Approaches for Exploring and Querying Scientific Workflow Provenance Graphs

Manish Kumar Anand[1], Shawn Bowers[2], Ilkay Altintas[1], and Bertram Ludäscher[1,3]

[1] San Diego Supercomputer Center, University of California, San Diego, USA
[2] Department of Computer Science, Gonzaga University
[3] UC Davis Genome Center, University of California, Davis
mkanand@sdsc.edu, bowers@gonzaga.edu, altintas@sdsc.edu, ludaesch@ucdavis.edu

Abstract. While many scientific workflow systems track and record data provenance, few tools have been developed that provide convenient and effective ways to access and explore this information. Two important ways for provenance information to be accessed and explored is through browsing (i.e., visualizing and navigating data and process dependencies) and querying (e.g., to select certain portions of provenance graphs or to determine if certain paths exist between items within a graph). We extend our prior work on representing and querying data provenance by showing how these can be effectively and efficiently combined into an interactive provenance browser. The browser allows different views of provenance to be explored and queried, where queries are expressed in a declarative graph-based provenance query language. Query results are expressed as provenance subgraphs, which can be further visualized and navigated through the browser. The browser supports a generic model of provenance that can be used with various workflow computation models, and has a direct translation to the Open Provenance Model. We present the provenance model, the query language, and describe the overall browser architecture and implementation.

1 Introduction

Scientific workflow provenance is commonly represented using data and process *dependency graphs* [1,2] in which dependencies represent causal relationships among data products and/or process invocations. As workflows are executed, many workflow execution environments (e.g., [3,4,5]) store the associated provenance graphs within dedicated *provenance stores* (i.e., databases). This provenance information is of great interest to scientists and other users, e.g., for determining data lineage, result interpretation, and evaluating the quality of workflow results.

While many workflow systems store provenance information, few provide users with tools for effectively and efficiently exploring, accessing, and querying the provenance graphs associated with workflow runs. In this paper, we address problems in exploring and querying provenance information by presenting approaches that combine provenance visualization and navigation with support for incremental query. These approaches have been implemented within a provenance browser application. The browser supports a generic model of provenance that is compatible with a number of existing

D.L. McGuinness, J.R. Michaelis, and L. Moreau (Eds.): IPAW 2010, LNCS 6378, pp. 17–26, 2010.

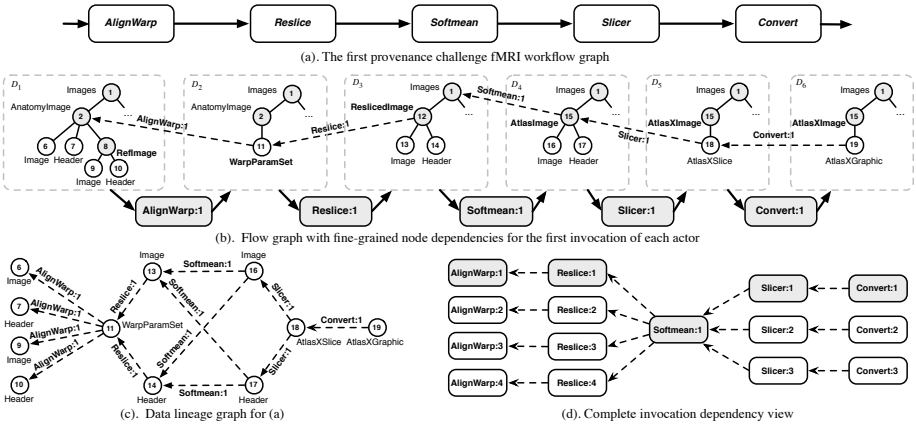

Fig. 1. (a) Workflow for the fMRI image analysis of the first provenance challenge; (b) Trace showing the first invocations of each actor (for a typical run); (c) Implied fine-grain data dependency graph for the data items in (b); and (d) Implied invocation dependency graph for the run, with the first invocations of each actor shown in gray

provenance models (including the Open Provenance Model [1]), and incorporates storage and query optimization that makes browsing and querying over (often large and complex [6]) provenance graphs feasible.

Contributions. This paper describes the different approaches used in implementing the provenance browser (first presented in [7]). In particular, we briefly describe the provenance model and query language (QLP) of the browser and show how these approaches can easily accommodate the Open Provenance Model (OPM). We then present the provenance browser[1] focusing on its architecture, implementation, and optimization techniques. The implementation allows users to easily navigate different aggregated views of underlying provenance graphs and specify new provenance views through incremental queries. Our approach also maintains the user's query history, allowing users to go back to and navigate previous query results.

2 Model of Provenance and Query Language

The Provenance Model. We assume workflow runs follow a standard dataflow-based workflow computation model (e.g., [3,4]). However, our provenance model also supports processes that (1) can execute multiple times in a workflow run (and in parallel), and (2) can recieve and produce data products that are *structured* via labeled, nested collections (e.g., data can be organized into XML structures).

Consider the simple workflow in Fig. 1(a) representing the First Provenance Challenge fMRI workflow [1]. Steps in the workflow are referred to as *actors* that are *invoked*

[1] The browser is open-source and can be freely downloaded from `http://www.daks.ucdavis.edu/projects/pb`, or from within Kepler as the "provenance-browser" extension module.

over input data supplied by previous steps. This example takes a set of anatomy images representing 3D brain scans and a reference image, and creates a graphical image for each 2D slice. In this example workflow implementation we assume each invocation of an actor receives an XML data structure, performs an update on a portion of that structure, and then sends the updated version of the structure to downstream actors (see Fig. 1(b)).

Fig. 1(b) shows the first invocation of each actor for a typical run of the workflow. The invocation of the AlignWarp actor (*AlignWarp:1*) modifies the first AnatomyImage collection (node 2), and replaces its contents with a WarpParamSet data token (node 11). The invocation of the Reslice actor uses this WarpParamSet to generate a new Image and Header data token (nodes 13 and 14, respectively). Since only a part of an XML data structure D may be modified by an invocation, we also represent explicit data dependencies as part of a run. For example, the dashed arrow from node 11 to node 2 in Fig. 1(b) states that the WarpParamSet was created from the AnatomyImage collection by the first invocation of AlignWarp. Note that node 11 implicilty depends on each of the descendents of node 2. Each descendent of a collection also implicitly inherits the dependencies of its ancestors, e.g., node 13 depends on node 11 since it is a descendent of node 12. Taken together, Fig. 1(b) denotes a portion of the *trace* for a run of Fig. 1(a), in this case corresponding to the first invocation of each workflow actor.

More formally, we define a *trace* as an acyclic digraph $T = (V, E, \tau, L)$. Each vertex $V = S \cup I$ represents either a data structure $s \in S$ or an actor invocation $i \in I$. Edges $E = E_{in} \cup E_{out}$ are *in-edges* $E_{in} \subseteq S \times I$ or *out-edges* $E_{out} \subseteq I \times S$. Each trace includes a function $\tau : S \rightarrow X$ that maps structures $s \in S$ to their corresponding XML trees $\tau(s) \in X$. We assume an underlying space of XML nodes N from which XML trees X are built, and a function $\text{nodes}(x) \subseteq N$ that gives the nodes of a structure $x \in X$. Further, we allow different versions of XML trees $\tau(s) \in X$ to *share* nodes from N. To support fine-grained dependencies we consider ternary node-level *lineage relations* $L \subseteq N \times I \times N$ such that $(n_1, i, n_2) \in L$ implies n_1 was required for the derivation of n_2 by i. We define the relations *in*, *out*, *ddep*, and *idep* using the following Datalog rules.

$$\begin{aligned}
\text{in}(n,i) &\text{ :- } E_{in}(s,i),\ n \in \text{nodes}(\tau(s)).\\
\text{out}(i,n) &\text{ :- } E_{out}(i,s),\ n \in \text{nodes}(\tau(s)).\\
\text{ddep}(n_2,n_1) &\text{ :- } L(n_1,i,n_2).\\
\text{idep}(i_2,i_1) &\text{ :- } L(n_1,i_i,n_2), L(n_2,i_2,n_3).
\end{aligned}$$

The last two relations can be used to construct standard dependency graphs, e.g., see Fig. 1(c) and (d).

Correspondence to the Open Provenance Model (OPM). OPM traces can be represented using the above model (with the major difference being that OPM lacks explicit support for modeling structured data). Within OPM, *artifacts* denote "opaque" data objects (atomic with respect to OPM), *used* edges relate artifacts to the processes they were input to, *wasGeneratedBy* edges relate artifacts to the process they were produced by, *wasDerivedFrom* edges define data dependencies between artifacts, and *wasTriggeredBy* edges denote dependencies between processes. OPM employs the following first-order constraints (i.e., "completion rules") over these edges [1].

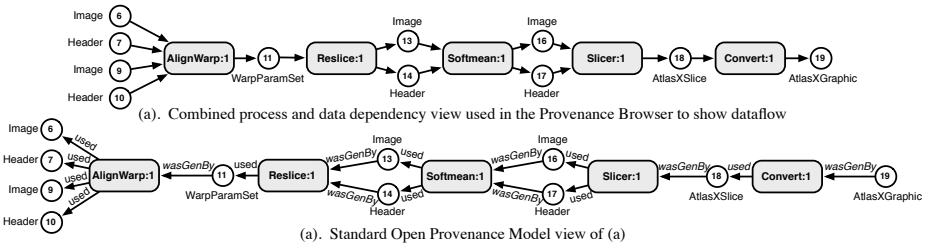

Fig. 2. (a) A standard provenance-browser view of a provenance graph; and (b) the same view with corresponding OPM edges

$$\forall a \forall p_1 \forall p_2 (\text{wasGeneratedBy}(a, p_1) \wedge \text{used}(p_2, a) \rightarrow \text{wasTriggeredBy}(p_2, p_1))$$
$$\forall p_1 \forall p_2 (\text{wasTriggeredBy}(p_2, p_1) \rightarrow \exists a (\text{wasGeneratedBy}(a, p_1)) \wedge \text{used}(p_2, a))$$
$$\forall a_1 \forall a_2 (\text{wasDerivedFrom}(a_2, a_1) \rightarrow \exists p (\text{used}(p, a_1) \wedge \text{wasGeneratedBy}(a_2, p)))$$

If OPM artifacts are represented as single-node tree structures, we have the following equivalences: *used* corresponds to the *in* relation; *wasGeneratedBy* corresponds to the *out* relation; *wasDerivedFrom* corresponds to the *ddep* relation; and *wasTriggeredBy* corresponds to the *idep* relation. Given these, it can easily be shown that the OPM completion rules for *wasDerivedFrom* and *wasTriggeredBy* are equivalent to the *ddep* and *idep* rules, respectively. Fig. 2(a) shows a standard provenance view used by the provenance browser that gives the *in* and *out* relations for the first actor invocations of the example in Fig. 1; and Fig. 2(b) shows the same basic view but using the corresponding OPM *used* and *wasTriggeredBy* edges. A similar graph can be constructed for the *ddep* to *wasDerivedBy*, and *idep* to *wasTriggeredBy* correspondences.

The Query Language for Provenance (QLP). QLP queries are expressed against provenance trace graphs, and can include constructs for querying the different dimensions of traces including *lineage relations* among nodes and invocations, *in-out edges* among input and output structures of invocations, and *structural relations* among nodes within and across data structures. The syntax of QLP is similar in spirit to tree and graph-based languages such as XPath and generalized path expressions, such as those used in Lorel [8]. However, QLP queries primarily act as *filters* over lineage relations. That is, given a set of lineage relations, a QLP query selects and returns a subset of the relations.

To illustrate QLP, the lineage queries "*.. 19", "6 .. *", and "#Softmean .. #Convert.. *" return lineage relations denoting sets of paths that: start from any node and end at node 19; start at node 6 and end at any node; and start at invocations of Softmean and pass through invocations of the Convert actor; respectively. See [9] for other QLP constructs and functions.

Applying QLP to OPM Provenance Graphs. QLP queries can be directly evaluated against OPM graphs based on the straightforward mapping described above. QLP queries expressed against OPM graphs of the form "n_1 .. n_2" select the set of OPM edges that lie on a *wasDerivedFrom* path starting at artifact n_2 and ending at artifact n_1.

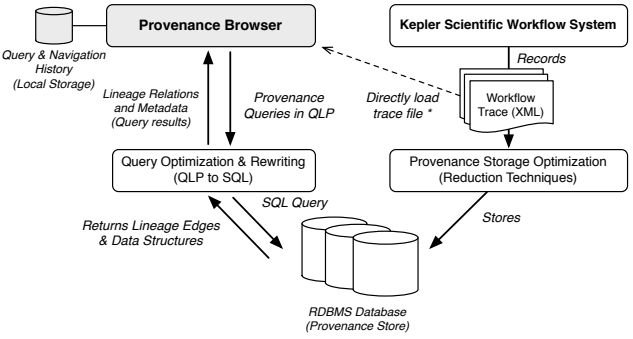

Fig. 3. The basic provenance browser architecture where trace files can be loaded either directly into the browser or through a dedicated provenance store (relational database)

Thus, if n_2 is connected to n_1 either directly or transitively through one or more *wasDerivedFrom* edges, then all edges, processes, and artifacts on this path are returned. QLP queries of the form "$\#p_1$ .. $\#p_2$" select the set of OPM edges, artifacts, and processes that lie on a *wasTriggeredBy* path from process p_2 to process p_1. Again, if p_2 is connected to p_1 through one or more *wasTriggeredBy* edges, then all edges, processes, and artifacts along the path are returned as a result of the query. For QLP queries of the form "n .. $\#p$" and "$\#p$ .. n", we first find the output and input artifacts of process p and then use these artifacts to find *wasDerivedFrom* paths to and from n, respectively. More complicated QLP expressions such as "n_1 .. $\#p_1$ .. n_2" and "$\#p_1$ .. n_1 .. n_2 .. $\#p_3$" are evaluated based on these simple patterns as described in [10,9].

3 The Provenance Browser

The provenance browser provides an interactive approach for visualizing and querying provenance traces. The basic architecture of the browser is shown in Fig. 3 and two standard views of provenance information are shown using the browser in Fig. 4.

The Provenance Browser Architecture. The provenance browser has been integrated with the Kepler scientific workflow system [3,7] and can also be run as a stand-alone application. Given a trace file, a set of pre-processing steps are applied to the trace prior to storage in a provenance database. The pre-processing steps perform storage reduction techniques (based on factorization) over the data lineage graph of the workflow trace as described in [9]. Using the provenance browser, a user can connect to a provenance store to select traces to view, issue QLP queries against traces, and then display, navigate, and further query these results. As shown in Fig. 3, QLP queries are parsed, optimized, and rewritten to corresponding SQL queries expressed against the provenance database. Optimized and translated SQL queries return sets of lineage edges as query results from which the browser constructs and displays the corresponding lineage graph. Finally, the browser maintains (i.e., caches) query results as well as navigation and query history

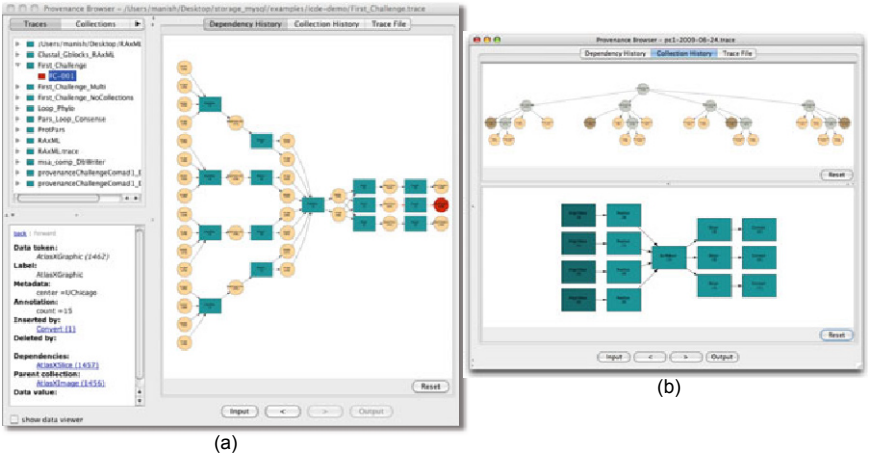

Fig. 4. Two basic provenance views supported by the browser: (a) The in-out edge representation over data and invocation dependencies; and (b) the corresponding collection structure and invocation graph after the first set of actor invocations

locally. Local storage, e.g., allows query results to be accessed and viewed efficiently within the browser.

Visualization and Navigation. As shown in Fig. 4(a), the left-side of the provenance browser displays the XML collection structure together with the details of actor invocations. Much like a web browser, this information can be navigated (e.g., to select among different data items and invocations). The browser also displays various provenance views of the execution trace: the *dependency history* view of Fig. 4(a) combines data dependency and process invocation graphs (where data nodes are denoted as circles and invocations as squares); the *collection history* view at the top of Fig. 4(b) shows the data structures input and output by invocations; and the *invocation dependency* view at the bottom of Fig. 4(b) shows process dependencies. Each of these views are synchronized, e.g., selection of a data item in the dependency history view also selects the corresponding item in the collection history view. Within a view, users can also step forward and backward ("VCR-style") through the execution history to display corresponding portions of the XML structures and data dependencies.

Incremental Querying. Fig. 5 shows an example of an incremental query session within the provenance browser. As shown in Fig. 5(c), the provenance browser contains a separate query window for users to issue QLP queries. In this example, we have selected the workflow run "FC-001" (see Fig. 4(a)). In general, users can select one or more traces from a provenance store using the query window. Once selected, the default views of the traces are displayed in the browser. If a workflow is not selected, all traces within the provenance store will be queried (but not initially displayed in the browser). After a query is entered into the query window and executed, the new provenance views that correspond to the query answer are constructed and displayed in the browser. In the

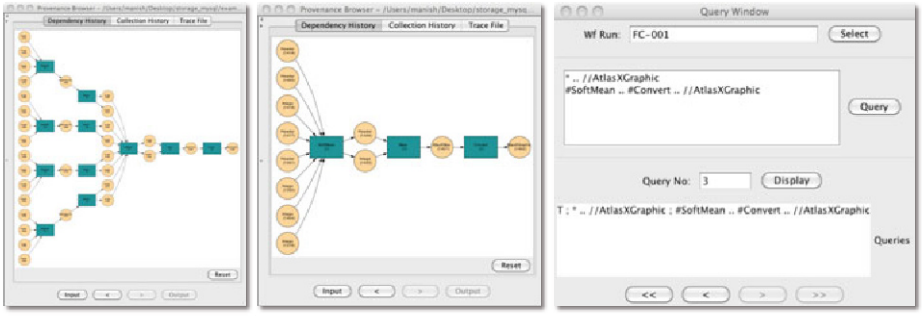

Fig. 5. Incrementally query support in the provenance browser: (a) the result of the first QLP query (expressed over the lineage graph of Fig. 4(a)); (b) the result of the second QLP query expressed over the result of the first query; and (c) the query window

example of Fig. 5, the user first enters the query "* .. //AtlasXGraphic" whose query result is shown in Fig. 5(a). This new view can be explored and navigated in exactly the same way as if the entire trace were displayed. The query result can also be further queried, as shown by the second query in Fig. 5(c) whose result is shown in Fig. 5(b).

Query History. The browser automatically captures the sequence of queries issued by the user (where the initial view is denoted as "T" in the query window). Users can return to a previous query result at any time by selecting the query from the query history or using the forward-backward buttons within the query window. In our example, selecting the first query (denoted query 2, since the initial trace is treated as query 1) would return the view from Fig. 5(b) back to Fig. 5(a). We could then continue browsing, return to the view in Fig. 5(b), move to the original view in Fig. 4(a), or issue a new query to generate a different view. The ability to incrementally query provenance graphs allows users to more easily inspect and explore relevant portions of large provenance graphs (containing, e.g., hundreds or thousands of nodes), which contrasts with more static approaches that simply display entire lineage graphs.

Optimization Techniques. Queries in QLP largely involve evaluation of transitive path queries, expressed in QLP whenever the '..' operator is used. To evaluate queries efficiently, we store both the immediate edges and transitive closure of dependency nodes. Naively materializing transitive closures for each dependency node can have prohibitively large overhead with respect to storage cost. To reduce this storage cost, we employ a "pointer-based" approach that partitions the transitive closure table into smaller tables (based on reduction techniques) where the original transitive closure table can be obtained by joining these smaller tables together [9]. In particular, a naive approach for storing nodes N and their dependencies D is as tuples $P(N,D)$ in which nodes involved in shared dependencies will be stored multiple times. For example, if nodes n_4, n_5, and n_6 each depend on nodes n_1, n_2, and n_3, nine tuples must be stored $P(n_4,n_1)$, $P(n_4,n_2)$, $P(n_4,n_3)$, $P(n_5,n_1)$, ..., $P(n_6,n_3)$, where each node is stored multiple times in P. Instead, we introduce additional levels of indirection through "pointers"

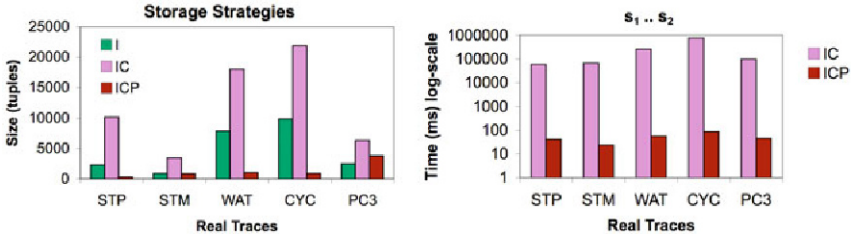

Fig. 6. Example sizes (left) and query time (right) for real provenance traces

(similar to vertical partitioning) for storing reduced sets of dependencies. Thus, we divide $P(N,D)$ into two relations $P_1(N,X)$ and $P_2(X,D)$ where X denotes a pointer to the set of dependencies D of N.[2] For instance, using this approach we store only six tuples $P_1(n_4,\&x)$, $P_1(n_5,\&x)$, $P_1(n_6,\&x)$, $P_2(\&x,n_1)$, $P_2(\&x,n_2)$, and $P_2(\&x,n_3)$ for the above example. Additional levels of indirection are also used to further reduce redundancies within dependency sets based on their common subsets, and similar techniques are used to reduce transitive dependency sets by applying reduction techniques directly to pointers (as described in [11]). We also use a set of optimization techniques [9] to efficiently evaluate the generic lineage path queries of the form $p = N_1 \,..N_2 \cdots N_m$, where each N_i can correspond to a set of nodes. It may be the case that not all the nodes in the set N_i share a lineage relationship with all the nodes in the other set. We use techniques to prune each of the sets N_i giving a new pruned set PN_i such that each node $n_i \in PN_i$ shares a dependency relationship with at least one node in all other sets, and vice versa. Each pruned set is computed as $PN_i = (\cap_{j=1}^{j=i-1} N_{F_i}) \cap N_i \cap (\cap_{k=i+1}^{k=m} N_{B_i})$, where N_{F_i} is "forward" lineage nodes for N_i (i.e., $N_i \,..*$), and N_{B_i} is the "backward" lineage nodes for N_i (i.e., $*\,..N_i$). We rewrite p to $PN_1 \,..PN_2 \cdots PN_m$ and evaluate each simple path $PN_i \,..PN_{i+1}$. Each of these simple paths are first evaluated to retrieve the set of nodes N on the lineage path from nodes in PN_i to nodes in PN_{i+1} as $PN_{F_i} \cap PN_{B_i}$. Finally, the set of lineage edges is computed from N.

The left side of Fig. 6 shows the sizes of actual provenance traces generated from metagenomic (STP, STM, and CYC), phylogenetic (WAT), and astronomy (PC3) workflows. As shown, the provenance graphs are relatively large, containing between ∼5–20K lineage edges, as shown by the number of tuples when only immediate edges (I) are stored. Even for simple QLP lineage queries, e.g., involving single-step derivation path expressions "$s_1 \,.. \,s_2$', standard evaluation techniques result in query execution times that are impractical. For instance, by storing both immediate and transitive dependencies (IC), these simple queries can take upwards of 1000 s (these times are worse if only immediate edges are stored, since recursion is required). As shown in Fig. 6, employing these storage and reduction techniques together with the query optimization approaches reduces query execution time to less than 100 ms, making common (but relatively complex) graph queries practical within the provenance browser.

[2] The actual partitioning is slightly more complex than this, but follows the same general idea.

4 Related Work

Many tools have been created (e.g., in Kepler [3] and Taverna [4], among others) that statically display provenance graphs or provide a log of provenance events. Exceptions include VisTrails [5], which provides a browser for displaying workflow edit operations, and the Zoom*UserViews prototype [12], which simplifies provenance graphs by inferring composite invocations. The provenance browser combines visualization with navigation and a declarative, high-level provenance query language (QLP). Standard approaches for querying provenance information (e.g., [13,5]) return sets of nodes (either sets of data items or process invocations) as query results that require additional steps (or queries) to reconstruct causal relations among nodes within a query answer. Instead, QLP returns sets of lineage edges, which enables incremental querying. We've shown here that a subset of our model has a direct mapping to OPM [1]. It also offers a more flexible approach for modeling nested data (e.g., compared to [14]) by not requiring processes to generate entirely new structures, and supporting update semantics.

5 Conclusion

We extend our prior work [15,7] by describing the architecture, implementation, and underlying approaches of the provenance browser. The browser is based on a general model of provenance and a high-level, declarative query language. In addition, we have also shown how OPM aligns with our model of provenance and query language. We would like to extend the provenance browser with the navigation operators [16], so that users visualize, summarize, and query provenance views from this environment.

Acknowledgements. This research was supported in part by NSF grants OCI-0722079, and DBI 0619060, DOE grant DE-FC02-07ER25811, and the Gordon and Betty Moore Foundation award to Calit2 at UCSD for CAMERA.

References

1. Moreau, L., et al.: The open provenance model. Technical Report 14979, ECS, Univ. of Southampton (2007)
2. Davidson, S.B., Boulakia, S.C., Eyal, A., Ludäscher, B., McPhillips, T.M., Bowers, S., Anand, M.K., Freire, J.: Provenance in scientific workflow systems. IEEE Data Eng. Bull. (2007)
3. Ludäscher, B., et al.: Scientific workflow management and the Kepler system. Concurr. Comput.: Pract. Exper. 18, 1039–1065 (2006)
4. Oinn, T., et al.: Taverna: lessons in creating a workflow environment for the life sciences. Concurr. Comput.: Pract. Exper. 18, 1067–1100 (2006)
5. Scheidegger, C., et al.: Tackling the provenance challenge one layer at a time. Comput.: Pract. Exper. 20, 473–483 (2008)
6. Chapman, A., et al.: Efficient provenance storage. In: SIGMOD (2008)
7. Bowers, S., McPhillips, T., Riddle, S., Anand, M., Ludäscher, B.: Kepler/pPOD: Scientific workflow and provenance support for assembling the tree of life. In: Freire, J., Koop, D., Moreau, L. (eds.) IPAW 2008. LNCS, vol. 5272, Springer, Heidelberg (2008)

8. Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J.L.: The Lorel query language for semistructured data. Intl. J. on Digitial Libraries 1, 68–88 (1997)
9. Anand, M.K., Bowers, S., Ludäscher, B.: Techniques for efficiently querying scientific workflow provenance graphs. In: EDBT (2010)
10. Anand, M.K., Bowers, S., McPhilips, T., Ludäscher, B.: Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs. In: SSDBM (2009)
11. Anand, M.K., Bowers, S., McPhilips, T., Ludäscher, B.: Efficient provenance storage over nested data collections. In: EDBT (2009)
12. Biton, O., Boulakia, S.C., Davidson, S.B., Hara, C.S.: Querying and managing provenance through user views in scientific workflows. In: ICDE (2008)
13. Holland, D., Braun, U., Maclean, D., Muniswamy-Reddy, K.K., Seltzer, M.: A data model and query language suitable for provenance. In: Moreau, L., Foster, I. (eds.) IPAW 2006. LNCS, vol. 4145, Springer, Heidelberg (2006)
14. Missier, P., Belhajjame, K., Zhao, J., Goble, C.: Data lineage model for taverna workflows with lightweight annotation requirements. In: Freire, J., Koop, D., Moreau, L. (eds.) IPAW 2008. LNCS, vol. 5272, pp. 17–30. Springer, Heidelberg (2008)
15. Anand, M.K., Bowers, S., Ludäscher, B.: Provenance browser: Displaying and querying scientific workflow provenance graphs (Demo) In: ICDE (2010)
16. Anand, M.K., Bowers, S., Ludäscher, B.: A navigation model for exploring scientific workflow provenance graphs. In: WORKS (2009)