MTCProv: A Practical provenance query framework for many-task scientific computing

Luiz M. R. Gadelha Jr.[1,2], Michael Wilde[3,4], Marta Mattoso[1], Ian Foster[3,4,5]
[1]Computer Engineering Program, COPPE, Federal University of Rio de Janeiro, Brazil
[2]National Laboratory for Scientific Computing, Brazil
[3]Mathematics and Computer Science Division, Argonne National Laboratory, USA
[4]Computation Institute, Argonne National Laboratory and University of Chicago, USA
[5]Department of Computer Science, University of Chicago, USA
lgadelha@lncc.br, wilde@mcs.anl.gov, marta@cos.ufrj.br, foster@anl.gov

Abstract

Scientific research is increasingly assisted by computer-based experiments. Such experiments are often composed of a vast number of loosely-coupled computational tasks that are specified and automated as scientific workflows. This large scale is also characteristic of the data that flows within such "many-task" computations (MTC). Provenance information can record the behavior of such computational experiments via the lineage of process and data artifacts. However, work to date has focused on data models, leaving unsolved issues of recording MTC behavior such as task runtimes and failures, or the impact of environment on performance and accuracy. In this work we contribute with MTCProv, a provenance query framework for many-task scientific computing that captures the runtime execution details of MTC workflow tasks on parallel and distributed systems, in addition to standard prospective and data derivation provenance. To help users query provenance data we provide a high level interface that hides relational query complexities. We evaluate MTCProv using applications in protein science and social network analysis, and describe how important query patterns such as correlations between provenance, runtime data, and scientific parameters are simplified and expressed.

1 Introduction

Scientific research is increasingly being assisted by computer-based (or *in-silico*) experiments that often manipulate large volumes of data and require high performance computing resources, such as parallel clusters or large distributed computational infrastructure such as grids and clouds. These experiments often comprise a large set of discrete, loosely-coupled computational tasks that can be specified and automated as scientific workflows. Their life-cycle [22] is usually given by a design phase, where its component tasks and data flow are specified; an execution phase, where data sets are transferred, and component tasks are mapped to the available computational resources and executed; and an analysis phase where scientists examine their outcome. These phases are typically iterated for experimental refinement. In the analysis phase, a scientist checks, for instance, which data sets were consumed and produced by each component task, what were the scientific domain parameters associated with these data sets, what was the runtime behavior of computational tasks (e.g., which of them failed, and why). All these phases can be supported by provenance management systems that describe how these scientific computations were specified (*prospective provenance*) [13], and executed (*retrospective provenance*) [32, 9]. A provenance database is in the middle of the experiment life-cycle [22].

Provenance information has many applications [32] such as verifying the results of scientific computations through re-execution, audit trail of data derivations, attribution [16], and the discovery of research

practices and protocols [18]. Domain scientists can explore scientific parameters used in their applications, and computational task variations for the same abstract workflow. Scientific software developers can examine the behavior of different versions of their applications, and debug them by input and output analysis. System administrators can analyze performance statistics about application execution on different computational resources, and quality of service information, such as failure rates per site. To allow for the exchange of provenance information between diverse systems, the Open Provenance Model (OPM) [25] was proposed. This standardization work is being continued as the PROV data model [26] by the Provenance Working Group within the World Wide Web Consortium[1].

The Swift parallel scripting system [37] supports the specification, execution, and analysis of many-task scientific computations [30]. The Swift language [35] has many characteristics commonly found in functional programming, such as functional abstraction without side effects, single-assignment variables, and implicitly parallel evaluation. Data sets are declared as variables of primitive types, *mapped types*, which associate a variable with persistent files; and collections (arrays and structures). Calls to application programs are expressed as *application interface functions* (i.e., primitive "leaf" functions) that consume and produce data sets. *Compound functions* compose sub-functions (including application interface functions), loops, and conditionals to form more complex data flows. Parallelism is implicit and pervasive in Swift: all expressions in a Swift script whose data dependencies are met are evaluated in parallel. A `foreach` loop processes all elements of an array in parallel.

A "site catalog" points Swift to available computational resources, and an "application catalog" lists the application programs that implement Swift leaf-functions, and the sites and locations where they can be executed. In this way, the specification of a many-task scientific computation in Swift is independent of location and expresses solely the functional data flow of the many-task computation. Swift automates load balancing between available computational resources, fault tolerance through execution retries, and replication of tasks to avoid queue latency on highly congested sites. These necessary features place complex demands on a provenance collection and query system. Swift is a successor to the Virtual Data System (VDS) [12], one of the pioneering systems supporting provenance of distributed parallel scientific computations [39, 8]. One problem with this model was the use of different data models, relational and semi-structured, to query provenance information, which makes it difficult to compose them into more complex queries. Another challenge is to support provenance while keeping the scalability of Swift in thousands of cores [20].

In this work, we are concerned with the problem of managing the provenance information of many-task scientific computations. Such computations are typified by a large number of tasks, often with data or control flow dependencies, which pass data via files. The tasks can execute in parallel on many, possible diverse and distributed nodes (as typified by scientific workflows expressed in Swift). Our main contributions are: a data model for representing provenance that extends OPM to capture events that are typical of parallel and distributed systems, such as a computational task invocation possibly having many execution attempts due to failures or high demand for computational resources; a provenance query interface that allows for visualization of provenance graphs and querying of provenance information using a simplified syntax with respect to plain SQL, that allows for simple query design by abstracting commonly found provenance query patterns into built-in functions and procedures, and by hiding the underlying database schema through database views and automated computation of relational joins; and a case study demonstrating the usability of this query interface and the applications of the provenance information gathered. This provenance data model and query interface, along with components for extracting provenance information from Swift script runs and storing it in a relational database, were implemented as a provenance query framework, that we call MTCProv, for the Swift parallel scripting system.

The remainder of this article is organized as follows. In section 2, we present the data model used to represent provenance in many-task scientific computations. In section 3, we describe the design and implementation of MTCProv, in particular how provenance is gathered, stored and queried. In section 4, we evaluate MTCProv in the context of a protein structure prediction application[5], demonstrating its usability

and utility. In section 5, we compare our approach with existing approaches for managing provenance of scientific computations. Finally, in section 6, we make some concluding remarks and describe future work.

2 A Provenance Model for Many-Task Scientific Computations

In this section, we present the MTCProv data model for representing provenance of many-task scientific computations. This MTC provenance model is a compatible extension of the Open Provenance Model. It addresses the characteristics of many-task computation, where concurrent component tasks are submitted to parallel and distributed computational resources. Such resources are subject to failures, and are usually under high demand for executing tasks and transferring data. Science-level performance information, which describes the behavior of an experiment from the point of view of the scientific domain, is critical for the management of such experiments (for instance, by determining how accurate the outcome of a scientific simulation was, and whether accuracy varies between execution environments). Recording the resource-level performance of such workloads can also assist scientists in managing the life cycle of their computational experiments. In designing MTCProv, we interacted with Swift users from multiple scientific domains, including protein science, and earth sciences, and social network analysis, to support them in designing, executing and analyzing their scientific computations with Swift. From these engagements, we identified the following requirements for MTCProv:

1. *Gather producer-consumer relationships between data sets and processes*. These relationships form the core of provenance information. They enable typical provenance queries to be performed, such as determining all processes and data sets that were involved in the production of a particular data set. This in general requires traversing a graph defined by these relationships. Users should be able, for instance, to check the usage of a given file by different many-task application runs.

2. *Gather hierarchical relationships between data sets*. Swift supports hierarchical data sets, such as arrays and structures. For instance, a user can map input files stored in a given directory to an array, and later process these files in parallel using a `foreach` construct. Fine-grained recording of data set usage details should be supported, so that a user can trace, for instance, that an array was passed to a procedure, and that an individual array member was used by some sub-procedure. This is usually achieved by recording constructors and accessors of arrays as processes in a provenance trace [25].

3. *Gather versioned information of the specifications of many-task scientific computations and of their component applications*. As the specifications of many-task computations (e.g., Swift scripts), and their component applications (e.g., Swift leaf functions) can evolve over time, scientists can benefit from keeping track of which version they are using in a given run. In some cases, the scientist also acts as the developer of a component application, which can result in frequent component application version updates during a workflow lifecycle.

4. *Allow users to enrich their provenance records with annotations*. Annotations are usually specified as key-value pairs that can be used, for instance, to record resource-level and science-level performance, like input and output scientific parameters, and usage statistics from computational resources. Annotations are useful for extending the provenance data model when required information is not captured in the standard system data flow. For instance, many scientific applications use textual configuration files to specify the parameters of a simulation. Yet automated provenance management systems usually record only the consumption of the configuration file by the scientific application, but preserve no information about its content (which is what the scientist really needs to know).

5. *Gather runtime information about component application executions*. Systems like Swift support many diverse parallel and distributed environments. Depending on the available applications at each site and on job scheduling heuristics, a computational task can be executed on a local host, a high performance computing cluster, or a remote grid or cloud site. Scientists often can benefit from having access to details about these executions, such as where each job was executed, the amount of time a job had to wait on a scheduler's queue, the duration of its actual execution, its memory and processor consumption, and the volume and rate of file system and/or network IO operations.

6. *Provide a usable and useful query interface for provenance information*. While the relational model is ideal for many aspects to store provenance relationships, it is often cumbersome to write SQL queries that require joining the many relations required to implement the OPM. For provenance to become a standard part of the e-Science methodology, it must be easy for scientists to formulate queries and interpret their outputs. Queries can often be of exploratory nature [33], where provenance information is analyzed in many steps that refine previous query outputs. Improving usability is usually achievable through the specification of a provenance query language, or by making available a set of stored procedures that abstract common provenance queries. *In this work we propose a query interface which both extends and simplifies standard SQL by automating join specifications and abstracting common provenance query patterns into built-in functions*.

Some of these requirements have been previously identified by Miles et al. [24]. However few works to date emphasize usability and applications of provenance represented by requirements 4 through 6, which are the main objectives of this work. As a first step toward meeting these requirements, we propose a data model for the provenance of many-task scientific computations. A UML diagram of this provenance model is presented in Figure 1. We simplify the UML notation to abbreviate the information that each annotated entity set (script run, function call, and variable) has one annotation entity set per data type. We define entities that correspond to the OPM notions of artifact, process, and artifact usage (either being consumed or produced by a process). These are augmented with entities used to represent many-task scientific computations, and to allow for entity annotations. Such annotations, which can be added post-execution, represent information about provenance entities such as object version tags and scientific parameters.

ScriptRun refers to the execution (successful or unsuccessful) of an entire many-task scientific computation, which in Swift is specified as the execution of a complete parallel script from start to finish. *FunctionCall* records calls to Swift functions. These calls take as input data sets, such as values stored primitive variables or files referenced by mapped variables; perform some computation specified in the respective function declaration; and produce data sets as output. In Swift, function calls can represent invocations of external applications, built-in functions, and operators; each function call is associated with the script run that invoked it. *ApplicationFunctionCall* represents an invocation of one component application of a many-task scientific computation. In Swift, it is generated by an invocation to an external application. External applications are listed in an application catalog along with the computational resources on which they can be executed. *ApplicationExecution* represents execution attempts of an external application. Each application function call triggers one or more execution attempts, where one (or, in the case of retries or replication, several) particular computational resource(s) will be selected to actually execute the application. *RuntimeSnapshot* contains information associated with an application execution, such as resource consumption. *Variable* represents data sets that were assigned to variables in a Swift script. Variable types can be atomic or composite. Atomic types are primitive types, such as integers and strings, recorded in the relation *Primitive*, or *mapped* types, recorded in the relation *Mapped*. Mapped types are used for declaring and accessing data that is stored in files. Composite types are given by structures and arrays. Containment relationships define a hierarchy where each variable may have child variables (when it is a structure or an array), or a parent variable (when it is a member of a collection). A variable may have as attributes a value, when

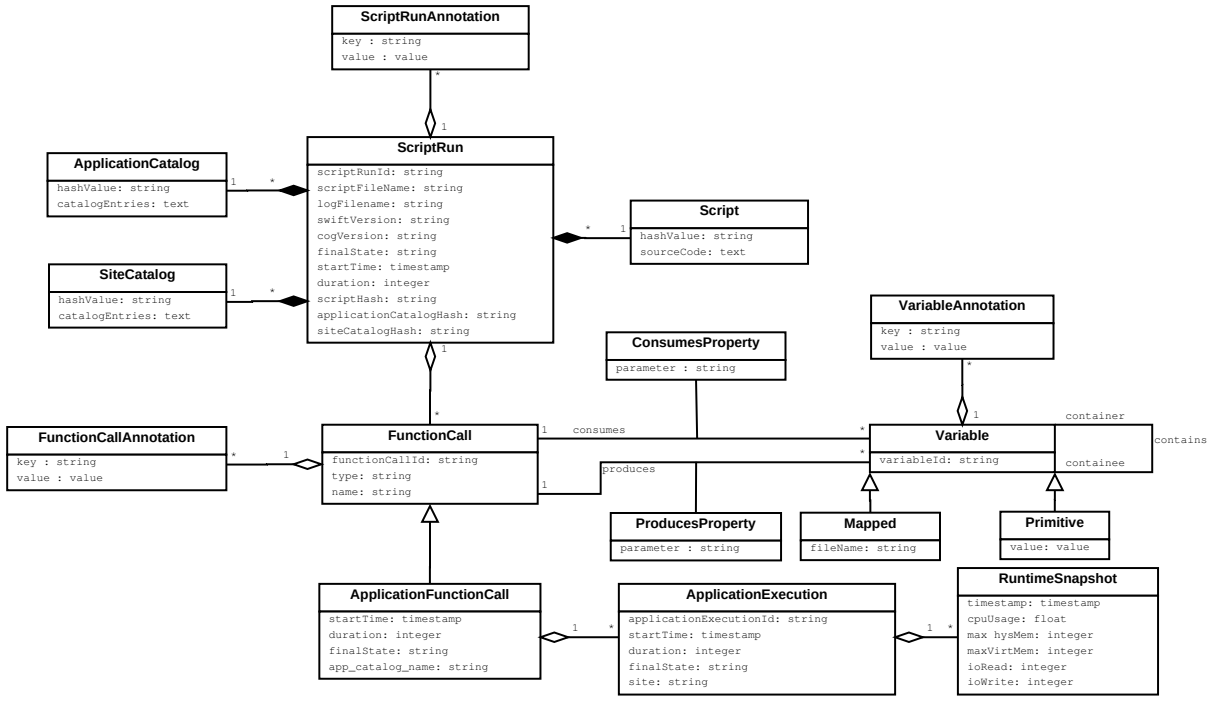

Figure 1: UML diagram of Swift's provenance database.

it is a primitive variable; or a filename, when it is a mapped file. An ⟨Entity set name⟩*Annotation* is a key-value pair associated with either a variable, function call, or script run. These annotations are used to store context-specific information about the entities of the provenance data model. Examples include scientific-domain parameters, object versions, and user identities. Annotations can also be used to associate a set of runs of a script related to a particular event or study, which we refer to as a *campaign*. The *produces* and *consumes* relationships between *FunctionCall* and *Variable* define a lineage graph that can be traversed to determine ancestors or descendants of a particular entity. Process dependency and data dependency graphs are derived with transitive queries over these relationships.

The provenance model presented here is a significant refinement of a previous one used by Swift in the Third Provenance Challenge [15], which was shown to be similar to OPM. This similarity is retained in the current version of the model, which adds support for annotations and runtime information on component application executions. *FunctionCall* corresponds to OPM processes, and *Variable* corresponds to OPM artifacts as immutable data objects. The OPM entity agent controls OPM processes, e.g. starting or terminating them. While we do not explicitly define an entity type for such agents, this information can be stored in the annotation tables of the *FunctionCall* or *ScriptRun* entities. To record which users controlled each script or function call execution, one can gather the associated POSIX userids, when executions are local, or the distinguished name of network security credentials, when executions cross security domains, and store them as annotations for the respective entity. This is equivalent to establishing an OPM *wasControlledBy* relationship. The dependency relationships, *used* and *wasGeneratedBy*, as defined in OPM, correspond to our *consumes* and *produces* relationships, respectively. Our data model has additional entity sets to capture behavior that is specific to parallel and distributed systems, to distinguish, for instance, between application invocations and execution attempts. We currently do not directly support the OPM concept of *account*, which can describe the same computation using different levels of abstraction. However, one could use annotations to associate one or more such accounts with an execution entity. Based on the mapping to OPM described here, MTCProv provides tools for exporting the provenance database into OPM provenance graph

interchange format, which provides interoperability with other OPM-compatible provenance systems.

3 Design and Implementation of MTCProv

In this section, we describe the design and implementation of the MTCProv provenance query framework. It consists of a set of tools used for extracting provenance information from Swift log files, and a query interface. While its log extractor is specific to Swift, the remainder of the system, including the query interface, is applicable to any parallel functional data flow execution model. The MTCProv system design is influenced by our survey of provenance queries in many-task computing [17], where a set of query patterns was identified. The *multiple-step relationships* (R*) pattern is implemented by queries that follow the transitive closure of basic provenance relationships, such as data containment hierarchies, and data derivation and consumption. The *run correlation* (RCr) pattern is implemented by queries for correlating attributes from multiple script runs, such as annotation values or the values of function call parameters.

3.1 Provenance Gathering and Storage

Swift can be configured to add both prospective and retrospective provenance information to the log file it creates to track the behavior of each script run. The provenance extraction mechanism processes these log files, filters the entries that contain provenance data, and exports this information to a relational SQL database. Each application execution is launched by a wrapper script that sets up the execution environment. These scripts can be configured to gather runtime information, such as memory consumption and processor load. Additionally, one can define a script that generates annotations in the form of key-value pairs, to be executed immediately before the actual application. These annotations can be exported to the provenance database and associated with the respective application execution. MTCProv processes the data logged by each wrapper to extract both the runtime information and the annotations, storing them in the provenance database. Additional annotations can be generated per script run using *ad-hoc* annotator scripts, as we demonstrate in our case study in section 4. In addition to retrospective provenance, MTCProv keeps prospective provenance by recording the Swift script source code, the application catalog, and the site catalog used in each script run. Figure 2(left) describes the components and information flow of MTCProv.

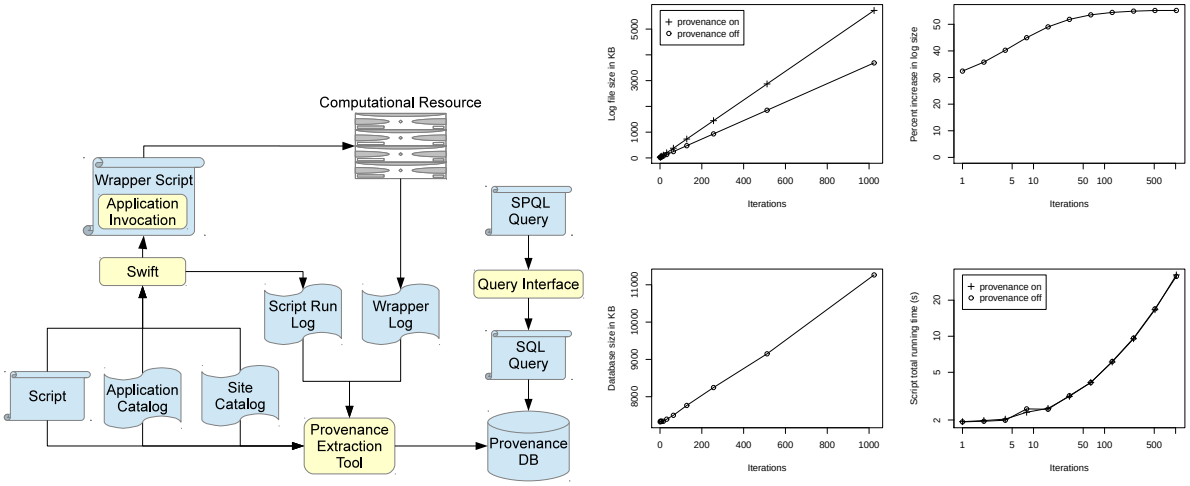

Figure 2: Left: MTCProv components and flow; Right: Impact of gathering provenance.

Provenance information is frequently stored in a relational database. RDBMS's are well known for their reliability, performance and consistency properties. Some shortcomings of the relational data model for managing provenance are mentioned by Zhao et al. [38], such as its use of fixed schemas, and weak support for recursive queries. Despite using a fixed schema, our data model allows for key-value annotations, which gives it some flexibility to store information not explicitly defined in the schema. The SQL:1999 standard, which is supported by many relational database management systems, has native constructs for performing recursive queries. Ordonez [29] proposed recursive query optimizations that can enable transitive closure computation in linear time complexity on binary trees, and quadratic time complexity on sparse graphs. Relationship transitive closures, which are required by recursive R* pattern queries, are well supported by graph-based data models, however as we present in our case study in section 4, many interesting queries require aggregation of entity attributes. Aggregate queries on a graph can support grouping node or edge attributes that are along a path [10]. However, some provenance queries require the grouping of node attributes that are sparsely spread across a graph. These require potentially costly graph traversals, whereas in the relational data model, well-supported aggregate operations can implement such operations efficiently.

To keep track of file usage across script runs, we record its hash function value as an alternative identifier. This enables traversing the provenance graphs of different script runs by detecting file reuse. Persistent unique identifiers for files could be provided by a data curation system with support for object versioning. However, due to the heterogeneity of parallel and distributed environments, one cannot assume the availability of such systems.

Figure 2(right) shows the impact of gathering provenance in Swift. It is based on runs of a sample iterative script using a different number of iterations on each run. The workflow specified in the script has a structure that is commonly found in many Swift applications. In this case, the log file size tends to grow about 55% in size when provenance gathering is enabled, but once they are exported to the provenance database they are no longer needed by MTCProv. The database size scales linearly with the number of iterations in the script. The initial set up of the database, which contains information such as integrity constraints, indices, and schema, take some space in addition to the data. As the database grows, it tends to consume less space than the log file. For instance, a 10,000 iteration run of the sample script produces a 55MB log file, while the total size of a provenance database containing only this run is 42MB. In addition, successful relational parallel database techniques can partition the provenance database and obtain high performance query processing. The average total time to complete a script run shows a negligible impact when provenance gathering is enabled. In a few cases the script was executed faster with provenance gathering enabled, which indicates that other factors, such as the task execution scheduling heuristics used by Swift and operating system noise, have a higher impact in the total execution time.

3.2 Query Interface

During the Third Provenance Challenge [15], we observed that expressing provenance queries in SQL is often cumbersome. For example, such queries require extensive use of complex relational joins, for instance, which are beyond the level of complexity that most domain scientists are willing, or have the time, to master and write. Such usability barriers are increasingly being seen as a critical issue in database management systems. Jagadish et al. [19] propose that ease of use should be a requirement as important as functionality and performance. They observe that, even though general-purpose query languages such as SQL and XQuery allow for the design of powerful queries, they require detailed knowledge of the database schema and rather complex programming to express queries in terms of such schemas. Since databases are often normalized, data is spread through different relations requiring even more extensive use of database join operations when designing queries. Some of the approaches used to improve usability are forms-based query interfaces, visual query builders, and schema summarization [36].

In this section, we describe our structured provenance query language, SQPL for short, a component of

MTCProv. It was designed to meet the requirements listed in section 2 and to allow for easier formation of provenance queries for the patterns identified in [17] than can be accomplished with general purpose query languages, such as SQL. SPQL supports exploratory queries [33], where the user seeks information through a sequence of queries that progressively refine previous outputs, instead of having to compose many subqueries into a single complex query, as it is often the case with SQL. Even though our current implementation uses a relational database as the underlying data model for storing provenance information, it should not be dependent on it, we plan to evaluate alternative underlying data models such as graph databases, Datalog, and distributed column-oriented stores. Therefore, in the current implementation, every SPQL query is translated into a SQL query that is processed by the underlying relational database. While the syntax of SPQL is by design similar to SQL, it does not require detailed knowledge of the underlying database schema for designing queries, but rather only of the entities in a simpler, higher-level abstract provenance schema, and their respective attributes.

The basic building block of a SPQL query consists of a selection query with the following format:

```
select (distinct) selectClause
(where            whereClause
(group by         groupByClause
(order by         orderByClause)))
```

This syntax is very similar to a selection query in SQL, with a critical usability benefit: hide the complexity of designing extensive join expressions. One does not need to provide all tables of the from clause. Instead, only the entity name is given and the translator reconstructs the underlying entity that was broken apart to produce the normalized schema. As in the relational data model, every query or built-in function results in a table, to preserve the power of SQL in querying results of another query. Selection queries can be composed using the usual set operations: union, intersect, and difference. A *select* clause is a list with elements of the form ⟨entity set name⟩(.⟨attribute name⟩) or ⟨built-in function name⟩(.⟨return attribute name⟩). If attribute names are omitted, the query returns all the existing attributes of the entity set. SPQL supports the same aggregation, grouping, set operation and ordering constructs provided by SQL.

To simplify the schema that the user needs to understand to design queries, we used database views to define the higher-level schema presentation shown in Figure 3. This abstract, OPM-compliant provenance schema is a simplified view of the physical database schema detailed in section 2. It groups information related to a provenance entity set in a single relation. The annotation entity set shown is the union of the annotation entity sets of the underlying database, presented in Figure 1. To avoid defining one annotation table per data type, we use dynamic expression evaluation in the SPQL to SQL translator to determine the required type-specific annotation table of the underlying provenance database.

Most of the query patterns identified in [17] are relatively straightforward to express in a relational query language such as SQL, except for the R* and RCr patterns, which require either recursion or extensive use of relational joins. To abstract queries that match these patterns, we included in SPQL the following built-in functions to make these common provenance queries easier to express:

- `ancestors`(object_id) returns a table with a single column containing the identifiers of variables and function calls that precede a particular node in a provenance graph stored in the database.
- `data_dependencies`(variable_id), related to the previous built-in function, returns the identifiers of variables upon which *variable_id* depends.
- `function_call_dependencies`(function_call_id) returns the identifiers of function calls upon which *function_call_id* depends.
- `compare_run`(list of ⟨`function_parameter`=string | `annotation_key`=string⟩) shows how process parameters or annotation values vary across the script runs stored in the database.

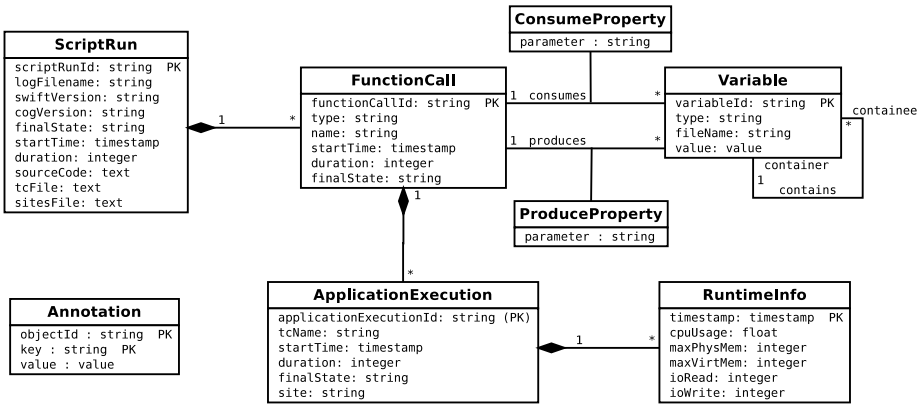

Figure 3: Higher-level schema that summarizes the underlying provenance database schema.

The underlying SQL implementation of the `ancestor` built-in function, below, uses recursive Common Query Expressions, which are supported in the SQL:1999 standard. It uses the *prov_graph* database view, which is derived from the *produces* and *consumes* tables, resulting in a table containing the edges of the provenance graph.

```
CREATE FUNCTION ancestors(varchar)  RETURNS SETOF varchar AS $$
WITH RECURSIVE anc(ancestor,descendant) AS
  (
       SELECT parent AS ancestor, child AS descendant
       FROM   prov_graph
       WHERE child=$1
     UNION
       SELECT prov_graph.parent AS ancestor,
              anc.descendant AS descendant
       FROM   anc, prov_graph
       WHERE  anc.ancestor=prov_graph.child
  )
  SELECT ancestor FROM anc $$ ;
```

To further simplify query specification, SPQL uses a generic mechanism for computing the *from* clauses and the join expressions of the *where* clause for the target SQL query. The SPQL to SQL query translator first scans all the entities present in the SPQL query. A shortest path containing all these entities is computed in the graph defined by the schema of the provenance database. All the entities present in this shortest path are listed in the *from* clause of the target SQL query. The join expressions of the *where* clause of the target query are computed using the edges of the shortest path, where each edge derives an expression that equates the attributes involved in the foreign key constraint of the entities that define the edge. While this automated join computation facilitates query design, it does somewhat reduce the expressivity of SPQL, as one is not able to perform other types of joins, such as self-joins, explicitly. However, many such queries can be expressed using subqueries, which are supported by SPQL. While some of the expressive power of SQL is thus lost, we show in the sections that follow that SPQL is able to express, with far less effort and complexity, most important and useful queries that provenance query patterns require. As a quick taste, this SPQL query returns the identifiers of the script runs that either produced or consumed the file `nr`:

```
select  compare_run(parameter='proteinId').run_id  where  file.name='nr';
```

This SPQL query is translated by MTCProv to the following SQL query:

```
select compare_run1.run_id
from   select run_id, j1.value AS proteinId
```

```
       from compare_run_by_param('proteinId') as compare_run1,
       run, proc, ds_use, ds, file
where  compare_run1.run_id=run.id and ds_use.proc_id=proc.id and
       ds_use.ds_id=ds.id and ds.id=file.id and
       run.id=proc.run_id and file.name='nr';
```

Further queries are illustrated by example in the next section. We note here that the SPQL query interface also lets the user submit standard SQL statements to query the database.

4 Case Study: Protein structure prediction workflow

We show here how MTCProv is used by scientists to assess and analyze their computational experiments. The workflows were executed on Beagle, a 186-node cluster with 17,856 processing cores; and PADS, a 48-node cluster with 384 processing cores. These machines are located at the Computation Institute, a joint institute of the University of Chicago and Argonne National Laboratory.

Open Protein Simulator (OOPS) [5] is a protein modeling application used for predicting the local structure of loops, and large insertion or chain ends in crystal structures and template-based models. It is used in conjunction with pre-processing and post-processing applications in a high-level workflow that is described in Figure 4, where the doLoopRound workflow activity is a compound procedure. Annotations are gathered by an application-specific script executed after an OOPS run, and stored in the provenance database.

Suppose a scientist runs an application script several times, in what we call a campaign, between April 4, 2010 and August 8, 2010 for a protein modeling competition. After completing the runs, he/she wants to know which ones were recorded in the database for the period. The following query lists runs of the OOPS application script (psim.loops) that were executed for the period. This is a simple query for the identifier, start time, and duration attributes of the run entity. The SPQL query is given by:

```
select script_run
where  script_run.start_time between '2010-04-04' and '2010-08-08' and
       script_run.filename='psim.loops.swift';
```

```
                id                 |         start_time          |   ...
-----------------------------------+-----------------------------+--------
 psim.loops-20100619-0339-b95ull7d | 2010-06-19 03:39:15.18-05  |   ...
 psim.loops-20100618-0402-qhm9ugg4 | 2010-06-18 04:02:21.234-05 |   ...
               ...                 |             ...             |   ...
```

One can annotate these runs with the key-value pair ("campaign", "casp2010"), so that the next queries can the answered with respect to this campaign only. Next, one can list how many times each protein was modeled by each of these runs. This identification of protein modeled is present in the modelIn (see figure 4) data set. Therefore one way to answer this query is to check the values of the variables that are used as parameter "proteinId":

```
select   compare_run(function_parameter='proteinId', annotation_key='campaign').proteinId,
         count(*)
where    compare_run.campaign='casp2010'
group by compare_run.proteinId;
```

```
 proteinId | count
-----------+-------
 T0588end  |     2
 T0601     |     1
 ...       |   ...
```

For the next queries, that illustrate the use of runtime execution information, we display their output graphically in figure 4. Even though the computational cost of loopModel is highly dependent on the content

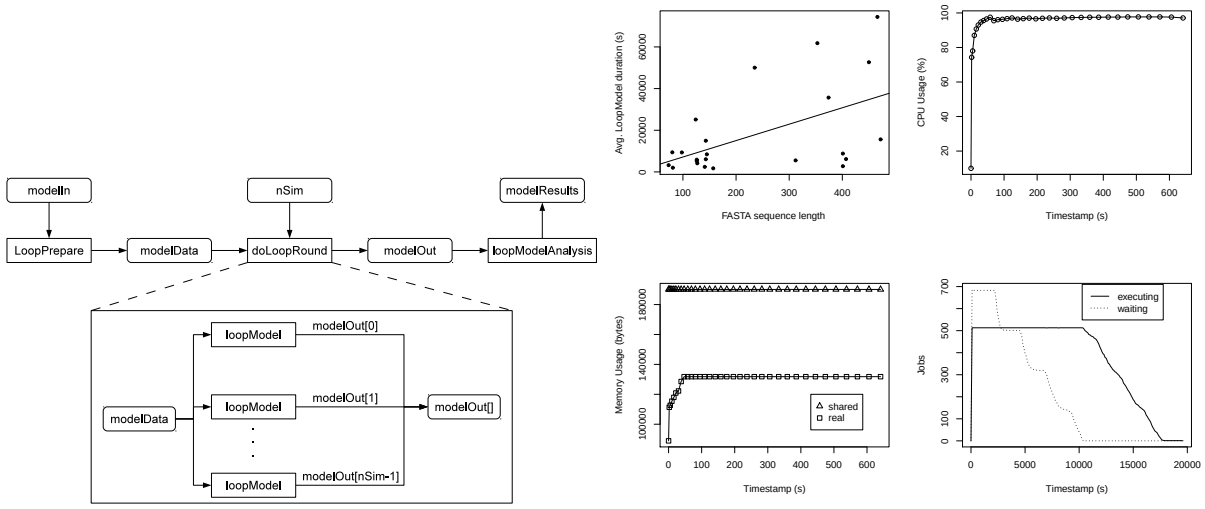

Figure 4: Left: Protein structure prediction workflow; Right: plots of query outputs.

of the FASTA sequence given as input, the following query comparing the FASTA sequence length with the duration of the loopModel task shows an example of the type of queries enabled by MTCProv:

```
select   compare_run(annotation_key='fasta_sequence_length'),
         avg(function_call.duration)
where    function_call.name='LoopModel'
group by compare_run.fasta_sequence_length;
```

Suppose one wants to analyze why an application execution failed by exploring, for instance, its computational resource consumption. In SPQL this can be obtained with the following queries:

```
select application_execution.id
where  application_execution.final_state='FAILED';

                      id
-------------------------------------------------
 execute:psim-loops-20100618-0402-qhm9ugg:0-1-0-0

select runtime_info
where  application_execution.id='execute:psim-loops-20100618-0402-qhm9ugg:0-1-0-0';
```

Also in figure 4, one can see a plot, derived from the provenance database, of how many jobs were waiting and how many were executing during a script run. In addition to runtime and domain-specific information, it is possible to design queries that explore lineage information, such as determining all processes and data sets that led to a data set:

```
select ancestors('dataset:20100618-0402-ia0bqb73:72000045');

                  ancestor
-------------------------------------------------
 execute:psim.loops-20100618-0402-qhm9ugg4:451006
 dataset:20100618-0402-ia0bqb73:72000039
 ...
```

These recursive queries can be combined with runtime and domain-specific information to determine, for instance, the range of specific scientific parameters, or the total amount of CPU time used by application

executions that preceded a particular dataset in a lineage graph, which can give an estimate of how long would it take to generate it without parallelism. In plain SQL, which can use the same provenance built-in functions, this query can be expresses as:

```
select sum(application_execution.walltime)
from   function_call_dependencies('psim.loops-20100618-0402-qhm9ugg4:451006') as predecessors,
       application_execution
where  predecessors.function_call_id=application_execution.function_call_id;

          sum
------------------------
 690449.134001970343328
```

In this example, the preceding application executions belong to a same script run. However, this is not always the case, since a file generated by a script run might be used by another one, causing the transitive closure to traverse multiple provenance graphs.

5 Related Work

One example of the importance of tracking resource consumption is the Technology Audit and Insertion Service (TAIS) [14] [2], an effort to develop adequate tools for monitoring and auditing of high performance computing systems, which is one of the main components of XSEDE [3], a large-scale distributed computational infrastructure. Capturing runtime performance information of parallel and distributed scientific workflows can be useful in planning task execution. Askalon [34] provides a performance monitoring service for scientific workflows that are executed in grid computing environments. A performance predictor service uses information gathered by the performance monitor to support better scheduling of computational tasks. Ogasawara et al. [28] describe components that can be included in specifications of MTC workflows to control the parallelization of activities, along with support for gathering provenance data about the execution of these activities. It enables reuse of the parallelization strategies recorded. ParaTrac [11] gathers resource consumption information about data-intensive scientific workflows, enabling detailed performance analysis to be performed, such as the amount of data flowing between component tasks. PDB [21] collects timing data about task execution in streaming workflows, where data flows occur in memory buffers instead of disk-based I/O, and also uses this information for execution planning. Our approach extends the functionality of these systems by gathering, in addition to performance information, lineage data and annotations about the scientific domain, and by allowing the design of queries exploring these different aspects.

Domain-specific languages for querying provenance have been an active area of research. The Vistrails [13] workflow management system supports exploratory computational scientific experiments, keeping track of how workflow specifications evolve and derivation relationships between data and processes. This provenance model also allows its entities to be annotated by the user, as key-value pairs. vtPQL [31] is a domain-specific language for querying provenance information in Vistrails. It can query workflow specification, evolution, and derivation relationships between data and processes. The query system is augmented by useful constructs in the context of provenance and workflows, such as functions for provenance graph traversal. Anand et al. [6] advocate the representation of provenance information as fine-grained relationships over nested collections of data. They present a *Query Language for Provenance* (QLP) that is independent of the underlying data model used for storage and workflow management system and closed under query composition. It provides constructs for accessing specific versions of structures produced by a workflow run. QLP operators for querying lineage act as filters over lineage relations, returning a subset of them. Lineage queries can be combined with queries over data structures. QLP supports constraining the structure of lineage paths using regular expressions. *Path Query Language* (PQL) [27] is used to query provenance gathered from different layers, such as the workflow layer or the operating system layer. It is

derived from the Lorel query language for semistructured data[4] and extends it to allow bi-directional graph traversal and queries using regular expressions on graph edges to define paths. Chebotko et al. [7] describe RDFProv, a provenance management system that is based on semantic web techniques and relational storage. It manages provenance ontologies using an inference engine; provenance data is represented as RDF triples, and SPARQL is used for querying.

The main semantic contribution of MTCProv relative to the existing approaches is its support for capturing, storing and querying events that are typical of high-performance parallel and distributed computations. These events include re-executions for fault-tolerance, redundant submission of tasks for improving execution throughput, and resource consumption in terms of filesystem input and output operations, memory usage, and processor load. Another contribution of MTCProv, not found in related work, is showing its scalability in MTC with more than 1000 cores. Our work is most distinguished in terms of its contribution to the utility and usability of provenance query capabilities: SPQL supports queries for patterns that are of high value to scientists but which are difficult to perform with existing querying approaches (such as correlating scientific parameter settings with the resource consumption behavior of various workflow runs). SPQL enables users with only a modest high level facility in SQL to use relational SQL query techniques on a powerful, OPM-compliant provenance database schema.

6 Concluding Remarks

In this work, we presented MTCProv, a provenance management system for many-task scientific computing that captures runtime execution details of workflow tasks on parallel and distributed systems, in addition to standard prospective and data derivation provenance. It allows this provenance information to be enriched with domain specific annotations. We have also described the use of SPQL for querying provenance, which abstracts frequent query patterns with built-in functions and automatically defines complex relational join expressions. Case studies in real, large-scale parallel distributed workflows, in use today on active investigations, demonstrated how useful queries were made possible by MTCProv, such as correlations between scientific parameters and computational resource consumption. As observed in this work, the impact of gathering provenance is low in terms of total script execution time, making it suitable for rather fine-grained tracking of highly parallel distributed workflows. The storage space requirements for a script run are a constant proportion of the total log file size.

In future work, we plan to continue to enhance the SPQL query model, adding more built-in functions as well as continuing to perfect the ability to seamlessly integrate standard SQL into SPQL queries. We plan to enable online provenance recording, which allows for provenance analysis during the execution of a Swift script. This could facilitate adaptive script execution by dynamically changing the mapping of tasks to computational resources. Our current implementation gathers workflow-level provenance about function calls and variables defined in Swift. This could be augmented by gathering additional lower level information about data transfers and job management strategies. These lower level provenance records could be linked to the respective workflow-level ones through a layering scheme analogous to the one presented in [27]. Finally, we plan to explore different underlying schemas and possible data models for storing provenance. For instance, column-oriented data stores and tools like Dremel [23] could enable distributed and highly scalable management of provenance data generated from a large set of high performance computing sites.

Acknowledgment

This work was supported in part by CAPES, CNPq, by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357, and by NSF under awards OCI-0944332 and OCI-1007115. We thank Swift users Aashish Adhikari, Andrey Rzhetsky

and Jon Monette, for providing and running applications using MTCProv, and for helping us understand their provenance requirements.

The original publication is available at www.springerlink.com:
Gadelha, L. M. R., Wilde, M., Mattoso, M., Foster, I. (2012). MTCProv: a practical provenance query framework for many-task scientific computing. Distributed and Parallel Databases, 30(5-6), 351–370. http://link.springer.com/article/10.1007/s10619-012-7104-4

References

[1] Provenance working group. http://www.w3.org/2011/prov/wiki/Main_Page, 2012.

[2] Technology audit and insertion service for TeraGrid. http://www.si.umich.edu/research/project/technology-audit-and-insertion-service-teragrid, 2012.

[3] XSEDE - extreme science and engineering discovery environment. https://www.xsede.org, 2012.

[4] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1:66–88, 1997.

[5] A. Adhikari, J. Peng, M. Wilde, J. Xu, K. Freed, and T. Sosnick. Modeling large regions in proteins: Applications to loops, termini, and folding. *Protein Science*, 21(1):107–121, 2012.

[6] M. Anand, S. Bowers, T. McPhillips, and B. Ludäscher. Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs. In *Scientific and Statistical Database Management*, volume 5566 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2009.

[7] A. Chebotko, S. Lu, X. Fei, and F. Fotouhi. RDFProv: a relational RDF store for querying and managing scientific workflow provenance. *Data & Knowledge Engineering*, 69(8):836–865, 2010.

[8] B. Clifford, I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Tracking provenance in a virtual data grid. *Concurrency and Computation: Practice and Experience*, 20(5):575, 2008.

[9] S. da Cruz, M. Campos, and M. Mattoso. Towards a taxonomy of provenance in scientific workflow management systems. In *Proc. IEEE Congress on Services, Part I, (SERVICES I 2009)*, pages 259–266, 2009.

[10] A. Dries and S. Nijssen. Analyzing graph databases by aggregate queries. In *Proc. Workshop on Mining and Learning with Graphs (MLG 2010)*, pages 37–45, 2010.

[11] N. Dun, K. Taura, and A. Yonezawa. ParaTrac: a fine-grained profiler for data-intensive workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 37–48. ACM, 2010.

[12] I. Foster, J. Vöckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proc. International Conference on Scientific and Statistical Database Management (SSDBM 2002)*, pages 37–46. IEEE Computer Society, 2002.

[13] J. Freire, C. Silva, S. Callahan, E. Santos, C. Scheidegger, and H. Vo. Managing Rapidly-Evolving scientific workflows. In *Provenance and Annotation of Data*, volume 4145 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 2006.

[14] T. Furlani, M. Jones, S. Gallo, A. Bruno, C. Lu, A. Ghadersohi, R. J. Gentner, A. Patra, R. DeLeon, G. von Laszewski, L. Wang, and A. Zimmerman. Performance metrics and auditing framework for high performance computer systems. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, page 16:1–16:1. ACM, 2011.

[15] L. Gadelha, B. Clifford, M. Mattoso, M. Wilde, and I. Foster. Provenance management in swift. *Future Generation Computer Systems*, 27(6):780, 2011.

[16] L. Gadelha and M. Mattoso. Kairos: An architecture for securing authorship and temporal information of provenance data in Grid-Enabled workflow management systems. In *IEEE Fourth International Conference on eScience (e-Science 2008)*, pages 597–602. IEEE, 2008.

[17] L. Gadelha, M. Mattoso, M. Wilde, and I. Foster. Provenance query patterns for Many-Task scientific computations. In *Proceedings of the 3rd USENIX Workshop on Theory and Applications of Provenance (TaPP'11)*, 2011.

[18] G. Goth. The science of better science. *Commun. ACM*, 55(2):13–15, February 2012.

[19] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 13–24. ACM, 2007.

[20] D. Katz, T. Armstrong, Z. Zhang, M. Wilde, and Justin. Wozniak. Many-Task computing and blue waters. *arXiv:1202.3943*, 2012.

[21] C. Liew, M. Atkinson, R. Ostrowski, M. Cole, J. van Hemert, and L. Han. Performance database: capturing data for optimizing distributed streaming workflows. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 369(1949):3268–3284, 2011.

[22] M. Mattoso, C. Werner, G. Travassos, V. Braganholo, E. Ogasawara, D. Oliveira, S. Cruz, W. Martinho, and L. Murta. Towards supporting the life cycle of large scale scientific experiments. *International Journal of Business Process Integration and Management*, 5(1):79–92, January 2010.

[23] S. Melnik, A. Gubarev, J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54(6):114–123, June 2011.

[24] S. Miles, P. Groth, M. Branco, and L. Moreau. The requirements of recording and using provenance in e-Science. *Journal of Grid Computing*, 5(1):1–25, 2007.

[25] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. Van den Bussche. The open provenance model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743–756, 2011.

[26] L. Moreau, P. Missier, K. Belhajjame, S. Cresswell, Y. Gil, R. Golden, P. Groth, G. Klyne, J. McCusker, S. Miles, J. Myers, and S. Sahoo. The PROV data model and abstract syntax notation. Technical report, World Wide Web Consortium (W3C), December 2011.

[27] K. Muniswamy-Reddy, U. Braun, D. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor. Layering in provenance systems. In *Proc. of the USENIX Annual Technical Conference*, 2009.

[28] E. Ogasawara, D. de Oliveira, F. Chirigati, C. Barbosa, R. Elias, V. Braganholo, A. Coutinho, and M. Mattoso. Exploring many task computing in scientific workflows. In *Proc. 2nd Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS 2009)*, 2009.

[29] C. Ordonez. Optimizing recursive queries in SQL. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2005)*, pages 834–839, 2005.

[30] I. Raicu, I. T Foster, and Yong Zhao. Many-task computing for grids and supercomputers. In *Workshop on Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008*, pages 1–11. IEEE, November 2008.

[31] C. Scheidegger, D. Koop, E. Santos, H. Vo, S. Callahan, J. Freire, and C. Silva. Tackling the provenance challenge one layer at a time. *Concurrency and Computation: Practice and Experience*, 20(5):473–483, 2008.

[32] Y. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, September 2005.

[33] R. White and R. Roth. *Exploratory Search: Beyond the Query–Response Paradigm*. Morgan & Claypool, 2009.

[34] M. Wieczorek, R. Prodan, and T. Fahringer. Scheduling of scientific workflows in the ASKALON grid environment. *SIGMOD Rec.*, 34(3):56–62, September 2005.

[35] M. Wilde, M. Hategan, J. Wozniak, B. Clifford, D. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):634–652, 2011.

[36] C. Yu and H. V. Jagadish. Schema summarization. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, page 319–330. VLDB Endowment, 2006.

[37] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Proc. 1st IEEE International Workshop on Scientific Workflows (SWF 2007)*, pages 199–206, 2007.

[38] Y. Zhao and S. Lu. A logic programming approach to scientific workflow provenance querying. In *Provenance and Annotation of Data and Processes (IPAW 2008)*, volume 5272 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2008.

[39] Y. Zhao, M. Wilde, and I. Foster. Applying the virtual data provenance model. In *Proc. 1st International Provenance and Annotation Workshop (IPAW 2006)*, volume 4145 of *Lecture Notes in Computer Science*, pages 148–161. Springer, 2006.

This manuscript was created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (“Argonne”). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

(This note is removed in camera-ready submissions.)