Choosing a Data Model and Query Language for Provenance

David A. Holland, Uri Braun, Diana Maclean,
Kiran-Kumar Muniswamy-Reddy, Margo I. Seltzer

Harvard University, Cambridge, Massachusetts
pass@eecs.harvard.edu

Abstract. The ancestry relationships found in provenance form a directed graph. Many provenance queries require traversal of this graph. The data and query models for provenance should directly and naturally address this graph-centric nature of provenance. To that end, we set out the requirements for a provenance data and query model and discuss why the common solutions (relational, XML, RDF) fall short. A semistructured data model is more suited for handling provenance. We propose a query model based on the Lorel query language, and briefly describe how our query language PQL extends Lorel.

1 Introduction

Our provenance aware storage system (PASS), which transparently and automatically collects file-level provenance from unmodified Linux applications [14], tackled the problem of automatic collection, but did not consider querying. We participated in a community effort to compare and contrast different approaches to provenance collection (the First Provenance Challenge [15, 13]), and quickly discovered that query capabilities were integral to the system.

Provenance differs from other forms of meta-data because it is based on relationships among objects. This includes both ancestry relationships (which must be acyclic) and relationships of object identity, which might form cycles. In practice, provenance forms a directed graph, and paths through this graph form the basis of many queries. Seven of the nine Provenance Challenge queries involve paths. Unfortunately, most existing data management and query models are ill suited to handle graphs and paths through graphs.

Most existing provenance systems use relational, XML, or RDF storage, and the corresponding languages for querying [13]. These models are not well suited for querying provenance data. The relational model does not provide native support for graphs, which must be treated as lists of nodes and edges. XML has a different problem; it is hierarchical and does not naturally represent objects with multiple parents. Finally, while the RDF query language SPARQL supports graphs, paths are not first class objects and it lacks various necessary features.

This paper discusses the difficulties with the prevalent query models and describes the PASS choice of an alternative, a semistructured data model. We adopted a variant of the Lore database manager's query language Lorel [1].

In Section 2, we review the essential characteristics of provenance. Sections 3, 4, and 5 describe the shortcomings of relational, XML, and RDF-based query processing respectively. We then turn to query languages for semistructured data in Section 6 and briefly introduce our query language PQL in Section 7. We conclude in Section 8.

2 The Nature of Provenance Data

Provenance data is based on object relationships and is inherently graph-oriented. When an object O is found to have been derived from some other object P, we say that P is an *ancestor* of O, or O is a *descendent* of P. This is an *ancestry* relationship. Because objects may be derived from multiple sources, and objects may have multiple new objects derived from them, ancestry exhibits "diamonds" and is a graph, not a tree. Because these relationships are not symmetric, the graph is directed. Because cyclic ancestry would violate causality, ancestry must be acyclic. Other relationships specifying object *identity* may appear as well, however, and these relationships need not be acyclic. Two objects O_1 and O_2 that together form a single data set might each point to the other. So in general, provenance forms a directed graph but not necessarily a directed acyclic graph. Because there are multiple kinds of ancestry relationships, it must be possible to label the edges of the graph.

While objects may have assorted other provenance-related attributes and annotations, it is the relationships and specifically the ancestry relationships that form the heart of provenance data. These relationships usually lie at the core of provenance queries. Moreover, these relationships are most valuable when considered not one at a time, but in aggregate: the *structure* of the ancestry in a provenance system is the most interesting and most valuable information it provides. We shadowed several computational science users and found that they were often interested in identifying *friends* — groups of files processed the same way. Such processing, a sequence of transformations, appears as a path through the ancestry graph. Handling friends requires not just following paths but comparing and manipulating them. Conversely, given a group of objects one might wish to inspect the processing steps that generated them.

The importance of such queries makes good support for paths necessary for querying provenance. This requires making paths first class objects; it must be possible to treat found paths as language-level objects and operate on them. It is also necessary to be able to follow paths whose exact structure is not known in advance. This requires pattern matching over sequences of graph edges as well as pattern matching against the labels on graph edges (that is, the names of relationship attributes) themselves. We believe that support for full regular expressions over graph edges is important, particularly if combining provenance data from multiple sources whose attribute names may not be fully consistent.

Provenance queries also need aggregation operators. For example, consider the queries that find all objects with at least ten immediate ancestors, or that

find results derived from data sets whose average calibration quality exceeds some threshold. Counting and averaging are aggregation operations.

Any query involving two or more disjoint collections of objects makes subquery support desirable. The question "Are there more objects derived from my data set than from my competitor's data set?" requires issuing two queries and comparing the results. While one can run these queries separately, for large provenance stores or more complex queries than this example it can be important for performance to allow the query planner to see both at once.

Any data model used for provenance should have a natural representation for directed graphs; and any query language should have direct, simple, and straightforward support for reasoning about graphs and paths through them.

Note that this is a matter of language expressivity (and, importantly, usability) at the front end. The choice of back-end storage in any particular provenance system should be driven by implementation considerations.

3 Shortcomings of Relational Provenance

The relational data model is, roughly speaking, the complete antithesis of a graph-oriented model. The only way to represent a graph as relations is as a list of nodes and edges, and the only way to create paths is to join the list to itself repeatedly. This does not mean that queries based on paths cannot be expressed, but it means that such queries must be mentally translated into relational algebra before being written down.

Moreover, queries involving arbitrary repetitions of edges in a path require mental translation to SQL transitive queries. These are now supported in SQL-99 [6] (SQL Server, DB2) and also (using a different syntax) in Oracle Server, but they are complex and awkward. The SQL-99 form requires constructing a view and two sub-queries. Writing complex path queries this way quickly becomes unwieldy; stored procedures are recommended as an alternative method. Oracle's custom syntax suffers from similar problems.

The SQL-99 transitive queries also cannot provide regular expressions over path edges, only paths consisting of simple repetition. It also does not support applying arbitrary predicates to the objects along a path, so path queries cannot readily be combined with additional conditions. For example, a query requesting all documents derived from processes running programs whose names include the string "Microsoft" (e.g., "Microsoft Word", "Microsoft Excel") requires both a transitive closure and a matching operation on object names.

And finally, because paths are not first class objects, there is no way to determine if two paths lead to the same object, cross the same objects, or if two objects are *friends* as defined above.

Two of the teams participating in the First Provenance Challenge used the relational data and query model, and both were hampered by it. ZOOM, using Oracle transitive queries, was unable to answer query seven, which asks for a comparison of two sets of paths. The REDUX team needed to build a special-purpose query tool to handle recursive queries. Two further groups used SQL

for some queries and other methods for other queries, which suggests that they found none of their methods truly satisfactory.

Provenance is fundamentally not relational, and the relational model is at best awkward. It is used regardless only because robust production-grade relational database systems are easily available and readily deployed.

4 Shortcomings of Provenance-as-XML

Unlike the relational data model, XML supports paths. XPath and XQuery [18] allow writing down paths as first-class objects and support placing conditions on path elements. XML thus appears to be a good choice for provenance queries. We in fact pursued this alternative for some time.

However, XML is tree-structured. An XPath path (or an XQuery path query) starts at the root of an XML document and selects some subsection of it, by passing through the various increasingly specific entities that contain that subsection. Because of the containment property, there can be no "diamonds" in XML data. There is no natural support for graphs; because ancestry forms a directed graph rather than a tree, XML and XQuery are a poor fit for provenance.

It is tempting to try to reuse the XML path model and XML query languages and extend them to allow reasoning about paths through a graph. We took this route at first. We spent several months pursuing an XQuery-related query system before ultimately abandoning it. It works, but is unpleasant to use. Furthermore, it turns out that the XML path model is more strongly dependent on the tree structure of XML than it appears at casual inspection. As a result, our query language needed to diverge substantially from XQuery, and its relationship to XQuery became a liability rather than an asset.

For example, in XQuery, a path always has the value of the object at the right end. While one can insert predicates into the path to restrict intermediate objects, it is impossible to extract those objects themselves for further inspection or comparison against elements of other paths. Intermediate objects can also sometimes be the desired query result, such as in "find me all programs that appear in the derivation chains leading from a common ancestor to the following files." These problems can sometimes be worked around by splitting paths into multiple parts; however, doing so sacrifices the expressive power of the path notation.

XQuery lacks general regular expressions on path elements. This allows simple queries like "find all objects", but falls down on more complex conditions that require alternatives or repeated subsequences. XQuery also fails on Query 2 from the Challenge, which asks for an ancestry list stopping at a particular point. The natural way to express this is a path that does not cross any object matching the stopping point. But this requires writing a path where this criterion is applied to a repeating entity, which XQuery does not support.

XQuery paths are composed of names of objects. This is wrong for provenance data; instead, paths should be composed of names of edges between objects. (This is the model used in Lorel.) In our experience objects in provenance queries are

not identified primarily by an object name; they are found by either attribute matching or their position in the graph structure. In the Second Provenance Challenge [16], most groups' data either did not have object names as such or had arbitrarily assigned (and not human-readable) ID codes as names. Typing these into a query engine is not the right approach.

A different way to use XML for provenance is simply to use it as a container, and use XML cross-references to establish a graph. XPath/XQuery paths cannot then traverse the graph, however; one must write procedural code, either in XQuery itself or in some other language, to do the traversals. All of the teams in the First Provenance Challenge that used XML (in the back end or at the query layer) took this approach and wrote code to handle the challenge queries.

Writing code to work around the query language is not desirable. While XML might or might not be an adequate back-end representation, we conclude that XPath and XQuery are not appropriate for provenance data.

5 Shortcomings of RDF and SPARQL

The Resource Description Framework (RDF) is a data format for directed labeled graphs. SPARQL [17] is a query language for graphs stored as RDF. These two points suggest that RDF might be a good data model for provenance data and that SPARQL might be a good query language. However, even though SPARQL is explicitly intended for graphs, it lacks fundamental features as well as other useful query support.

SPARQL has well-known shortcomings. It lacks support for sub-queries, many aggregation functions, and expressions in select clauses. There are extensions that address almost every one of these limitations [7, 11], but no single extension includes everything one wants and extensions are not readily combined.

For provenance, the most serious problem is that SPARQL does not support path variables, constraints on path expressions, or path expressions of arbitrary length [3, 9]. There are extensions for these features also [2, 12]. Of these, SPARQLeR appears to have the most powerful path expressions, but it lacks the other features previously mentioned.

Some of these limitations can be worked around. In the Challenge, teams that used SPARQL (or its predecessor TriQL) worked around the lack of path expressions using two techniques. One approach involved explicitly coding the endpoint node in the query thus avoiding the necessity for a path search in the first place. Another method was to explicitly build the full path as a sequence of single steps. Finding all grandparents of x is thus a three-step process: 1. "let y be the parents of x", 2. "let z be the parents of y", and 3. "return all z's." This sacrifices the power of path notation and does not generalize to arbitrary length paths with repeating elements.

SPARQL is not presently suitable for provenance. It is possible that some future combination of extensions could yield a sufficiently powerful dialect. Currently, however, there are competing designs for each extension feature and no single dialect with all the necessary parts.

6 Semistructured Data

For a solution, we turn to *semistructured data*, a model providing a system of objects with linkages interconnecting them and with no formal predetermined structure. This model, from the object-oriented database community, provides a clean representation of graph-oriented data, which naturally fits provenance.

Furthermore, provenance may include arbitrary application-specified or user-specified annotations whose names and relationships are not known in advance and may vary from site to site. A semistructured data model supports this environment. Traditional object query languages like OQL [5] rely on a fixed schema of possible relationships that is not suitable for storing and integrating provenance collected from multiple sources.

A number of projects have tackled querying of semistructured data, resulting in the query languages UnQL [4], StruQL [8], GOOD [10], and Lorel [1].

UnQL provides a procedural query language based on structural recursion through tree-oriented data. The power of the language is carefully restricted to allow a suitable level of query optimization. However, it only handles graphs by recursively expanding them into trees; this is not good for provenance. It also does not support any concept of paths.

StruQL's primary concern is the ability to transform graphs; a StruQL query is a set of path-based rules for traversing a graph and producing another graph as the output. While this is a potentially useful ability for provenance, it does not seem suitable as the *only* query modality. Furthermore, while StruQL does support paths with full regular expressions over graph edges, the paths are not first-class objects and there is no straightforward way to manipulate or relate them. StruQL also does not appear to be able to address objects found along sections of paths matched by repetition operators, which we believe necessary for cleanly representing queries like Query 2 from the Challenge.

GOOD (Graph Oriented Object Database), like StruQL, is primarily concerned with transforming graphs. It provides five transformation operations to apply independently to a input graph to obtain the desired output graph. Beyond simple forms, these transformations are applied procedurally rather than declaratively. It also does not support paths. These properties make it not suitable for provenance.

Lorel, however, provides almost exactly what we want. The basic element of a Lorel query is a path. Paths are specified as regular expressions rather than mere sequences, and are formed from edge names rather than node names. Paths are first-class objects and can be manipulated at the language level. Objects found within a path (as well as edges and subpaths) can be bound to variables for inspection or restriction. Paths can furthermore be joined to one another in the relational sense. These features allow expressing more complex matching structures, such as "friend" queries, and allow querying the *structure* of the data in addition to the contents. The support for generating complex graphs as results is, however, rather limited.

7 PQL

We have developed our own query language called PQL, which is based on Lorel with extensions for handling provenance. A complete discussion of PQL is beyond the scope of this paper; we describe only the key differences from Lorel.

The most visible difference is that we have extended graph edges to be bidirectional. Lorel's object-oriented worldview has, essentially, pointers: one-way linkages. In provenance, however, every descendent relationship is also an ancestor relationship, and it makes sense to be able to traverse the corresponding graph edge in either direction. This requires a corresponding extension to edge naming. Since navigation is by edge name, and edges generally have directional names (such as `ancestor` or `input`), using the same name to traverse an edge in either direction would cause mass confusion. Maintaining and enforcing a master list of arbitrary pairs of names for each direction of a relationship would defeat one of the main purposes of using semistructured data. We also wanted to rule out being left with half of a bidirectional linkage. We extended the grammar to allow appending `-of` to edge names to specify the reverse direction. Thus in our database you can follow `input` edges forward or `input-of` edges backward.

We also made two relatively minor extensions to edge naming. In Lorel, graph edges and thus relationship names may only be identifiers. We allow integers as well, because some of our data (`argv` arrays) most naturally uses integer indexes. We also allow converting any primitive value (rather than just a previously encountered path segment) to the name of a graph edge in a path. This adds flexibility and also allows handling edges that contain non-identifier characters. We expect to use file system path names (such as `/bin/sh`) as the names of graph edges.

Other extensions include lifting Lorel's restrictions on the complexity of regular expressions in paths, allowing boolean values to appear in the database or be query results, and adding string matching by shell globs (`*.gif`) to Lorel's support for text regular expressions (`^.*\.gif$`). The last is important for our environment, which contains many file system names. We hope to add more powerful graph construction in the future.

8 Conclusions

Querying provenance introduces new and interesting questions about query models. The idea of a path, so central to provenance, is notably absent or seriously limited in most existing models and languages. While one might build systems to support provenance queries in those contexts, we have found no existing complete solution. The Lorel query language and our extended form PQL appear to address the shortcomings of existing systems. However, only further use and experimentation will reveal if we have overlooked some further fundamental property of provenance.

References

1. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel query language for semistructured data. International Journal on Digital Libraries, 1(1):68–88, 1997.
2. F. Alkhateeb, J. Baget, and J. Euzenat. RDF with regular expressions. Research report 6191, INRIA Rhône-Alpes, Grenoble (FR), 2007.
3. K. Anyanwu, A. Maduko, and A. P. Sheth. SPARQ2L: towards support for subgraph extraction queries in RDF databases. In WWW, pages 797–806, 2007.
4. P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. VLDB Journal, 9(1):76–110, March 2000.
5. R. Cattell. The Object Database Standard: DBMG-93. Morgan Kaufman, 1994.
6. A. Eisenberg and J. Melton. Sql:1999, formerly known as sql3. SIGMOD Record, 1999.
7. SPARQL//Extensions. `http://esw.w3.org/topic/SPARQL/Extensions`.
8. M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. A query language for a web-site management system. SIGMOD Record, 26(3):4–11, 1997.
9. GLEEN: Regular Paths for ARQ SparQL. `http://sig.biostr.washington.edu/projects/ontviews/gleen/index.html`.
10. M. Gyssens, J. Paredaens, J. V. den Bussche, and D. van Gucht. A graph-oriented object database model. IEEE Transactions on Knowledge and Data Engineering, 6(4):572–586, 1994.
11. SPARQL Extensions. `http://jena.hpl.hp.com/wiki/SPARQL_Extensions`.
12. K. Kochut and M. Janik. SPARQLeR: Extended Sparql for semantic association discovery. In ESWC, pages 145–159, 2007.
13. L. Moreau et al. The First Provenance Challenge. *Concurrency and Computation: Practice and Experience*. Published online. DOI 10.1002/cpe.1233, April 2008.
14. K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In Proceedings of the 2006 USENIX Annual Technical Conference, June 2006.
15. The First Provenance Challenge. `http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge`.
16. The Second Provenance Challenge. `http://twiki.ipaw.info/bin/view/Challenge/SecondProvenanceChallenge`.
17. E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF, January 2008.
18. XQuery 1.0: An XML query language.