that new tools and paradigms are needed. Furthermore, the widespread popularity of interactive processing in many domains has led to a desire among scientists for more interactive access to grid resources. As a result, a number of large, multi-disciplinary efforts have been started by scientists to define the next generation of grid models, tools and infrastructure. These include the Grid Physics Network (GriPhyN) [28], the International Virtual Data Grid Observatory [14], the Particle Physics Data Grid [34], and the European Union DataGrid [47].

GriPhyN, in particular, is built around the notion of "Virtual Data" [51], which aims to put the concept of "data" on an equal footing with that of "process" in grid computing. Our GridDB project, which is being done in the context of GriPhyN, elevates the importance of data one step further, by proposing a *data-centric* view of the grid.

As illustrated in Figure 1, GridDB provides a veneer, or *overlay*, on top of existing process-centric grid services that enables clients to create, manage, and interactively access the results of grid computations using a query language interface to manipulate tables of input parameters and results. The benefits of GridDB include:

- **Declarative Interface**: Scientific computing is a data-intensive task. The benefits of declarative interfaces for such tasks are well-known in the database field, and include: ease of programming, resilience to change, and support for transparent, system-directed optimization.
- **Type Checking**: In contrast to process-centric ap-

proaches, which have little or no knowledge of data types, GridDB provides a language for describing data types and function signatures that enables a number of features including more timely error reporting, and support for code sharing [19, 36].

- **Interactive Query Processing**: Scientific computing jobs are often long running. The batch-oriented mode of interaction supported by existing process-centric middleware severely limits scientists' ability to observe and steer computations based on partial results [35, 18]. GridDB's data-centric interface directly supports *computational steering*, giving scientists more effective control over the use of grid resources.

- **Memoization Support**: A key feature advocated by many current grid efforts is the minimization of resources wasted due to the recomputation of previously generated data products [28]. GridDB's data-centric approach provides the necessary infrastructure for supporting such *memoization* [30].

- **Data Provenance**: Because grid data production will entail promiscuous, anonymous, and transparent resource sharing, scientists must have the ability to retroactively check information on how a data product was created [51, 28]. GridDB's model of function-based data processing lends itself well towards tracking the lineage, or *provenance*, of individual data products.

- **Co-existence**: Perhaps most importantly, GridDB provides these benefits by working *on top* of process-centric middleware, rather than replacing it. This allows users to continue to employ their existing (or even new), imperative data processing codes while selectively choosing which aspects of such processing to make visible to GridDB. This approach also enables incremental migration of scientific workflows into the GridDB framework.

1.2 Contributions and Overview

In this paper, we describe the design, implementation and evaluation of GridDB, a data-centric overlay for scientific grid computing. GridDB is based on two core principles: First, scientific analysis programs can be abstracted as typed functions, and program invocations as typed function calls. Second, while most scientific analysis data is not relational in nature (and therefore not directly amenable to relational database management), 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. We use this principle to provide users with a SQL-like interface to grid analysis along with the benefits of data-centric processing listed previously.

Following these two principles, we have developed a grid computing workflow and data model, the Functional Data Model with Relational Covers (FDM/RC), and a schema definition language for creating FDM/RC models. We then developed GridDB, a software overlay that models grid analyses in the FDM/RC. GridDB exploits the FDM/RC's modeling of both workflow and data to provide new services previously unavailable to scientists. We demonstrate its usefulness with two example data analysis workflows taken from a High Energy Physics experiment and an Astronomy survey, and report on experiments that examine the benefits of GridDB's memoization and compuational steering features.

2 High-Energy Physics Example

In this section we introduce a simplified workflow obtained from the ATLAS High-Energy Physics experiment [18, 21]. We refer to this workflow as `HepEx` (High Energy Physics Example) and use it as a running example[1].

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 response of the new and old simulations for various physics events. A workflow achieving these comparisons is shown in Fig. 2(a). It consists of three programs: an event generator, `gen`; the fast simulation, `atlfast`; and the original, slower simulation, `atlsim`. `gen` is called with an integer parameter, $pmas$, and creates a file, $\langle pmas \rangle$`.evts` that digitally describes a particle's decay into subparticles. $\langle pmas \rangle$`.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 [2]. All three programs are long-running, and compute-bound, thus requiring grid processing.

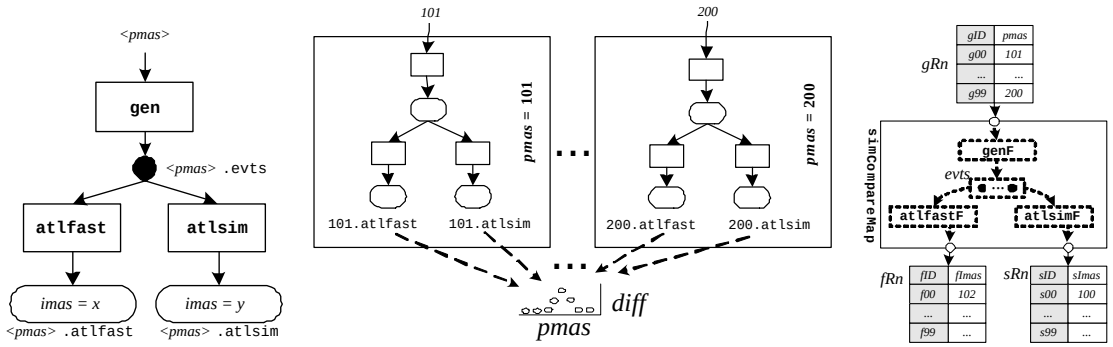
Before describing GridDB, it is useful to examine how `HepEx` would be deployed in a process-centric system. We identify three types of users who would contribute to such a deployment: *coders*, who write programs; *modelers*, who compose these programs into analysis workflows; and *analysts*, who execute workflows and perform data analysis.

To deploy `HepEx` , *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, in perl or some other procedural language[1]. Each program execution is represented by a *process specification* (proc-spec) file, which contains a program, a command-line to execute the program, and a set of input files [2, 5]. 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 extremely effective in presenting a single-machine interface to the grid. Thus, the *analyst* works as if he/she is submitting jobs on a single (very powerful) machine and the grid middleware handles the execution and management of the jobs across the distributed grid resources. The *analyst* creates a *grid job* by executing another script that invokes the AWF script mul-

[1]The GridDB implementation of a more complex scientific workflow is described in Section 6.

[2]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.



(a) (b) (c)

Figure 2: (a) `HepEx` abstract workflow, (b) `HepEx` grid job, (c) GridDB's `simCompareMap` replaces(a) and (b)

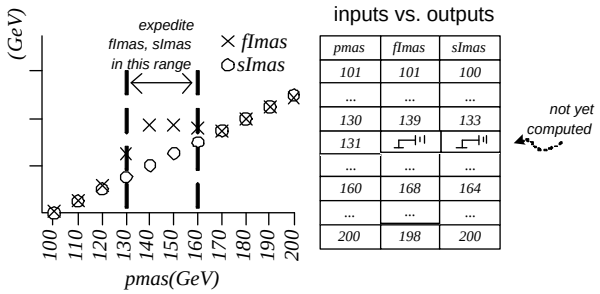


Figure 3: Interactive Query Processing

tiple 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. Fig. 2(b) shows the `HepEx` grid job consisting of these invocations.

3 Data Analysis With GridDB

In the previous section, we identified three roles involved in the deployment of grid 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 grid users. In contrast, the *modeler* sees a major change: instead of encoding the AWF in a procedural script, he encodes it with a schema definition language, conveying workflow and data structure information to GridDB, thus empowering GridDB to provide data-centric services. The *analyst*'s interaction with the grid is also changed dramatically.

We describe the workflow and data model and schema definition language used by the *modeler* in detail in Section 4. Here, we focus on the *analyst*'s interactions using GridDB's data manipulation language (DML).

3.1 Motivating Example: IQP

To illustrate the benefits of data-centric grid analysis, we describe how Interactive Query Processing (IQP) can provide increased flexibility and power to the data analysis process.

Recall from Section 1 that GridDB provides a relational interface for data analysis; for example, the table on the right-side of Fig. 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 the figure, we show streaming partial results in the table's corresponding scatter plot, which 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 phenomenom that needs investigation. Using IQP, an analyst can prioritize data in this range simply by selecting the relevant portion of the x-axis (between the dashed lines) and prioritizing it with a GUI command. GridDB is capable of expediting the minimal set of computations that materialize the points of interest. Through this graphical, data-centric interface, the user drives grid process execution. In contrast, users of process-centric middleware usually run jobs in batch (consider the term "batch" scheduler used to describe many process-centric middlewares).

GridDB's workflow and data model enables it to provide such data-centric services, which are unavailable in process-centric middleware.

3.2 GridDB Modeling Principles

Before walking through an analyst's interaction with GridDB, we describe GridDB's modeling principles and provide a conceptual model for `HepEx`. The GridDB model rests on two main principles: (1) programs and workflows can be represented as *functions*, and (2) an important subset of the data in a workflow can be represented as relations. We refer to this subset as the *relational cover*.

Representing programs and workflows as typed functions provides GridDB with the knowledge necessary to perform type checking of program inputs and outputs and enables a functional representation for AWFs — composite functions. Note, however, that this does not require a change to the programs themselves, but rather, consists of wrapping programs with functional definitions, as we describe in Section 4. In contrast, the process-centered approach uses opaque command-line strings, thereby depriving the middleware of any knowledge of data types for type checking.

In terms of data, 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 pro-

```
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 1: DML for analysis in HepEx

curement, as described in Section 2: 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. 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 [49]. 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[4] and *root* or *paw* in high energy physics[16, 6]. Within these tools, scientists manipulate — with operations like *projection* and *selection* — workflow data to generate multidimensional, multivariate graphs [48]. Such graphs — including scatter plots, time-series plots, histograms, and barcharts — are fundamentally visualizations of relations.

Figure 2(c) shows a HepEx model using these two principles[3]. 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.

3.3 Data Manipulation Language

Having described the main modeling principles, we now describe the data manipulation language for *analyst* interaction. With GridDB, analysts first create their own computational "sandbox", in which to perform their analyses. This sandbox consists of private copies of relations to be populated and manipulated by GridDB. Next, the analyst specifies the analysis 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 1. Line 1 declares the sandbox relations. For example, gRn is declared with a type of set(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, has already declared simCompareMap, with the signature: set(g) $\rightarrow$ set(f) $\times$ set(s), an action we show in Section 4.3.2.

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 submission of procspecs for grid execution. This is conceptually similar to the execution represented by Fig. 2(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.

[3]This model is created by schema definition commands written by the modeler, as we describe in Section 4

Analysts commonly seek to discover relationships between different physical quantities. To support this analysis 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 5.1.2. The autoview mechanism also plays in an important role in the provision of IQP, as is discussed in Section 7.1.

3.4 Summary

To summarize, GridDB provides a SQL-like DML that allows users to initiate grid analysis tasks 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" also play an important role in supporting Interactive Query Processing. By understanding both workflow and data, GridDB provides data-centric services unavailable in process-centric middleware.

4 Data Model: FDM/RC

The previous section illustrated the benefits of GridDB data analysis provided that the modeler has specified a workflow and data schema to GridDB. In this section, we describe a model and data definition language for these schemas. The model is called the Functional Data Model with Relational Covers(FDM/RC) and has two main constructs: *entities* and *functions*. Functions, which map from entities to entities, and can be composed together and iterated over, are used to model workflows. Data entities may be modeled opaquely as blobs, or transparently as tuples, providing the modeling power of the relational model. By modeling both workflows and their data (as tuples), the FDM/RC enables GridDB to provide not only services found in either workflow and database systems, but also services not found in either.

4.1 Core Design Concepts

We begin our discussion with two concepts: (1) the inclusion of *transparent* and *opaque* data and (2) the need for *fold/unfold* operations.

4.1.1 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, as GridDB interacts 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 Fig. 2(a) is modeled as a transparent entity: it can be modeled

as a tuple with one integer attribute, $pmas$. Knowing the contents of this data, that the input is a tuple with an integer attribute, GridDB can perform type-checking on `gen`'s inputs. Other services enabled by transparent data are a declarative SQL interface and IQP based on the declarative interface.

On the other hand, there are times when a modeler wants GridDB to catalog a file, but has neither the need for enhanced services, nor the resources to describe data semantics. GridDB allows lighter weight modeling of these entities as *opaque* objects. As an example, consider an output file of `gen`, `x.evt`. The entity is used to execute programs `atlfast` and `atlsim`. GridDB must catalog and retrieve the file for later use, but the modeler does not need extra services.

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

4.1.2 Unfold/Fold

To provide a well-defined interface for grid programs, GridDB allows a modeler to wrap programs behind typed function interfaces. 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 programs outputs files, upon termination, must map to the functions return values. These two mappings are defined by the *fold* and *unfold* operations, respectively.

In Section 4.3.2, we describe *atomic* functions, which encapsulate imperative programs and employ fold and unfold operations. In Section 4.3.3, we elucidate the operations with an example.

4.2 Definition

Having described the core concepts of our grid data model, we now define it:

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[4]. 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.

4.3 Data Definition Language

An FDM/RC schema, as we have just described, is defined by a data definition language (DDL). The DDL is divided into type and function definition constructs. In this section, we describe the constructs, and illustrate them with `HepEx`'s data definition, when possible. Its DDL is shown in Listing 2.

4.3.1 Types

As suggested in Section 4.1.1, 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 prefixed 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.

4.3.2 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 header 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 4.1.2, *atomic functions* embody grid 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 tranforming 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`(function headers in Listing 2, at Lines 12-14). Because the body of an atomic function definition is quite involved, we defer its discussion to Section 4.3.3.

Composite functions are used to express complex anal-

[4] one of τ_t and τ_o can be null, but not both

yses, 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`. `sOut`, 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, consider the following statement, which 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 grid machinery.

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 6, we will show an SQL workflow function that simplifies a spatial computation with a spatial "overlaps" query, as used in an actual astronomy analysis.

4.3.3 Fold/Unfold Revisited

Finally, we return to defining atomic functions and their *fold* and *unfold* operations. Recall from Section 4.1.2 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 maps to. The second and third arguments are lists that specify the *unfold* and *fold* operations for each input or output

```
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 2: Abridged `HepEx` DDL

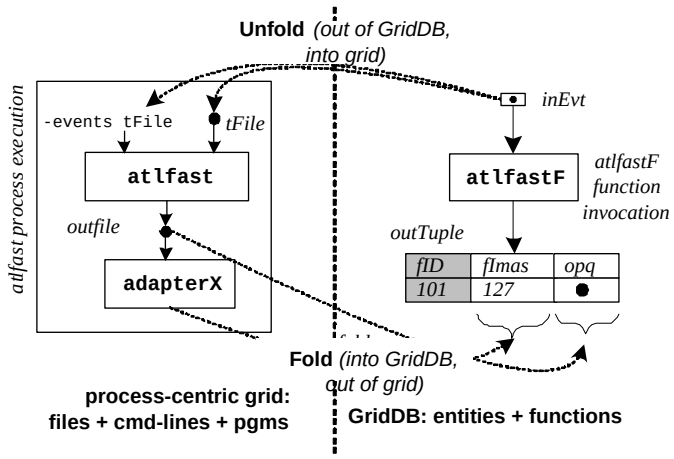


Figure 4: Unfold/Fold in *atomic* functions

entity.

The second argument is a list of pairs (only one here) which specifies how arguments are unfolded. In this case, because evtsIn 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 Fig. 4).

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 Fig. 4). 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

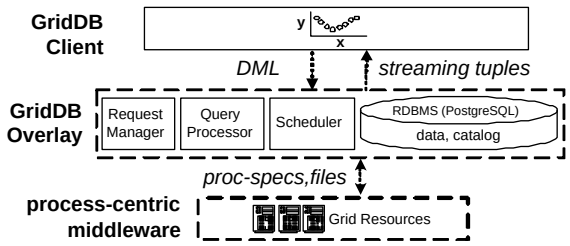


Figure 5: GridDB's Architecture

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.

Object-to-file mappings have previously been studied in the ZOO project [10].

5 Design and Implementation

In this section, we discuss the design of GridDB, focusing on the query processing of *analyst* actions, as embodied in DML statements.

GridDB's software architecture is shown in Fig. 5. 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* [44]) stores and manipulates data, as well as the system catalog.

In the rest of this section, we describe how GridDB processes DML statements. We do not discuss the processing of schema definition statements, as they are straightforward updates to the system catalog (similar to data definition statements of relational databases).

5.1 Data Representation and Query Processing

Our general implementation strategy is to translate DML statements into SQL, re-using a pre-existing relational query processor for most processing. One consequence of this strategy is that our 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.

5.1.1 Tabular Data Structures

GridDB uses three kinds of tables: two for entities and function mappings and the last for process state. We describe these three in turn.

Entity Tables: Recall from Section **??** 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 a bitstream. 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* [30]. We describe the table of a function with one input and one output, $F : X \rightarrow Y$, as the description easily extends to functions with multiple inputs and outputs. F 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 IDs for 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.

Process Table: As described in Section 4.3.3, function evaluations are resolved through process process invocation. Process invocations are stored in a table with the following attributes: $(PID, FID, funcName, priority, status)$. PID is a unique process ID; FID is a foreign key to the memo table of function $funcName$, representing the process' associated function evaluation; $priority$ is the process' priority, used for computational steering; and $status$ is one of *done*, *running*, *ready*, or *pending*.

5.1.2 DML Query Processing: Translation to SQL

Having represented entities, functions and processes in RDBMS tables, query processing can proceed by translating the DML to standard SQL.

In this section, we describe how each *analyst* action is processed and show, as an example, query processing for the `HepEx` analysis. The internal data structures for `HepEx` are shown in Fig. 6. The diagram is an enhanced version of the analyst's view of Fig. 2(c).

Recall the three basic analyst actions from Section 3: *workflow setup* creates sandbox entity-sets and connects them as inputs and outputs of a map; *data procurement* is the submission of inputs to the workflow, triggering function evaluations to create output entities. Finally, streaming partial results can be perused with *automatic views*. We repeat the listing 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, tables are created for the entity-sets and workflow functions. At this step, GridDB also stores a *workflow DAG* for the analysis. In the example, workflow setup (Lines 1-2) creates a table for each of the four entity-sets (`gRn`, `fRn`, `sRn`, `evts`), as well as each of the three functions ($genFMemo, atlfastFMemo, atlsimMemo$). An internal data structure stores the workflow DAG, represented by the solid arrows between tables.

Data procurement and Process Execution

Data procurement is performed with an `INSERT` statement into a map's input entity-set variables. In GridDB, an `INSERT` into an entity table connected to a function triggers evaluations of that function. Function outputs are ap-

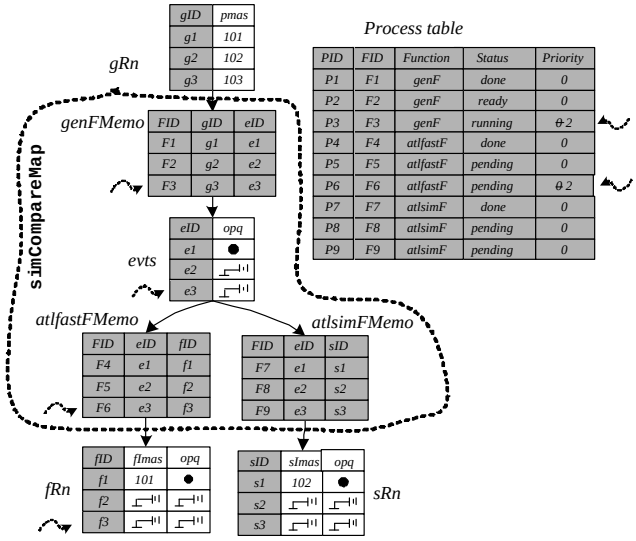


Figure 6: Internal data structures representing `HepEx` functions, entities, and processes. Shaded fields are system-managed. Dashed arrows indicate interesting tuples in our IQP discussion (Sec. 7.1)

pended 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 4.3.3. To summarize: first, the function's input entities are converted to files and a command-line string using the *unfold* operation; second, the process (defined by program, input files and command-line) is executed on the grid; and third, the *fold* operations ingest the process' output files into GridDB entities.

In the example, 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 Fig. 6 indicates the completion of three processes ($P1$, $P4$, $P7$), whose results have been folded back into entity tables (entities $e1$, $f1$, $s1$, 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 them can be created by joining all function- and entity-tables on the path.

In Fig. 7, 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, non-opaque attributes.

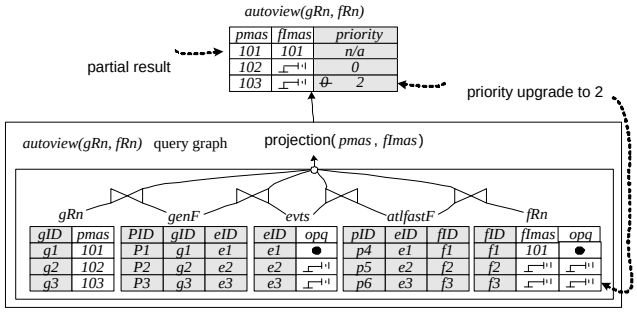


Figure 7: `autoview(gRn, fRn)`

6 `ClustFind`: A Complex Example

Up until this point, we have demonstrated GridDB concepts using `HepEx`, a rather simple analysis. In this section, we describe how GridDB handles a complex astronomy application. First, we describe the application science. Next, we describe how the analysis can be modeled as an FDM/RC schema. Finally, we describe how `ClustFind` is enhanced by memoization and data-centric computational steering.

6.1 The Science: Finding Clusters of Galaxies

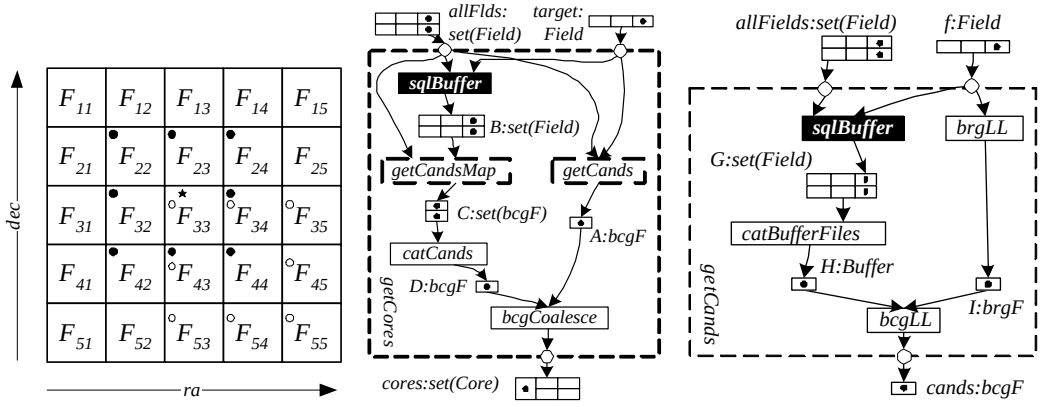
The Sloan Digital Sky Survey (SDSS) [7] is a 12 TB digital imaging survey mapping 250,000,000 celestial objects with two orders of magnitudes greater sensitivity than previous large sky surveys. `ClustFind` is a computationally-intense SDSS analysis that detects galaxy clusters, the largest gravitation-bound objects in the universe. The analysis uses the MaxBCG cluster finding algorithm [11], requiring 7000 CPU hours on a 500 MHz computer [12].

In this analysis, all survey objects are characterized by two spatial coordinates, *ra* and *dec*. All objects fit within a two-dimensional mesh of fields such that each field holds objects in a particular square of (ra, dec)-space (Fig. 8(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 Fig. 8(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 Fig. 8(a), each field in the buffer of F_{33} is annotated with a $\bullet$). Finally, the algorithm applies a core selection algorithm, which selects cores from the target candidates based on interactions with buffer candidates and other core candidates.

6.2 An FDM/RC Schema for `ClustFind`

In this section, we describe the top-level `ClustFind` workflow as an FDM/RC function, `getCores`. Given a target field entity, `getCores` returns the target's set of cores. `getCores` is shown as the outermost function of Fig. 8(b). The analysis would actually build a map function using `getCores` as its body, in order to find cores for a set of 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 candidates, D, and target candidates, A, returning a file of target cores, `cores`. During



(a) (b) (c)

Figure 8: (a) `ClustFind` divides sky objects into a square mesh of buckets in (ra, dec) space. (b) `getCores`, the body of the top-level `ClustFind` map function. (c) `getCands`, a composite subfunction used in `getCores`.

the fold operation, `cores` is ingested as a set of `Core` entities (shown at the bottom of Fig. 8(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*[39]. We have decided that even though such a model provides sophisticated support for set types, its added complexity is not justified in the system.

In Fig. 8(c), we show `getCands`, a function used in `getCores`, standalone and also as the body of `getCandsMap`. 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 Fig. 8(a). Note that the `bcgLL` function within `getCands` is the most expensive function to evaluate [12], making `getCands` the bottleneck in `getCores`.

The analysis touches upon 2 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 grid process executions, are the solid boxes.

6.3 Memoization & IQP in `ClustFind`

Embedded in our description is this fact: `getCands` is called ten times per field. `getCands` is called twice in computing the field's cores and once each for computing the cores of its eight neighbors. By modeling this workflow in a GridDB schema, an astronomer automatically gains the performance of memoization, without needing to implement it himself.

Finally, it is common for astronomers to point to a spot on an image map — for instance, the using SkyServer interface[7] — and query for results from those coordinates. These requests translate to (ra, dec) coordinates, and are accomodated by GridDB's data-driven prioritization of interesting computations.

7 Performance Enhancements

Previous sections have shown how GridDB provides basic services. In this section, we show that GridDB's model serves as a foundation for two other performance-enhancing services: Interactive Query Processing and Memoization. We describe these two services and validate their benefits using a prototype GridDB implementation.

7.1 Interactive Query Processing

Due to the conflict between the long-running nature of grid jobs and the iterative nature of data analysis, scientists have expressed a need for data-centric computational steering (IQP) [18, 35].

In this section, we describe how the FDM/RC enables IQP through a relational interface. We introduce IQP with an example. Consider the autoview at the top of Fig. 7. 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$= 103$) using an SQL `UPDATE` statement:

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

By defining a relational cover, GridDB allows prior-

itization of data, rather than processes. In GridDB, the `UPDATE` statement is enhanced; one can update a special `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 [30]. Any view tuple can always be traced back to entities of the relational cover, using basic data lineage techniques [50]. Each entity also has a functional expression, which encodes its computational recipe. Since function evaluations are associated with process execution, GridDB can prioritize only the minimal process executions, delaying the computation of other, irrelevant computations.

As an example, consider the processing of the prioritization request in Fig. 7. The only missing uncomputed attribute is *fImas*, which is derived from from relational cover tuple $f3$. Fig. 6 (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, have provided a data-centric interface for computational steering.

7.2 Memoization

Recall from Section 5.1.1 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 analysis programs are often deterministic, as repeatibility is paramount to computational science [8]. 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.

7.3 Implementation

We have implemented a java-based prototype of GridDB, consisting of almost 19K lines of code. Modular line counts are in Table. 1. The large size of the client is explained by its graphical interface, which we implemented for a demonstration of the system during SIGMOD 2003 [32]. Currently, the system uses condor [31] as its process-centric middleware; therefore, it allows access to a cluster of machines. In the future, we plan to use Globus, in order for the system to leverage distributively-owned computing resources. The change should not be conceptually different, as both provide the same process-centric interface.

7.4 Validation

To demonstrate the effectiveness of IQP and memoization, we conducted validation experiments with our prototype implementation and the cluster testbed of Fig. 9. Measurements

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 1: LOCs for a java-based GridDB prototype.

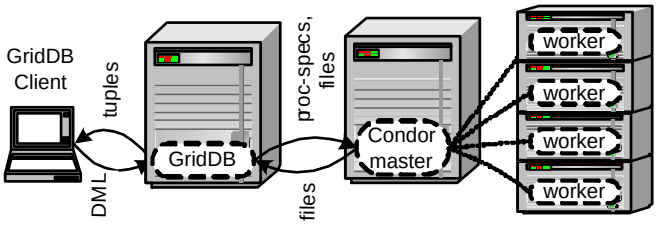


Figure 9: Experimental setup.

were conducted on a miniature "grid" consisting of six nodes (Fig. 9). The GridDB client issued results from a laptop while the GridDB overlay, a "Condor Master" batch scheduler [31] 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 Mbps network.

7.4.1 Validation 1: IQP

In the first validation experiment, we show the benefits of IQP 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, inserting 100 values of $pmas$ into `simCompareMap`. 200 hundred seconds after submission, we inject a IQP request, prioritizing 25 as yet uncomputed `f` tuples:

```
UPDATE autoview(gRn, fRn) SET PRIORITY = 2
WHERE 131 ≤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 Fig. 10, we plot *Number of Prioritized Data Points* returned vs. *time*. In the plot, GridDB(dynamic) has delivered all 20 interesting results. The figure shows that *dynamic* has delivered all 20 interesting results within 1047s. 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 7.1.

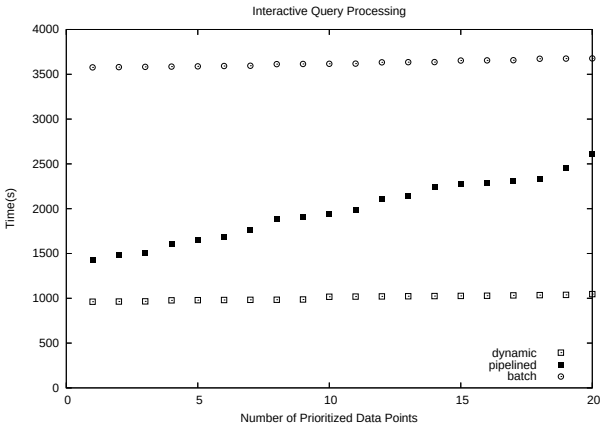


Figure 10: Validation 1, IQP for `HepEx`

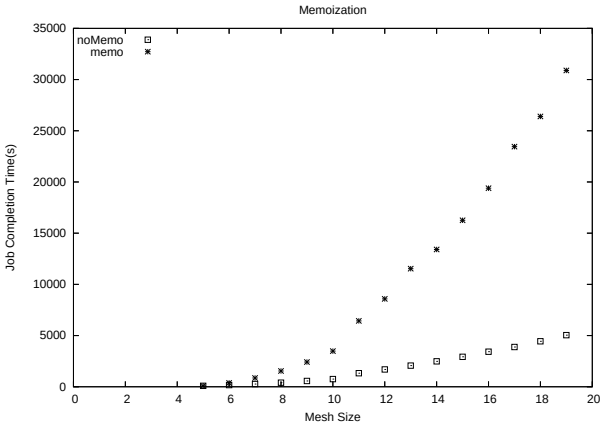


Figure 11: Validation 2, Memoization in `ClustFind`

7.4.2 Validation 2: Memoization Speedup

We validated the GridDB memoization implementation by testing how well it exploits `ClustFind` memoization opportunities (from Section 6). 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. 8(a). Each field was of length 0.1×0.1 degrees. Recall from Section 6.3 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. 11, 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. Analysis using Amdahl's law dictates an upper bound of 7, when exploiting memoization.

To precisely understand these performance gains, we profiled the `ClustFind` analysis, recording the amount of time spent in the four main modules of Fig. 8(b): `sqlBuffer`, `getCands`(and `getCandsMap`) , `catCands`, and `bcgCoalesce`. We discovered 95% of the execution time was spent in `getCands`. Our profiling showed that memoization reduced the time calling `getCands` by up to 86%, where 90% is optimal (each call is made a maximum of 10 times, so a 10 to 1 reduction is optimal).

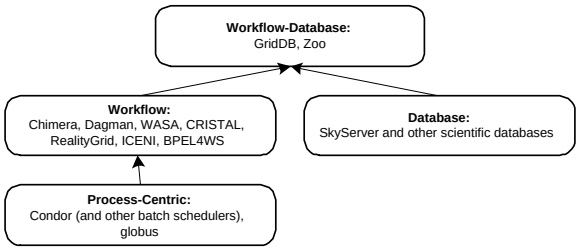


Figure 12: A categorization of related systems.

8 Related Work

We classify related systems into four categories: *Process-Centric*, *Workflow*, *Database* and *Workflow-Database*. In this section, we summarize the salient features of each category, deferring discussions of individual systems to [33]. Using this classification, we argue that by combining the models of both *Workflow* and *Database* systems, GridDB not only provides the feature-sets of both systems, but also provides features absent from both categories.

The four categories are related in Figure 12, where an arrow from category A to category B indicates that B derives from A. *Process-centric* systems such as Condor [31] and Globus [24] provide an OS-like process submission interface. *Workflow* systems, such as WASA [45], CRISTAL [20], chimera [12, 51] and DAGMan [3], focus on the management of processes that create data, but typically do not exploit the structure of the data created by their workflows. Naturally, *Workflow* systems build on top of process-centric systems. On the other hand, *Database* systems used for scientific computing, such as SQL Server (used to build SkyServer [42, 43]), model and manipulate data, the final product of workflows, but do not provide workflow management facilities.

Finally, *Workflow-Database* systems model both workflows and their data. GridDB, along with ZOO [29, 9], fit into this category. To our knowledge, ZOO is the first, and only other system to incorporate workflow and data modeling into a scientific analysis framework. GridDB and ZOO share many similarities; for example, ZOO also provides a data model and query language [46], and maps between files and objects [10].

However, we have identified three main differences: (1) GridDB is focused on the grid environment rather than the desktop environment, (2) GridDB uses the simpler relational model, which we argue is sufficient for input and output modeling (Section 3), and (3) GridDB exploits synergy between workflow and data modeling, providing features unavailable in ZOO, such as IQP.

An alternative parallel programming model is represented by MPI [23] and MPI-based libraries, such as Master-

Worker[26]. These programming models can increase the execution speed of an individual process by running it on multiple machines, but stipulate a lower-level programming model when compared to any of the *Workflow*, *Database*, or *Workflow-Database* systems described above.

Finally, GridDB builds upon the Functional Data Model (FDM) [40, 41, 17] and relational Interactive Query Processing (IQP) [37, 38, 27]. We expand the use of the FDM to model grid workflows and extend previous IQP systems by steering grid computation, rather than the relational queries of previous work.

9 Conclusion

In this paper, we have presented GridDB, a software overlay that provides data-centric services for scientific grid computing. We exploit 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 relational interface to the full data set. As such, we have built a workflow and data model (FDM/RC) and query language for representing workflows and accessing their data through a relational interface. Additionally, GridDB exploits the synergy between workflow and data modeling to provide previously unavailable data-centric grid services. Finally, we have demonstrated the use of GridDB in modeling High Energy Physics and Astronomy analyses and have validated our ideas by measuring a prototype implementation.

10 Acknowledgements

We thank many collaborators from the GriPhyN, ATLAS and SDSS projects and other members of the UC Berkeley Database Group. This work was funded in part by the NSF under ITR grants SCI-0086044 and SI-0122599, by the IBM Faculty Partnership Award program, and by research funds from Intel, Microsoft, and the UC MICRO program.

References

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

[2] Condor Manual. Chapter 2.6: Submitting a Job to Condor. Available at `http://www.cs.wisc.edu/condor/manual/`.

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

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

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

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

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

[8] *Handbook of Mathematics and Computational Science*. Springer Verlag, 1998.

[9] A. Ailamaki, *et al.*. Scientific workflow management by database management. In *Statistical and Scientific Database Management*, pp. 190–199. 1998.

[10] V. Anjur, *et al.*. FROG and TURTLE: Visual bridges between files and object-oriented data. In *Proceedings of the Eighth International Conference on Scientific and Statistical Database Management*, pp. 76–85. IEEE, Stockholm, Sweden, 18–20 1996.

[11] Annis, *et al.*. MaxBCG Technique for Finding Galaxy Clusters in SDSS Data . In *AAS 195th Meeting*. 2000.

[12] J. Annis, *et al.*. Applying chimera virtual data concepts to cluster finding in the sloan sky survey. In *Supercomputing*. 2002.

[13] Grid Physics Network High-Energy Particle Physics Description. `http://www.griphyn.org/projinfo/physics/highenergy.php`. Accessed 11/19/03.

[14] P. Avery. iVDGL ITR proposal: An International Virtual-Data Grid Laboratory for Data Intensive Science. `http://www.phys.ufl.edu/~avery/ivdgl/itr2001/proposal_all.pdf`, 2001.

[15] A. Bayucash, *et al.*. Portable Batch System: External Reference Specification. Technical report, MRJ Technology Solutions, November 1999.

[16] R. Brun, *et al.*. ROOT - An Interactive Object Oriented Framework and its application to NA49 data analysis. In *Proceedings of Computing in High Energy Physics*. May 1997.

[17] P. Buneman *et al.*. FQL–A Functional Query Language. In *ACM SIGMOD International Conference on Management of Data*. May 1979.

[18] Carminati, F., et al. HEPCAL II: Common Use Cases for a HEP Common Application Layer for Analysis. Technical report, LHC Grid Computing Project, 2003.

[19] Charles W. Krueger. Software Reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[20] Concurrent Repository Information. CRISTAL. URL `citeseer.ist.psu.edu/417078.html`.

[21] 2003. Personal communication with Craig Tull.

[22] K. Czajkowski, *et al.*. A resource management architecture for metacomputing systems. *LNCS*, 1459, 1998.

[23] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[24] I. Foster *et al.*. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[25] I. Foster, *et al.*. The anatomy of the grid. In *International Journal of Supercomputer Applications*. 2001.

[26] J.-P. Goux, *et al.*. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *HPDC*, pp. 43–50. 2000.

[27] J. M. Hellerstein, *et al.*. Informix under control: Online query processing. In *Data Mining and Knowledge Discovery 4(4)*, pp. 281–314. 2000.

[28] Ian Foster *et al.*. Grid Physics Network (GriPhyN) White Paper, 2003. `http://www.griphyn.org/`.

[29] Y. E. Ioannidis, *et al.*. Zoo: a desktop experiment management environment. In *Proceedings of the 22 nd Conference on Very Large Data Bases (VLDB), 1996*, pp. 580–583. 1997.

[30] John Hughes. Lazy memo-functions. *Functional Programming Languages and Computer Architecture*, (201):129–146, September 1985.

[31] M. Litzkow, *et al.*. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*. 1988.

[32] D. T. Liu, *et al.*. Demo. GridDB: A Relational Interface to the Grid. In *SIGMOD*. 2003.

[33] D. T. Liu *et al.*. GridDB: Data-Centric Services in Scientific Grids. Technical report, UC Berkeley, EECS Department, March 2004. UCB//CSD-04-1311. Available at `http://www.cs.berkeley.edu/~dtliu/pubs/griddb_tr.pdf`.

[34] M. Livny, *et al.*. Particle physics data grid collaboratory pilot. `http://www.ppdg.net/docs/SciDAC/PPDG_overview.pdf`, September 2001.

[35] D. Olson *et al.*. 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.

[36] L. Prechelt *et al.*. An experiment to assess the benefits of intermodule type checking, 1996.

[37] V. Raman, *et al.*. Online dynamic reordering for interactive data processing. In *The VLDB Journal*, pp. 709–720. 1999.

[38] V. Raman *et al.*. Partial results for online query processing. In *SIGMOD Conference*, pp. 275–286. 2002.

[39] Serge Abiteboul, *et al.*. *Foundations of Databases: The Logical Level*, chapter Chapter 20: Complex Values. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201537710.

[40] D. W. Shipman. The functional data model and the data language daplex. *ACM Transactions on Database Systems (TODS)*, 6(1):140–173, 1981.

[41] E. H. Sibley *et al.*. Data architecture and data model considerations. In *In Proceedings of the AFIPS National Computer Conference, Dallas, Texas*. American Federation of Information Processing Societies, june 1977.

[42] A. S. Szalay, *et al.*. Designing and mining multi-terabyte astronomy archives: the Sloan Digital Sky Survey. pp. 451–462. 2000.

[43] A. S. Szalay, *et al.*. The SDSS skyserver: Public Access to the Sloan Digital Sky Server Data. In *SIGMOD*, pp. 570–581. 2002.

[44] T. P. D. Team. The PostgreSQL Development Team. PostgreSQL User's Guide, 1999.

[45] M. Weske, *et al.*. Wasa: A workflow-based architecture to support scientific database applications. In *DEXA*. 1995.

[46] J. L. Wiener *et al.*. A moose and a fox can aid scientists with data management problems. In *Workshop on Database Programming Languages*, pp. 376–398. 1993.

[47] Wolfgang Hoschek *et al.*. Data Management in an International Data Grid Project. In *IEEE/ACM International Workshop on Grid Computing Grid*. 2000.

[48] P. Wong *et al.*. 30 years of multidimensional multivariate visualization, 1997.

[49] Yihong Zhao and Prasad M. Deshpande and Jeffrey F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *1997 ACM SIGMOD*, pp. 159–170. 1997.

[50] Yingwei Cui, *et al.*. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2):179–227, 2000.

[51] Y. Zhao, *et al.*. Chimera: A virtual data system for representing, querying, and automating data derivation. In *14th Conference on Scientific and Statistical Data Management*. 2002.