Zoom*UserViews: Querying Relevant Provenance in Workflow Systems*

[Demonstration Proposal]

Olivier Biton
University of Pennsylvania
Philadelphia, USA
biton@seas.upenn.edu

Sarah Cohen-Boulakia
University of Pennsylvania
Philadelphia, USA
sarahcb@seas.upenn.edu

Susan B. Davidson
University of Pennsylvania
Philadelphia, USA
susan@seas.upenn.edu

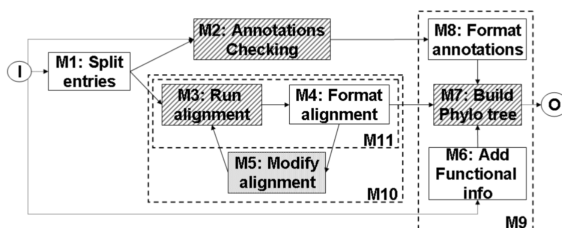

Figure 1: Phylogenomic workflow specification

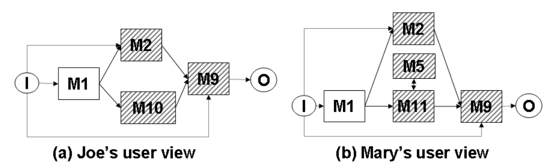

Figure 2: Induced phylogenomic workflows

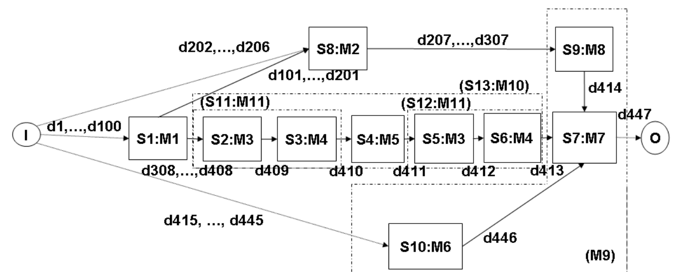

Figure 3: Phylogenomic workflow run

ABSTRACT

In this demonstration, we present the ZOOM*UserView system, and focus on the module which generates a "user view" based on what tasks the user perceives to be relevant in the workflow specification. We will show how user views can be used to reduce the amount of information returned by provenance queries, while focusing on information the user finds relevant. User views are based on the notion of composite tasks, and induce a higher-level specification of a workflow.

1. MOTIVATION

Workflow management systems (e.g. [2, 5]) have become increasingly popular as a way of specifying and implementing large-scale in-silico experiments. In such systems, a workflow can be graphically designed by chaining together bioinformatics *tasks* (e.g. downloading sequences, building a phylogenetic tree) which can be grouped together to form *composite* tasks. Composite tasks are an important mechanism for abstraction, privacy, and reuse between workflows.

*This research is supported by the National Science Foundation under Grants No. 0513778, 0415810, 0612177, 0629846, 0630033, and 0629702. (Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.)

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

As an example, the *workflow specification* in Figure 1 represents a common analysis in molecular biology: *Phylogenomic inference of protein biological function*. Each node in this specification represents a task, and will be called a *Module* (M). Some of the modules are considered by the user as *relevant*, which we indicate using shading. For example, "Annotations checking" (M2), "Run alignment" (M3) and "Build Phylo tree" (M7) may be considered relevant by the users Joe and Mary, while modules M1, M4, M6, and M8 are not since they merely re-format data. Joe may not be interested in the alignment modification step (M5), while Mary may be (lightly shaded in Figure 1). By grouping together atomic modules, *composite* modules such as M9, M10 and M11 can be formed, allowing the user to visualize the workflow in a simpler way. Figure 2 represents Joe's and Mary's visualizations of the workflow induced by their respective *user views*, i.e., set of composite modules. Each user can then define the level of granularity at which he wishes to view the workflow.

A typical lab executes a given workflow several times a month, resulting in vast amounts of intermediate and final data products. Since it is easy to lose track of how a data object came to be, i.e. what sequence of steps and input data were used to produce that data product, scientists must be

able to ask *provenance* queries such as: *What are all the data objects / sequence of steps which have been used to produce this tree?*

Figure 3 shows an execution of the workflow in Figure 1 in which one hundred sequences are taken as initial input (d1 to d100), minor modifications are done on the annotations (d202 to d206), and thirty additional annotations are used (d415 to d445). In this run both M2 and M3 have been executed, and two executions of modules M3 and M4 were performed. In another run, M2 may be skipped and six executions of M3-M5 performed. The execution of consecutive steps within the same composite module causes a virtual execution of the composite module, shown in Figure 3 by dotted boxes.

It is clear that the answer to a provenance query should depend on the level at which the user can see the workflow. For example, in Figure 3, Mary will see two executions of M11, while Joe will see only one execution of M10. The input used to produce d413 would include the data passed between executions of M11 and M5 (d410, d411) for Mary, while this data would not be visible to Joe. Thus the answer to a provenance query depends on the user view. While several workflow systems are able to answer provenance queries [4], none take user views into account.

In this demonstration, we present the ZOOM*UserViews system, which allows users to define user views and use them in provenance queries to reduce the amount of data returned to the user while ensuring that relevant information is returned. Technical contributions include:

- A model for tracking and querying provenance through user views which details the information that must be provided by a workflow system.
- Properties of a "good" user view, and algorithm which takes as input a workflow specification and set of relevant modules and constructs a user view.
- A provenance reasoning system which assists in the construction of user views, stores provenance information in an Oracle warehouse, and provides a user interface for querying and visualizing provenance information with respect to a user view.

It should be noted that although our approach is illustrated using scientific workflows, it is *generic* in the sense that it can be used by any workflow system which provides basic logging information.

2. ZOOM ARCHITECTURE

The goal of ZOOM is to provide users with an interface to query provenance information provided by a workflow system as well as to help them construct an appropriate user view.

The architecture of ZOOM is presented in Figure 4. The warehouse of provenance information (top of figure) is populated by the workflow system which provides information about (i) workflow specifications and user view definitions (converted to tables *contains* and *userView*); and (ii) log information obtained following (or during) a workflow execution (tables *instanceOf*, *input*, *output*). The meaning of these tables will be discussed in the next subsection (provenance model). Note that the Twiki web page of the second "provenance challenge" [4] provides samples of log information (in XML format) generated by 14 workflow systems.

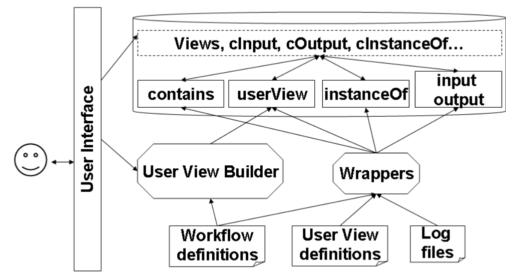

Figure 4: Architecture

Constructing wrappers to convert the log information of those systems into our model is thus straightforward.

Users interact with the system by posing a provenance query or building a user view. Provenance information is displayed graphically to aid in understanding the results.

The provenance model for scientific workflows, UserViewBuilder and query processing engine are described in more detail in the following subsections.

2.1 Provenance Model

The *provenance* of a data object is the sequence of modules and input data objects on which it depends [3, 4]. If the data is a parameter or was input to the workflow execution by a user, its provenance is whatever metadata information is recorded, e.g. who input the data and the time at which the input occurred. Following other work (e.g. [1, 2]), we assume data is never overwritten or updated in place. Each data object therefore has a unique identifier and is produced by at most one step.

We assume that the workflow run generates a log of events, which allows us to reason about what module a step is an instance of, what data objects and parameters were input to that step, and what data objects were output from that step. For example, the log could include the start time of each step, the module of which it was an instance, and read (write) events that indicate which step read (wrote) which data objects at what time.

2.1.1 Warehouse Model

ZOOM*UserViews's model is composed of base tables which capture information about the workflow execution, the workflow specification, the relationship between the specification and execution, the inclusion relationship between composite modules and modules, and information about user views.

The abstracted log information is captured using the following relations, where *did* is the id of a data object, *sid* is the id of a step, m is the name of a module, *ts* is an integer that captures the partial order of input and output events to a step, and $(label, value)$ is the metadata information recorded for user or parameter data:[1] $instanceOf(sid, m)$

$input(sid, did, ts)$
$output(sid, did, ts)$
$metadata(did, label, value)$

A *user view* is a partition of the modules of a workflow specification (ignoring input (I) and output (O)). This information is captured in the base relations $userView(u, cm)$

[1] If the system does not record the time at which read and write events occur, the timestamp can be the start time of the step.

and $contains(cm, m)$, where u represents the name of the user view, cm is the name of a composite module, and m is the name of a module. Thus $userView$ defines what composite modules (partitions) are in a user view, while $contains$ specifies what modules are in a composite module. For example, *userView*(Joe,M_9) and *userView*(Mary,M_9) are tuples stating that both Joe and Mary see the module M_9; *contains*(M_9, M_6) states that M_9 contains the module M_6.

As discussed earlier, user views affect what provenance information the user can see and query, and are defined at the specification level (*userView* and *contains*). However, since execution occurs at the lowest-level modules, we must calculate composite steps (executions of composite modules) as the longest sequence of steps in the same composite module. This information is captured in the calculated relations $cStep(csid, sid)$ and $cInstanceOf(csid, cm)$, where $csid$ is the (invented) id of a composite step which is an execution of composite module cm and contains the step sid. We also calculate the input and output of composite steps, *cInput* and *cOutput*, as follows: Input to a composite step is input to any contained step that is not produced as output by some contained step; output to a composite step is output of any contained step that is not consumed as input by any contained step.

Using *cInput*, *cOutput*, $cInstanceOf$ and $userView$, we can now calculate the provenance of a data item *did* as a function of a user view u.

2.1.2 Implementing provenance relations

The database used for our prototype is Oracle 10.2, extended with stored procedures. The user interface and wrappers are developed using Java with JDBC.

Information about the workflow and user view definitions together with workflow runs are stored in the database.

From the tables described earlier, we can express queries to generate the immediate provenance (*immProv*) for an input data object *did* as the step *sid* which produced it, and data objects that were input before *did* was output. The deep provenance (*prov*) for a data object is all steps and input data that were used to produce it.

Query *immProv* is easily implemented in SQL using the base tables $instanceOf$, *input*, *output* and *metadata*. Relation *cStep* is computed by wrappers while loading the execution (log), and is used to calculate *immUprov* (immediate provenance through user view).

Implementing *prov* and *uProv* (deep provenance through user view) implies recursive queries which are, by nature, extremely expensive. Oracle 10.g provides an extension of SQL for recursive queries on *hierarchical data* (CONNECT BY operator). Although CONNECT BY queries are more expensive than other SQL queries, the operator takes advantage of the query optimizer and is in the core system. Computation is also done on the server, and less data needs to be sent to the client. We compared the performance of CONNECT BY with a local computation using Java and JDBC, and found the processing time to be much better using CONNECT BY.

However, CONNECT BY was developed for trees, and does not work for arbitrary DAGs. We therefore extended it to check for nodes already seen in a DAG. As an example, the following query Q computes *prov* for the data id 'd447'.

```
SELECT
    output.execid, output.sid prodstep,
    outinst.m prodmodule,
    output.did dataid, output.ts ts,
    input.sid consstep,
    inputinst.m consmodule,
    markdata(input.sid, output.did)
  FROM
    output INNER JOIN input
      ON (output.did = input.did
       AND output.execId = input.execId)
    INNER JOIN instanceOf inputInst
      ON (input.sid = inputInst.sid)
    INNER JOIN instanceOf outInst
      ON (output.sid = outInst.sid)
  WHERE execId=1
  START WITH output.did='d447'
  CONNECT BY PRIOR output.sid=input.sid
     AND isMarked(input.sid, output.did)=0
     AND PRIOR output.ts>output.ts;
```

This query first creates an inline view (FROM clause) of the data exchange, with, for each row: (i) a data id, (ii) *prodStep* (*consStep*) the step that produced (consumed) this data id. It then moves recursively through this (flattened) data exchange, finding a row's predecessor using the condition that this row produced some data which was consumed by the current *prodStep*, and this data was read by *prodStep* before it output the current data id (test on timestamps). The call to ISMARKED in the CONNECT BY condition is a stored procedure used to stop the recursion when a node has already been seen in the DAG.

Relation $uProv$ could have been implemented by first calculating *cInput* and *cOutput* using our extended operator, and using it again to yield *uProv*. However, this involves three transitive closures together with union, which is very expensive. We therefore implement $uProv$ as a stored procedure as follows: Compute *prov* (using the extended operator), and join with *cStep* to remove all data exchange that occur inside a composite step (produced/consumed by steps in a same composite step). Then replace remaining step ids with the composite step id in which they are included. This is equivalent to joining *prov* with *cInput* and *cOutput* (which hides data exchanges inside the user view composite step). However, as it only has one call to a recursive procedure it is much more efficient, making it possible to cope with databases containing large data sets.

2.2 UserViewBuilder

UserViewBuilder assists users when user views have not been defined by the designer of the workflow or when the defined user views are not what users want. At the core is an algorithm which takes as input a workflow specification and set of relevant modules, and produces a user view.

But what is a "good" user view? Intuitively, the user should see in the induced workflow a composite module for each relevant module. Futhermore, paths between relevant modules in the workflow specification should be preserved in the induced workflow specification formed by the user view: None should be lost and none should be added. The inputs to and outputs from relevant modules must also be preserved. Finally, the induced workflow specification should hide as much detail about uninteresting modules as possible.

UserViewBuilder takes as input the workflow definition, loads it in a graphical environment and allows users to specify which modules are relevant by flagging them through the interface. The algorithm for building the user view is run interactively, so that the user can visualize each new user

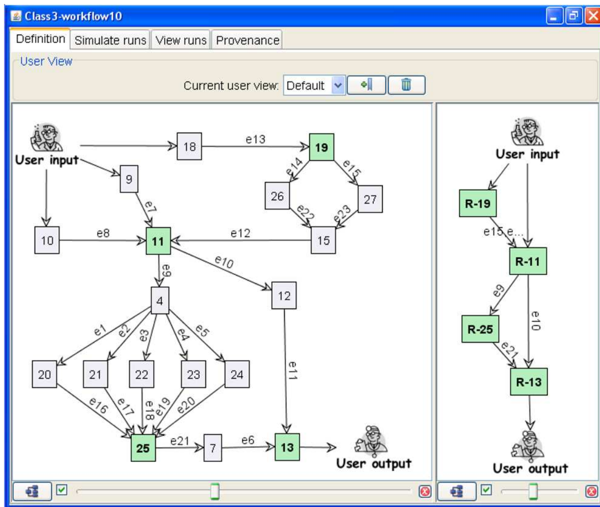

Figure 5: Constructing and Visualizing user views

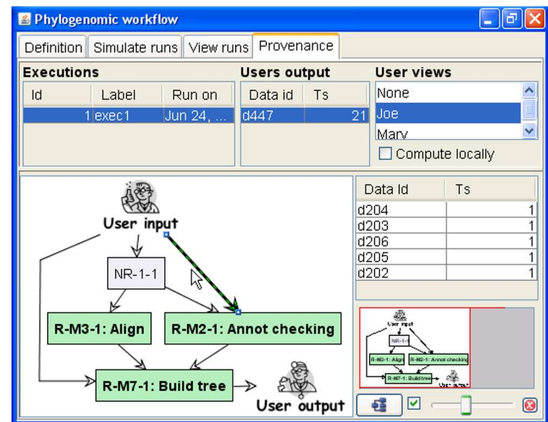

Figure 6: Graph of provenance information (Q)

view as they flag or unflag nodes.

3. DEMONSTRATION

ZOOM*UserViews has been implemented in Java. Our demonstration will highlight the following features:

Loading Data: Users may load a workflow specification and a user view into the system. We will also show how to generate synthetic runs, which can be used to validate the user view or to demonstrate the numerous capabilities of our approach.

Generating user views: Given a graphical display of the workflow definition, the user can flag a module as relevant. The user view is then computed, and the workflow specification induced by the user view displayed. The left hand side of Figure 5 shows an example of a complex workflow where four modules (19, 11, 25, and 13) have been flagged as relevant. The user view produced by UserViewBuilder is shown on the right hand side.

Querying and Visualizing Provenance information: Runs are displayed graphically. By selecting a run and clicking on an edge between two steps, the user can see the data set passed between them. To query provenance, the user selects the data id of interest. Requested provenance information is then calculated with respect to the user view, and displayed as a graph. When run or provenance graphs are long, the user can navigate over the portion of graph he is interested in. As the user's needs evolve, the user may modify (add or remove) the set of modules he considers to be relevant. The provenance graph is then automatically modified for the new user view. For example, the answer to the deep provenance of the final output (data id d447) using Joe's view (Section 1) is shown in Figure 6.

What are the database challenges in ZOOM? The database challenges in ZOOM include:

1. Modeling the information typically available in the logs of scientific workflow systems (see [4]), and showing how to use it in reasoning about provenance.

2. Developing abstraction techniques for reducing the amount of provenance information returned in queries, while ensuring that "relevant" information is returned.

3. Automatically constructing user views based on input from the user.

4. Efficiently implementing provenance queries.

Why would this demo be interesting for the database community? Within the database community, *data integration* has been a long-standing challenge and *provenance* has been a topic of increasing interest. Recent VLDB keynote talks, many research papers, as well as specialized workshops have focused on those two topic areas. However, within the scientific database community there is increasing recognition that (i) data integration is frequently captured through workflows rather than by just queries and (ii) provenance information is frequently overwhelming for the end-user (scientist). Abstraction techniques are thus of increasing interest. This demonstration is at the intersection of these two topic areas, and provides an abstraction technique that is theoretically and practically interesting to the database community.

4. REFERENCES

[1] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with Uncertainty and Lineage. In *Proc. Conference on Very Large Data Bases (VLDB)*, pages 953–964, 2006.

[2] S. Bowers and B. Ludäscher. Actor-oriented design of scientific workflows. In *Proc of ER'05, International Conference on Conceptual Modeling*, pages 369–384, 2005.

[3] S. Cohen, S. Cohen-Boulakia, and S. Davidson. Towards a model of provenance and user views in scientific workflows. In *Proc. of Data Integration in the Life Sciences (DILS)*, volume 4075 of *Lecture Notes in Bioinformatics*, pages 264–279. Springer, 2006.

[4] L. Moreau and J. Freire. The first and second provenance challenges., 2006. http://twiki.ipaw.info/bin/view/Challenge/.

[5] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. Greenwood, K. Carver, M. G. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(1):3045–3054, 2003.