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: 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 [16]. 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* [26] 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 [34, 11] and distributed cluster federation [23]. As such, grid computing 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[25] and Condor [34]) 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.

The process-centric approach is a direct extension of the techniques used by scientists in the past. But, with

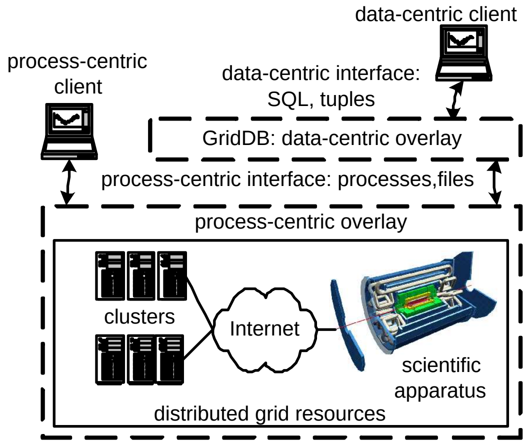

data-centric client
process-centric client
data-centric interface: SQL, tuples
GridDB: data-centric overlay
process-centric interface: processes,files
process-centric overlay
clusters
Internet
scientific apparatus
distributed grid resources

Figure 1: Grid Access Overview

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 lead 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 [37], the Particle Physics Data Grid [35], and the European Union DataGrid [9].

GriPhyN, in particular, is built around the notion of "Virtual Data" [50], 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, takes 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-like 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**: A related benefit is type checking. In contrast to process-centric approaches, 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 [20, 38].

- **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 prioritize computations based on partial results of jobs [36, 19]. GridDB's data-centric interface directly supports interactive query processing, enabling scientists to more effectively exploit grid resources for data analysis tasks.

- **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" [32], even for highly sophisticated analysis routines.

- **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 [50, 28]. GridDB's model of function-based data processing lends itself well towards tracking the lineage of individual data products, as is required for supporting data provenance.

- **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, that 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 data model, the Functional Data Model with Relational Covers (FDM/RC), and a data definition language for creating FDM/RC schemas. We then developed a set of software services that implement the data-centric, GridDB model using existing process-centric middleware. In this paper we describe this model and its implementation. 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 the memoization and interactive query processing features of the system.

The remainder of the paper is structured as follows. Section 2 provides background on grid applications and process-centric middleware. Section 3 describes the GridDB analysis interface. Section 4 describes the FDM/RC data model. Section 5 the design and implementation of a GridDB prototype. Section 6 describes GridDB's modeling of a complex analysis. Section 7 describes advanced performance-enhancing features. Sections 8 and 9 discuss Related Work and our Conclusion.

2 High-Energy Physics Example

In this section we introduce a simplified workflow obtained from the ATLAS High-Energy Physics experiment [19, 21]. We refer to this workflow as `HepEx` (High Energy Physics Example) and use it as a running example throughout the paper[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

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

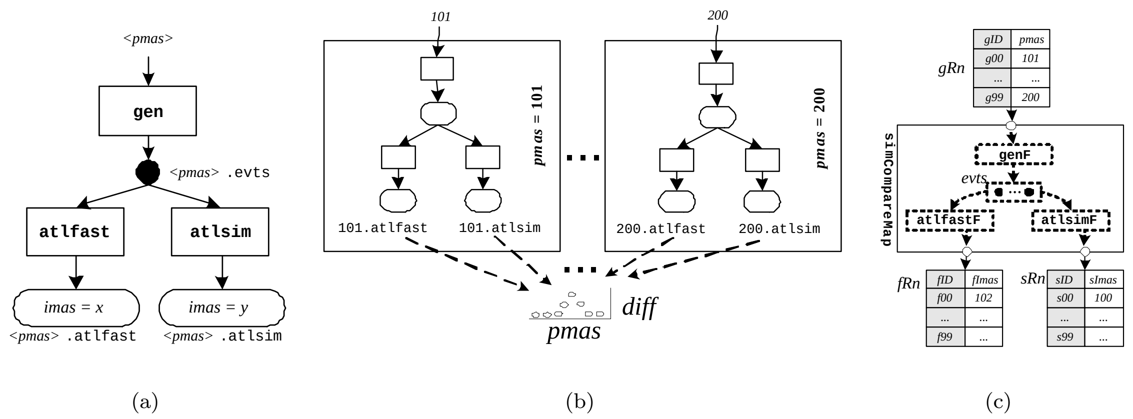


Figure 2: (a) `HepEx` abstract workflow (b) `HepEx` grid job (c) GridDB's `simCompareMap` replaces(a) and (b).

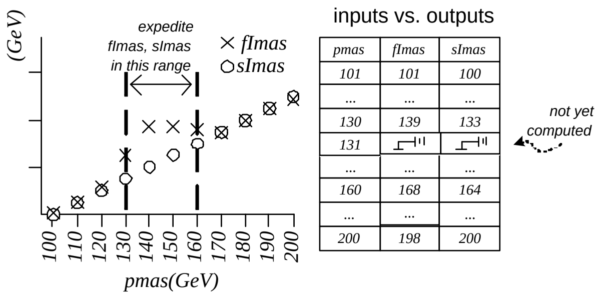
both simulations across a range of *pmas* values [2]. All three programs are long-running, and compute-bound, thus requiring grid processing.

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 create 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[1], 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 [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 [1]. 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 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

[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.



pmas	fImas	sImas
101	101	100
...	...	...
130	139	133
131		
...	...	...
160	168	164
...	...	...
200	198	200

Figure 3: Interactive Query Processing

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 expresses it in a GridDB data definition language (DDL), conveying semantics to GridDB, and allowing it to provide data-centric services and interfaces. The analyst's interaction with the grid is also changed dramatically.

We describe the data model and DDL used by the *modeler* in detail in Section 4. Here, we focus on the *analyst*'s interactions using the GridDB 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 prcoess.

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 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 the IQP service, an analyst can prioritize data in this range simply by selecting the data in this range (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 grid process execution. In contrast, users of process-centric middleware usually run jobs in batch.

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

3.2 GridDB Overview

The GridDB model rests on two main principles: (1) it represents programs and workflows as *functions*, and (2) an important subset of the data in a workflow can be represented as relations. We refer to this subset of data as the *relational cover*.

Representing programs and workflows as functions provides GridDB with the knowledge necessary to perform type checking of program inputs and outputs and enables a functional representation for AWFs; namely, 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 described in the 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 and supporting compostion.

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 procurement, 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[17, 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 bar-charts — are fundamentally visualizations of relations.

Analyst Actions

```
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);
```

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 creates the sandbox relations. For example, `gRn` is declared with a type of *set of* `g`. That statement 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 issue of proc-specs for grid execution, conceptually similar to the execution shown in 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.

Analyses commonly seek to discover relationships between different physical quantities. To support this analysis task, GridDBautomatically creates relational

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

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 provison of Interactive Query Processing, 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" are useful for analysis in their own right, and also play an important role in supporting Interactive Query Processing. By understanding the semantics of workflow and data, GridDB is able to provide 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 exposed workflow and data semantics to GridDB, essentially defining a schema. In this section, we describe a grid data model for the schema and its data definition language. The data model is called the Functional Data Model with Relational Covers(FDM/RC) and has two main constructs, data *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.

4.1 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.

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.

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

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 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

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

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 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`(see their headers in Listing 2, 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 analyses, 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 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

Listing 2: Abridged HepEx DDL

```
//opaque-only type definitions
opaque type evt;

//transparent-only type declarations
transparent type g = (pmas:int);

//both opaque and transparent types
type f = (fImas:int);
type s = (sImas:int);

//headers of atomic function definitions for
    genF, atlfastF, atlsimF
atomic fun genF(params:g):(out:evt) = ...;
atomic fun atlsim(evtsIn:evt):(outTuple:s)
    = ...;
atomic fun atlfastF(inEvt:evt):(outTuple:f) =
 exec(''atlfast'',
      [(''events'',inEvt)],
      [(/.atlfast\$/, outTuple, ''adapterX'')
          ]);

//composite function simCompare definition
fun simCompare(in:g):(fOut:f,sOut:s) =
 ( atlfast(gen(in)), atlsim(gen(in)) );

//a map function for simCompare
fun simCompareMap = map(simCompare);
```

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

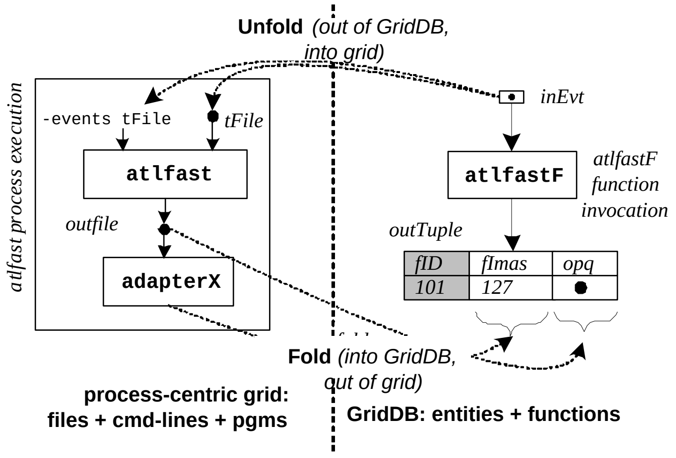
Finally, we return to defining atomic functions and their *fold* and *unfold* operations. Recall from Section 4.1.2 that the fold and unfold operations define how data moves between GridDB and process-centric middleware.

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 entity.

The second argument is a list of pairs (only one here) which specifies how arguments are unfolded. In this



Unfold (out of GridDB, into grid)
atlfast process execution
-events tFile
tFile
atlfast
outfile
adapterX
inEvt
atlfastF
atlfastF function invocation
outTuple
fID
fImas
opq
101
127
●
Fold (into GridDB, out of grid)
process-centric grid: files + cmd-lines + pgms
GridDB: entities + functions

Figure 4: Unfold/Fold in *atomic* functions

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 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.

5 GridDB Design

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*) 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

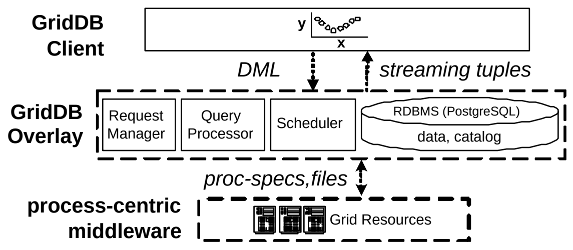


Figure 5: GridDB's Architecture

the processing of DDL statements, as they are straightforward updates to the system catalog.

5.1 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 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; 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 4.1.1 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* [32]. 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*.

5.1.2 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 each analyst action is processed and show, as an example, query processing for `HepEx` analysis. Internal data structures for `HepEx` are shown in Fig. 6. The diagram is an enhanced version of the analyst's view (Fig. 2(c)).

Recall from Section 3, 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 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 (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 ($genFMemo, atlfastFMemo, atlsimMemo$) . At this step, GridDB also stores a *workflow graph* (represented by the solid arrows in the figure) for the analysis.

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 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

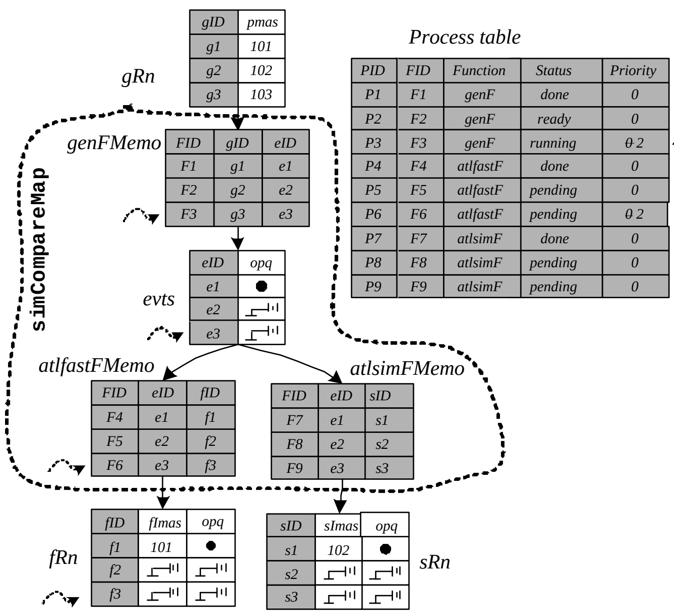


Figure 6: Internal data structures representing HepEx functions, entities, and processes. Shaded fields are system-managed. Dashed arrows indicate interesting tuples in our IQP dicussion (Sec. 7.1)

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 $e1$, $f1$, $r1$, 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 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 attributes.

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 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

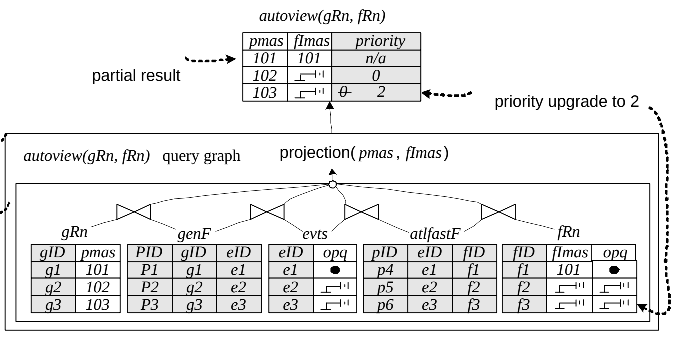


Figure 7: `autoview(gRn, fRn)`

describe in the following section.

6.1 Finding Clusters of Galaxies

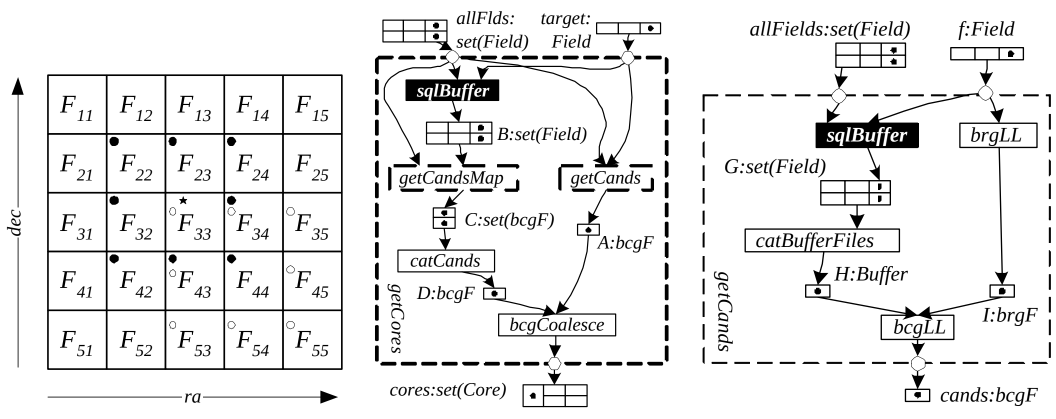
The Sloan Digital Sky Survey (SDSS) [7] is a 12 TB digital imaging survey mapping 250,000,000 celestial objects with two orders of magnitudes greater sensitivity than previous large sky surveys. `ClustFind` is a computationally-intense SDSS analysis that detects galaxy clusters, the largest gravitation-bound objects in the universe. The analysis uses the MaxBCG cluster finding algorithm [14], requiring 7000 CPU hours on a 500 MHz computer [15].

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 (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 $\bullet$). It then 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 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 Fig. 8(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 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 Fig. 8(b)).



(a) (b) (c)

Figure 8: (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`.

`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*[41]. 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`, 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 Fig. 8(a). Note that the `bcgLL` function within `getCands` is the most expensive function to evaluate [15], 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: 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, an astronomer automatically gains the performance of memoization, without needing to implement it himself.

We describe the implementation of both of these advanced features in the next section.

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. As these requests translate to (ra, dec) coordinates, GridDB's data-driven IQP service accomodates selective prioritization of interesting fields.

We describe the implementation of both of these advnaced features in the next section.

7 Performance Enhancements

Previous sections have shown GridDB providing basic grid 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 [19, 36] have expressed a need for interactive query processing (IQP) [39].

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) 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. GridDB the `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 [32]. Any view tuple can always be traced back to entities of the relational cover, using basic data lineage techniques [22]. 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 necessary and sufficient 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, tabular interface for grid process scheduling.

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 experimental 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 [?].

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.

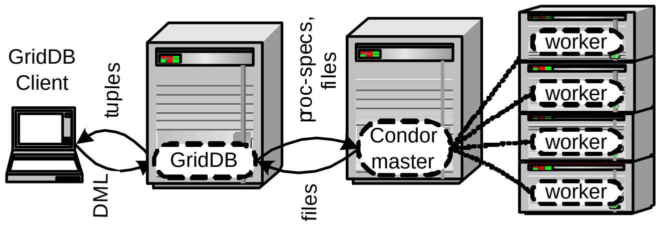


Figure 9: Experimental setup.

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 [24]. Currently, the system uses Condor [34] 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 are process-submission interfaces.

7.4 Validation

To demonstrate the effectiveness of IQP and memoization, we conducted validation experiments with our prototype GridDB implementation and the cluster testbed of Fig. 9. Measurements 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 [34] 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.

7.4.1 Validation 1: Data Prioritization

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*.

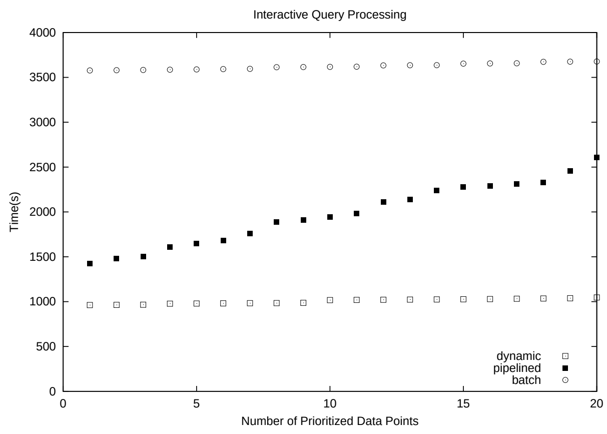


Figure 10: Validation 1, IQP for `HepEx`

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 an IQP request, prioritizing 25 as yet uncomputed `f` tuples:

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

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.

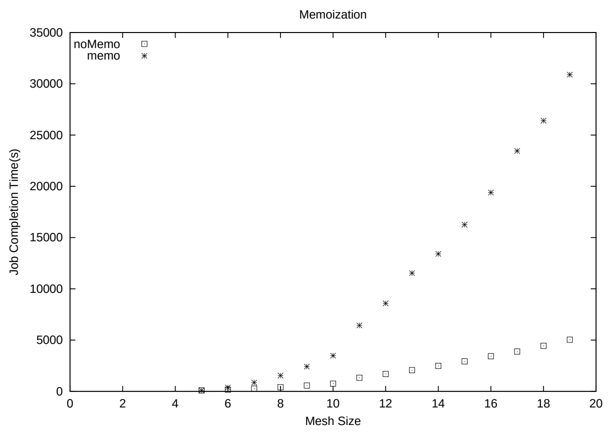


Figure 11: Validation 2, Memoization in `ClustFind`

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.

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).

8 Related Work

To streamline our discussion, we classify related systems into three categories: *process-centric*, *Workflow* and *Database*.

The relationships amongst the three categories is shown in Figure 12, where an arrow goes from category A to category B if B derives from A. *Process-centric* systems provide an OS-like process submission interface. *Workflow* systems focus on the management of process workflows and are naturally built on top of process-centric systems. *Database* systems provide a declarative interface to data — the final product of workflows — but do not support workflow management. Also in the diagram is a fourth category, *Workflow-Database*, which contains systems that exhibit characteristics of both workflow and database systems. GridDB is a member of this category.

Process-centric software systems such as Condor [34]

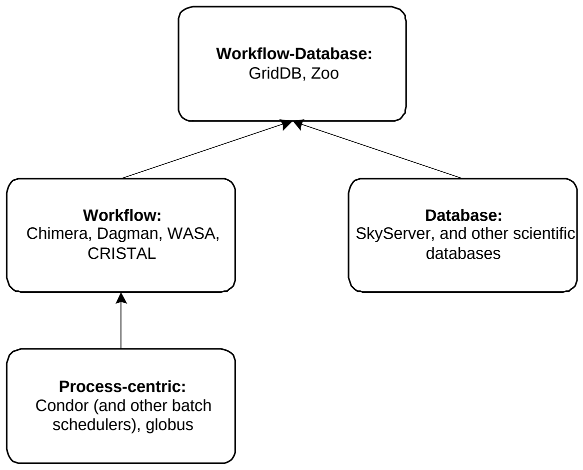

Workflow-Database:
GridDB, Zoo
Workflow:
Chimera, Dagman, WASA, CRISTAL
Database:
SkyServer, and other scientific databases
Process-centric:
Condor (and other batch schedulers), globus

Figure 12: A cateogorization of related systems.

and Globus [25] provide an OS-like interface, providing process submission services. While such systems provide an essential computational substrate, scientific collaborations are seeking to abstract these low-level interfaces and use data-centric interfaces so they can focus on data analysis (as we describe in Section 1).

Workflow systems build on top of process-centric middleware, modeling sets of dependent processes. In general, workflow systems model human-tasks in addition to computer-tasks[33]. The task sets often exhibit dependencies and are long-running. While workflow systems were also built for other domains, such as business process modeling, we focus our attention on those that support scientific data analysis.

Some scientific workflow systems, such as WASA [46], and CRISTAL [30] model general human and computer tasks. In contrast, systems such as Chimera [15, 50] and DAGMan [3], specialize in the modeling of computer tasks (which we have called *processes* in this paper), similar to GridDB. All of these workflow systems lack one characteristic, which is present in GridDB: they do not understand the data which is being created by their tasks, as embodied by an analysis' *relational cover.* As such, these systems do not provide a declarative interface, as is provided by GridDB.

At the other end of the spectrum are *Database* systems, which focus on the querying of *final* data values, but provide minimal support for process management.

An example of a recently developed scientific database system is the SkyServer [44, 45], which provides a web interface to terabytes of astronomy data from the Sloan Digital Sky Survey (the `ClustFind` analysis of Section 6 used this data). With about 500 users per day, the SkyServer has established itself as a useful astronomy resource. However, it does not model the processing pipeline that converts raw telescope data into SQLServer tables, as is possible in a workflow system. Memoization of data products as well as interactive query processing cannot be provided in grid environments without this modeling information.

Workflow-Database systems accomodate both the modeling of final data values (as is done in *Database* systems) and workflows (as is done in *Workflow* systems). GridDB fits into this category by allowing the modeling of final values using a relational cover and the modeling of workflows through typed functions. To our knowledge, ZOO [31, 12], a *Desktop Experiment Management Environment* from the University of Wisconsin, is the first, and only other, system to incorporate workflow and database functionality into a scientific analysis framework. In fact, 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 [47]. Also, the notions of *folding* and *unfolding* file data (as described in Section 4.1.2) are comprehensively addressed through the Frog and Turtle object-to-file mapping framework [13].

Unlike GridDB, however, Zoo was designed before the conception of grid computing and therefore was focused towards the desktop, rather than a more powerful computional grid. Additionally, GridDB uses the simpler, relational data model, as opposed to an object-oriented data model [47], as is used in ZOO. We believe this simplicity to be an advantage and have argued in Section 3 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 dat ato provide computational steering services, which are unavailable in ZOO.

In contrast to GridDB, ZOO and other high-level *Workflow* and *Database* systems, some systems provide a lower-level interface for managing grid resources. For example, the Master-Worker [27] framework is a software library for programming data parallel applications on clusters. While one can imagine performing computation steering within such a framework, it is done by extending C++ classes — a procedural programming model.

The Functional Data Model (FDM) and Interactive Query Processing are two related ideas that GridDB builds upon. The FDM was developed in the late 70s [42, 43, 18] as an alternative to the relational model with a more "natural" querying interface . While the FDM never displaced the relational model, it has been recently established as a "common-denominator" data model to federate distributively managed data sources [10]. We have used the FDM for an unrelated purpose, that of modeling programs and workflows.

Interactive Query Processing (IQP) has recently been proposed for RDBMSs [39, 40, 29]. The past work has focused *relational* query processing. While GridDB provides the same relational interface for steering computations, it allows the steering of *non-relational* "operators," programs that execute on grid nodes.

9 Conclusion

In this paper, we have presented GridDB, a software overlay that provides a relational interface, and data-centric interfaces to the grid. 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 window to the full data set. As such, we have built a data model (FDM/RC) and query language (a DDL and a DML) for representing workflows and accessing their data through a relational interface. 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. Next, we plan to deploy the software overlay with our scientific collaborators in the GriPhyN project.

10 Acknowledgements

We thank Mike Wilde, Jens-S. Vöeckler, Yong Zhao, and Ian Foster, our collaborators from the GriPhyN and Chimera projects, for helping us understand scientific grid workflows; the Nordugrid project for deploying a portable version of ATLAS software; the PPDG-CS 11 *Working Group on Interactive Analysis* for specifying scientific grid requirements; Yuri Smirnov, Craig Tull, Wim Lavrijsen, Rob Gardner and Marco Mambelli from the ATLAS project for helping us construct an ATLAS use-case and execute its software; Jim Annis for documenting the cluster finding use case, and making cluster finding code amenable to grid execution; Jim Gray for discussions of databases in science collaborations; and Sirish Chandarasekaran, Mehul Shah, Boon Thau Loo and Sailesh Krishnamurthy for discussion on this work.

This work was funded by the NSF under grant ITR-0086044 (GriPhyN) and by research grants from IBM, Intel and Microsoft.

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.

[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] *Data Management in an International Data Grid Project*, 2000.

[10] The Functional Approach to Data Management: Modeling, Analyzing and Integrating Heterogeneous Data, 2003.

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

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

[13] 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.

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

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

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

[17] 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.

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

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

[20] Charles W. Krueger. Software Reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992. ISSN 0360-0300.

[21] 2003. Personal communication with Craig Tull.

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

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

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

[25] 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.

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

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

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

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

[30] C. R. Information. Cristal. URL `citeseer.ist.psu.edu/417078.html`.

[31] 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.

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

[33] F. Leymann *et al.*. *Production workflow: concepts and techniques*. Prentice Hall PTR, 2000. ISBN 0-13-021753-0.

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

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

[36] 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.

[37] e. a. Paul 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. ”Proposal 0122557”.

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

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

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

[41] Serge Abiteboul and Richard Hull and Victor Vianu. *Foundations of Databases: The Logical Level*, chapter Chapter 20: Complex Values. Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201537710.

[42] D. W. Shipman. The functional data model and the data language daplex. *ACM Transactions on Database Systems (TODS)*, 6(1):140–173, 1981. ISSN 0362-5915.

[43] 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.

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

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

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

[47] 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.

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

[49] Y. Zhao, *et al.*. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pp. 159–170. ACM Press, 1997. ISBN 0-89791-911-4.

[50] 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.