Managing Scientific Workflow Provenance

By

MANISH KUMAR ANAND

B.Tech. (International Institute of Information Technology, Hyderabad) 2004
M.S. (University of California, Davis) 2009

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

_______________________________
Bertram Ludäscher (Chair)

_______________________________
Shawn Bowers

_______________________________
Felix Wu

Committee in Charge

2010
Managing Scientific Workflow Provenance

Abstract

An advantage of scientific workflow systems over traditional approaches is their ability to automatically record the provenance (or lineage) of intermediate and final data products generated during workflow execution. The provenance of a data product contains information about how the product was derived, and is crucial for enabling scientists to easily understand, reproduce, and verify scientific results. This work addresses challenges for managing large amounts of provenance information, and describes efficient approaches to model, store, query, visualize, and explore provenance information.

Specifically, a model of provenance is presented that extends the conventional provenance model, supports nested data, and captures fine-grained lineage information. Novel reduction techniques are described to optimize storage size, update time, and query-response time. A high-level query language is proposed to allow non-experts to easily express provenance graph queries over the model. Query optimization techniques that leverage the storage reductions are described. These optimizations scale with the size of provenance and query complexity, and can also be used in more general settings to efficiently answer a broad range of path queries over labeled, acyclic directed graphs. To further allow users to explore relevant provenance information, a navigation model for provenance is proposed that provides an integrated approach for creating provenance views, navigating between views, and summarizing views. To demonstrate the approaches presented, a Provenance Browser application has been developed and integrated into the Kepler scientific workflow system.
Contents

Abstract ii

List of Figures vii

List of Tables xi

1 Introduction 1
  1.1 Motivation ........................................ 1
  1.2 Scientific Workflow Challenges ...................... 3
  1.3 Provenance Management Challenges ................... 7
  1.4 Dissertation Starting Point ........................ 11
  1.5 Research Questions ................................ 11
  1.6 Research Contributions ............................ 13
  1.7 Dissertation Structure .............................. 16

2 Preliminaries 18
  2.1 Kepler Workflow System ............................ 18
  2.2 Collection-Oriented Modeling and Design ............. 21

3 A Unified Provenance Model 25
3.1 Provenance Models ................................................. 26
3.2 Basic (Conventional) Model .................................... 29
3.3 Multiple Invocation Model ..................................... 30
3.4 Nested Model (Copy Semantics) ............................... 31
3.5 Nested Model (Update Semantics) ............................ 32
3.6 The Unified Model .............................................. 33
3.7 Running Example ............................................... 35
3.8 Related Work ................................................... 38
3.9 Conclusion ...................................................... 42

4 Efficient Provenance Storage ..................................... 43
4.1 Applying Update Semantics .................................... 44
4.2 Condensed Unified Provenance Model ....................... 46
4.3 Formalization of Condensed Unified Provenance Model ....................... 48
4.4 Minimizing Provenance Annotations ......................... 50
4.5 Storage Strategies .............................................. 56
4.6 Storage Reduction Techniques .............................. 58
4.7 Reducing overall Provenance Storage Size .................. 64
4.8 Rewriting Provenance Queries ............................... 66
4.9 Evaluation of Storage Strategies and Reduction Techniques .......... 68
4.10 Related Work ................................................... 79
4.11 Conclusion ...................................................... 81

5 A Query Language for Provenance (QLP) ....................... 83
5.1 Limitations of Existing Approaches to Query Provenance ........... 84
5.2 Desiderata of a Provenance Query Language ................. 85
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3 Provenance Queries using QLP</td>
<td>88</td>
</tr>
<tr>
<td>5.3.1 Queries over Lineage Relations</td>
<td>90</td>
</tr>
<tr>
<td>5.3.2 Queries over Flow Relations</td>
<td>94</td>
</tr>
<tr>
<td>5.3.3 Queries over Structural Relations</td>
<td>96</td>
</tr>
<tr>
<td>5.3.4 Queries over Each Dimension</td>
<td>97</td>
</tr>
<tr>
<td>5.3.5 Additional Functions over Lineage Relations</td>
<td>98</td>
</tr>
<tr>
<td>5.4 QLP Implementation and Evaluation</td>
<td>100</td>
</tr>
<tr>
<td>5.5 Related Work</td>
<td>104</td>
</tr>
<tr>
<td>5.6 Conclusion</td>
<td>107</td>
</tr>
</tbody>
</table>

6 Efficiently Querying Provenance Graphs   108

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Querying Provenance Graphs</td>
<td>108</td>
</tr>
<tr>
<td>6.2 QLP Syntax, and Semantics</td>
<td>111</td>
</tr>
<tr>
<td>6.2.1 QLP Syntax</td>
<td>112</td>
</tr>
<tr>
<td>6.2.2 QLP Semantics</td>
<td>113</td>
</tr>
<tr>
<td>6.3 QLP Evaluation Techniques</td>
<td>117</td>
</tr>
<tr>
<td>6.4 Evaluation of Query Optimization Techniques</td>
<td>127</td>
</tr>
<tr>
<td>6.5 Related Work</td>
<td>136</td>
</tr>
<tr>
<td>6.6 Conclusion</td>
<td>138</td>
</tr>
</tbody>
</table>

7 A Navigation Model for Provenance             139

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Exploring Large Provenance Graphs</td>
<td>139</td>
</tr>
<tr>
<td>7.2 Navigation Model</td>
<td>141</td>
</tr>
<tr>
<td>7.2.1 Provenance Views</td>
<td>143</td>
</tr>
<tr>
<td>7.2.2 Navigating Provenance Views</td>
<td>146</td>
</tr>
<tr>
<td>7.2.3 System Architecture</td>
<td>157</td>
</tr>
</tbody>
</table>
7.3 Related Work .............................................. 161
7.4 Conclusion .................................................. 165

8 Provenance Browser ........................................ 167
8.1 Provenance Browser Architecture ...................... 168
8.2 Displaying Provenance Views ........................... 170
8.3 Navigating Provenance Views ........................... 172
8.4 Incremental Querying .................................... 175
8.5 Storage and Query Optimization ....................... 177
8.6 Conclusion .................................................. 179

9 Conclusions .................................................. 180
9.1 Summary of Contributions ............................... 180
9.2 Future Work ............................................... 184
9.3 In Conclusion .............................................. 185
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A Kepler representation of the fMRI workflow</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>A COMAD representation of the fMRI workflow</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>An actor employing <em>update semantics</em> and fine-grained data dependencies</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>A phylogenetics workflow implemented in Kepler</td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>An intermediate snapshot of a workflow run</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>The conventional provenance model</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>The process network (PN) provenance model</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>The <em>unified</em> provenance model</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>An example XML-based workflow implementing the fMRI workflow</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>Fine-grained dependency graph of data nodes for Figure 3.4b</td>
<td>36</td>
</tr>
<tr>
<td>4.1</td>
<td>A workflow graph with actors and dataflow edges</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>An execution (or <em>run</em>) of workflow showing versions of intermediate data</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>products received/produced by invocations</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>“Condensed” workflow trace of Figure 4.2</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>Rules for computing fully annotated traces</td>
<td>50</td>
</tr>
<tr>
<td>4.5</td>
<td>Logic rules for computing fully annotated traces</td>
<td>51</td>
</tr>
<tr>
<td>4.6</td>
<td>The same trace shown with minimal and full annotation</td>
<td>53</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.7</td>
<td>Basic trace storage strategies</td>
<td>57</td>
</tr>
<tr>
<td>4.8</td>
<td>Algorithm for Duplicate-Set Reduction</td>
<td>60</td>
</tr>
<tr>
<td>4.9</td>
<td>Algorithm for Subsequence Reduction</td>
<td>61</td>
</tr>
<tr>
<td>4.10</td>
<td>Algorithm for Subset Reduction</td>
<td>62</td>
</tr>
<tr>
<td>4.11</td>
<td>Expand-reduction view for immediate dependencies</td>
<td>66</td>
</tr>
<tr>
<td>4.12</td>
<td>Expand-reduction view for transitive dependencies</td>
<td>66</td>
</tr>
<tr>
<td>4.13</td>
<td>Expand-annotation view for dependencies</td>
<td>67</td>
</tr>
<tr>
<td>4.14</td>
<td>Provenance storage size for different patterns of synthetic traces</td>
<td>71</td>
</tr>
<tr>
<td>4.15</td>
<td>Comparison of reduction techniques</td>
<td>72</td>
</tr>
<tr>
<td>4.16</td>
<td>Storage compression of RE and RC w.r.t. NC storage strategy on Mixed pattern</td>
<td>72</td>
</tr>
<tr>
<td>4.17</td>
<td>Update time, query response time, and storage size for real execution traces</td>
<td>73</td>
</tr>
<tr>
<td>4.18</td>
<td>Comparison of update time steps for NE and RE storage strategies on Mixed pattern</td>
<td>74</td>
</tr>
<tr>
<td>4.19</td>
<td>Update steps of RE storage strategy on Mixed pattern</td>
<td>74</td>
</tr>
<tr>
<td>4.20</td>
<td>Provenance update time for the Mixed pattern</td>
<td>75</td>
</tr>
<tr>
<td>4.21</td>
<td>Query response time for MIXED pattern traces</td>
<td>76</td>
</tr>
<tr>
<td>5.1</td>
<td>An example XML-based workflow implementing the fMRI workflow (same as Figure 3.4)</td>
<td>89</td>
</tr>
<tr>
<td>5.2</td>
<td>The combined structure for Figure 5.1b.</td>
<td>90</td>
</tr>
<tr>
<td>5.3</td>
<td>The fine-grained dependency graph of data nodes for Figure 5.1b.</td>
<td>90</td>
</tr>
<tr>
<td>5.4</td>
<td>Pictorial representation of how a query that spans in each dimension is evaluated</td>
<td>98</td>
</tr>
<tr>
<td>5.5</td>
<td>The query response time for the naive and reduced-transitive approaches</td>
<td>102</td>
</tr>
</tbody>
</table>
5.6 The storage size for reduced-transitive, naive, and simple-transitive for
Figure 5.5 ................................................................. 103

6.1 An example XML-based workflow implementing the fMRI workflow
(same as Figure 3.4) .................................................... 109

6.2 Semantics of step expressions $s(T)$. ................................ 115

6.3 Semantics of simple immediate path expressions $s_1 .. s_n (T, L)$. 116

6.4 Semantics of $\text{exists}$ (i.e., reachability) queries ............ 116

6.5 Semantics of transitive path expressions. ......................... 117

6.6 Lineage graph with corresponding pointer-based representations. 122

6.7 Algorithm for evaluating single node-based path expressions ($ICP$) 124

6.8 Algorithm for efficiently evaluating set-based path expressions ($ICP(S)$) 125

6.9 An example execution of set-based path expressions using set-based
algorithm ................................................................. 126

6.10 Dependency complexity and storage sizes of experimental traces . 128

6.11 Evaluating $* .. n$ over storage strategies. ..................... 129

6.12 Evaluating $n_1 .. n_2$ over storage strategies. ................... 130

6.13 Evaluating $n_1 .. n_2 .. n_3$ over storage strategies. .......... 131

6.14 Comparing node-based $ICP(N)$ and set-based $ICP(S)$ approaches for
evaluating $N_1 .. N_2$. ............................................... 132

6.15 Evaluating $N_1 .. N_2 .. N_3 .. N_m$ using $ICP(S)$. ............. 133

6.16 Storage size and query time for real traces .................... 134

7.1 Different provenance graphs displayed using the provenance browser . 140

7.2 Three separate levels of granularity for viewing flow graphs ....... 144

7.3 Three representations of a flow dependency graph ............. 145
List of Tables

4.1 The result of applying reduction techniques on dependency sets . . . 59
4.2 Node and node dependency schemas . . . . . . . . . . . . . . . . . . 63
4.3 Provenance storage strategies . . . . . . . . . . . . . . . . . . . . . . 69
4.4 Workflow execution trace patterns . . . . . . . . . . . . . . . . . . . 69
4.5 Real-world workflow execution traces used in experimental evaluation 70
5.1 Summary of QLP constructs and corresponding descriptive notations 91
6.1 Semantics of QLP constructs . . . . . . . . . . . . . . . . . . . . . . . . 114
Chapter 1

Introduction

The contributions of this dissertation are a model for representing multi-version nested data structures that include labeled node-level lineage relationships; techniques for efficiently storing instances of this model; a generic path-based query language; optimization techniques for efficiently evaluating queries; and a navigation model for exploring and summarizing relevant portions of model instances. This work is largely motivated by the need for managing provenance of scientific workflow systems. The rest of this chapter describes the motivation for this work and summarizes research contributions.

1.1 Motivation

Many significant scientific discoveries are achieved through complex and distributed scientific experiments [Kep, Vis, Tav, Tri, KDG+08]. To analyze, evaluate, and understand results of scientific experiments, it is required that computationally complex processes be automated. The automation of scientific data analyses often requires “chaining together” distinct software programs and services in complex ways. These
1.1. Motivation

Computationally and data intensive processes may generate large volumes of information, including intermediate and final data products, and scientists often wish to record this information to better support reproducibility, validity, and re-use of scientific experiments [DF08, ABML09a]. Traditionally, scientists use batch files and scripting languages (e.g., Perl) to automate the execution of individual programs and the routing of data between them [DF08]. These scripting approaches have certain limitations, such as, they require substantial effort and significant technical skills to manage data and record provenance information (i.e., processing history) of scientific experiments. In addition, these approaches can be cumbersome, time-consuming, and error-prone for scientists, especially in data- and compute-intensive applications.

Scientific workflow systems (e.g., [LAB+06, Tom06, TSWH07, ZHC+07, CFS+06, Tav, Tri, Vis, FVWZ02, DSS+05, SPGM06b]) have recently emerged to address these challenges, and promise to better support scientists wishing to automate their computational tasks [GDE+07]. Scientific workflows systems are increasingly used by scientists to design and execute data analysis and to perform other tool integration tasks. Scientific workflow employ dataflow languages [JHM04], where a **scientific workflow** is represented as a directed graph of nodes denoting computation steps (e.g., for data acquisition, integration, analysis, or visualization) implemented by **actors**, and edges (connections) denoting the required dataflow between steps. Systems execute workflow graphs according to various **models of computation** [LAB+06, JHM04], which generally specify how workflow steps should be scheduled and how data should be passed (and managed) between steps.

As compared to traditional programming and scripting approaches, scientific workflows have a number of advantages. They adopt a simple programming model where a series of computational processes are composed by connecting the outputs
of one process to the inputs of another. Also, scientific workflow systems often provide visual programming interfaces, which make them easier for scientists to model their scientific processes that would otherwise require substantial programming expertise. In addition to automating data analyses, these systems additionally offer support for workflow design and analysis [GRD+07, MBZL09], efficient execution and deployment of workflows [DPF06, GKD05], and automatic recording of provenance [DF08, MLA+08, SPG05]. We define provenance of a data product as information about the process and data used to derive the product.

This dissertation addresses a number of challenges related to managing large amounts of provenance information, especially in the area of efficient modeling, storage, querying, visualization, and summarization (or abstraction) of large provenance graphs. The overarching goal of this work is to devise efficient techniques to enable practical and usable provenance systems.

1.2 Scientific Workflow Challenges

We adopt workflow terminology used by the Kepler [Kep, LAB+06] community, which is introduced informally here, but see Chapter 2 for a more detailed descriptions of terms. These terms can easily be mapped to those used in other workflow systems (e.g., [Tom06, TSWH07, ZHC+07, CFS+06, Tav, Tri, Vis, FVWZ02, DSS+05, SPGM06b]) without loss of generality. To make the ideas described above more concrete, we consider an example workflow shown in Figure 1.1. This workflow processes fMRI images [Hor03] and was used in the first Provenance Challenge [MFF+08]. The first Provenance Challenge was a community activity aiming at understanding the expressiveness of provenance representation and capabilities of provenance systems. We refer to steps in the workflow as actors that are invoked over input data supplied
1.2. Scientific Workflow Challenges

Figure 1.1: The fMRI workflow of the first provenance challenge implemented within Kepler, which takes as input 3D brain scan (i.e., anatomy) images and then: (1) compares each input image to a reference image to determine “warping” parameters for alignment with the reference; (2) transforms the input image according to the warping parameters; (3) averages the collection of transformed images into an atlas image; (4) produces three different 2D slices of the atlas image (in the x, y, and z dimensions); and (5) converts each resulting 2D slice into a graphical image.

by previous steps. This workflow takes a set of anatomy images representing 3D brain scans and a reference image, and applies the actors as shown in Figure 1.1 as follows.

1. AlignWarp is invoked over each anatomy image to produce a set of “warping” parameters;

2. Reslice is invoked over each set of warping parameters to transform the associated anatomy image;

3. Softmean averages the transformed images into an atlas image;

4. Slicer produces three different 2D slices of the atlas; and

5. Convert is invoked over each 2D slice to create a graphical image.
Figure 1.2: A COMAD representation of the fMRI workflow where: (1) the CollectionReader actor is configured with an input XML structure that specifies the input images used; (2) the AlignWarp, ResliceWarp, SoftMean, Slicer, and Convert actors are similar to those of Figure 1.1 but work over portions of the XML data stream determined by the scope parameters (show below actors, e.g., //Images); and (3) the ReplicateCollection actor is configured to create $n$ copies of the resulting SoftMean images, where each copy induces a separate Slicer invocation.

The workflow shown in Figure 1.1 has been designed in the Kepler scientific workflow system [Kep]. This workflow is a straight-forward design and is hardwired with respect to the number and size of input data sets the workflow can be applied to (stages 1 through 3 are wired to average exactly four input brain anatomy images to yield a single brain atlas image) and the size and contents of the output data set (three slices of the atlas image). Performing these computations on a different number of input images would require significant modifications to the workflow definition. For instance, new workflow components and additional wires would need to be introduced at stages 1, 3, 4, and 5.

Data for many applications (e.g., phylogenetics, or bioinformatics in general) can be large, complex, and nested in structure. Workflows must operate efficiently on these nested collections of data, while maintaining data associations they signify. Actor invocations within scientific workflows frequently operate over collections of data to produce new collections of results. When carried out one after the other, these operations can yield increasingly nested data collections, where different invocations potentially operate over different nesting levels. This complexity is compounded
1.2. Scientific Workflow Challenges

when the steps in scientific workflows execute in pipeline parallel mode [ZBML09],
e.g., when multiple sets of input data are supplied to and processed by a single
workflow run.

Figure 1.2 illustrates the advantages of employing nested data structures in work-
flows. Shown is a workflow definition with the same intent as that in Figure 1.1, but
implemented using the Collection-oriented modeling and design (i.e., COMAD) model
of computation [MBZL09, MBL06] developed in Kepler. Workflows in COMAD are
executed over XML data streams in which actors employ update semantics by
making modifications to (i.e., updating) portions of the overall XML structure and
passing the updated structure to downstream actors. Specifically, actors in COMAD
receive fragments of an incoming XML token stream based on their declared read-
scope parameters (shown as XPath expressions below the actors in Figure 1.2), insert
and remove fragments within their matching scopes, and pass the modified stream on
to subsequent actors. Actors are invoked over each corresponding read-scope match
within an incoming token stream. For instance, the COMAD implementation of the
AlignWarp actor (Stage 1) is invoked once over each AnatomyImage collection within
the workflow input, as shown in Figure 1.3a for the first invocation of AlignWarp.
We denote invocations of a process named $P$ as $P : n$, where $n$ is the $n^{\text{th}}$ invocation
of $P$.

The workflow of Figure 1.2 differs from Figure 1.1 in that it can be executed over
multiple collections of anatomy images of varying sizes without requiring modifica-
tions to the workflow graph: e.g., to have the workflow average five anatomy images,
rather than four, the user needs to only add the additional image to the input data
set.

\[^{3}\text{Kepler is unique in that it supports multiple computation models (or directors) including standard dataflow models such as Synchronous DataFlow (SDF) and Process Networks (PN) [LAB}^{+06}].\]
1.3 Provenance Management Challenges

Though the provenance model developed for COMAD workflows supports multiple invocations of processes, handles complex data, and captures fine-grained (token-level) data dependencies, a number of challenges related to efficiently modeling, storing, querying, visualizing and summarizing (or abstracting) large provenance graphs remain. The following is a summary of the challenges we focus on in this work.

Representing (modeling) provenance of nested data. While using nested data collections can provide benefits for workflow design and execution, this approach requires richer models of provenance than those used by other conventional systems (e.g. [MFF+08, BBDH08, SFC07, CFL+07, KDG+08, BD08, CJR08] where actors represent pure data transformers (as in Figure 1.1). Specifically, these transformers (i) treat data as opaque objects (or tokens), (ii) produce sets of new output tokens from each set of input tokens they receive, and (iii) assume all input tokens are used
1.3. Provenance Management Challenges

to derive all output tokens within a single actor invocation.

Provenance management systems tailored to such workflows (e.g., [ABJF06, Car08, MFF+08]) generally store the input and output tokens of each actor invocation, and later use this information to infer dependencies between data tokens and workflow steps. When applied to computation models that support nested data collections (e.g., COMAD), however, this standard approach to recording and reporting provenance relationships often yields incomplete and even incorrect data and invocation dependency inferences.

For example, Figure 1.3b shows the actual data dependencies introduced by the first invocation of the AlignWarp actor; specifically, the object representing the warping parameters (node 11) was derived from each of the image and header objects contained in the AnatomyImage collection matching the read scope of the invocation (node 2). However, if these input and output structures were represented as atomic data objects ($s_1$ and $s_2$), the standard model would infer only a single dependency between the output version ($s_2$) and the input version ($s_1$), which is too coarse. Alternatively, if XML structures are represented as token streams, then the standard model would incorrectly infer dependencies between every output node and every input node.

**Efficiently storing large amounts of provenance information.** When a large number of operations are performed on data and fine-grained provenance information is stored, then the number of provenance annotations can grow very quickly. In addition, workflows are often executed multiple times over different data sets and parameter settings, and most scientific research projects consist of many distinct workflows [BBD+09, ABML09a]. Maintaining provenance information for all nodes in a workflow trace can lead to prohibitively expensive storage costs in terms of
1.3. Provenance Management Challenges

database size and update time (i.e., the time needed to load provenance information into the database) [CJR08, HA08, DF08]. Further, common provenance queries are centered around dependency relations [MLA+08]. To answer such provenance queries across dependency relations, it is often required to consider the transitive closure of dependencies, e.g., to find the nodes or invocations that directly or indirectly contributed to a specific node. Storing dependency closures can further increase storage cost, whereas if only immediate dependencies are stored, more expressive query constructs are required, such as recursion, which can be expensive and therefore increase query time [HA08].

**User-friendly query language for complex provenance queries.** While most systems record and store data and process dependencies, few provide easy-to-use and efficient approaches for accessing provenance information. Instead, users write provenance graph queries directly against physical data representations (e.g., relational, XML, or RDF databases) through corresponding query languages (SQL, XQuery, or SPARQL) [DF08]. This approach often leads to complex query expressions, even for answering basic provenance questions (see Chapter 5 for some query examples). In addition, to query over lineage information stored within a relational database, users must typically specify join operations over multiple tables and compute transitive closures [DF08, CJR08, HA08, ABML09a]. Expressing such queries is both cumbersome and error-prone, and requires considerable user expertise. Furthermore, as our provenance model supports nested data structures, we need mechanisms to allow users to pose queries not only over data lineage relationships, but also over versions of nested data structures produced during a workflow run.

**Efficiently evaluating complex provenance queries.** Provenance queries are often recursive in nature, and evaluating them is expensive, as this typically re-
quires join operations over multiple tables and computing transitive closures over large graphs. Also, due to dependency relations shared between data and processes, it is meaningful to explore provenance query results as a sub-graph of original dependency graphs. But most typical approaches for querying provenance information (e.g., [Car08, HBM+08, ZL08, BBDH08]) return sets of nodes (either sets of data items or process invocations) as the query result. This approach requires additional steps (queries) to reconstruct causal relations among nodes within a query answer. All these operations result into inefficiency in evaluating provenance queries.

**Exploring large provenance graphs.** Provenance information is often displayed to users visually as one or more dependency graphs [MFF+08, BMR+08], e.g., where nodes denote data items or processes, and edges denote dependency relationships between nodes. Displaying such graphs is especially useful for small workflows, involving only a small number of data sets and processes, since users can quickly see every data product, process, and dependency associated with a run. By analyzing and creating insightful visualizations of provenance data, scientists can debug their tasks and obtain a better understanding of their results. However, for many real-world scientific workflows provenance graphs may be large (e.g., thousands of nodes and edges) due to the complexity of the workflow, the size of input data sets, and the number of intermediate data sets produced [CJR08, ABML09a, DF08]. For large provenance graphs, understanding and exploring provenance information becomes a significant challenge for users.
1.4 Dissertation Starting Point

This work builds on the prior work of COMAD [MB05, BML+06, BML08] in modeling and representing provenance information of scientific workflows. COMAD defines a provenance model [BMR+08] that supports multiple invocations of processes (in dataflow process networks [LP95], handles complex data (i.e., nested data collections, represented in XML), and captures fine-grained (token-level) data dependencies through “provenance-aware” actors. Provenance information in COMAD is recorded during a workflow run by adding special provenance annotation tokens directly into the token stream containing explicit data-dependency information. During workflow execution, input structures are operated upon by actor invocations to produce new output structures, which is essentially a new version of the data structure. When a module inserts a node then the provenance annotation is recorded for the highest collection\(^2\) ancestor, and the provenance information for that node implicitly applies to all descendant nodes. Similarly, if a node is derived from an ancestor node, an insertion annotation is created that refers just to the ancestor identifier rather than to all descendants. Thus, workflow traces generated by COMAD reduce the amount of provenance annotations. A more detailed description of COMAD and provenance model is described in Chapter 2.

1.5 Research Questions

Broadly stating, this work addresses the research problems of efficiently modeling, storing, querying, visualizing and summarizing (or abstracting) large provenance graphs. This work addresses problems in the following areas: how to efficiently

\(^2\)We represent data collections as XML like labeled ordered trees.
represent multi-version nested data structures that include labeled node-level lineage relationships; how to efficiently store direct and transitive dependency relationships; how to allow users to easily express path-based lineage query expressions; how to optimize evaluation of path-based lineage queries; and how to explore and summarize relevant portions of directed acyclic graphs. This work addresses these research problems in the context of scientific workflow provenance (see Section 1.3). The approaches and results of this dissertation are not dependent on any particular workflow system.

Specifically this dissertation addresses the following research questions:

- How to extend the conventional model of provenance to support nested data collections and capture data dependencies explicitly at a fine-grained (node) level (Chapter 3).

- How to devise efficient storage techniques to optimize update time, storage size, and query time (Chapter 4).

- How to make it simple and intuitive for users to formulate provenance queries, such that the formulation of queries are independent from storage representations and database schemas (Chapter 5).

- How to incorporate query optimization techniques to make provenance querying practical and efficient (Chapter 6).

- How to develop techniques to explore large provenance graphs through an integrated approach such that users can visualize, navigate, summarize (or abstract), and query relevant sections of provenance views (Chapter 7).
1.6 Research Contributions

Broadly stating, the contributions of this work include (1) a collection-oriented approach to model provenance of nested data, along with its formalism; (2) a set of storage strategies and reduction techniques to decrease the size of provenance store, update time, and query time; (3) a novel high-level query language of provenance (QLP) that allows users to express complex provenance queries easily and concisely; (4) a set of optimization techniques to efficiently evaluate provenance queries; (5) a novel navigational model for provenance that provides an integrated approach to visualize, navigate, summarize (or abstract), and query provenance views; and (6) development of the Provenance Browser, a provenance visualization tool that allows users to store provenance traces in relational database systems; and users can view, navigate, and query various provenance views.

In particular, the contributions of this work are as follows:

1. A unified provenance model. A logical model of provenance that subsumes conventional approaches for representing workflow provenance is described. The provenance model supports more advanced workflow computation models that permit multiple invocations of processes (e.g., for pipelining and loops), structured data, and update semantics. A formal description of the provenance model is given in Chapter 3 and published in SSDBM09 [ABML09b].

2. A set of storage strategies, and reduction algorithms. The main contributions in this area are the development and comparison of different strategies for efficiently storing, updating, and querying provenance graphs expressed in our unified model using a relational database system. We define: (i) inference techniques for expanding and collapsing the implied data and process dependen-
cies within a trace; (ii) reduction techniques for minimizing the size of provenance storage for direct and transitive data and process dependencies (these techniques are generic and applicable to both hierarchical and non-hierarchical data structures); and (iii) relational views for recovering all implied and reduced dependencies generated by the inference and reduction techniques. We compare trade-offs in terms of provenance storage size, update time, and query response time, and show using real-world and synthetic provenance traces that our techniques can improve each of these. Storage strategies, and reduction algorithms are presented in Chapter 4 and have been published in EDBT09 [ABML09a].

3. **A specialized provenance query language.** A high-level Query Language for Provenance (QLP) is proposed, which provides specialized constructs for expressing both structural (i.e., parent-child) and lineage (i.e., data dependency) queries, and their combination (“hybrid” queries). QLP allow users to formulate provenance queries concisely by using simple query expressions. QLP is a closed query language, in the sense that results are provenance preserving, i.e., result of a provenance lineage graph is a sub-graph that can be queried further. QLP is presented in Chapter 5 and has been published in SSDBM09 [ABML09b].

4. **Optimization techniques for efficiently evaluating provenance queries.** A formal semantics for QLP constructs is described, and a set of novel query optimization techniques are developed that leverage the lineage-graph reduction approaches described in Chapter 4. The approach simultaneously reduces the amount of storage required to represent scientific workflow provenance without negatively affecting query performance. The optimizations can also be used in
1.6. Research Contributions

more general settings to efficiently answer a broad range of path queries over labeled, acyclic digraphs. Unlike using standard approaches (which are often based on in-memory implementations, e.g., [ZL08, HS08, LHL+98]), answering QLP queries with our optimization techniques over a relational database system is both practically feasible and scales linearly with the size of provenance information and query complexity. The semantics of QLP and our query optimization techniques are presented in Chapter 6 and have been published in EDBT10 [ABL10b].

5. **A provenance navigation model.** A novel navigation model is proposed that allows users to explore and view relevant portions of provenance information using intuitive and natural graph-based provenance representations. The navigation model provides an integrated approach to: (i) abstract views of provenance information (actor, invocation, and flow dependency graphs), (ii) seamlessly navigate between these views, (iii) summarize portions of views through the grouping actors or invocations; and (iv) filter (or query) provenance views using QLP, a high-level provenance query language. This navigation model is the first approach to the best of our knowledge that combines navigation, abstraction (through composition), and query capabilities. The navigation model is described in Chapter 7 and has been published in WORKS09 [ABL09].

6. **Provenance Browser.** The relational schema for various storage strategies, reduction algorithms, relational views over different schemas to query provenance information, a parser for converting QLP queries into SQL queries over relational storages, and optimization techniques to efficiently evaluate provenance queries have all been implemented and integrated into the Provenance Browser, a provenance visualization and querying tool. The Provenance Browser has
been successfully integrated into the Kepler scientific workflow system. From the integrated environment of Kepler, the users can now not only develop and execute scientific workflows, but also efficiently store the provenance information into a relational database, browse the provenance information via meaningful provenance views, and query the these views through concise QLP expressions. The Provenance Browser is described in Chapter 8 and has been published in ICDE10 [ABL10a].

1.7 Dissertation Structure

This dissertation is organized as follows.

Chapter 2 briefly describes Kepler, a scientific workflow system, and gives a brief description of COMAD (a Collection-Oriented Modeling and Design) framework. Chapter 3 formalizes and extends conventional provenance models to develop a unified provenance model that supports complex data (i.e., nested data), captures explicit data dependencies and employs update semantics. Chapter 4 describes a set of strategies for efficiently storing instances of our unified provenance model, and also proposes a set of novel reduction techniques for minimizing redundancy in immediate and transitive dependency representations. Chapter 5 describes a general approach for querying provenance using our proposed Query Language for Provenance (QLP), which provides specialized constructs for expressing both structural (i.e., parent-child) and lineage (i.e., data dependency) queries. Chapter 6 gives a formal semantics for QLP query constructs. Based on this formal semantics, we describe different approaches for evaluating QLP lineage queries against various storage strategies, including our proposed query optimization techniques that leverage the lineage-graph reduction approaches described in Chapter 4. Chapter 7 describes
our proposed navigation model that provides an integrated approach for exploring, summarizing, and querying portions of provenance graphs through a set of navigation operators. Chapter 8 describes the Provenance Browser that integrates various applications that resulted from this work and describes how all those applications have been successfully integrated into the Kepler scientific workflow system. Finally, Chapter 9 summarizes the results and suggests areas of future works.
Chapter 2

Preliminaries

While the work presented here was conducted in the context of Kepler [Kep] as a concrete scientific workflow system, the approaches described in this work are not limited to Kepler and generally apply to other systems (e.g., [Tom06, TSWH07, ZHC+07, CFS+06, Tav, Tri, Vis, FVWZ02, DSS+05, SPGM06b]).

2.1 Kepler Workflow System

This section presents a brief overview of Kepler, as pertinent to this dissertation. Please refer to [Kep, LAB+06] for more detailed description of the system.

Kepler is a scientific workflow system for multiple disciplines that provides a workflow environment in which scientists can design and execute workflows. Kepler builds on top of the mature Ptolemy II software [Pto, BLL+07], which is a Java-based system and a set of APIs for heterogeneous hierarchical modeling. The focus of Ptolemy II is to build models based on the composition of existing components, which are called actors, and observe the behavior of these simulation models when executed using different computational semantics, which are implemented as components called
directors. In contrast to Ptolemy II, Kepler provides an environment for designing and executing scientific workflows.

Workflows are composed by selecting appropriate components called “actors” (e.g., from actor libraries or wrapping web services as actors) on a design canvas, and “wiring” them together to form the desired workflow graph. Actors have input ports and output ports that provide the communication interface to other actors. Actors communicate by sending tokens, which encapsulate data or messages, to other actors through ports. Workflow graphs can exhibit branching, merging, and loops.

Kepler makes explicit the distinction between the workflow graph on the one hand, and the Model of Computation (MoC) used to interpret and enact the workflow on the other. This is achieved by requiring workflow authors to specify a director for each workflow. A director specifies and mediates all inter-actor communication, separating workflow scheduling and runtime orchestration (a director’s concern) from individual actor execution (an actor’s concern). This separation results in more reusable actor components. Kepler inherits from Ptolemy II a variety of directors that implement synchronous data flow (SDF), process network (PN), discrete event (DE), continuous time (CT), and finite state transducer (FST) semantics. In addition, a Kepler specific director for Collection-Oriented Modeling and Design (COMAD) has been developed at UCDavis [MB05]. Taken together, workflows, actors, ports, connections and directors represent the basic building blocks of Kepler workflow design.

We will now have a closer look at an example workflow created in Kepler to describe basic concepts and abstractions in dataflow-oriented workflow systems.

A phylogenetics workflow. Figure 2.1 shows an example workflow for inferring phylogenetic (evolutionary) relationships between a set of homologous DNA
2.1. Kepler Workflow System

Figure 2.1: A phylogenetics workflow implemented in the Kepler system. Kepler workflows are built from *actors* (boxes) that perform computational tasks. Users can select actors from component libraries (panel on the left) and connect them on the canvas to form a workflow graph (center/right). Connections specify dataflow between actors. Configuration parameters can also be provided (top center), e.g., the location of input data and the initial and maximum *jumble seed* values are given. A *director* (top left corner on the canvas) is a special component, specifying a model of computation and controlling its execution.

sequences. DNA sequences for homologous genes from a number of taxa are provided as input to the workflow. The Actor **Read-sequences** reads a collection of DNA sequences, the actor **Compute-alignment** performs an initial alignment of the sequences and the actor **Refine-alignment** refines this initial alignment. The Actor **Find-MP-trees** searches the most parsimonious phylogenetic trees relating to the sequences, and the actor **Compute-consensus** computes the consensus of these trees, if more than one was found. The workflow shown in Figure 2.1 uses a particular MoC called COMAD, developed at UCDavis [MB05]. This model has several advantages and is described in the next section.
2.2 Collection-Oriented Modeling and Design

The Collection-Oriented Modeling And Design (COMAD) framework [MBL06] in Kepler permits data to be grouped explicitly into nested collections similar to the tree structure of XML documents. These trees of data are input, manipulated, and output by collection-aware modules. However, unlike a general XML transformer, a COMAD actor (short coactor) generally preserves the structure and content of input data, accessing particular collections and data items of relevance to it, and adding newly computed data and new collections to the data structure it received. COMAD workflow designers declare the read-scope and write-scope for each actor while composing the workflow specification. A read-scope specifies the type of data and collections relevant to a module (using an XPath-like expression), and a write-scope specifies where a module should add new data and collections to the stream. The system iteratively invokes modules over relevant portions of the input stream matching read-scope expressions, adding output data to the stream according to write-scope expressions.

Figure 2.2 illustrates the state of a COMAD run at a particular point in time, and contrasts the logical organization of the data flowing through the workflow in Figure 2.2(b) with its tokenized realization at the same point in time in Figure 2.2(c). To emphasize the pipelining capabilities of COMAD two independent sets of DNA sequences (represented by two trial collections) were provided to the workflow. Figure 2.2(c) shows the tokenized realization of the data stream as it flows through three of the coactors. Note that two of the coactors, COMPUTE-CONSENSUS and FND-MP-TREES, are working concurrently, the former computing the consensus of the phylogenetic trees derived from the first batch of sequences, the latter computing phylogenetic trees for the second batch of sequences.
2.2. Collection-Oriented Modeling and Design

Figure 2.2: An intermediate snapshot of a workflow run: (a) example workflow; (b) logical organization of data at a point in time during the run; (c) tokenized version of the collection structure: three actors are working *concurrently* on different parts of the data stream. Nested collections organize and relate data objects from domain-specific types (DNA sequences, alignments, phylogenetic trees). A *Proj* collection containing two *Trial* sub-collections is used to pipeline multiple sets of input sequences through the workflow. Provenance events (insert-data, insert-collection), insertion dependencies, and deletions (from the stream) are added directly as metadata tokens to the stream (c), yielding data-dependency graphs as in (b).

**Recording and Representing Provenance.** Unlike other models [MLA+08, MFF+08, BBD07], the COMAD provenance model takes into account nested data collections, pipeline-parallelism (in addition to the usual task parallelism), and actor scope expressions. The primary goal is to capture the information necessary to reconstruct and effectively present the derivation history of scientific data products, thereby supporting the main provenance needs of scientists.

Figure 2.2 also illustrates how data provenance is captured and represented at runtime. As COMAD modules insert new data and collections into the data stream, they also insert metadata tokens containing explicit data-dependency information.
For example, the fact that Alignment2 was computed from Alignment1 is stored in the insertion-event metadata token immediately preceding the A2 data token in Figure 2.2(c), and displayed as the dashed arrow from A2 to A1 in Figure 2.2(b).

The products of a COMAD workflow may be saved as an XML-formatted trace file, in which provenance records are embedded directly within the file as XML elements annotating data and collection elements. Every node in the nested data collection has a unique identifier, and every data node has a label that represents the type of value encapsulated. Furthermore, a write event corresponds to inserting or deleting a node in the XML tree. Deleting a node means annotating it as deleted so that it is skipped by future read-scope expressions. Detailed data dependencies can be inferred from the trace file, i.e., from the embedded provenance annotations together with the nested data collections output by the workflow run.

A deletion token to the stream for each data and collection item they remove from the stream. A deletion token consists of (i) the actor invocation-id that removed the data or collection item; and (ii) the token-id of the item that was removed. We note that to maintain the dependencies between data products, our system simply tags removed items as deleted, which prevents downstream actors from using deleted items while allowing these items to be tracked by the provenance system.

In certain cases, actors also add invocation-dependency tokens into the stream that specify an ordering relation between two actor invocations. Specifically, invocation dependencies are added when actor invocations insert data or collection items that depend on a collection that was modified by a previously invoked actor. Here, a collection is considered modified when a data or collection node is added or removed (either a direct child or a descendent).
The provenance model employed within COMAD also provides a number of extensions beyond the core model described here. Extensions are provided for representing metadata (name-value pairs) associated with collection and data tokens, actor invocation parameters, annotations specifying the tokens that were read by actors (i.e., those tokens within an actor’s read scope), and deletion dependencies. Similar to the provenance annotations described above, these annotations also implicitly cascade (or propagate) to descendent nodes [BMWL07].
Chapter 3

A Unified Provenance Model

This Chapter discusses shortcomings of the conventional provenance models in accurately capturing data dependencies in many computational scenarios and when complex data is used. A unified provenance model is described that naturally extends the conventional provenance model by adding support for nested data and for accurately capturing detailed lineage information of processes employing update semantics. A number of models of provenance are developed, starting with a basic model containing simple task dependency graphs over atomic data and atomic (single invocation) processes. The basic model is gradually extended to the unified model where the later supports multiple invocations of processes, handles complex data (i.e., nested data collections, represented in XML), captures fine-grained (token-level) data dependencies correctly and uses update semantics as natural extensions to basic model.
3.1 Provenance Models

Many workflow systems (e.g., [SFC07, BCS+05, ABJF06, ZGST08, ZWF06, OCP+08]) and provenance approaches (e.g., [CBD06, BD08, CJR08, HA08, MFF+08]) employ a simple provenance model that is data “agnostic” [DF08]. This model generally can be characterized as recording the inputs and outputs for each actor invocation occurring within a workflow run. Conceptually, a workflow execution trace in such a model consists of pairs of records \(\text{in}(x, a)\) and \(\text{out}(a, y)\), stating that \(x\) was an input and \(y\) an output of an actor invocation \(a\).\(^1\) This information is used to infer data and process dependencies. For example, a run of a simple workflow

\[
X \rightarrow A \rightarrow Y \rightarrow B \rightarrow Z
\]

is captured by a set of trace facts, say \(\text{in}(x, a)\), \(\text{out}(a, y)\), \(\text{in}(y, b)\), and \(\text{out}(b, z)\), implying that the invocation \(a\) of actor \(A\) directly preceded invocation \(b\) (of \(B\)), and that the output \(y\) directly depended on \(x\), while \(z\) indirectly depended on \(x\). In most systems, each input and output of an invocation is denoted by one or more tokens that encapsulate data values (e.g., DNA sequences) or references to values (e.g., filenames or accession numbers). Tokens are assumed to be immutable (once created their values cannot be modified), are assigned unique identifiers, and may be further organized into records, lists, trees, streams, etc. [MFF+08, MBZL09, MBZG08].

This “conventional” model of provenance is useful for representing data and process dependencies of scientific workflows consisting primarily of black-box transformations [CW03], i.e., actors which (1) produce new outputs from their inputs; and (2) use all inputs to derive their outputs (implying that all outputs of an invocation

\(^1\)Other information may also be captured, e.g., timestamps of invocation reads and writes, and invocation parameters.
3.1. Provenance Models

depended on all inputs to the invocation).

However, many scientific workflows do not operate under these assumptions, and in such cases directly employing the conventional model can result in inefficient provenance storage and insufficient (or even incorrect) inferences of data dependencies.

For instance, many systems (e.g., [MBZL09, MBZG08, QF07]) and approaches (e.g., [FCB07, KS07, MAA+05, WMe08]) support actors that make only small changes or updates to incoming data, passing on some or all of their input to downstream actors. Thus, if invocation $a$ above retains within its output $y$, some unchanged substructure $s$ from its input $x$, denoted\(^2\)

\[
x = (s \oplus x_0), \quad y = (s \oplus y_0)
\]

then $s$ will be stored twice: once in the trace record $\text{in}(x, a)$ (call this occurrence $s_x$) and once in $\text{out}(a, y)$ (call this occurrence $s_y$). In the conventional model, all parts of output $y$ are said to depend on all parts of input $x$, so these records may also incorrectly imply that $s_y$ depends on $x_0$.

Furthermore, because actors often wrap complex external applications and services, various patterns of data dependencies can arise in which not all parts of the output depend on all parts of the input [CW03, MBK+08, MBZG08, BML+06, QF07]. Assume, e.g., that invocation $a$ above receives input $x$ and produces output $y$ as follows

\[
x = (x_1 \oplus \ldots \oplus x_n), \quad y = (y_1 \oplus \ldots \oplus y_m).
\]

Simple examples of $a$ with data dependency patterns not supported by the conventional model include [BML+06]:

\(^2\)“$\oplus$” splits complex data into subparts (e.g., into list elements, subtrees, etc.); the details are not important here.
3.1. Provenance Models

(a) actors having subtasks (such as filtering input data) prior to applying a scientific function, resulting in dependencies where each $y_i$ depends only on some of the $x_j$’s;

(b) actors that process each input token in turn, resulting in dependencies where each $y_i$ depends on a single $x_j$ ($j = i$);

(c) actors that perform running aggregates over their input, resulting in dependencies where each $y_i$ depends on the set $\{x_1, \ldots, x_i\}$; and

(d) actors that apply functions over their input using sliding windows of a fixed size $w$, resulting in dependencies where each $y_i$ depends on the window $\{x_{i-w}, \ldots, x_i\}$.

Because many possible dependency patterns may occur (e.g., via combinations or variations of the above), in general all data dependencies must be recorded explicitly for each workflow run to accurately represent data lineage.

Despite efforts to better understand and even standardize provenance models, e.g., in the Open Provenance Model (OPM) initiative [MLA+08], a universally accepted model has yet to emerge. This is due in part to the differences in the underlying models of computation employed by workflow systems. In the following, we develop a number of models of provenance, starting with a basic model capturing the conventional view of scientific workflows as simple task dependency graphs over atomic data and atomic (single invocation) processes, similar to the OPM model. Next, we extend this basic model to handle computations where a workflow step (i.e., task or process) consists of multiple invocations (or firings [LM08]) over a stream of incoming tokens. The resulting model is a variant of process networks [Kah74] that is well-suited for stream processing, and comes with “built-in” pipeline parallelism. A second, and for our purposes crucial extension of the basic provenance model, al-
3.2 Basic (Conventional) Model

The conventional (or basic) model of provenance consists of a trace structure (or simply a trace) $T = (V, E)$ forming an acyclic flow graph, where each node in $V = S \cup I$ represents an (atomic) data structure $s \in S$ or a process invocation $i \in I$. Edges $E = E_{in} \cup E_{out}$ are in-edges $E_{in} \subseteq S \times I$ or out-edges $E_{out} \subseteq I \times S$, representing the flow of data during a workflow run. A trace $T$ in this model links atomic data tokens to atomic processes (i.e., having a single process invocation). Data structures are consumed (destroyed) by an invocation via in-edges. Similarly, process invocations create new output structures via out-edges. Thus, in this model, processes are viewed as data transformers. To avoid write conflicts among multiple invocations, we require that $E_{out}^{-1} : S \rightarrow I$ be a function, associating with each output structure $s \in S$ the unique invocation $i_s \in I$ that created it.

A flow graph gives rise to two natural views, a data dependency graph $G_d = (S, E_{ddep})$ and an invocation dependency graph $G_i = (I, E_{idep})$, defined by $E_{ddep} = \pi_{3,1}(E_{in} \times E_{out})$ and $E_{idep} = \pi_{3,1}(E_{out} \times E_{in})$, respectively. These views are equivalently defined in Datalog as follows.

\[
\text{ddep}(s_2, s_1) \leftarrow \text{in}(s_1, i), \text{out}(i, s_2)
\]

\[
\text{idep}(i_2, i_1) \leftarrow \text{out}(i_1, s), \text{in}(s, i_2)
\]
3.3 Multiple Invocation Model

Figure 3.1: The Conventional model: (a) a workflow definition with steps A through D; (b) an example flow graph with process invocations a through d and atomic data objects s1 through s5; and (c) data and (d) invocation dependencies inferred from the flow graph of (b).

Our basic model closely resembles the Open Provenance Model (OPM) [MLA+08]. In OPM, an atomic data structure $s \in S$ is called an artifact, an invocation $i \in I$ is called a process, an in-edge $s \rightarrow i$ corresponds to a used edge $s \leftarrow i$, and an out-edge $i \rightarrow s$ corresponds to a wasGeneratedBy edge $i \leftarrow s$. Similarly, the above dependency views $E_{ddep}$ and $E_{idep}$ simply have different names in OPM: For $ddep(s_2, s_1)$ we say in OPM that the artifact $s_2$ was derived from the artifact $s_1$; and for $idep(i_2, i_1)$ we say that the process $i_2$ was triggered by the process $i_1$.

3.3 Multiple Invocation Model

The basic model views processes as atomic, i.e., for each task A, B, ... in the workflow definition, there is a single invocation a, b, ... in the flow graph (see Figure 3.1b). Other models of computation, notably process networks [Kah74] and related dataflow variants with firing [LM08] give rise to finer-grained process models for incremental
computations over data streams. Figure 3.2a shows the execution of process A modeled as two independent invocations, a:1 and a:2, which may be executed concurrently over the input stream s_1, s_2, ... (similarly for B and b:1, b:2). Here, the second invocation a:2 of A does not “see” (is independent of) the earlier input s_1 used by a:1. This is the case, e.g., if A is stateless, i.e., has no memory between invocations. Figure 3.2b is a variant of Figure 3.2a in which A is stateful, and thus preserves information between invocations a:i, resulting in additional dependencies. More formally, in the multiple invocation model, a trace $T = (V, E, \alpha)$ includes a function $\alpha : I \to A$ returning for each invocation $i \in I$ the actor $\alpha(i) \in A$ that created $i$. Conversely, for any actor $A$, $\alpha^{-1}(A) = \{i \in I \mid \alpha(i) = A\}$ is the set of invocations created by $A$ during a workflow run.

The underlying model of computation (MoC) of a workflow language determines the kinds of traces that can be generated at execution time. One can understand (and formalize) a MoC as a mapping that associates with a workflow definition $W$ and input $s$, a set of possible traces $T(s)$. The basic model, e.g., with its atomic data tokens and atomic processes can create flow graphs as in Figure 3.1b, but not those in Figure 3.2, which can support pipeline parallel execution.

### 3.4 Nested Model (Copy Semantics)

So far we have considered data structures as atomic tokens $s \in S$, i.e., without further access to any internal structure (e.g., $s$ might denote a string, file, or Java object). This model is often too coarse and thus inadequate when dealing with workflows over nested data such as XML. Thus, we refine the multiple invocation model by “drilling down” to the level of data items within structures. In the nested model with copy semantics, a trace $T = (V, E, \alpha, \tau)$ includes a function $\tau : S \to X$, which maps
3.5 Nested Model (Update Semantics)

Consuming XML objects and recreating parts of them (through fresh copies) is both inconvenient and inefficient, e.g., with respect to both workflow execution and provenance storage. We now assume that different versions of trees \( \tau(s) \in X \) share nodes...
3.6. The Unified Model

from \( N \).\(^3\) In particular, invocations are modeled as sets of updates that produce new versions of their input. For example, in Figure 3.3a, \( \Delta_{a:1}(s_1) \) creates an updated version \( s'_1 \) (not shown in the figure) of the structure \( s_1 \). Viewing invocations as updates can increase the concurrency of workflow execution for independent invocations. For invocations \( a:i \) and \( a:j \) of \( A \) and inputs \( s \in S \), if

\[
\Delta_{a:1}(\Delta_{a:j}(s)) = \Delta_{a:j}(\Delta_{a:i}(s)),
\]

then we say that \( A \) has independent invocations. There are different ways to achieve this independence. In COMAD, e.g., one can (i) employ non-overlapping scope expressions, and (ii) require further that actors are stateless across multiple invocations of the same actor. As shown in Figure 3.3b, we also relax the earlier constraint that \( E_{out}^{-1} \) be a function. Thus, in the nested model with update semantics, a trace \( T = (V, E, \alpha, \tau, \gamma) \) includes a function \( \gamma : N \rightarrow I \) that returns for each node the unique invocation that created it. This avoids write conflicts at the node level, while still allowing parallel updates to the same tree, e.g., as shown in Figure 3.3b.

We also restrict the types of updates that can be performed by invocations to insertions and deletions of nodes (i.e., arbitrary structural modifications are not permitted). This is also a restriction in COMAD to support branching and efficient merging of XML streams.

### 3.6 The Unified Model

Finally, we combine the above models into a unified trace structure supporting both multiple invocations and nested XML with update semantics. Specifically, we extend

\(^3\)We can view \( s \) as the version-id of \( \tau(s) \)
3.6. The Unified Model

the previous model with fine-grained dependencies for relating nodes according to dependency relationships (as opposed to relating coarse-grained structures as in a flow graph). If a node $n$ is a fine-grained dependency of a node $n'$, then we say that $n$ was directly used in the creation (or derivation) of $n'$. We represent fine-grained dependencies using lineage relations, which includes the invocation $i$ that created $n'$, denoted $n \rightarrow_i n'$ in Figure 3.3b. Note that if $n'$ depends on a collection node $n$ within the structure $s$, then $n'$ is also assumed to depend on the descendents of $n$ with respect to $s$; and we typically show only the “highest” dependencies of a node, as in Figure 3.3b.

A trace in the unified model is of the form $T = (V, E, L, \alpha, \tau, \gamma)$ where fine-grained dependencies are captured by the ternary lineage relation $L \subseteq N \times I \times N$. In the case of the basic model, $L$ is assumed to be given by the in-edges and out-edges of the flow graph (as shown in Figure 3.1c), which may not accurately represent data.

Figure 3.3: The unified model, which supports COMAD and similar models of computation: (a) the complex (i.e., XML) data structure $s_1$ is updated by invocations giving $s_2 = \Delta_{a:2}(\Delta_{a:1}(s_1))$, and for stateless $A$, $s_2 = \Delta_{a:1}(\Delta_{a:2}(s_1))$; and (b) the labeled dependency (i.e., lineage) edges representing the correct data dependencies for the nested data structures of (d).
3.7 Running Example

Consider the workflow in Figure 3.4a showing a straightforward XML-based implementation of the fMRI image processing pipeline used in the First Provenance Challenge [MFF+08]. We refer to steps in the workflow as *actors* that are *invoked* over input data supplied by previous steps.\(^4\) This workflow takes a set of anatomy images representing 3D brain scans and a reference image, and applies the actors in Figure 3.4a as follows.

1. AlignWarp is invoked over each anatomy image to produce a set of “warping”

\(^4\)In general, we also consider more complex workflow graphs involving branching, merging, and loops, as in [LAB+06, ABML09a, BMR+08].

Figure 3.4: (a) Example XML-based workflow implementing the fMRI image analysis of the first provenance challenge; (b) The trace showing the first invocations of each actor for a typical run; (c) The implied fine-grain data dependency graph for the data items in (b); and (d) The implied invocation dependency graph for the run, with the first invocations of each actor shown in red.
parameters;

2. Reslice is invoked over each set of warping parameters to transform the associated anatomy image;

3. Softmean averages the transformed images into an atlas image;

4. Slicer produces three different 2D slices of the atlas; and

5. Convert is invoked over each 2D slice to create a graphical image.

In this implementation of the workflow, each invocation of an actor receives an XML structure, performs an update on a portion of that structure, and then sends the updated version of the structure to downstream actors (see Figure 3.4b). This is often carried out by streaming XML through invocations. Here we assume that each XML structure denotes an unranked, labeled ordered tree representing workflow data products, each tree node is assigned a unique identifier, and tree nodes represent either collection tokens or data tokens (which wrap complex objects or reference external data, e.g., stored within a file). A collection token may be an internal node (for non-empty collections) or a leaf node (for empty collections), whereas data tokens are leaf nodes only.
3.7. Running Example

Representing Provenance with Flow Graphs. Figure 3.4b shows the first invocation of each actor for a typical run of the workflow. The invocation of the AlignWarp actor (shown as AlignWarp:1) modifies the first AnatomyImage collection (node 2), and replaces its contents with a WarpParamSet data token (node 11). Similarly, the invocation of the Reslice actor uses this WarpParamSet to generate a new Image and Header data token (nodes 13 and 14, respectively). As shown, explicit “fine-grained” data dependencies are represented as part of the provenance of a run for cases when only a portion of an input data structure $d$ is modified by an invocation. Since only a portion of an input data structure $D$ is typically modified by an invocation, we also represent explicit (i.e., “fine-grained”) data dependencies as part of the provenance of a run. These dependencies are declared as part of the computation model, e.g., as in [BMR+08], or may be later inferred (e.g., based on constraints) as in [MBK+08]. For example, the dashed arrow from node 11 to node 2 in Figure 3.4b states that the WarpParamSet was created from the AnatomyImage collection by the first invocation of AlignWarp. Note that implicitly, node 11 depends on each of the descendents of node 2 (which includes nodes 6-10 in the figure). Similarly, each descendent of a collection implicitly inherits the dependencies of its ancestors (unless otherwise given). In our example, node 13 is a descendent of node 12 (a ReslicedImage collection), and thus implicitly depends on node 11. Taken together, Figure 3.4b denotes a portion of the trace for a run of Figure 3.4a; in particular, this trace shows only the information associated with the first invocation of each workflow actor.

Flow graphs can be used to derive standard process (or invocation) and data dependency graphs. For instance, Figure 3.4d shows an invocation dependency graph for the flow graph shown of Figure 3.4b, where nodes represent invocations of actors and edges represent dependency relationships between invocations. Figure 3.4c shows
3.8. Related Work

a corresponding data dependency graph, where nodes denote data items and edges denote dependency relations between data items. In general, edges within data dependency graphs are not labeled. In Figure 3.5, we explicitly label dependency edges with the invocation that created the item at the end of the arrow. For example, a dependency $x \xrightarrow{i} y$ states that the data item $y$ was produced by invocation $i$ from the data item $x$. Note that while the flow graph can be used to infer data and invocation dependency graphs, the flow graph cannot be reconstructed from these two graphs alone.

The unified model consists of three dimensions: (i) flow relations among input and output structures (defined by the flow graph), (ii) parent-child relations among nodes of structures, and (iii) lineage relations defining fine-grained data dependencies. These dimensions are independent in the sense that information in one dimension cannot be used in general to infer information of another dimension.

3.8 Related Work

Numerous approaches have been proposed in recent years for representing provenance information from scientific workflows runs (e.g. [SPG05, ABJF06, MLA+08, DF08, DC08, SFC07, BCS05, ZGST08, ZWF08, OCP+08, CBD06, BD08, BML+06, MFF+08] among others).

The Kepler Provenance Recorder [ABJF06] is modeled as a separate concern in the Kepler system and provides an extensible framework for capturing provenance information in actor-oriented scientific workflows. Provenance recorder listens to the execution and saves information customized by a set of parameters: context (who, what, where, when, and why that is associated with the run), input data and its associated metadata, workflow outputs and intermediate data products and workflow
3.8. Related Work

definition (entities, parameters, connections). The provenance recorder is paramet-

eric and customizable, allowing the user to choose different levels of granularity of

provenance data for recording. Various types of provenance information is captured

including data provenance (intermediate and end results including files and db refer-

ences), process provenance, error and execution logs and workflow design provenance.

The RWS model [BML+06] is designed to explicitly support scientific workflows

using pipeline parallelism over streaming data as well as cycles. The RWS model

assumes that each output token depends on all tokens input so far in the current

round. The RWS model records read, write, and state-reset events for each actor in

a workflow run and stores them in a relational event log. An approach is proposed

to reconstruct various dependency graphs from the event log for a workflow run to

support a wide range of scientific provenance queries. This model focuses on only

a minimal set of observables that allow us to answer many science-oriented user

questions, while ignoring non-functional observables such as timestamps, although

such information can be easily added.

ZOOM [CBD06] provides a model of provenance for scientific workflows, which

relies on log information comprising of start step events, input (read) events, output

(write) events, and the time at which these events occurred. ZOOM*UserViewsOs

model is composed of base tables which capture information about the workflow ex-

ecution, the workflow specification, the relationship between the specification and

execution, the inclusion relationship between composite modules and modules, and

information about user views. ZOOM provides user views allowing provenance infor-

mation to be calculated at different levels of granularity. Only visible and necessary

data (and steps) are returned. As for the queries, recursive queries expressing the

transitive closure on inputs and outputs are used, and both the flow of data and steps
3.8. Related Work

(that is, both data and events) can be queried.

The Taverna system [SZG07] uses Semantic Web technologies for representing provenance metadata at four levels: process, data, organization, and knowledge. Two levels of ontologies are used. A domain-independent schema ontology is used to describe the classes of resources and the properties between them that are needed to represent the four levels of provenance. A domain ontology is used to classify various types of resources such as data types, service types, and topic of interest for a particular domain.

The VisTrails system [BCS05] supports provenance tracking of workflow evolution in addition to data derivation history. In VisTrails, workflow evolution provenance is represented as a rooted tree, in which each node corresponds to a version of a workflow, and each edge corresponds to an update action that was applied to the parent workflow to create the child workflow. The above provenance systems are tightly coupled with their scientific workflow environments.

A couple of stand-alone provenance systems have also been developed, including the PReServ system [GJM06] and the Karma system [SPGM06a]. PReServ supports the recording of interaction provenance, actor provenance, and input provenance with the Provenance Recording Protocol (PReP), which specifies the messages that actors can asynchronously exchange with the provenance store to support provenance submission.

Although these above models differ in many ways as how they represent, store and access provenance of workflow execution, most of these models have mechanisms to infer process and data dependencies. Most of these models adopt a “conventional” provenance model where they record input and outputs of process invocations and inference rule is used (based on time or causal relationship between processes) in which
invocation outputs are assumed to depend on all associated inputs, and invocations only produce new data by employing transform semantics, i.e., they do not employ an update (or \( \Delta \)) semantics. In \([\text{MBK}^+08]\) and \([\text{MBZG}08]\), these assumptions are relaxed such that constraints are used to describe anticipated dependency patterns of actors. In contrast, our goal is to record explicit invocation dependencies (e.g., generated by “dependency-aware” actors), thus expanding the possible data dependencies that can be recorded for actor invocations.

Alternatively, workflow models that work over structured data (e.g., \([\text{BMR}^+08, \text{Slo}07, \text{MBZG}08, \text{Tom}06]\)) often employ update semantics, where only a portion of an incoming XML stream is modified by each workflow step. Applying conventional provenance approaches to such models results in provenance information that is either too coarse or potentially incomplete and even incorrect, which can lead to incorrect analysis of scientific results \([\text{ABML}09b]\).

In addition, conventional models of provenance \([\text{MFF}^+08, \text{CJR}08, \text{ABJF}06, \text{DF}08]\) are largely based on the assumption that data structures and processes are atomic. These assumptions do not hold, however, for many computation models employed within current scientific workflow systems \([\text{MBZL}09, \text{LAB}^+06]\). Our approach extends the conventional model by supporting workflows composed of actors that can have multiple invocations (e.g., as in process networks \([\text{LAB}^+06]\)) and that employ. Our provenance model subsumes conventional approaches for representing workflow provenance by supporting more advanced workflow computation models that permit multiple invocations of processes (e.g., for pipelining and loops), structured data, and update semantics.

Our provenance model is inspired by approaches for reference-based versioning of XML \([\text{CTZ}02, \text{MACM}01, \text{BCC}06]\) and semistructured data \([\text{CAW}98]\), and employs
a versioning model which supports a small set of operations (insert and delete), allows a partial ordering of versions, and which adds data and invocation dependency annotations.

3.9 Conclusion

In this Chapter, we have described a formal provenance model that generalizes the conventional model used for representing data and process provenance from workflow runs, by supporting a wider range of workflow types and workflow systems. Our provenance model does not limit invocations to simple “black-box” transformers, but allows the workflow system to declare and record data dependencies, based on an underlying update semantics. Our approach also simplifies provenance recording by not requiring implied provenance information to be given explicitly, while at the same time allowing detailed provenance to be recorded efficiently.

In the coming Chapters, we describe techniques to efficiently store, query, visualize, and summarize large provenance graphs.
Chapter 4

Efficient Provenance Storage

When a workflow is executed with nested data structures, actors operate upon these structures by either adding new nodes to collections or removing older nodes from collections, thereby producing new versions of data structures. Users might be interested to query across such versions of data structures, but the storage requirements to explicitly store different versions of data structures can be very expensive. In addition, when a large number of operations are performed on data and fine-grained provenance information is stored, then the number of provenance annotations can grow very quickly. Thus, provenance information may be many times larger than the workflow and input data itself [CJR08], workflows are often executed multiple times over different data sets and parameter settings, and most scientific research projects consist of many distinct workflows [BBD+09, ABML09a]. Maintaining provenance information for all nodes in a workflow trace can lead to prohibitively expensive storage costs in terms of database size and update time (i.e., the time needed to load provenance information into the database) [CJR08, HA08, DF08]. Further, common provenance queries are centered around dependency relations [MLA+08]. In order to answer such provenance queries across dependency relations, it is often required
that dependencies to be transitively closed, e.g., to find the nodes or invocations that directly or indirectly contributed to a specific node. Storing dependency closures can further increase storage cost. In this Chapter, I present techniques to store only unified data structure with some additional provenance annotations on process and data nodes, so that different versions of data structures can be materialized from the unified structure, as and when required.

4.1 Applying Update Semantics

Our model uses nested collections (i.e., XML trees) for representing scientific data—similar tree-structured models also have been employed in Kepler [MBZL09] and other approaches [MBZG08, HKS+05, CJR08]. We support actor update semantics and explicit data dependencies on these nested collection structures by employing embedded (or “inline”) provenance annotations, which describe the changes (or deltas $\Delta$) made to token streams by invocations at each workflow step. These annotations are used to record the tokens that were inserted (added to the token stream) and deleted (removed from the token stream by invocations, and the token dependencies that were introduced. Thus, instead of separately storing input and output structures of each actor invocation, our traces consist of a single, “condensed” version of the overall structure (Figure 4.3) and the complete version history including all intermediate “snapshots” (Figure 4.2) can be reconstructed as needed from the provenance annotations of the trace.

For instance, consider that a run of a simple workflow

$$X \rightarrow [A] \rightarrow Y \rightarrow [B] \rightarrow [Z]$$
4.1. Applying Update Semantics

is captured by a set of trace facts, say \text{in}(x, a), \text{out}(a, y), \text{in}(y, b), \text{and} \text{out}(b, z), implying that the invocation \(a\) of actor \(A\) directly preceded invocation \(b\) (of \(B\)), and that the output \(y\) directly depended on \(x\), while \(z\) indirectly depended on \(x\). In this example, invocation \(b\) received input \(y\) and produced output \(z\). Let \(\Delta_b\) denote the changes that \(b\) made to \(y\) to produce \(z\), so \(z = \Delta_b(y)\). Our approach is to store only the most recent version of data (here: \(z\)), but add provenance annotations (here: \(\Delta_b\)) that allow us to “undo” any updates and obtain all earlier versions of data products in the processing history. In this way, we can represent \(y\) as the pair \((\Delta_b^-, z)\), where the \textit{backward delta} \(\Delta_b^-\) denotes the inverse of \(\Delta_b\), i.e., we have

\[
y = \Delta_b^-(z).
\]

Similarly, the input \(x\) of invocation \(a\) can be represented by the pair \((\Delta_a^-, y)\) and we have

\[
x = \Delta_a^-(y) = \Delta_a^- (\Delta_b^-(z)).
\]

One advantage of this approach is that the provenance of invocations with update semantics can be represented without introducing unnecessary redundancy. For example, if \(y\) and \(z\) share a subtree \(s\), then \(s\) is stored only once in the resulting trace. Our model can also reduce the number of stored data dependencies by exploiting the nested structure of data collections to “cascade” dependencies within a trace.
4.2 Condensed Unified Provenance Model

We use unranked, labeled, ordered trees (similar to XML) to represent workflow data products. Tree nodes are either collection tokens or data tokens. A collection token may be an internal node (representing a non-empty collection) or a leaf node (empty collection); data tokens occur as leaf nodes only. All tokens are assumed to have unique identifiers (e.g., positive integers), and labels are used to tag (or type) nodes. For example, a data token containing a single DNA sequence may have the label dna_sequence; a collection token representing a DNA sequence alignment (i.e., a list of aligned sequences) may have the label dna_alignment.

Figure 4.1 shows an example workflow graph; a workflow run is given in Figure 4.2. In this example, invocation a receives the overall workflow input but only uses the
subtree at node 3 as input to generate a new output subtree at node 6 (integers ‘3’ and ‘6’ are node identifiers). We say that node 6 depended on node 3; or alternatively, 3 contributed to 6. The remaining parts of the input are passed on with node 6 to downstream actor invocations b and c. The changes \( \Delta_a \) made by a can be represented by the following provenance annotations:

\[
\Delta_a = \{ \text{ins}(6, a), \text{del}(3, a), \text{dep}(6, 3) \},
\]

stating that (i) node 6 was inserted by invocation a, (ii) node 3 was deleted by invocation a, and (iii) 6 depended on 3. Annotations like these are recorded during workflow execution and constitute the provenance of the invocation. The remaining invocations of Figure 4.2 are represented by the following sets of provenance annotations:

\[
\Delta_b = \{ \text{ins}(9, b), \text{dep}(9, 2) \}
\]

\[
\Delta_c = \{ \text{ins}(12, c), \text{dep}(12, 6) \}
\]

\[
\Delta_d = \{ \text{ins}(15, d), \text{dep}(16, 9), \text{dep}(17, 12) \}
\]

In the conventional model described above, the input and output of each invocation (i.e., \( x_1, \ldots, x_5 \) in Figure 4.2) would be recorded as part of the workflow trace. This approach would require storing the same sets of nodes multiple times, e.g., nodes 1 and 2 are part of every intermediate data product of the run. Instead, in our approach, only the unique set of nodes across all inputs and outputs are recorded together with the above provenance annotations.

Figure 4.3 shows this “condensed” trace for the workflow run of Figure 4.2. We use the following conventions to represent annotations in the figure: “+a” denotes

\(^1\)We say ‘node n’ or simply ‘n’ for ‘the subtree rooted at n’.
4.3 Formalization of Condensed Unified Provenance Model

Data collections are represented as (XML) trees; these token streams constitute the dataflow shown in Figure 4.1. From the perspective of provenance, we assume that actor invocations operate by: (1) inserting new subtrees into the tree (i.e., data stream); and (2) deleting node fragments from the tree, thereby removing subtrees from the data stream and not passing them on to downstream invocations.2

A workflow execution trace $T = (S, I, A, \prec_I)$ consists of a tree structure $S$, invocations $I$, node annotations $A$, and a strict, partial invocation order $\prec_I$ (i.e. repre-

2While supporting only these two operations may seem restrictive, in practice they allow workflow systems to better support concurrent execution of actors by avoiding potential race conditions caused by allowing ad-hoc structural modifications [MBZL09].
senting a DAG). The tree $S = (N, E)$ is unranked, labeled, and ordered; $N$ is a set of nodes, and $E \subseteq N \times N$ is a set of (ordered) parent-child edges. A node annotation $A$ is a (partial) mapping

$$A : N \rightarrow I^+ \times I^- \times 2^N,$$

associating with a node $n \in N$ the invocation that inserted $n$ (else $\perp$), the invocation that deleted $n$ (else $\perp$), and a subset of nodes from $N$ on which an insertion depended.

We consider both node annotations $A$ and invocation order $\prec_I$ as constituting provenance annotations. For a node $n$ such that

$$A(n) = (+a, -b, \{x_1, \ldots, x_n\}),$$

the invocation that inserted $n$ is $a$, the invocation that deleted $n$ is $b$, and the insertion of $n$ depended on all $x_i$ ($1 \leq i \leq n$). We use $\text{child}(x, y)$ to denote that the child of node $x$ is $y$. The following relations express provenance annotations:

1. $\text{ins}(n, a)$ states that a node $n$ was inserted by invocation $a$. Nodes are either inserted by an invocation or else were part of the overall workflow input.

2. $\text{del}(n, a)$ states that a node $n$ was deleted (i.e., not passed on) by an invocation $a$. The node is still stored within the trace, but is “marked” deleted. Marked deletions allow us to reconstruct intermediate versions of the trace.

3. $\text{dep}(n, d)$ states that the insertion of a node $n$ directly depended on a node $d$ (i.e., $d$ “contributed to” $n$).

4. $a \prec b$ states that invocation $a$ preceded invocation $b$, i.e., $b$ received input which includes (part of) $a$’s output.\(^3\)

\(^3\)a $\prec b$ does not imply that $a$ finished executing before $b$ started. Streaming execution models can exploit this flexibility [MBZL09].
4.4 Minimizing Provenance Annotations

An important feature of our model is that not all nodes in a trace require explicit provenance annotations. Instead, these annotations can be inferred from the annotations of other nodes in a trace. In general, this approach allows traces to be stored in a more compact form as compared to an equivalent trace that is fully annotated. The hierarchical structure of a trace is used to minimize the number and size of provenance annotations recorded. For example, insertion annotations can be recorded at
4.4. Minimizing Provenance Annotations

(1) \( \text{child}(p, c) \land \text{ins}(p, x) \land \text{ins}_{\bot}(c) \rightarrow \text{ins}(c, y) \)

(2) \( \text{child}(p, c) \land \text{del}(p, x) \land \text{del}_{\bot}(c) \rightarrow \text{del}(c, y) \)

(3) \( \text{child}(p, c) \land \text{ins}(p, x) \land \text{ins}(c, y) \land x \neq y \rightarrow (x < y) \)

(4) \( \text{child}(p, c) \land \text{del}(p, x) \land \text{del}(c, y) \land x \neq y \rightarrow (y < x) \)

(5) \( \text{ins}(n, x) \land \text{del}(n, y) \rightarrow (x < y) \)

(6) \( \text{dep}(n, d) \land \text{ins}(d, x) \land \text{ins}(n, y) \rightarrow (x < y) \)

(7) \( \text{dep}(n, d) \land \text{del}(d, x) \land \text{ins}(n, y) \rightarrow (y < x) \)

(8) \( \text{dep}(n, d) \land \text{ins}(d, x) \land \text{del}(n, y) \rightarrow (x < y) \)

(9) \( \text{dep}(n, p) \land \text{child}(p, c) \land \text{ins}(n, y) \land \\
((\text{ins}(c, x) \land (x < y)) \lor \text{ins}_{\bot}(c)) \land \\
((\text{del}(c, z) \land (z \not< y)) \lor \text{del}_{\bot}(c)) \rightarrow \text{dep}(n, c) \)

(10) \( \text{child}(p, c) \land \text{dep}(p, n) \land \text{ins}(p, y) \land \text{ins}(c, y) \rightarrow \text{dep}(c, n) \)

(11) \( \text{ins}(n, a) \land \text{ins}(n, b) \rightarrow (a = b) \)

(12) \( \text{dep}(n, d) \rightarrow \exists a \ \text{ins}(n, a) \)

(13) \( \text{del}(n, a) \land \text{del}(n, b) \rightarrow (a \not< b) \land (b \not< a) \)

Figure 4.5: Logic rules for computing fully annotated traces (1–10), and additional constraints (11–13) for checking well-formedness. Free variables are implicitly \( \forall \)-quantified.
4.4. Minimizing Provenance Annotations

the highest node \( n \) in the trace where they apply, and implicitly cascade to descendent nodes (i.e., all subtree nodes of \( n \)) that existed at the time of the insertion (similarly for deletion annotations). Nodes that depend on a collection (node) \( c \) also depend on all descendents (subtree nodes) of \( c \) that were present at the time of insertion of nodes. In Figure 4.3, e.g., the insertion and dependency annotations for nodes 7 and 8 are the same as for node 6; nodes 6, 7, and 8 have (implicit) dependencies on nodes 4 and 5.

A partially annotated trace can be expanded by applying a set of inference rules, shown graphically in Figure 4.4 and as logic rules in Figure 4.5. Rules 1–10 in Figure 4.5 correspond to those in Figure 4.4; whereas rules 11–13 place additional constraints on traces. We employ unary relations \( \text{ins}_\bot(N) \) and \( \text{del}_\bot(N) \) in Figure 4.5 to mark nodes of a trace without initial insertion or deletion annotations.

In Figure 4.4, the first set of inference rules (1–2) are used to propagate insertion and deletion annotations from collection nodes to their children nodes. For example, if a collection (or parent) node \( p \) has an insertion annotation \( \text{ins}(p, x) \), and a child \( c \) of \( p \) does not have an associated insertion annotation, then applying the inference rule results in the annotation \( \text{ins}(c, x) \). The second set of rules (3–5) are used to infer invocation order according to the insertion and deletion annotations of parent-child relationships and individual nodes. The third set of rules (6–8) are also used to infer invocation order, but using insertion and deletion annotations on token dependencies. The final set of rules (9–10) are used to expand the set of dependencies of nodes and to propagate dependencies to child nodes.

The additional rules of Figure 4.5 define constraints on traces requiring that: (11) each node be inserted by at most one invocation; (12) any node that is dependent on another node be inserted by some invocation; and (13) an invocation receiving a
4.4. Minimizing Provenance Annotations

Figure 4.6: The same trace shown minimally annotated in (a) and (b) and fully annotated in (c).

deleted node cannot delete the node again.

Expanding and Collapsing Traces. We use structural recursion to expand and collapse traces, i.e., to construct equivalent traces with larger or smaller sets of provenance annotations. Specifically, we can complete a trace by traversing its collection and data nodes, and at each step in the traversal apply the annotation rules of Figure 4.4. In general, it is not possible to fully expand a given trace in a single pass, i.e., a single preorder, inorder, postorder, or breadth-first traversal, in which each node of the trace is visited only once. As a simple counter-example, consider the trace in Figure 4.6(a). Its completed version cannot be computed using one of the above traversal algorithms, since node 6 must be visited before 2, node 2 must be visited before 3, and node 3 must be visited before 4.

Our approach is to expand traces using three distinct preorder (i.e., top-down, left-to-right) traversals of \( S \). The first pass propagates insertion and deletion annotations according to inference rules (1–2) of Figure 4.4, followed by applications of rules (3–5) to infer invocation order from nodes and parent-child relationships. The second pass generates the remaining invocation precedence relationships based on the rules (6–8). Finally, the third pass expands dependencies sets and propagates dependencies to child nodes using the rules (9–10). Note that insertion and deletion annotations must
be propagated before the second pass since both sides of a dependency require an
insertion and/or deletion to infer the remaining invocation precedence relationships.
Similarly, invocation order must also be known before dependencies can be expanded
and propagated.

Expanded traces can also be minimized using a similar approach. Specifically,
collapsing a fully annotated trace can be performed using a postorder (i.e, bottom-
up, left-to-right) traversal of the trace where the following annotations are removed
for each node \( n \): (i) \( \text{dep}(n, c) \) if \( \text{dep}(n, p) \) and \( \text{child}(p, c) \); (ii) \( \text{dep}(n, d) \) if \( \text{child}(p, n) \) and \( \text{dep}(p, d) \); (iii) \( \text{ins}(n, x) \) if \( \text{child}(p, n) \) and \( \text{ins}(p, x) \); and (iv) \( \text{del}(n, y) \) if \( \text{child}(p, n) \) and \( \text{del}(p, y) \). Finally, we remove the invocation order annotations that are implied
according to the rules in Figure 4.4(3–8).

**Determining Versions.** Another important feature of our provenance model is
that although each token is stored only once in a trace, the input and output of each
invocation can be reconstructed by computing the corresponding “version” of the
final workflow output. As described previously, the versions of the tree structures of
Figure 4.3 are shown in Figure 4.2.

**Determining Intermediate Versions.** Given a fully annotated trace, we compute
the corresponding version of the input structure used by an invocation \( b \) as follows.
Let \( P = \{ a \in I \mid a \prec b \} \) be the set of invocations that preceded \( b \). The version
 correponding to the input of \( b \) includes all nodes not deleted by an invocation in
\( P \) such that either the node (1) was inserted by an invocation in \( P \) or (2) has no
insertion annotation and thus was an input to the workflow run. The output of \( b \) is
computed similarly, i.e., by removing from the input of \( b \) the nodes deleted by \( b \) and
adding the nodes inserted by \( b \).

**Determining Workflow Input Versions.** Given a fully annotated trace, we com-
pute the versions used by workflow execution as its input structures as follows. The input versions of workflow execution includes all nodes that have no insertion annotation.

**Determining Workflow Output Versions.** Given a fully annotated trace, we compute the versions output by workflow execution as its output structures as follows. The output versions of workflow execution includes all nodes that (1) have insertion annotation, and (2) have no deletion annotation.

Maintaining provenance information for all nodes in a workflow trace can lead to prohibitively expensive storage costs in terms of database size and update time (i.e., the time needed to load provenance information into the database) [CJR08, HA08, DF08]. We show that these costs can be decreased using our provenance model by exploiting trace equivalence to collapse provenance annotations. However, *reducing* the number of annotations in this way can lead to *increases* in query execution time, e.g., queries that need to access collapsed annotations will need to expand these annotations as part of the query.

Moreover, most of the provenance queries are centered around lineage graph (see, Figure 5.3). Seven out of nine provenance queries in first provenance challenge [MLA⁺08] was on this lineage graph. In order to answer such provenance queries it is often required that dependencies to be transitively closed, e.g., to find the nodes or invocations that directly or indirectly contributed to a specific node. Storing dependency closures can further increase storage cost, whereas storing only immediate dependencies requires more expressive query constructs, such as recursion, that often increase query time [HA08].

We present optimizations for balancing this trade-off between storage size, update time, and query-response time. Our focus is on efficiently storing, loading, and
querying individual traces within a relational database system. We also define a set of strategies for storing traces (Figure 4.7), each of which consist of: (i) workflow run identifiers; (ii) nested collection structures; (iii) insertions and deletions; (iv) immediate and transitive node dependencies; and (v) immediate and transitive invocation order. Similar to other approaches [CJR08, HA08], we define new techniques for storing direct and indirect node dependencies and invocation order, allowing provenance queries to be expressed using purely relational languages instead of the more typical approach of using expressive but less efficient recursive query constructs (e.g., [BBDH08, BCC06]). We use these strategies and reduction techniques to compare the effectiveness of approaches for minimizing storage size, update time, and query time on real-world and synthetic traces.

4.5 Storage Strategies

The basic setting we consider for storing workflow traces is summarized in Figure 4.7. Given an XML file containing a workflow trace, we first perform a number of pre-processing tasks before loading the file into a database. Pre-processing includes parsing the trace file, applying trace inference rules, determining interval-encodings [Koc06] for nested data collections, and computing transitive dependency and invocation-order closures. Interval encodings are used to efficiently access the ancestors and descendents of nested collections. Similarly, we store both direct and indirect node dependencies and invocation orders (although in a reduced form) to increase overall query performance and to allow lineage queries to be expressed using relational queries.

Once a trace is pre-processed, one of the following three storage strategies is selected (shown in the middle of Figure 4.7):
### 4.5. Storage Strategies

Figure 4.7: Basic trace storage strategies. Traces are stored in either their collapsed, asserted, or expanded form; reduction techniques are applied to minimize storage size; and relational views are used to rewrite provenance queries expressed against the store.

**Collapsed Strategy.** The first strategy stores a *collapsed*, but not minimal, representation of the trace. In particular, we only collapse the trace with respect to node dependency sets. The trace resulting from the pre-processing step is partially collapsed by removing all implied dependencies of nodes (applying the inverse of the dependency-expansion rule 9 of Figure 4.5). Thus, in the collapsed strategy, insertions and deletions annotations of nodes have been propagated to their children. As a node can be inserted and deleted only once, it is not expensive to store insertion and deletion for all nodes. But a node can have many other nodes as its immediate dependency nodes, thus storing dependencies in collapsed state reduces dependencies annotation.

** Asserted Strategy.** The second strategy stores the *asserted* trace, i.e., the input trace to the system (Figure 4.7). An asserted trace may contain redundant data dependencies and invocation orders. For both the collapsed and asserted strategies, provenance queries are rewritten (via the expand-annotation view of Figure 4.7; see Section 4.8) to dynamically reconstruct the relevant portion of the expanded trace for the query. In the asserted strategy, we also record during the update process which
4.6 Storage Reduction Techniques

Nodes in the trace are expanded. Although not described further here, this approach allows the expand-annotation view to be applied only when an unexpanded portion of the trace is being queried.

**Expanded Strategy.** The third strategy stores the *fully-annotated* (i.e., *expanded*) representation of the trace, which includes all annotations that are derived from applying the inference rules of Figure 4.5.

Each of these three strategies employ space-reduction techniques for storing only once the common subsets of node dependency sets, node dependency closures, invocation orders, and invocation order closures. Similar to denormalization, these techniques require splitting dependency and closure relations into multiple tables. Thus, provenance queries for each case are rewritten (via the *expand-reduction* view of Figure 4.7; see Section 4.8) to reconstruct the relevant portion of the original tables representing dependencies and invocation order. Update time \((U)\) consists of the time required to pre-process the trace; compute interval encodings of nested collections; collapse or expand the trace; reduce common dependency and closure subsets; and load the resulting records into the database.

### 4.6 Storage Reduction Techniques

Nodes within a trace often have similar sets of immediate dependencies. Using a straightforward representation that stores node-dependency pairs as tuples \(R(N, D)\), each shared dependency node in \(R\) will be stored multiple times. For example, if \(n_1\) and \(n_2\) both depend on \(n_3\) and \(n_4\), \(R\) will contain the tuples \(R(n_1, n_3)\), \(R(n_1, n_4)\), \(R(n_2, n_3)\), and \(R(n_2, n_4)\), resulting in \(n_3\) and \(n_4\) being stored twice.

Below we describe four approaches that can be applied sequentially to reduce the
4.6. Storage Reduction Techniques

Table 4.1: The result of sequentially applying reduction techniques on example dependency sets

<table>
<thead>
<tr>
<th>Technique</th>
<th>Input Dependencies</th>
<th>Nodes</th>
<th>Output Dependencies</th>
<th>Subsequence</th>
<th>Subset</th>
<th>Dep. Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0) None</td>
<td>100 → {10, 20, 30, 40, 50}</td>
<td>100</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>17</td>
</tr>
<tr>
<td>(1) Duplicate-Set Reduction</td>
<td>100 → {10, 20, 30, 40, 50}</td>
<td>100</td>
<td>&amp;1 → {10, 20, 30, 40, 50}</td>
<td>-</td>
<td>-</td>
<td>12</td>
</tr>
<tr>
<td>(2a) Subsequence Reduction</td>
<td>&amp;1 → {10, 20, 30, 40, 50}</td>
<td>&amp;1 → &amp;1</td>
<td>&amp;1 → {10, 20, 30, 40, 50}</td>
<td>&amp;2 → &amp;1</td>
<td>&amp;2 → &amp;1</td>
<td>10</td>
</tr>
<tr>
<td>(2b) Subset Reduction</td>
<td>&amp;1 → {10, 20, 30, 40}</td>
<td>100</td>
<td>&amp;1 → &amp;1</td>
<td>&amp;2 → &amp;1, [10, 40]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3) Subsequence-Subset Reduction</td>
<td>&amp;1 → {10, 20, 30, 40}</td>
<td>100</td>
<td>&amp;1 → &amp;1</td>
<td>&amp;2 → &amp;1, [10, 40]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Redundancy of dependency sets. Each approach requires slight changes to the simple dependency schema $R(N, D)$ to minimize the number of duplicate node occurrences within a provenance database, thereby reducing the overall size of the database. While many possible reduction techniques can be considered, we employ the techniques below because they are simple enough to be applied efficiently at runtime and the reduced provenance store can be queried directly through relational views defined over the reduced schema. Table 4.1 demonstrates these techniques using four simple dependency sets. The unique dependency nodes in this example are 10, 20, 30, 40, and 50; whereas nodes 100, 200, 300, and 400 are dependent on subsets of these dependency nodes. We use the notation $n → \{n_1, ..., n_k\}$ in the figure to denote tuples of the form $R(n, n_1), ..., R(n, n_k)$. When no reductions are applied (first row of Table 4.1), the dependencies given in the example consist of 5 unique dependency nodes requiring 17 total dependency nodes to be stored.

**Duplicate-Set Reduction.** This reduction technique is a common storage optimization in which nodes indirectly relate to their dependency sets through “pointers” (i.e., integer identifiers $x$, denoted by &x). Storage size is reduced by using...
4.6. Storage Reduction Techniques

**Input:** Fully-annotated trace \( T \)

**Output:** Dictionaries \( D \) and \( P \)

1. \( D \leftarrow \[ \], \ P \leftarrow \[ \], \ S \leftarrow \[ \], \ i \leftarrow 1 \)

2. for \( n \) in nodes of \( T \) do

3. \( D_n \leftarrow \{ d \mid \text{dep}(n, d) \} \)

4. if \( S[D_n] = \text{nil} \) then

5. \( D[n] \leftarrow \&i, \ P[\&i] \leftarrow D_n, \ S[D_n] \leftarrow \&i \)

6. \( i \leftarrow i + 1 \)

7. else

8. \( D[n] \leftarrow S[D_n] \) /* Reduction*/

9. end if

10. end for

---

**Figure 4.8: Algorithm for Duplicate-Set Reduction**

the same pointer for each node that depends on the same set of dependencies. In Table 4.1, applying this technique reduces the total number of nodes stored to 12.

Duplicate-set reduction divides the original dependency relation into two distinct relations \( R_N(N, \&P) \) and \( R_D(\&P, D) \), represented in Table 4.1 using \( n \rightarrow \&p \) and \( \&p \rightarrow \{n_1, ..., n_k\} \). Figure 4.8 describes this technique in more detail. The duplicate-set algorithm takes \( O(N) \) time, where \( N \) is the number of nodes in the trace.

**Subsequence Reduction.** This reduction technique identifies dependency sets that represent subsequences of larger dependency sets, and stores only the range of the subsequence relative to the larger set. This technique is applied after duplicate-set reduction. Each unique dependency set is sorted (e.g., using node identifiers) to identify subsequences. The approach stores the largest dependency sequences and creates a new relation \( R_{\text{subseq}}(\&P_{\text{from}}, \&P_{\text{to}}, i, j) \) to store subsequences in which \( i \) and \( j \) denote the (inclusive) range of the subsequence. Only subsequences of length 3 or greater are considered. In Table 4.1, applying this technique reduces the total number of nodes stored to 10. Subsequence reduction is effective in cases where actors perform a form of running aggregation, either within their invocation or with
4.6. Storage Reduction Techniques

**Input:** Dictionary $P$ mapping pointers to sorted dependency sets

**Output:** Dictionaries $P$ and $S$

1. $S \leftarrow []$
2. for $p_s$ in keys of $P$ and $|P[p_s]| > 2$ do
3.    found $\leftarrow$ false, maxlen $\leftarrow 0$, $p_b \leftarrow \bot$, $D_s \leftarrow P[p_s]$
4.    for $p$ in keys of $P$ and $p \neq p_s$ do
5.        $D_b \leftarrow P[p]$
6.        if $D_s$ is a subsequence of $D_b$ and $|D_b| > $ maxlen then
7.            found $\leftarrow$ true, $p_b \leftarrow p$, maxlen $\leftarrow |D_b|$
8.        end if
9.    end for
10. if found then
11.    $i \leftarrow $ first($D_s$), $j \leftarrow $ last($D_s$)
12.    $S[p_s] \leftarrow (p_b, i, j)$ /* Reduction*/
13.    $P[p_s] \leftarrow \emptyset$ /* Reduction*/
14. end if
15. end for

Figure 4.9: Algorithm for Subsequence Reduction

respect to the overall token stream. Figure 4.9 describes this technique in more detail.

Subsequence reduction takes $O(|P|^2)$ time, in which for each unique dependency set (where $|P|$ is the number of unique sets) the largest containing sequence is found.

**Subset Reduction.** This reduction technique identifies dependency sets that are proper subsets of larger dependency sets. This technique is also applied after duplicate-set reduction. Unlike in subsequence reduction, this approach stores the smaller set and creates a new relation $R_{\text{subset}}(\&P_{\text{from}}, \&P_{\text{to}})$ such that $P_{\text{to}} \subseteq P_{\text{from}}$, where $P$ denotes the dependency set pointed to by $\&P$. Only subsets of length 2 or greater are considered. We also only permit one level of subset reduction, i.e., a dependency set that is stored as a subset of another set cannot itself be represented through a subset. If the number of nested subset relationships is not held constant (in our case we allow one level of nesting), recursive views would be required in general to reconstruct dependency sets. In Table 4.1, applying our subset reduction technique
4.6. Storage Reduction Techniques

Input: Dictionary $P$ mapping pointers to dependency sets
Output: Dictionaries $P$ and $S$

1: $S \leftarrow []$, savings $\leftarrow$ true, $M \leftarrow \emptyset$
2: while savings do
3:   savings $\leftarrow$ false, $r \leftarrow 0$, $p_s \leftarrow \perp$, $B \leftarrow \emptyset$
4:   for $p'_s$ in keys of $P$, $p'_s$ not in $M$, and $|P[p'_s]| > 1$ do
5:     $B' \leftarrow \emptyset$
6:     for $p'_b$ in keys of $P$, $p'_b \neq p'_s$, and $p'_b$ not in $M$ do
7:       if $P[p'_s] \subseteq P[p'_b]$ then
8:           $B' \leftarrow B' \cup \{p'_b\}$
9:       end if
10:      end for
11:     if $|P[p'_s]| * |B'| > r$ then
12:        savings $\leftarrow$ true, $r \leftarrow |P[p'_s]| * |B'|$, $p_s \leftarrow p'_s$, $B \leftarrow B'$
13:    end if
14:   end for
15:  if savings then
16:     for $p_b$ in $B$ do
17:       $P[p_b] \leftarrow P[p_b] - P[p_s]$, $S[p_b] \leftarrow p_s$ /* Reduction*/
18:     end for
19:  end if
20: $M \leftarrow M \cup \{p_s\}$ /* Reduction*/
21: end while

Figure 4.10: Algorithm for Subset Reduction

reduces the total number of nodes stored to 8. Figure 4.10 describes this technique in more detail. Subset reduction takes $O(|P|^3)$ time, in which the subset that gives the largest reduction is identified in $O(|P|^2)$ time and in the worst case $|P|$ such subsets are found. In each pass, the subset that maximizes the product of its size and the number of sets that contain it is selected, and the set is not considered for reduction in subsequent passes.

Subsequence-Subset Reduction. This reduction technique first applies the subsequence reduction, and then applies the subset reduction to the result. In Table 4.1, applying this technique reduces the total number of nodes stored to 7. This approach always provides better reduction than using only subsequence reduction. However,
4.6. Storage Reduction Techniques

Table 4.2: Node and node dependency schemas

<table>
<thead>
<tr>
<th>Before Reduction</th>
<th>After Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>node(R, N, I_{ins}, I_{del})</td>
<td>r-node(R, N, I_{ins}, I_{del}, &amp;P_{dep}, &amp;P_{depc})</td>
</tr>
<tr>
<td>dep(R, N, N_{dep})</td>
<td>r-dep(R, &amp;P_{dep}, N_{dep})</td>
</tr>
<tr>
<td></td>
<td>r-dep-subseq(R, &amp;P_{from}^{dep}, &amp;P_{to}^{dep}, i, j)</td>
</tr>
<tr>
<td></td>
<td>r-dep-subset(R, &amp;P_{from}^{dep}, &amp;P_{to}^{dep})</td>
</tr>
<tr>
<td>depc(R, N, N_{depc})</td>
<td>r-depc(R, &amp;P_{depc}, &amp;P_{dep})</td>
</tr>
<tr>
<td></td>
<td>r-depc-subseq(R, &amp;P_{from}^{depc}, &amp;P_{to}^{depc}, i, j)</td>
</tr>
<tr>
<td></td>
<td>r-depc-subset(R, &amp;P_{from}^{depc}, &amp;P_{to}^{depc})</td>
</tr>
</tbody>
</table>

in certain situations applying only the subset technique can provide better overall reduction than applying subsequence followed by subset. Thus, our current implementation applies both techniques (during the pre-processing phase) and selects the result with better overall reduction. Similar to subset reduction, this algorithm runs in $O(|P|^3)$ time.

**Closure Reduction.** The above reduction techniques can be generically applied to any transitive binary relation. We apply the reductions to immediate data dependencies, dependency closures, immediate invocation order, and invocation-order closures. A portion of the schema we use to represent immediate data dependencies is shown in Table 4.2; similar relations are defined for immediate invocation order and its closure. As shown in Table 4.2, we employ the reduction techniques over closures of pointers instead of closures of nodes (similarly, for invocations). This approach results in an overall smaller representation compared to directly reducing node closures, since pointer closures do not store nodes and also do not store the immediate-node reduction.

For example, the result of applying the duplicate-set reduction technique to the
4.7. Reducing overall Provenance Storage Size

trace of Figure 4.3 gives the immediate dependencies $17 \rightarrow &3$, $12 \rightarrow &2$, and $6 \rightarrow &1$ such that $&3 \rightarrow \{12, 13, 14\}$, $&2 \rightarrow \{6, 7, 8\}$, and $&3 \rightarrow \{3, 4, 5\}$. The transitive dependency closure for node 17 is $\{3, 4, 5, 6, 7, 8, 12, 13, 14\}$, which is represented in our approach by the set of immediate dependency pointers $\{&1, &2, &3\}$. We further reduce these pointer-based closures using our reduction techniques, resulting in a representation that is smaller than the corresponding reduction of the node-based closure.

As shown in Table 4.2, we store multiple workflow traces that are distinguished by a run identifier $R$. Additionally, each node is stored with its insertion and deletion annotation as well as its immediate-dependency $&P_{dep}$ and transitive-dependency $&P_{depc}$ pointers.

4.7 Reducing overall Provenance Storage Size

Apart from the reduction techniques, we also apply other strategies to reduce the overall provenance store size. We list down some of those strategies.

1. **Storing only one copy of data value.** Scientists often run workflows many times with similar data values to repeat their experiments. Instead of storing values for every data (input, intermediate, output) of workflow execution, we maintain a single repository (table), where we store data values only once, and while inserting the data attributes into provenance tables, we assign a pointer to data to refer to its value in the repository table. We have found that this technique reduces provenance size store by huge factor where workflow is data-intensive and workflow execution use or generate data with same values.

2. **Storing distinct set of provenance annotations pointers for nodes.**
4.7. Reducing overall Provenance Storage Size

During a workflow execution, many nodes get associated with similar provenance annotations (Insertion, deletion, dependencies). When we run our equivalence algorithm over dependencies set each node is assigned it dependency pointer. Insertion and deletion invocations are refereed by local id of invocation assigned during parsing of trace. Instead of storing these annotations for each node in different columns, we can store only distinct set of such annotation pointers by running our one-level equivalence algorithm over these annotations pointer set. Each node is assigned a pointer that that points to their annotation set of Insertion, deletion, dependencies pointers.

3. **Inferring input and output of invocations from provenance annotations.** In our model, we don’t store input and output for each invocations, but they are computed based on insertion annotation and dependencies annotations. For an invocation, output refers to all nodes that have insertion pointer, referring to the invocation, where as input refers to all nodes that have deletion pointer referring to the invocation and all nodes in the dependency set which are referred by node that has insertion pointer referring to the invocation. This approach saves the storage space by not explicitly storing input and output of invocation, but computing it when required.

4. **Using interval encoding to fetch cascading provenance annotations for nodes.** Our model is designed such that instead of explicitly assigning all annotations and attributes to a node, it assigns only the difference between cascading attributes and annotations between related nodes. Similar to techniques like that of structural inheritance [BMWL07, CJR08] and hierarchical provenance [BCC06], if parent have attributes or annotations that would cascade to its child, then such attributes or annotations are recorded only for parent and
4.8. Rewriting Provenance Queries

For expanded traces, provenance queries over the schema shown on the left-side of Table 4.2 are answered using the view definition expressed over the schema for storing reductions (Table 4.2, right). We consider the case of node dependencies, however,

\[
\text{node}(R, N, I_{ins}, I_{del}) :\quad \text{r-node}(R, N, I_{ins}, I_{del}, \omega, \omega).
\]

\[
\text{dep}(R, N, N_{dep}) :\quad \text{r-node}(R, N, \omega, P, \omega), \text{v-dep}(R, P, N_{dep}).
\]

\[
\text{v-dep}(R, P, N) :\quad \text{v-dep}_0(R, P, N).
\]

\[
\text{v-dep}(R, P, N) :\quad \text{r-dep-subseq}(R, P, P_{to}, I, J), \text{v-dep}_0(R, P_{to}, N), N \geq I, N \leq J.
\]

\[
\text{v-dep}_0(R, P, N) :\quad \text{r-dep}(R, P, N).
\]

\[
\text{v-dep}_0(R, P, N) :\quad \text{r-dep-subset}(R, P, P_{to}), \text{r-dep}(R, P_{to}, N).
\]

Figure 4.11: Expand-reduction view for immediate dependencies.

\[
\text{depc}(R, N, N_{depc}) :\quad \text{r-node}(R, N, \omega, \omega, \omega, P), \text{v-depc}(R, P, P_{depc}), \\
\quad \text{v-dep}(R, P_{depc}, N_{depc}).
\]

\[
\text{v-depc}(R, P, P_{depc}) :\quad \text{v-depc}_0(R, P, P_{depc}).
\]

\[
\text{v-depc}(R, P, P_{depc}) :\quad \text{r-depc-subseq}(R, P, P_{to}, I, J), \text{v-depc}_0(R, P_{to}, P_{depc}), \\
\quad P_{depc} \geq I, P_{depc} \leq J.
\]

\[
\text{v-depc}_0(R, P, P_{depc}) :\quad \text{r-depc}(R, P, P_{depc}).
\]

\[
\text{v-depc}_0(R, P, P_{depc}) :\quad \text{r-depc-subset}(R, P, P_{to}), \text{r-dep}(R, P_{to}, P_{depc}).
\]

Figure 4.12: Expand-reduction view for transitive dependencies.

not for child. For example, if a set of metadata or annotation is attached to a parent, then only new or overriding metadata and annotations specific for children nodes is stored. In order to get all metadata or annotation for a node, one would need either some recursive procedure [BCC06] or some iterator [CJR08], to fetch all metadata or annotations up from children to root node, which are very expensive if the height of tree is large. We use interval encoding to write simple relational queries that fetch such information from all ancestor nodes. Such simple relational queries are very fast.

### 4.8 Rewriting Provenance Queries

For expanded traces, provenance queries over the schema shown on the left-side of Table 4.2 are answered using the view definition expressed over the schema for storing reductions (Table 4.2, right). We consider the case of node dependencies, however,
invocation-order is handled in a similar way. The views are relational queries written in non-recursive Datalog and can be expressed, e.g., using SQL. The view definitions in Figure 4.11 are used to recover nodes and immediate dependencies. Similarly, the view definitions in Figure 4.12 are used to recover dependency closures.

We also define a relational view in Figure 4.13 for expanding traces stored using the collapsed (or asserted) strategy. A relational query can be used to expand a collapsed trace since: (i) we explicitly store the transitive closure of the invocation order; (ii) we store the insertion and deletion annotations of each node directly within the node relation; (iii) we store the dependency set pointer of each node directly within the node relation; and (iv) we use interval encoding for nested collections. Thus, a traced stored via the collapsed strategy can be expanded by simply applying rule 9 in Figure 4.5, since (i), (ii), and (iii) already store the provenance information that would be derived from the remaining rules. We use (iv) to access collection descendents instead of children as in rule 9 of Figure 4.5. The variable $X$ is the insertion invocation of $N$, and $Y$ and $Z$ are the insertion and deletion invocations of node $C$, respectively. Node $C$ here is a descendent of $N_{dep}$, where $N_{dep}$ corresponds
4.9 Evaluation of Storage Strategies and Reduction Techniques

Here we evaluate our provenance model, storage strategies, and reduction techniques on both real-world and synthetic traces. We compare our storage strategies to the conventional provenance model discussed in Chapter 3 as well as to approaches that do not employ our reduction techniques. Our experiments evaluate the various approaches in terms of storage size, update time, and query time. All experiments were performed using a 2.4GHz Intel Core 2 duo PC with 2 GB RAM and 120 GB disk space. Our implementation stores all provenance information and base data using MySQL 5. Our algorithms were implemented in Java 1.5 and use JDBC to communicate with the provenance database.

Table 4.3 lists the storage strategies used for our evaluation. These strategies were compared using a set of synthetic traces that consist of actors employing the dependency patterns shown in Table 4.4. For each pattern, we consider traces of increasing numbers of nodes, ranging from 100 to 6000, and dependency annotations ranging from $10^3$ to $10^6$ with $10^4$ to $10^8$ transitive dependency edges. In addition to these synthetic traces, we also evaluate our approaches using the following real-world traces: the CGR and PLC workflows [BMR+08] infer phylogenetic trees from
### Table 4.3: Provenance storage strategies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Label</th>
<th>Annotations</th>
<th>Dep. Closure</th>
<th>Reduced</th>
<th>Optimizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Expanded</td>
<td>SE</td>
<td>Expanded</td>
<td>Node-Based</td>
<td>No</td>
<td>Immediate &amp; Transitive Queries</td>
</tr>
<tr>
<td>Naive Expanded</td>
<td>NE</td>
<td>Expanded</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Naive Collapsed</td>
<td>NC</td>
<td>Collapsed Dependencies</td>
<td>No</td>
<td>No</td>
<td>Storage size</td>
</tr>
<tr>
<td>Reduced Expanded</td>
<td>RE</td>
<td>Expanded</td>
<td>Pointer-Based</td>
<td>Yes</td>
<td>Storage Size, Immediate &amp; Transitive Queries</td>
</tr>
<tr>
<td>Reduced Collapsed</td>
<td>RC</td>
<td>Collapsed Dependencies</td>
<td>Pointer-Based</td>
<td>Yes</td>
<td>Storage Size</td>
</tr>
</tbody>
</table>

### Table 4.4: Workflow execution trace patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Label</th>
<th>Dependencies</th>
<th>Removes Input Nodes from Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependency All</td>
<td>DA</td>
<td>All output depend on all inputs in token stream at time of insertion</td>
<td>No</td>
</tr>
<tr>
<td>Transform Add</td>
<td>TA</td>
<td>All invocation outputs depend on all invocation inputs</td>
<td>No</td>
</tr>
<tr>
<td>Transform Delete</td>
<td>TD</td>
<td>All invocation outputs depend on all invocation inputs</td>
<td>Yes</td>
</tr>
<tr>
<td>Mixed</td>
<td>MIXED</td>
<td>Repeatedly perform DA followed by TA followed by TD</td>
<td>Yes (via TD)</td>
</tr>
</tbody>
</table>

protein and morphological sequence data; the FPC workflow was used within the first provenance challenge [MLA+08]; and the Steam workflow characterizes microbial communities by clustering and identifying DNA sequences of 16S ribosomal RNA. The dependency annotations of these traces range from $2 \cdot 10^3$ to $2 \cdot 10^4$ with $5 \cdot 10^3$ to $5 \cdot 10^4$ transitive dependency edges.

**Storage Size.** Figure 4.14 shows the storage sizes observed for the TD, DA, and Mixed synthetic traces. Note that we only consider the conventional (Conv) approach for the TD case, since it cannot be used to represent the other dependency patterns. The storage space for SE grows exponentially because it stores provenance and closure records for each node (which ranges from thousands to millions of records). The storage size for NE also grows exponentially, similarly requiring millions of records to
Table 4.5: Real-world workflow execution traces used in experimental evaluation

<table>
<thead>
<tr>
<th>Workflow</th>
<th>Label</th>
<th>Nodes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Clustal-Gblocks-RAxML</strong></td>
<td>CGR</td>
<td>147</td>
<td>Infers phylogenetic trees from protein sequence data by aligning a protein sequences using Clustal, refining the alignment using Gblocks, and then determining the maximum-likelihood tree using RAxML</td>
</tr>
<tr>
<td><strong>Pars Loop Consensus</strong></td>
<td>PLC</td>
<td>152</td>
<td>Infers phylogenetic trees from morphological sequence data. The PLC workflow searches for the most parsimonious trees that explain the observed morphological character states, executing the Pars program iteratively in a loop, and then determining the consensus of the equally multiple, equally parsimonious trees that are discovered.</td>
</tr>
<tr>
<td><strong>First Provenance Challenge</strong></td>
<td>FPC</td>
<td>185</td>
<td>The FPC workflow creates a brain atlas from functional MRI data. It warps a set of 3-dimensional fMRI images to align them to reference image, averages the warped images, and then extracts 2-dimensional slices of the average image to yield the image atlas.</td>
</tr>
<tr>
<td><strong>Steam</strong></td>
<td>Steam</td>
<td>515</td>
<td>The Steam workflow characterizes the microbial communities represented in a sample from a natural environment by clustering and identifying DNA sequences of 16S ribosomal RNA amplified from the sample.</td>
</tr>
</tbody>
</table>

store the larger traces. Alternatively, the storage size of RC and RE grows linearly, and is comparable to NC, although RC and RE require slightly less storage space. Thus, storing the closure information for all nodes results in exponential growth in storage space for SE, whereas our reduction techniques reduce the same information such that RE and RC have linear growth in storage space and both require even less space than NC. Figure 4.16 shows effectiveness of our reduction techniques such that RE and RC reduce the overall storage size to less than 10% of NC (i.e., a 10-times reduction). Similar reductions are achieved for the real execution traces as shown in Figure 8.8(c).
4.9. Evaluation of Storage Strategies and Reduction Techniques 71

Figure 4.14: Provenance storage size for different patterns of synthetic traces.

Note that for strategies that collapse provenance annotations, less storage space is used than for their corresponding expanded strategies (e.g., NC versus NE, and RC versus RE). The relative ranking of the strategies in Figure 4.7 for storage size is RC < RE < NC < NE < SE.

Figure 4.15 shows the comparative reductions (to NE) achieved when different techniques are applied across each of the synthetic and real traces. Different techniques give minimal reduction depending upon the nature of the trace annotations. For example, for FC the subset reduction technique provides the greatest reduction, whereas in each of the other cases, the subsequence-subset technique provides the best
Figure 4.15: Comparison of reduction techniques.

Figure 4.16: Storage compression of RE and RC w.r.t. NC storage strategy on Mixed pattern.

reduction. Figure 4.15 also shows the additional space required to store pointer-based closures (NE+depc).

Update Time. Parsing trace files, expanding and collapsing annotations, and loading provenance records into the database comprise the update time for the NC, NE, and SE strategies. Update time for RE and RC consists of the time required to pre-process the trace, collapse or expand annotations, apply reduction techniques,
and load the resulting records into the provenance database. Figure 4.20 shows the
time taken to upload provenance records for the MIXED pattern traces. Similar to
storage size, because SE must upload provenance and closure records for each node,
its update time grows exponentially. NE also requires significant update time as the
size of the traces increase. The update times for RC and RE are linear in the MIXED
pattern and are less that that for NC (which does not store closures), demonstrating
that the costs of pre-processing and applying reductions is low.

Figure 4.19 shows the comparative time taken by various steps during update
4.9. Evaluation of Storage Strategies and Reduction Techniques

Figure 4.18: Comparison of update time steps for NE and RE storage strategies on Mixed pattern.

Figure 4.19: Update steps of RE storage strategy on Mixed pattern.

phase, when RE storage strategy is applied. All the steps grow linearly with the increasing trace annotations. As shown in Figure 4.18, the cost of applying our reduction algorithms grows linearly with increasing numbers of dependency annotations. While SE and NE do not incur the overhead associated with reduction, as shown in Figure 4.18 they exhibit exponential growth in update time over increasing numbers of annotations, as they have to insert a large number of records into tables. RE and RC have relatively low update times for the real execution traces as
4.9. Evaluation of Storage Strategies and Reduction Techniques

Figure 4.20: Provenance update time for the Mixed pattern.

shown in 8.8(a). Similar to storage size, the relative ranking of these strategies with respect to update time is $RC < RE < NC < NE < SE$ for the MIXED, TA, and TD patterns. For the DA pattern, dependencies grow exponentially resulting in RE and RC having a corresponding exponential increase in update time due to the cost of expanding annotations. For the DA pattern, the relative ranking of these strategies with respect to update time is $NC < RC < RE < NE < SE$.

**Query Time.** Because our reduction techniques partition the immediate and transitive dependency and invocation order tables into multiple tables for storing the reduction, we need to apply the views defined in Figures 4.11 and 4.12 to answer
Figure 4.21: Query response time for MIXED pattern traces.
4.9. Evaluation of Storage Strategies and Reduction Techniques

Provenance queries. To evaluate the cost of applying these views, we consider three distinct provenance queries. Given a specific node, we measure the query time to find: (1) the immediate dependencies of the node; (2) the transitive dependencies of the node; and (3) the sequence of invocations that contributed to the node being inserted. In our experience, (2) is the most widely used of these within provenance queries. Figure 4.21 shows the query response times for the MIXED pattern traces. Figure 4.21(a) shows that to find immediate dependencies, the RE strategy performs as good as the other methods (NE and SE). The overhead of applying the view in this case is the cost of joining the reduction tables. RC, however, takes significantly more time to find immediate dependencies, since the query must use the expand-annotation view which involves a large number of joins (Figure 4.13).

Figure 4.21(b) shows that for transitive dependencies, RE does as good as SE (again with only minor overhead of joins). Because NE does not store the transitive closure, a recursive stored procedure is used to answer the query. In RC, the expand-annotation view must be used for all nodes in its lineage path. NC will take even more time to answer this query than RC, since each transitive dependency node must be identified recursively and the expand-annotation view must be applied to all such nodes.

Figure 4.21(c) shows that determining the sequence of invocations that led to the insertion of a node takes less than .1 seconds for NE, SE, RE, and RC even for larger traces. The reason for this low query time is that the number of invocations in a trace typically is only on the order of at most 100s of invocations.

Figure 8.8(b) shows the total query time across these queries for the real execution traces. RE and SE both have almost the same query response time for these real traces. The relative ranking of these strategies with respect to query time is SE <
4.9. Evaluation of Storage Strategies and Reduction Techniques

RE < NE < RC.

**Analysis of Storage Strategies.** NC optimizes storage and update time, but requires recursive stored procedures to answer both immediate and transitive dependency queries, and thus exhibits the worst query response time. NE is better than NC for answering direct provenance annotation queries, but also requires recursive stored procedures to answer transitive dependency queries. In addition, as the number of provenance annotations increase, NE also leads to exponential growth in storage space and an increasingly high update time. SE optimizes query response time, but incurs exponential storage size and upload time, making it impractical except for small traces. RE optimizes all three metrics, i.e., storage space, update time, query response time. Applying the expand-reduction views for RE incurs only minor overhead for query response time. RC optimizes storage size and update time, but incurs significant overhead by having to apply both expand-reduction views and expand-annotation views. The following is a relative ranking of these strategies for storage size, update time, and query response time.

<table>
<thead>
<tr>
<th>metric</th>
<th>ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>RC &lt; RE &lt; NC &lt; NE &lt; SE</td>
</tr>
<tr>
<td>Update time</td>
<td>RC &lt; RE &lt; NC &lt; NE &lt; SE</td>
</tr>
<tr>
<td>Query time</td>
<td>SE &lt; RE &lt; NE &lt; RC &lt; NC</td>
</tr>
</tbody>
</table>

SE is generally impractical with respect to storage size and update time, and executing queries using RC is expensive and provides only minor storage savings over RE. In addition, RE reduces the exponentially growing storage size of storing provenance and dependency relations to linearly growing storage size by applying reduction techniques. Thus, we consider RE to be the best of these strategies when considering all
4.10 Related Work

With the increasing adoption of scientific workflow approaches and the need for book-
keeping and recording of large amounts of provenance information, techniques have
recently been proposed to efficiently store and query provenance metadata [CJR08,
HA08, DF08]. All these techniques identify important properties of provenance to
reduce the size of provenance store.

Specific reduction techniques (a family of factorization processes and two meth-
ods based on inheritance) are presented in [CJR08] to reduce the storage size of pro-
cess (i.e., invocation) lineage that contributed to data items. Unlike our approach,
these techniques assume a conventional provenance model and restrict process lin-
eage to simple sequences of invocations, while most other models use DAGs (e.g.,
[ABJF06, MFF+08, CBD06, BML+06] and others). In [CJR08], provenance is mainly
process dependencies, while in our unified provenance model, we consider both pro-
cess and data dependencies which are required to query lineage information about
data products. Also, algorithms in [CJR08] reduce only process dependencies, while
our algorithms reduce immediate and transitive closures of both data and invocations.
Further, in their provenance store each individual data is associated with process de-
pendencies, thus their provenance store is already exhibits a lot of redundancy be to
reduced. In our approach, each data item is associated with a pointer and only sin-
gle copy of process dependencies is maintained and our reduction techniques reduces
these unique set of process dependencies further. Similar to our approach, in [CJR08]
data is represented using an XML-based model and reduction techniques include re-
moval of duplicate invocation sets. In [CJR08], structural inheritance technique is
4.10. Related Work

used to store provenance for the highest ancestor node in a tree (this has also been
used in earlier approaches, e.g., [BMWL07, BCC06]). The provenance for descendant
nodes is accessed through a recursive approach. Our experimental evaluation, shows
that such query approach is highly expensive, and thus we store provenance pointer
for each individual items and use relational views to access provenance. In [CJR08],
the goal is to reduce the size of an existing provenance store, while our approach de-
fines explicit modeling constructs for minimizing provenance annotations, and allows
reduction techniques to be applied before the data is loaded into a database. This
provides opportunities to optimize both storage size and update time.

Unlike in [CJR08], but similar to our approach, [HA08] proposes reduction tech-
niques that are generically applicable to transitive, acyclic binary relations. These
techniques employ complex graph manipulations to reduce the number of duplicate
nodes of a closed dependency graph. Our approach of storing lineage information is
based on strategy of storing only closure dependency pointers, rather than storing all
dependency closure nodes, which reduces the size of lineage information significantly.
When compared with their algorithms, our reduction techniques on the examples in
[HA08] provide similar optimizations. For example, on the fMRI [MLA+08, HA08]
transitive dependency graph, our reduction techniques store only 60 total nodes,
whereas their approach stores 107 total nodes [HA08]. The reduction approaches
both store 22 total nodes for the primary example used in [HA08]. Thus, our experi-
mental evaluation shows that our reduction algorithms achieves greater reduction for
representing dependency closure, where the number of dependency nodes stored in
provenance store is comparable to that stored in [HA08]. Both [HA08] and our ap-
proach aim at enabling lineage queries using standard relational constructs, whereas
most others (e.g., [CJR08, BCC06, ABF06]) require recursive queries to reconstruct
4.11. Conclusion

We have proposed a framework to address open issues in the representation and storage of provenance information for scientific workflows. Specifically, we have described a formal provenance model that generalizes the conventional model used for representing data and process provenance from workflow runs, by supporting a wider range of workflow types and workflow systems. Our provenance model does not limit invocations to simple “black-box” transformers, but allows the workflow system to
4.11. Conclusion

declare and record data dependencies, based on an underlying update semantics. Our approach also simplifies provenance recording by not requiring implied provenance information to be given explicitly, while at the same time allowing detailed provenance to be recorded efficiently.

This work extends our prior work [BMWL07, BMR+08] by: (i) developing a formal description of our provenance model together with a set of first-order inference rules for expanding and collapsing traces; (ii) a set of strategies for efficiently storing and querying instances of the model; (iii) novel reduction techniques for minimizing redundancy in direct and transitive dependency representations; and (iv) a detailed experimental evaluation of our approach. Our reduction techniques can be applied efficiently and are applicable to both hierarchical and non-hierarchical data structures, and our results show that these reduction techniques can decrease provenance storage size, update time, and query-response time.
Chapter 5

A Query Language for Provenance (QLP)

While many approaches [MLA+08] have been proposed to record and represent provenance in scientific workflows, the problem of storing and querying provenance information has recently started getting prominence. The ability to query provenance not only allows users to explore and better understand results but also facilitates re-use. For example, workflow developers can verify that a workflow design executes properly by examining the inputs and outputs of actor invocations, or by ensuring that all input data is used to derive at least one corresponding output data product. Similarly, scientists can use provenance information to better understand how workflow results were derived by determining the data, actors, and invocation parameters that contributed to workflow data products. Provenance information can also be used to discover data and workflows, e.g., by enabling users to find data products, possibly across workflow runs, derived from certain input data (in case data was is revised or found to contain errors), or to find workflows used to produce data of a certain
5.1 Limitations of Existing Approaches to Query Provenance

Although many approaches [MLA+08] to store and query across “lineage” dimension of provenance information have been developed, few query languages have been specifically developed for querying provenance information. Most approaches (e.g., see [MLA+08]) define their provenance models in terms of physical schemas represented within general-purpose data management frameworks and their query mechanism is closely tied to the storage models used. This approach often leads to complex query expressions, even for answering basic provenance questions. For instance, most existing provenance systems use storage models where provenance information is stored either in a relational database, (RDBMS), XML views or in specialized semantic web languages (RDF, OWL). Such storage models require users to write queries in the corresponding languages, like SQL [BD08], Prolog [BML08], XQuery [MGM+08] and SPARQL [ZGST08, KDG+08, CFL+07]. In RDBMS approach, users need to be exposed to the different schemas of different provenance entities and to query over lineage information users must typically specify join operations over multiple tables and compute transitive closures [DF08, CJR08, HA08, ABML09a]. Such queries are very complex and difficult to write and extend. In XML approach, we need different query templates to implement different use cases. In order to define schema information in RDF approach, a lot statements are required through triple style.

In addition, execution of a workflow run forms a directed acyclic graph (DAGs
5.2. Desiderata of a Provenance Query Language

5.2 Desiderata of a Provenance Query Language

Our goal is to provide generic support for querying provenance information to enable a wide range of users and applications. Common types of provenance queries we want to support include standard lineage queries \cite{MLA08, BML06} for determining the data and invocations used to derive other data; queries that allow users to ensure that specific data and invocation dependencies were satisfied within a run; queries for determining the inputs and outputs of invocations (e.g., based on the actors used and their parameters); and queries for determining the structure of data produced by a workflow run. Further, to address shortcomings of current approaches that either do not provide explicit query support or else use query approaches not naturally suited for querying provenance, we also seek an approach that satisfies the following desiderata.

1. **Independent of Physical Storage Representations.** Many existing approaches for querying provenance information \cite{MLA08} are closely tied to
5.2. Desiderata of a Provenance Query Language

physical storage representations, e.g., relational, XML, or RDF schemas, where
users express provenance queries using corresponding manipulation languages,
i.e., SQL, XQuery, or SPARQL, respectively. Because most provenance queries
require transitive closures [CJR08, HA08], users must have considerable ex-
pertise in these underlying schemas and query languages. Some systems also
support multiple storage technologies (e.g., [ABJF06]) in which specific prove-
nance representations are selected by workflow designers. Thus, to query a
given workflow run, users must know which representation was used to store
the run and be proficient in the associated query language. A generic approach
for querying provenance information that allows multiple representations to be
used with the same fundamental query language, must be independent of the
underlying physical representation and storage system.

2. Independent of Specific Workflow Systems. Many different workflow sys-
tems record provenance information. Users should be able to express queries
without having to know which workflow system was used to produce prove-
nance information, and without having to use a different language for each
system. OPM [MFF+08], e.g., is independent of any particular workflow sys-
tem, whereas the Kepler provenance recorder in [ABJF06] requires users to
understand the details of Kepler workflow computation models to construct
basic lineage information.

3. Meaningful in the Absence of Workflow Definitions. It is often possible
to make use of provenance information gathered during a workflow run without
accessing or understanding the workflow definition. A generic query approach
should allow users to query provenance information without having prior knowl-
edge of the workflow definition, or the types of data input and output by the
workflow run. Similarly, a generic query approach should make it possible for users to discover the actors that were invoked during a workflow run, and the types of data used in a run. However, when data types and workflow definitions are known, users should be able to query provenance conveniently using this information.

4. **Provenance Relationship Preserving.** It is often convenient to visualize provenance information using data and invocation dependency graphs \[BMR+08, MLA+08, CJR08\], and these graphs are often constructed from the result of a provenance query. In many provenance approaches (including those based on path expressions \[HBM+08\]), typical queries return only sets of nodes or invocations, requiring additional queries to reconstruct the corresponding lineage graphs. Thus, a general approach for querying provenance information should make it simple for users and applications to construct such graphs by returning paths over lineage information.

5. **Incrementally Queryable.** A consequence of preserving provenance relationships within query results is that these results can be further queried. This can allow users to decompose complicated queries into smaller subqueries, and also incrementally refine query results. Thus, a generic query language should be *closed*, i.e., similar to relational algebra, where the result of a provenance query should be queryable using the same language.

6. **Transparencyly Optimizable.** Because the amount of provenance information produced by a workflow run can be large (e.g., due to the number of workflow steps, and input and output data sets), systems that manage provenance information must provide efficient storage and query approaches
5.3 Provenance Queries using QLP

[DF08, CJR08, HA08, ABML09a]. Thus, generic query approaches should be amenable to query optimization techniques, and these optimizations should be independent of the query language itself (i.e., users should not have to modify queries for efficiency).

The first two desiderata are addressed directly through our unified provenance model, which does not depend on any particular workflow system and can be implemented using different underlying provenance storage representations. The rest of this section describes techniques for querying our model, with the goal of helping address the remaining issues. We first describe how QLP can be used to query the different dimensions of our unified provenance model, and discuss how these dimensions can be combined through hybrid queries that mix lineage information with information about the structure of data and the versions of these structures produced by a workflow run. We also give examples of common provenance queries that can be expressed in QLP and discuss our current implementation.

5.3 Provenance Queries using QLP

Provenance model. As described in Section 3.6, our model of provenance subsumes conventional approaches for representing workflow provenance while supporting more advanced workflow computation models that permit multiple invocations of processes (e.g., for pipelining and loops), structured data, and update semantics [ABML09a]. Our approach naturally extends the conventional provenance model by adding support for nested data and for accurately capturing detailed lineage information of processes employing update semantics. Please see Section 3.6 and Section 3.7 for details on our provenance model.
5.3. Provenance Queries using QLP

Figure 5.1: (a) Example XML-based workflow implementing the fMRI image analysis of the first provenance challenge; (b) The trace showing the first invocations of each actor for a typical run; (c) The implied fine-grain data dependency graph for the data items in (b); and (d) The implied invocation dependency graph for the run, with the first invocations of each actor shown in red. This figure is same as Figure 3.4, and has been included here for easy reference in this Chapter.

Figure 5.1b shows a portion of the trace for a typical run of the fMRI workflow, which we use below in describing the different types of provenance queries supported in QLP.

QLP queries are posed against provenance traces using the constructs outlined in Table 5.1. In particular, different QLP constructs are used to query over distinct dimensions of the trace, representing: (i) lineage relations among nodes and invocations; (ii) flow relations among input and output data structures of invocations; and (iii) structural relations among nodes within and across data structures. These dimensions are independent in the sense that information in one dimension cannot be used in general to infer information of another dimension.
5.3. Provenance Queries using QLP

![Diagram of combined structure for Figure 5.1b.](image1)

![Diagram of fine-grained dependency graph for Figure 5.1b.](image2)

**5.3.1 Queries over Lineage Relations**

The majority of provenance queries are expressed over lineage relations (e.g., see [MLA+08, BML+06]). Figure 5.3 is an example of a portion of the lineage relations for the trace of Figure 5.1b, showing only the fine-grained dependencies of data nodes (i.e., collection nodes are not shown). The QLP operators for querying lineage (see Table 5.1) act as filters over lineage relations $L$. QLP lineage queries are closed under lineage relations such that a lineage query takes as input a set of lineage relations (edges) $L$ and returns a subset of these lineage edges as output. A lineage edge in $L$ is of the form $(n_1,i,n_2)$ stating that a node $n_1$ was used to derive a node $n_2$ via invocation $i$. In this way $L$ denotes the “derived” relation such that each edge in $L^{-1}$ is a “dependency”. We draw lineage edges as “dependencies” (see Figure 5.1) in which edges are labeled by their corresponding invocations. For example, consider
Table 5.1: Summary of QLP constructs and corresponding descriptive notations

<table>
<thead>
<tr>
<th>Construct</th>
<th>Descriptive Form</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node and invocation expressions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n, x, *$</td>
<td>$n, x, *$</td>
<td>single node $n$, XPath expression $x$, or set of trace nodes *</td>
</tr>
<tr>
<td>$#i, #a$</td>
<td>$i, a$</td>
<td>Invocation $i$ or actor $a$ (denoting the set of $a$ invocations)</td>
</tr>
<tr>
<td>$n #i \in #i$</td>
<td>$n #i \in i$</td>
<td>node $n$ input to invocation $#i$. If $i$ is not given, then the node $n$ input to the workflow run.</td>
</tr>
<tr>
<td>$n #i \out #i$</td>
<td>$n #i \out i$</td>
<td>node $n$ output by invocation $i$. If $i$ is not given, then the nodes $n$ output by the workflow run.</td>
</tr>
<tr>
<td><strong>Lineage-preserving queries (examples)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$*..n$</td>
<td>$* \text{ derived } n$</td>
<td>The lineage edges for node $n$.</td>
</tr>
<tr>
<td>$n..*$</td>
<td>$n \text{ derived } *$</td>
<td>The lineage edges for all nodes derived from node $n$.</td>
</tr>
<tr>
<td>$n_1..n_2$</td>
<td>$n_1 \text{ derived } n_2$</td>
<td>The lineage edges containing all edges from node $n_1$ to node $n_2$.</td>
</tr>
<tr>
<td>$n_1..#i..n_2$</td>
<td>$n_1 \text{ through } i \text{ derived } n_2$</td>
<td>The lineage edges containing all edges from node $n_1$ to node $n_2$ that pass through an invocation $i$.</td>
</tr>
<tr>
<td><strong>Functions over lineage edges</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exists($L$)</td>
<td>exists($L$)</td>
<td>True if the trace contains a path defined by $L$.</td>
</tr>
<tr>
<td>invocations($p$)</td>
<td>invocations($p$)</td>
<td>The invocations of the lineage edges $L$.</td>
</tr>
<tr>
<td>actors($L$)</td>
<td>actors($L$)</td>
<td>The actors of invocations of the lineage edges $L$.</td>
</tr>
<tr>
<td>nodes($L$)</td>
<td>nodes($L$)</td>
<td>The nodes of the lineage edges $L$.</td>
</tr>
<tr>
<td>input($L$)</td>
<td>input($L$)</td>
<td>The source nodes of the lineage edges $L$.</td>
</tr>
<tr>
<td>output($L$)</td>
<td>output($L$)</td>
<td>The sink nodes of the lineage edges $L$.</td>
</tr>
</tbody>
</table>

A lineage edge $\langle 2, \text{AlignWarp}:1, 11 \rangle$ of Figure 5.1b stating that node 2 was used by the first invocation of the AlignWarp actor to produce node 11 (i.e., node 11 depends on node 2). Sets of lineage relations define lineage graphs, and QLP lineage queries act as filters over these lineage relations. For example, consider the following QLP lineage queries over the trace of Figure 5.1b.
5.3. Provenance Queries using QLP

\* derived 19  \hspace{1cm} (1)

6 derived * \hspace{1cm} (2)

\* through Slicer:1 derived *

These queries return (1) lineage relations denoting the set of paths starting from any node and ending at node 19, (2) lineage relations denoting the set of paths starting at node 6 and ending at any node, and (3) lineage relations denoting the set of paths that go through the first invocation of the Slicer actor. QLP supports both a descriptive form (used above) and a shorthand notation. Using the shorthand notation, the above queries can be equivalently expressed as: \*..19, 6..*, and \*..#Slicer:1..* (or simply \*..#Slicer:1 or #Slicer:1..*). Nodes and invocations form steps within lineage queries (generalized to XPath expressions and actor names below), and lineage queries consist of two or more steps.

Queries 1-3 select lineage paths of length one or greater (referred to as transitive paths). Although not shown in Table 5.1, QLP also supports queries that select lineage paths having length one (i.e., immediate paths) using the . operator, or in the descriptive form, using the operators 1\_derived (for nodes) and 1\_through (for invocations). Immediate and transitive path operators can be combined in QLP. For example, the following query returns the lineage relations defining dependency paths that start at an input node of the first invocation of the AlignWarp actor.

\* 1\_through AlignWarp:1 derived *

Finally, lineage queries can be chained together, e.g., the following queries

2 derived 12 derived 15 \hspace{1cm} (5)

2 through Reslice:1 through Slicer:1 1\_derived *

return (5) lineage relations denoting paths that start at node 2 and end at node 15 while passing through node 12, and (6) lineage relations that start at node 2 and
end at an output node of the first invocation of Slicer while passing through the first invocation of Reslice.

Given a set of lineage relations $L$, let $\text{paths}(L)$ be the set of paths implied by $L$, where a lineage path takes the form

$$n_1.i_1.n_2 \ldots i_j \ldots n_{k-1}.i_{k-1}.n_k$$

$k \geq 2$

For any set $P$ of such paths, let $\text{edges}(P)$ be the set of lineage relations for $P$, such that $L = \text{edges}(\text{paths}(L))$. Given lineage relations $L$, the query ‘$n_1 \text{ derived } n_k$’ returns the subset of $L$ consisting of paths of any length starting at $n_1$ and ending at $n_k$. Similarly, the query ‘$n_1 \text{ through } i_j \text{ derived } n_k$’ returns the subset of $L$ consisting of paths of any length starting at $n_1$ and ending at $n_k$ that also pass through invocation $i_j$. If $\text{1-derived}$ (i.e., one-step or immediately derived) is used in place of derived, then only paths of the form $n_1.i_j.n_k$ are returned. That is, in $\text{1-derived}$ each selected path is defined by a single lineage relation. These operators can be combined to form more complex lineage paths. In this case, each individual expression specifies a segment of a larger lineage path. For instance, a query such as ‘$n_1 \text{ derived } n_j \text{ derived } n_k$’ selects lineage relations that form paths from $n_1$ to $n_j$ and from $n_j$ to $n_k$.

Lineage queries may also be expressed using a short-hand notation (see Table 5.1). Similar to the use of ‘/’ and ‘//’ in XPath expressions for navigating parent-child relationships of nodes, we use ‘.’ to navigate immediate (i.e., one-step) lineage paths and ‘..’ to navigate transitive lineage paths (i.e., paths consisting of one or more steps). When using the short-hand notation, we distinguish actor invocations from structural nodes by prefixing invocations with ‘#’. For example, queries (7-9) can be expressed as ‘*..19’, ‘6..*’, and ‘*..#slicer:1..*’ (or simply ‘#slicer:1’) using the QLP short-hand notation.
5.3. Provenance Queries using QLP

5.3.2 Queries over Flow Relations

QLP also provides constructs for accessing specific versions of structures produced by a workflow run. As a node in $N$ may occur within multiple versions, we denote an occurrence of a node $n \in N$ within a structure $d$ as $n@d$. The expression $n@d$ positions node $n$ at structure $d$ (i.e., $n$ is positioned at $d$), whereas the expression $n$ leaves the node unpositioned. The following QLP queries use the @in and @out constructs (see Table 5.1) to access the different versions of structures within a trace (according to the flow relations). For example, the following queries

@in
@out
@out Slicer:1
18 @out Slicer:1

return (7) the input structure $d_1$ of the run, (8) the output structure $d_6$ of the run, (9) the version of the output structure $d_5$ produced by the first invocation of the Slicer actor, and (10) the version of node 18 in the structure $d_5$ produced by the first invocation of Slicer.

Unlike querying combined structures, the @in and @out operators return sets of positioned nodes. For example, the set of nodes returned by query (7) when applied to the example in Figure 5.1b would include 1@d_1, i.e., node 1 positioned at $d_1$ (in addition to all of the descendents of node 1 in $d_1$). The @in and @out operators return the nodes of structures input to and output by the run when no invocations are given, and when invocations are given, the input and output structures of the invocation, respectively. An (unpositioned) node may also be supplied as an argument to these operators, in which case the set of nodes returned contains only a positioned version of the node (i.e., the node without its descendents), or the empty set if no
such node exists in the corresponding structures. For example, query (10) applied to
the run graph of Figure 5.1b returns the node $18@d_5$.

Considering some more example for queries over flow relations, the following
queries

\begin{align}
* \text{in derived} & \ 19 \\
6 \text{ derived} & \ * \text{out} \\
* \text{in Slicer:1 derived} & \ *
\text{(11) (12) (13) (14) (15) \ 
}\end{align}

return (11) lineage relations denoting paths that start at a node in the input data
structure provided to the workflow run and end at node 19, (12) lineage relations
denoting paths that start at node 6 and end at a node in an output structure of the
workflow run, (13) lineage relations denoting paths that start at a node in an input
structure of the first invocation of Slicer, (14) lineage relations denoting paths that
start at the occurrence of node 15 in the input of the first invocation of Slicer, and
(15) lineage relations denoting paths that start at the occurrence of node 15 in the
output of the first invocation of Slicer. Note that queries 1 and 11 (and similarly,
queries 2 and 12) are not equivalent for all traces. In particular, query 11 only
returns lineage graphs with paths starting at an input node of the workflow run,
whereas query 1 may contain paths starting at an intermediate node (e.g., a node
created by an invocation, but without data dependencies). Also note that for the
trace of Figure 5.1b, query 15 returns an empty set of lineage relations since node 15
is not a dependency of another node after the first invocation of Slicer.
5.3.3 Queries over Structural Relations

Because data structures in our provenance model are represented using XML, these structures can be queried using standard XML languages. In QLP, we adopt XPath as our query language for accessing nested data. For each trace, we also define a simple view that merges the different versions of structures within the trace into a combined data structure. As an example, Figure 5.2 shows the combined data structure $D$ for the flow graph associated with Figure 5.1b that consists of the structures $d_1$ to $d_6$. Queries over combined structures provide general access to all the nodes used and produced by a run, e.g., to return nodes of a specific type or with specific metadata annotations. The following XPath expressions are valid QLP queries, which are posed against the combined data structure. Thus, in addition to specific node identifiers, QLP queries can also contain XPath expressions for accessing nodes based on their type (i.e., tag name) and parent-child relationships. For example, consider the following queries.

\[
\star \text{ derived } //\text{AtlasImage}/*/ \quad (16)
\]

\[
/\text{AnatomyImage[@modality=\text{"speech"}]}/*/ \text{ derived } \star \quad (17)
\]

These queries return (16) lineage relations denoting paths that end at a descendent node of an AtlasImage collection, and (17) lineage relations denoting paths that start at a descendent node of an AnatomyImage collection having the value “speech” for the modality metadata attribute. QLP also allows actor names to be used in place of specific invocations. In this case, the actor name denotes the set of invocations of the actor within the trace, returning lineage relations denoting paths that pass through one of the actor’s invocations. Similar to attributes in XPath, we also allow invocation expressions to be selected based on their parameters, e.g., the expression ‘Slicer[@x=\text{“0.5”}]’ selects invocations of Slicer in which the parameter x is set to
Given a trace \( T = (V, E, L, \alpha, \tau, \gamma) \), (see Section 3.6 for details), a combined structure \( D = d_1 \cup \ldots \cup d_n \) is defined as the union of structures \( d_i \in V \). XPath queries expressed over combined structures \( d \) return ordered subsets of nodes within the tree \( \tau(d) \). We note that the combined structure of a trace is distinct from the final versions of structures, and instead, contain all data input to and produced by a run (including deleted nodes).

### 5.3.4 Queries over Each Dimension

Each of the above dimensions can be combined into a single query. For example, the following query returns the set of lineage relations denoting paths that end at a descendent node of an AtlasImage collection output by a Slicer invocation.

\[ \text{* derived} \ //\text{AtlasImage}/** \ @\text{out} \ \text{Slicer} \]

Combined queries can be (naively) evaluated by (i) obtaining the structures resulting from \@in\ and \@out\ version operators, (ii) applying XPath expressions to these structures, and (iii) applying lineage queries to the resulting nodes. For example, when applied to the portion of the trace shown in Figure 5.1b, query 18 is evaluated by: (i) obtaining the output structure of the Slicer invocation; (ii) executing the XPath query ‘//AtlasImage/**’ over the structure obtained in (i), returning node 18; and (iii) issuing a separate lineage query for node, i.e., ‘* derived 16’, ‘* derived 17’, ‘* derived 18’, and ‘* derived 19’, where the answer contains the unique set of resulting lineage relations.

Also, consider the following query that returns the lineage relations denoting paths through Softmean, Slicer, and Convert invocations, and ending at nodes of type AtlasGraphic.
5.3. Provenance Queries using QLP

What is lineage of all sub-elements of AtlasImage collection produced by slicer?

Figure 5.4: Pictorial representation of how a query that spans in each dimension is evaluated.

* through Softmean derived * through Slicer derived * through Convert derived //AtlasGraphic

(19)

For queries that specify complex lineage paths, it is often more convenient to use the short-hand notation, e.g., query (19) can equivalently be written as

#Softmean .. #Slicer .. #Convert .. //AtlasGraphic (short-hand version of 19)

Consider Figure 5.4 that illustrates how the queries that span in multiple dimensions are evaluated.

5.3.5 Additional Functions over Lineage Relations

QLP also supports additional operations that can be applied to sets of lineage relations, which are summarized in Table 5.1 and used in the following examples.
5.3. Provenance Queries using QLP

invocations(AlignWarp[m="12", dateOfExecution=“Monday"])) (20)

output//Header[max=”4096” derived //AtlasGraphic) (21)

output(#AlignWarp[m=“12"] .. #Softmean) (22)

input//Image derived //AtlasGraphic @out) (23)

actors//Image derived //AtlasGraphic @out) (24)

(//Image @in) – input//Image @in derived //AtlasGraphic @out) (25)

Query (20) returns the set of invocations of AlignWarp that used the values 12 and “Monday” for the “m” and “dateOfExecution” parameters, respectively. Query (21) returns the output nodes for lineage paths starting at Header nodes having the value “4096” for the global maximum metadata field “max” and ending at AtlasGraphic nodes. Specifically, the output operation (similarly, input) returns the nodes within lineage relations that do not have outgoing (incoming) lineage edges. Query (22) returns the output nodes produced by invocations of SoftMean that were derived from AlignWarp invocations using the value 12 for parameter “m”. This query combines multiple lineage operators and is expressed using the QLP shorthand notation. Query (23) returns the input nodes of paths starting from Image nodes and ending at AtlasGraphic nodes that were part of the output of the run. Query (24) is similar to (23) but returns the actors of invocations used in the corresponding lineage paths. Query (25) finds the Image nodes input to the workflow run that were not used to derive any output Image nodes. This is achieved by first finding all Image nodes input to the workflow run, and then subtracting from this set the input Image nodes used to derive an output AtlasGraphic node.

QLP also provides support for constraining the structure of lineage paths using regular expressions. For instance, the query ‘n_1 through (i_1 | i_2) derived n_2’ selects lineage relations denoting paths from n_1 to n_2 that pass through either i_1 or i_2. These
queries can be used by workflow developers, e.g., to ensure that complex workflows (e.g., involving multiple branches) executed correctly.

To summarize, Table 5.1 gives a number of additional functions that can be applied to the results of QLP lineage queries. The \texttt{exists} function determines whether a trace contains a given path; \texttt{invocations} and \texttt{actors} return for a set of lineage relations the invocations and actors, respectively; \texttt{nodes} returns the nodes contained in a set of lineage relations; and \texttt{input} and \texttt{output} return the source and sink nodes in a set of lineage relations, respectively.

\textbf{Storage requirements}. In order to answer lineage queries, it is required that we either store or represent all paths between any two nodes like triples of form \((n_x, i, n_y)\), so that query results between any two nodes or invocations can be returned as a set of such triples. In a naive approach, we would store all paths between any two nodes, which will require a huge storage requirements. We argue that it is sufficient enough to store only the immediate and transitive data dependencies of nodes to material such triples of form \((n_x, i, n_y)\) through some simple join operations over relational tables. As demonstrated in [ABML09a] storing transitive dependencies leads to exponential growth of provenance store. Thus, we need efficient storage techniques to reduce the provenance store size by applying effective reduction algorithms over transitive dependencies [ABML09a, HA08].

5.4 QLP Implementation and Evaluation

Our current implementation of QLP supports queries expressed using the shorthand notation described in Table 5.1, and answers these queries using the relational storage strategies described in [ABML09a], which store both immediate and transitive
dependencies in a reduced form. In particular, we implemented a Java application that takes as input a QLP query, transforms the query into an equivalent SQL query, and executes the SQL query over the underlying database.

Using our implementation, we were able to express and answer the queries of the first provenance challenge [MLA+08] as well as queries similar to those in [BML+06]. A number of variants of these queries are also given in (1–20) of Chapter 4. The QLP versions of these queries are significantly more concise than those expressed against the underlying storage structures [MLA+08], and in general, are easier to formulate and comprehend.

To evaluate the feasibility and scalability of our approach for executing QLP queries, we describe below the results of executing lineage queries over traces of increasing numbers of nodes and lineage relations. We also compare our reduced-transitive approach (R), which transforms QLP queries to SQL queries to our underlying relational schema that stores immediate and transitive lineage relations in reduced form, to a naive approach (N) in which only immediate dependencies are stored, and corresponding QLP lineage queries are transformed to recursive stored procedures.

Our experiments were performed using a 2.4GHz Intel Core 2 duo PC with 2 GB RAM and 120 GB of disk space. Each approach used MySQL to store provenance information. We compare query response time and storage size using a set of synthetic traces ranging from 100 to 3000 nodes, $10^3$ to $10^4$ immediate dependencies, $10^4$ to $10^6$ transitive dependencies, and lineage paths of length 25 to 150, respectively.

We consider the following basic lineage queries for evaluating query response time. In general, queries over lineage relations are considerably more expensive [HA08, ABML09a] than queries over only combined structures (i.e., XPath queries),
or queries that select only versions of structures within a trace (e.g., using the \texttt{@in} and \texttt{@out} QLP operators).

\begin{align*}
* & .. n \\
n & .. * \\
\textbf{exists} & n_1 .. n_2 \\
n_1 & .. n_2 \\
n_1 & .. n_2 .. n_3
\end{align*}

These queries return (Q1) lineage relations denoting paths that lead to a node \( n \) (e.g., to return the full lineage of \( n \)); (Q2) lineage relations denoting paths that start from a node \( n \) (i.e., the “progeny” of \( n \) \cite{BML06}); (Q3) true if there is a lineage path from node \( n_1 \) to node \( n_2 \) (where ‘\textbf{exists}’ denotes a boolean query); (Q4) lineage relations denoting paths starting at node \( n_1 \) and ending at node \( n_2 \); and (Q5) lineage relations denoting paths starting at node \( n_1 \) and ending at node \( n_3 \) that pass through node \( n_2 \).

The Figure 5.5 demonstrates that the query response time for the reduced-transitive approach grows linearly with increasing trace size, and is significantly faster than the naive approach. The increased query response time for the reduced-transitive
Figure 5.6: The storage size for reduced-transitive, naive, and simple-transitive for Figure 5.5

approach, when compared to the naive approach, is not based on increases in storage size. In particular, Figure 5.6 shows that when compared to the naive approach, storage size is also significantly reduced using our reduction strategies (for details see [ABML09a]). Thus, we can improve query response time by materializing transitive lineage relations, while at the same time reducing the provenance storage size using our reduction techniques. These results demonstrate that the standard approach of using recursive stored procedures to answer lineage queries does not scale linearly with trace size.

Our evaluation also demonstrates the advantages of decoupling provenance query languages from underlying representation schemes. In particular, the same QLP queries can be answered using different storage strategies, in this case using the naive and reduced-transitive approaches, to transparently improve query execution time and provenance storage size.

It is required that provenance information be stored, so that it can be queried. In the next chapters 5 and 6, we describe a set of storage strategies and reduction techniques to store the provenance information efficiently.
5.5 Related Work

Although many approaches [MLA+08] to store and query across "lineage" dimension of provenance information have been developed, their query mechanism is closely tied to the storage models used. Most existing provenance systems use storage models where provenance information is stored either in a relational database, (RDBMS), XML views or in specialized semantic web languages (RDF, OWL). Such storage models require users to write queries in the corresponding languages, like SQL [BD08], Prolog [BML08], XQuery [MGM+08] and SPARQL [ZGST08, KDG+08, CFL+07].

In RDBMS approach, users need to be exposed to the different schemas of different provenance entities, and the queries often span across multiple tables (i.e., joins) and transitive relationships. Such queries are very complex and difficult to write and extend. In XML approach, we need different query templates to implement different use cases. In order to define schema information in RDF approach, a lot statements are required through triple style. Our approach differs by separating the logical provenance model from its physical representation and providing provenance-specific query constructs. Thus, users express queries against the logical model using more natural constructs, which are automatically translated to equivalent queries expressed against the underlying physical representation.

In addition, execution of a workflow run forms a directed acyclic graph (DAGs e.g., [ABJF06, MFF+08, CBD06, BML+06]) and many provenance queries are based on paths through such DAG like structure. For example, seven out of nine queries in First Provenance Challenge [MLA+08] were based on such DAGs. Also standard query languages are not natural for querying provenance information efficiently, as these are not suited to handle paths through DAGs. RDBMS languages do not support querying for DAG, while XML query languages support querying only over
a tree structure. Though we can query a graph structure through SPARQL (where data is stored in RDF), it does not have essential features to query over paths (it lacks support for sub-queries and expressions in select clauses).

Although based on the conventional provenance model, the VisTrails provenance query language (vtPQL) [Car08] uses a layered approach to answer provenance queries. These layers refer to versions, workflows and executions. vtPQL has identified basic operations that simplify query syntax and queries are answered through API. For example, \textit{upstream}(x) returns all modules that proceed \(x\) in the workflow execution. This construct is used to answer lineage queries, e.g., to find dependencies between two invocations of modules \(x\) and \(y\) within a run, by using ”minus” operator, like \textit{upstream}(y) – \textit{upstream}(x), that returns a set of modules that precede between \(y\) and \(x\). Such construct works only when there is only one path to reach \(y\), but when there are more than one path to reach \(y\) and that lie outside the path between \(x\) and \(y\), then this approach returns incorrect query result. In our approach, we use construct \(x..y\) to answer such queries, and our query re-writing technique uses fundamental QLP constructs, like ”\((x..*) \cap (\ast..y)\)”. Conceptually, ”\(\ast..y\)” construct in QLP is similar to vtQPL \textit{upstream}(x), and we also overcome the limitation of \textit{upstream}(x), with additional construct \(x..\ast\), which returns all modules that follow \(x\). In addition, the result of a vtPQL query is workflow identifiers and attributes that is displayed visually. We have observed that visual display of query result is good when the execution graph has a few nodes and links between them, but if the execution graph has a larger number of nodes or links between them, then the visual representation gives a ”messy” view, making the visual display difficult to follow. This is one of the limitation of visually displaying graph like structures for larger traces as we have noticed for some of the pPOD [BMR+08] traces, when viewed
through our provenance browser [BMR+08], a tool to visually display different views of provenance information.

FLOQ [ZL08] describes a logic programming approach to representing and querying workflow provenance. FLOQ approach suffers from scalability issue of querying large-scale provenance information, as the entire provenance information might not fit into main main memory. FLOQ will have to rely on techniques to externally store provenance information in databases and load them on-demand, which would affect the query execution time. Also the result of a FLOQ query is a set of identifiers (like a set of nodes, or a set of invocations), while result of a QLP over dependency graphs is a set of edges, which is a more natural representation for lineage query results.

In [HBM+08], Lorel [AQM+97] is used to represent and query provenance information over a semi-structured data model (PASS [MRHBS06]). Similar to our approach, [HBM+08] employs generalized path expressions for querying lineage information. [HBM+08] has a new implementation of Lorel, as per Lorel specification, and extends it in several ways to enable its implementation to answer provenance queries. Thus, their approach is tied to a specific database model, Lore, and thus is not flexible enough to meet our requirements of not exposing the schema.

In [HA08] lineage information is stored such that only queries of the form *..n can be answered, while in our QLP approach, we support queries of various forms (e.g., *..n, n..*, n1..n2, exists n1..n2, n1..n2..n3, etc.). Our query approach is far more expressive than that of [HA08].

While both vtPQL and our approach provide a closed language over lineage edges, e.g., to support the visual representation of provenance lineage graphs, languages based on path expressions (Lorel, XPath, OQL, etc.) primarily return sets of identifiers (e.g., representing nodes and invocations) that require additional queries for
constructing lineage graphs. Further, approaches such as [HBM+08] are still closely tied to physical representations of provenance.

The recent Open Provenance Model [MFF+08] is an effort to explore fundamental relationships in provenance systems, such as the transitive relationship between data products and process invocations. OPM is currently not tied to any specific query language implementation, and we think that adopting our QLP desiderata is a natural step to explore.

5.6 Conclusion

In this Chapter, we describe a general approach for querying provenance using our Query Language for Provenance (QLP), which provides specialized constructs for expressing both structural (i.e., parent-child) and lineage (i.e., data dependency) “hybrid” queries. We also show how these constructs can be used to express a wide range of provenance queries (including those of [MLA+08]) and that answering QLP queries using relational technology can be feasible when both immediate and transitive dependency edges are stored according to [ABML09a].

In Chapter 6, we extend our current implementation of QLP with additional optimization techniques, with the goal of providing a generic and efficient approach for addressing challenges in managing the provenance of scientific workflow runs.
Chapter 6

Efficiently Querying Provenance Graphs

6.1 Querying Provenance Graphs

Scientific results are often based on complex data analysis pipelines that integrate multiple domain-specific applications [DC08]. Automating the use of these applications is frequently performed using traditional scripting languages, and more recently, through scientific workflow systems (e.g., [LAB+06, Tom06, CFS+06]). An advantage of scientific workflow systems over traditional approaches is their ability to automatically record the provenance of intermediate and final data products generated during workflow execution. This provenance information generally consists of data and process dependencies introduced during a workflow run, and is crucial for enabling scientists to more easily understand, reproduce, and verify scientific results [DF08].

Scientific workflow provenance is typically represented using data and process
dependency (i.e., causal graphs) [MFF+08]. The ability to effectively store and query large numbers of dependency graphs remains a significant challenge in managing provenance information [DF08], since provenance graphs are typically heterogeneous (i.e., they do not share a common “schema”) and each such graph may require considerable storage space. In particular, provenance information may be many times larger than the workflow and input data itself [CJR08], workflows are often executed multiple times over different data sets and parameter settings, and most scientific research projects consist of many distinct workflows [BBD+09, ABML09a]. To help address these challenges, a number of approaches have recently been proposed for efficiently representing and storing provenance dependency graphs [CJR08, HA08, ABML09a, BD08].

Approaches for querying stored provenance information, however, are largely still
based on underlying physical data representations [DF08] (e.g., relational, XML, or RDF schemas), where users express provenance queries through corresponding query languages (i.e., SQL, XQuery, or SPARQL). Typical provenance queries (e.g., see [MLA+08]) are exploratory and involve finding some or all of the data and process dependencies that led to the creation of one or more data products. Posed over dependency graphs, these queries require the computation of transitive closures as well as applying selection conditions on corresponding lineage paths. For most users, expressing these queries against the physical schemas used to represent provenance information requires considerable expertise and is cumbersome even for simple queries [DF08]. The complexity of these queries also presents a number of challenges for efficient query evaluation [HS08].

In this chapter, we present a high-level query language for provenance (QLP; pronounced “clip”) and novel evaluation techniques to address these challenges. Our query language provides basic constructs that are tailored to querying provenance information, which can help make common provenance queries easier to express for non-experts. In addition, unlike other standard languages that return sets of nodes (e.g., [Car08, HBM+08, ZL08, CFL+07, BBDH08]), QLP is closed under lineage relationships. That is, answers to lineage queries are sets of lineage dependencies (edges) forming provenance subgraphs. This approach simplifies the expression of complex queries, better support provenance view definitions and graph visualization [DF08, BMR+08], and leads to query optimizations described in this chapter.

In addition, QLP employs a general model of provenance that can support a wide range of scientific workflow systems. Most existing approaches for representing provenance do not consider workflow computation models that work over structured data, including XML. These standard provenance representation schemes (e.g.,
[MFF+08, SPGM06b, HBM+08, BD08, BBDH08] among others) largely assume that workflow models are based on *transformation semantics* in which each workflow step consumes all input data and produces entirely new output data. Alternatively, workflow models that work over structured data (e.g., [BMR+08, MBZG08, Slo07, Tom06]) often employ *update semantics*, where only a portion of an incoming XML stream is modified by each workflow step. This in turn can lead to a number of benefits for workflow design and reuse [ABML09b]. However, standard provenance approaches often cannot represent dependency relationships correctly or efficiently for XML-structured data [ABML09a, MBZG08]. The model of provenance used by QLP extends conventional approaches by explicitly supporting workflow steps that process XML data and employ update semantics [ABML09b], and includes query constructs for accessing both the structure and lineage of data items. In addition, QLP allows these constructs to be used together within a single query, which can further simplify the expression of complex provenance questions.

The following section provides a more detailed and formal treatment of the constructs presented here, and QLP evaluation techniques are presented in Section 6.3.

### 6.2 QLP Syntax, and Semantics

**Provenance model.** As described in Section 3.6, our model of provenance subsumes conventional approaches for representing workflow provenance while supporting more advanced workflow computation models that permit multiple invocations of processes (e.g., for pipelining and loops), structured data, and update semantics [ABML09a]. Our approach naturally extends the conventional provenance model by adding support for nested data and for accurately capturing detailed lineage information of
processes employing update semantics. Please see Section 3.6 and Section 3.7 for details on our provenance model.

We consider the following QLP provenance model. A workflow trace \( T = \langle V, F, L \rangle \) consists of:

(i) a set of vertices \( V = D \cup I \), where \( D \) is a set of XML data structures defined over nodes \( N \), and \( I \) is a set of invocations over actors \( A \);

(ii) a set of flow edges \( F = F_{in} \cup F_{out} \), where \( F_{in} \subseteq D \times I \) and \( F_{out} \subseteq I \times D \) denote invocation inputs and outputs, respectively; and

(iii) a set of lineage edges \( L \subseteq N \times I \times N \) denoting an acyclic digraph, where \( \langle n_1, i, n_2 \rangle \in L \) states that invocation \( i \) used \( n_1 \) to create \( n_2 \).

An XML data structure \( d \in D \) consists of nodes \( n \in N \) such that \( \text{nodes}(d) \) are the nodes of \( d \). We write \( A(i) \) to denote the actor associated with an invocation \( i \in I \) and \( I(a) \) to denote the set of invocations associated with an actor \( a \in A \). The input and output structures of a trace \( T \) are defined as:

\[
\text{in}(T) = \{ d \in D \mid \neg \exists i : \langle i, d \rangle \in F_{out} \} \\
\text{out}(T) = \{ d \in D \mid \neg \exists i : \langle d, i \rangle \in F_{in} \}.
\]

### 6.2.1 QLP Syntax

**Querying Provenance with QLP.** Our provenance model supports a high-level query language for provenance (QLP) [ABML09b], that allows users to easily express complex provenance queries. QLP queries can be posed against flow graphs through a number of different QLP constructs, some of which are described below. These
constructs are used to query distinct dimensions of the flow graph representing: (i) dependency paths over nodes and invocations; (ii) flow relations among input and output structures of invocations; and (iii) structural relations among nodes within and across data structures. For a detailed description of QLP, please refer to Section 5.3.

QLP (shorthand) path expressions $p$ are built from the following grammar.

\[
p ::= s . s | s .. s | s . p | s .. p \\
s ::= e_n | e_n q | e_i \\
e_n ::= n | x | * \\
e_i ::= #a | #i \\
q ::= @in | @out | @in e_i | @out e_i
\]

A path $p$ is composed of two or more steps $s$. A step is either a node expression $e_n$ or an invocation expression $e_i$. A node expression can be either an XPath $x$, a wildcard $*$, or a single node identifier $n$. An invocation expression specifies either an actor $#a$ denoting all invocations of $a$, or an invocation identifier $#i$. As a convention, invocations are written $a:1$, denoting in this case the first invocation of actor $a$. Node expressions are optionally qualified with either an $@in$ or $@out$ operator. A simple path consists of exactly two steps in which $s_1 . s_2$ is referred to as an immediate simple path and $s_1 .. s_2$ a transitive simple path.

### 6.2.2 QLP Semantics

Let $T = (V, F, L)$ be a trace defined over nodes $N$, invocations $I$, and actors $A$. Each individual step $s$ is defined as a function that maps traces $T$ to sets of vertices such that $s(T)$ returns a subset of $V$, as shown in Figure 6.2. For XPath expressions $x$
Table 6.1: Semantics of QLP constructs

<table>
<thead>
<tr>
<th>Construct</th>
<th>Descriptive Form</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node and invocation expressions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( n, x, * )</td>
<td>( n, x, * )</td>
<td>Node expressions ( e_n ) as a single node ( n ), XPath expression ( x ), or set of trace nodes ( * ).</td>
</tr>
<tr>
<td>(#i, #a)</td>
<td>( i, a )</td>
<td>Invocation expressions ( e_i ) as an invocation ( i ) or actor ( a ) (denoting the set of ( a ) invocations).</td>
</tr>
<tr>
<td>( e_n @in e_i )</td>
<td>( e_n @in e_i )</td>
<td>Nodes of ( e_n ) input to invocations of ( e_i ). If ( e_i ) is not given, then the nodes of ( e_n ) input to the workflow run.</td>
</tr>
<tr>
<td>( e_n @out e_i )</td>
<td>( e_n @out e_i )</td>
<td>Nodes of ( e_n ) output by invocations of ( e_i ). If ( e_i ) is not given, then the nodes of ( e_n ) output by the workflow run.</td>
</tr>
</tbody>
</table>

**Lineage-preserving path queries (examples)**

| \(*..e_n\)                      | \(* \) derived \( e_n \) | The lineage graph for nodes in \( e_n \).                                                                                           |
| \(e_n..\)                      | \( e_n \) derived \( * \) | The lineage graph for all nodes derived from nodes in \( e_n \).                                                                        |
| \(e_{n1}..e_{n2}\)            | \( e_{n1} \) derived \( e_{n2} \) | The lineage graph containing all paths from nodes in \( e_{n1} \) to nodes in \( e_{n2} \).                                           |
| \(e_{n1}..e_i..e_{n2}\)       | \( e_{n1} \) through \( e_i \) derived \( e_{n2} \) | The lineage graph containing all paths from nodes in \( e_{n1} \) to nodes in \( e_{n2} \) that pass through an invocation in \( e_i \). |

**Functions over path queries**

| \( \text{exists}(p) \)         | \( \text{exists}(p) \) | True if the trace contains a path defined by path query \( p \).                                                                       |
| \( \text{invocations}(p) \)    | \( \text{invocations}(p) \) | The invocations of the lineage graph returned by path query \( p \).                                                                   |
| \( \text{actors}(p) \)         | \( \text{actors}(p) \) | The actors of invocations of the lineage graph returned by path query \( p \).                                                         |
| \( \text{nodes}(p) \)          | \( \text{nodes}(p) \) | The nodes of the lineage graph returned by path query \( p \).                                                                          |
| \( \text{input}(p) \)          | \( \text{input}(p) \) | The source nodes of the lineage graph returned by path query \( p \).                                                                    |
| \( \text{output}(p) \)         | \( \text{output}(p) \) | The sink nodes of the lineage graph returned by path query \( p \).                                                                     |
6.2. QLP Syntax, and Semantics 115

\[
\begin{align*}
[n] & := \{n\} \\
[x] & := \{n \mid d \in D, n \in x(d)\} \\
[*] & := \{n \in N\} \\
[#a] & := \{i \in I(a)\} \\
[#i] & := \{i\}.
\end{align*}
\]

\[
\begin{align*}
[e_n @in] & := \{n \in [e_n] \mid d \in in(T), n \in nodes(d)\} \\
[e_n @out] & := \{n \in [e_n] \mid d \in out(T), n \in nodes(d)\} \\
[e_n @in e_i] & := \{n \in [e_n] \mid d \in D, i \in [e_i], \langle d, i \rangle \in E_{in}, n \in nodes(d)\} \\
[e_n @out e_i] & := \{n \in [e_n] \mid d \in D, i \in [e_i], \langle i, d \rangle \in E_{out}, n \in nodes(d)\}
\end{align*}
\]

Figure 6.2: Semantics of step expressions \(s(T)\).

and XML structures \(d\), in the normal way, \(x(d)\) returns those nodes in \(d\) satisfying the expression \(x\).

We define QLP path expressions \(p\) as functions that take traces \(T = (V, F, L_0)\) and lineage graphs \(L \subseteq L_0\) and return the given trace together with a new lineage graph \(L_p\). Specifically, if \(p\) is a path expression, \(T = (V, F, L_0)\) is a trace, and \(L\) is a lineage graph, \(p(T, L) = \langle T, L_p \rangle\) such that \(L_p \subseteq L\). Simple immediate path expressions of the form \(s_1 . s_2\) for traces \(T\) and lineage graphs \(L\) are defined in Figure 6.3. For convenience we generally omit \(T\) when writing steps (as in Figure 6.2) and paths (as in Figs. 6.3, 6.4, and 6.5).

Before defining more complex path expressions, we first consider the \texttt{exists} operator in Figure 6.4, which returns true if a given lineage graph contains a path \(p\) and false otherwise. Checking whether a path \(s_1 . s_2\) exists in \(L\) (i.e., the second definition in Figure 6.4) employs recursion: a path exists if an immediate path exists between \(s_1\) and \(s_2\) or else if there is a node \(n\) such that an immediate path exists from \(s_1\) to \(n\) and a transitive path exists from \(n\) to \(s_2\).
\[
[e_{n_1} \cdot e_{n_2}](L) := \{ (n_1, i, n_2) \mid n_1 \in [e_{n_1}], n_2 \in [e_{n_2}] \}
\]

\[
[e_{n_1} \cdot e_i](L) := \{ (n_1, i, n_2) \mid n_1 \in [e_{n_1}], i \in [e_i] \}
\]

\[
[e_i \cdot e_{n_2}](L) := \{ (n_1, i, n_2) \mid i \in [e_i], n_2 \in [e_{n_2}] \}
\]

\[
[e_{i_1} \cdot e_{i_2}](L) := \{ (n_1, i_1, n_2), (n_2, i_2, n_3) \mid (n_1, i_1, n_2) \in L, (n_2, i_2, n_3) \in L \}
\]

Figure 6.3: Semantics of simple immediate path expressions \(s_1 \cdot s_n (T, L)\).

\[
[\exists (s_1 \ldots s_2)](L) \equiv [s_1 \cdot s_2](L) \neq \emptyset
\]

\[
[\exists (s_1 \ldots s_2)](L) \equiv [\exists (s_1 \ldots s_2)](L) \lor (\exists n \in N : [\exists (s_1 \cdot n)](L) \land [\exists (n \cdot s_2)](L))
\]

\[
[\exists (s_1 \cdot s_2)](L) \equiv \exists n \in [e_n] : [\exists (s_1 \cdot n)](L) \land [\exists (n \cdot s_2)](L)
\]

\[
[\exists (s_1 \cdot s_2)](L) \equiv \exists n \in [e_n] : [\exists (s_1 \cdot n)](L) \land [\exists (n \cdot s_2)](L)
\]

Figure 6.4: Semantics of \(\exists\) (i.e., reachability) queries over path expressions \(s_1 \circ_1 s_2 \circ_2 \cdots \circ_m s_m\) for \(m \geq 3\) and \(\circ\) an immediate or transitive path.

The last two definitions in Figure 6.4 consider the general case of paths of length \(m > 2\). As shown, invocation expressions are treated as a special case since in general it is difficult to replace an invocation with a corresponding node (without introducing a number of additional conditions). For example, consider a lineage graph comprising of two edges \(\langle n_1, i_1, n_2 \rangle\) and \(\langle n_2, i_2, n_3 \rangle\), and the 3-step query \(\exists (n_1 \cdot i_1 \cdot n_2)\). Replacing \(i_1\) in the query with its input \(n_1\) results in the two subqueries \(\exists (n_1 \cdot n_1)\) and \(\exists (n_1 \cdot n_2)\), whereas replacing \(i_1\) with its output \(n_2\) results in the subqueries \(\exists (n_1 \cdot n_2)\) and \(\exists (n_2 \cdot n_2)\), both of which consist of unsatisfiable subqueries of the form \(\exists (n \cdot n)\). This issue is addressed by the fourth definition in Figure 6.4, which for this example first computes \(n_1 \cdot n_2\) over \(L\), and then checks if both \(n_1 \cdot i_1\) and \(i_1 \cdot n_2\) exist in the result (i.e., if \(i_1\) can be reached from \(n_1\), and \(n_2\) can be reached from \(i_1\)). Note that for general paths from \(s_1\) to \(s_m\), after applying \(s_1 \cdot s_m\), if it is possible to reach an invocation \(i\) from \(s_1\), then trivially \(i\) falls on a path to \(s_m\), and conversely, if it is possible to reach \(s_m\) from \(i\), then \(i\) must trivially fall on a path from \(s_1\).
6.3 QLP Evaluation Techniques

Transitive path queries are defined in Figure 6.5. Similar to Figure 6.4, invocations in path expressions of length \( m > 2 \) are defined as a special case. Note also that in each definition of Figure 6.5, we require that a given path exists prior to returning lineage relations. Without such a check, it is possible that edges forming incomplete paths could be returned.

Finally, we note that because QLP lineage queries return sets of dependency edges, more complex queries can be built from path expressions using standard set operations. For instance, given two QLP path expressions \( p_1 \) and \( p_2 \), it is possible to combine these into more complex queries using set intersection \( p_1 \cap p_2 \), union \( p_1 \cup p_2 \), and difference \( p_1 - p_2 \).

6.3 QLP Evaluation Techniques

In this section we describe techniques for evaluating transitive path queries in QLP, i.e., queries that use the ‘..’ operator, since these are often considerably more expensive to evaluate (e.g., see [HA08, ABML09a]) than XPath queries or queries that simply select specific versions of structures within a trace (using @in and @out).

The evaluation of transitive path queries is closely tied to how lineage relations are stored. Thus, the techniques proposed here consider different approaches for storing lineage relations and associated techniques for answering queries.

We consider path expressions \( p \) that take the form \( s_1 .. s_2 .. s_3 \cdots s_m \) of length
m > 1 such that for a set of lineage relations \( L \), we select the set of paths in \( L \) satisfying \( p \), and return these paths as a subset of the given lineage relations. To simplify the discussion, we assume that each step \( s_i \) in \( p \) can be evaluated to a set of nodes \( N_i \). Thus, a path expression \( p = s_1 \ldots s_2 \) can be rewritten as \( N_1 \ldots N_2 \) denoting a set of subqueries such that the answer to \( p \) is comprised of the answers to each of the simpler subqueries \( n_1 \ldots n_2 \) for \( n_1 \in N_1 \) and \( n_2 \in N_2 \).

If a step \( s \) in \( p \) is an invocation expression, we construct the corresponding set of nodes as follows. First, we use \( \text{in}(s) \) and \( \text{out}(s) \) to denote the set of input and output nodes across invocations of \( s \). Let \( p = s_1 \ldots s_2 \). If \( s_1 \) is an invocation expression and \( s_2 \) a node expression evaluating to \( N_2 \), we rewrite \( p \) as \( \text{in}(s_1) \ldots (\text{out}(s_1) \cup N_2) \). Similarly, if \( s_2 \) is also an invocation expression, we rewrite \( p \) as \( \text{in}(s_1) \ldots \text{out}(s_2) \). Finally, if \( s_1 \) is a node expression evaluating to \( N_1 \) and \( s_2 \) an invocation expression, we rewrite \( p \) as \( (N_1 \cup \text{in}(s_2)) \ldots \text{out}(s_2) \). Rewriting lineage queries in this way can be easily extended to paths containing more than two steps.

We start by presenting different strategies (denoted \( I \), \( ICP \), \( ICP(N) \), and \( ICP(S) \)) for storing lineage graphs and discuss their trade-offs in terms of storage and query efficiency. We then describe additional optimization techniques for queries involving paths containing more than 2 steps and for cases when steps evaluate to multiple nodes.

**Immediate Lineage Edges** (\( I \)). A common approach for storing lineage graphs is to store only the immediate edges in an edge table \( \text{Edge}(N_1, I, N_2) \). Evaluating path expressions in this case requires recursion or iteration over the edges of the graph. For example, the following Datalog program can be used to compute the set of lineage relations \( \langle x, i, y \rangle \) between any two nodes \( N_1 \) and \( N_2 \).
In the above query, Path is the transitive closure of the Edge relation. Query $Q$ computes lineage relations as path nodes are traversed by recursively performing a breadth-first search (using both $\text{Path}$ and $Q$).

Here, we propose a more efficient approach in which we (i) compute the set of nodes on paths starting at node $n_1$ and ending at node $n_2$, and then (ii) join the resulting path nodes with the Edge table to return the corresponding lineage edges. The following Datalog program implements this approach, which returns the same set of edges as the previous program. In general, we adopt this query approach when considering the immediate (edge) storage strategy, which we denote as storage strategy $I$. 
Note that this approach has a single recursive rule (Path) whereas the previous approach has two recursive rules (Path and Q).

**Immediate Edges and Dependency Closure (IC).** The recursion in the previous approach can be removed by materializing the Path relation. Thus, this strategy (denoted IC) stores both the immediate edges in an edge table $\text{Edge}(N_1, I, N_2)$ and the transitive closure over dependency nodes $\text{Path}(N_1, N_2)$. To compute the set of nodes on lineage paths (PathNode), this approach must perform an expensive self-join on a potentially large Path table. In addition, to return a set of lineage relations, a join must still be performed between PathNode and the Edge table. Note that this approach increases storage size by materializing Path (at a cost of $O(n^2)$) to reduce query time by eliminating recursion.

**Immediate Edges and Closure via Pointers (ICP).** To address the issue of performing a self-join on large, materialized Path tables, we employ a “pointer-based” strategy (denoted here as ICP) that partitions the Path table into three smaller tables.
(based on reduction techniques) where the original Path table can be obtained by joining these smaller tables together. This pointer-based storage strategy is discussed in our earlier work [ABML09a]. In particular, the Path table is partitioned into the following three tables (see Figure 6.6)

- \text{Node}(N_2, I, P_{\text{dep}}, P_{\text{depc}})
- \text{DepV}(P_{\text{dep}}, N_1)
- \text{DepcV}(P_{\text{depc}}, P_{\text{dep}})

where $P_{\text{dep}}$ is a pointer to the set of immediate dependencies of $n_2$; and $P_{\text{depc}}$ is a pointer to a set of immediate dependency pointers $P_{\text{dep}}$, representing the transitive dependency closure of $n_2$. Using this approach, the lineage graph shown in Figure 6.6(a) is reduced to the graph in Figure 6.6(b).

The basic idea of the approach is to use pointers to store a single copy of shared sets of dependency nodes. For instance, in Figure 6.6(a), the dependency set for node 16 is $\{13, 14\}$ and the dependency set for node 17 is also $\{13, 14\}$. Instead of storing multiple copies of the same dependency set, a pointer &9 is used to refer to the dependency set $\{13, 14\}$ in the DepV relation, and nodes 16 and 17 refer to pointer &9 in the Node relation. In addition, instead of storing the transitive closure between dependency nodes, DepcV stores the transitive closure of the immediate pointers, e.g., as shown by the dashed lines in Figure 6.6(b). For instance, the transitive dependency closure of node 16 is the set of nodes $\{6, 7, 9, 10, 11, 13, 14\}$, whereas in the pointer-based representation, node 16 has the transitive dependency pointer &9′ that refers to the transitive dependency (pointer) set \{&1, &5, &9\} in the DepcV relation.
6.3. QLP Evaluation Techniques

(a). Node-based lineage graph

(b). Pointer-based (reduced) lineage graph for (a)

Figure 6.6: Lineage graph with corresponding pointer-based representations.

The following rules can be used to reconstruct a node-based lineage graph from a pointer-based (reduced) lineage graph.

\[
\text{Edge}(N_1, I, N_2) : \neg \text{Node}(N_2, I, P_{dep}, P_{depc}), \text{DepV}(P_{dep}, N_1).
\]

\[
\text{Path}(N_1, N_2) : \neg \text{Node}(N_2, I, P_{dep}, P_{depc}), \text{DepcV}(P_{depc}, P_{dep}),
\text{DepV}(P_{dep}, N_1).
\]

As described in [ABML09a], both \text{DepV} and \text{DepcV} are further reduced by factoring out common subsets of dependency sets. In this way, \text{DepV} and \text{DepcV} are simple views defined over the further reduced tables.

Note that these Edge and Path queries can be used with PathNode defined above to answer QLP path expressions of the form \(n_1 .. n_2\). While computing the PathNode
6.3. QLP Evaluation Techniques

relation is more efficient in the pointer-based approach (since we avoid self-joins over the materialized Path table), it still requires a number of joins over intermediate relations (i.e., DepV, DepcV, Path, and PathNode). To more efficiently evaluate path expressions, we (i) temporarily materialize the views DepV and DepcV (which together contain fewer overall tuples than Path, as shown in Figure 6.6); (ii) evaluate and materialize the queries Path($N_1, N$) and Path($N, N_2$); (iii) use these materialized Path queries to evaluate and materialize the query PathNode($N_1, N_2, N$); and (iv) use the materialized PathNode query to answer Q($N_1, N_2, X, I, Y$), returning the set of lineage edges $\langle x, i, y \rangle$.

**Evaluating Simple Reachability Queries.** To answer reachability queries between two nodes $n_1$ and $n_2$, i.e., \( \exists (n_1 .. n_2) \), we simply evaluate the query Path($n_1, n_2$).

**Evaluating Single Node-Based Path Expressions.** We evaluate general path expressions \( p = n_1 .. n_2 .. n_3 \cdots n_m - 1 .. n_m \) for \( m > 2 \) using ICP by transforming the expression into a set of simple path expressions \( n_i .. n_{i+1} \) (for \( 1 \leq i \leq m \)). If each simple path is reachable, then we find the set of nodes for each simple path, and compute the set of edges over these nodes (see Figure 6.7).

**Naive Evaluation of Set-Based Path Expressions (ICP($N$)).** A straightforward approach for evaluating set-based path expressions \( p = N_1 .. N_2 \cdots N_m \) for \( N_i \) a set of nodes, is to rewrite \( p \) into multiple node-based path expressions \( n_1 .. n_2 \cdots n_m \) for each \( n_i \in N_i \). Each of these node-based path expressions are evaluated in a similar way as in Figure 6.7, i.e., for each node-based path expression we add the corresponding set of nodes returned to \( N_p \) and then compute the set of edges as in step 10 (i.e., computing edges is deferred until each of the node-based expressions is computed). Note however, that in this approach the number of node-based path
6.3. QLP Evaluation Techniques

**Input:** $p = n_1 \ldots n_2 \ldots n_3 \ldots n_m$

**Output:** Set of lineage relations

1. for $i \leq m - 1$ do
2.  if not exists($n_i.. n_{i+1}$) then
3.    return $\emptyset$ /* no lineage path for $p$*/
4.  end if
5. end for
6. $N_p = \emptyset$
7. for $i \leq m - 1$ do
8.  $N_p = N_p \cup \{n | \text{PathNode}(n_i,n_{i+1},n)\}$
9. end for
10. return $\{(x,i,y) | x \in N_p, y \in N_p, \text{Edge}(x,i,y)\}$ /* lineage edge */

Figure 6.7: Algorithm for evaluating single node-based path expressions ($ICP$)

expressions that must be computed from $p$ is $|N_1| \times |N_2| \times \cdots \times |N_m|$. This node-based approach for evaluating path expressions is referred to as $ICP(N)$.

**Efficient Evaluation of Set-Based Path Expressions ($ICP(S)$).** To avoid the cost of computing a potentially large number of node-based path expressions (referred to as $ICP(S)$). For simple paths $N_1..N_2$, we first evaluate $N_1..*$ as follows: (i) compute the set of pointers $P_{dep}$ for the set of nodes $N_1$ from the DepV relation; (ii) from the set $P_{dep}$ we select the set of transitive pointers $P_{depc}$ from DepcV; (iii) from Node relation we select all nodes $N_F$ (i.e., the “forward” lineage nodes) that have $p_{depc} \in P_{depc}$; and (iv) we add only those nodes from $N_1$ to the nodes obtained in (iii) that have a $P_{dep}$ associated with them. We then use a similar approach to evaluate the expression $*..N_2$, resulting in a set of nodes $N_B$ (the “backward” lineage nodes). The set of nodes $N$ on the lineage path from nodes in $N_1$ to nodes in $N_2$ is $N_F \cap N_B$. Finally, the set of lineage edges is computed from $N$.

Given a path expression of the form $p = N_1..N_2 \cdots N_m$ 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
Input: \( p = N_1 .. N_2 .. N_3 \cdots N_m \)
Output: Set of lineage relations

1. for \( i \leq m \) do
2. \( N_{Fi} = \{ n | \text{Path}(n_i, n), n_i \in N_i \} \quad /\!* \text{forward lineage nodes} \*/
3. \( N_{Bi} = \{ n | \text{Path}(n, n_i), n_i \in N_i \} \quad /\!* \text{backward lineage nodes} \*/
4. end for
5. for \( i \leq m \) do
6. \( PN_i = N_i \)
7. for \( j \leq i - 1 \) do
8. \( PN_i = PN_i \cap N_{Fj} \)
9. end for
10. for \( i + 1 \leq j \leq m \) do
11. \( PN_i = PN_i \cap N_{Bj} \quad /\!* \text{pruned set} \*/
12. end for
13. if \( (PN_i == \emptyset) \) then
14. return \( \emptyset \quad /\!* \text{no lineage path for} \ p \!*\/
15. end if
16. end for
17. for \( i \leq m \) do
18. \( PN_{Fi} = \{ n | \text{Path}(n_i, n), n_i \in PN_i \} \)
19. \( PN_{Bi} = \{ n | \text{Path}(n, n_i), n_i \in PN_i \} \)
20. end for
21. \( N_p = \emptyset \)
22. for \( i \leq m - 1 \) do
23. \( N_p = N_p \cup (PN_{Fi} \cap PN_{Bi+1}) \)
24. end for
25. return \{ \langle x, i, y \rangle | x \in N_p, y \in N_p, Edge(x, i, y) \} \quad /\!* \text{lineage edge} \*/

Figure 6.8: Algorithm for efficiently evaluating set-based path expressions (ICP(S))

other set. In ICP(S), we prune each of the sets \( N \) giving a new set \( PN \) 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. We rewrite \( p \) to \( PN_1 .. PN_2 \cdots PN_m \) and evaluate each simple path \( PN_i .. PN_{i+1} \) as in Figure 6.8. Lines 1–4 compute “forward” and “backward” lineage nodes for each set \( N_i \). Lines 5–16 prune each set \( N_i \) giving \( PN_i = (\cap_{j=1}^{i-1} N_{Fi}) \cap N_i \cap (\cap_{k=i+1}^{m} N_{Bi}) \) such that all nodes in the pruned sets \( PN_i \) are reachable from at least one node in each of the other sets. Lines 17–20 compute
6.3. QLP Evaluation Techniques

![Diagram](image)

Figure 6.9: Example execution of set-based path expressions using set-based algorithm (Figure 6.8) for $p = N_1..N_2..N_3$: (a) the initial lineage graph where $N_1 = \{1, 2, 3\}$, $N_2 = \{4, 5, 6\}$ and $N_3 = \{7, 8, 9\}$; (b) the sets of nodes corresponding to $N_1$, $N_2$, and $N_3$; (c) the “forward” and “backward” lineage nodes for each $N_i$; (d) the pruned sets $N_i$ such that all nodes in $PN_i$ are reachable from at least one node in all other sets; (e) the pruned sets $p' = PN_1..PN_2..PN_3$; (f) the lineage nodes over pruned sets $N_p = \text{nodes}(PN_1..PN_2..PN_3) = \{2, 6, 9\}$; and (g) the lineage edges $(2, 6)$ and $(6, 9)$ returned as a result of the lineage nodes $\{2, 6, 9\}$.

“forward” and “backward” lineage nodes for pruned sets $PN_i$. Lines 21–24 evaluate lineage nodes $N_p$ for path expressions over pruned sets. Finally, line 25 computes lineage edges from the lineage nodes.

To help see how this approach works, consider a QLP path expression query $p = N_1..N_2..N_3$ for the lineage graph shown in Figure 6.9(a), where $N_1 = \{1, 2, 3\}$ (red color), $N_2 = \{4, 5, 6\}$ (blue color), and $N_3 = \{7, 8, 9\}$ (green color). Figure 6.9(c–g) shows the various stages of Figure 6.8 for evaluating set-based path expressions. Figure 6.9(c)) shows all those nodes that are reachable from the nodes in each sets $N_i$. Figure 6.9(d) selects those nodes from sets $N_i$ such that all those nodes are reachable from at least one node in all other sets. Figure 6.9(e) shows the pruned sets $PN_i$,.
where \( P_{N_1} = \{1, 2\} \), \( P_{N_2} = \{6\} \), and \( N_3 = \{8, 9\} \), such that each node in \( P_{n_i} \) is reachable from at least one node in other sets. Figure 6.9(f) shows the nodes that lie on lineage path over the pruned sets. Finally, Figure 6.9(g) shows the lineage edges \( \langle 2, \_ , 6 \rangle \) and \( \langle 6, \_ , 9 \rangle \) over the lineage nodes \( \{2, 6, 9\} \).

Unlike in \( ICP(N) \), which requires \( |N_1| \times |N_2| \times \cdots \times |N_m| \) simple path expressions \( (N_i..N_{i+1}) \) to be computed, \( ICP(S) \) requires only \( 2 \times m \) simple path expressions to be computed.

### 6.4 Evaluation of Query Optimization Techniques

Here we evaluate the efficiency and scalability of answering QLP queries over the storage strategies of Section 6.3 on both real and synthetic traces. Real traces were generated from existing workflows implemented within the Kepler scientific workflow system (using the extensions described in [BMR+08, ABML09b]). Our experiments were performed using a 2.4GHz Intel Core 2 duo PC with 2 GB RAM and 120 GB of disk space. Each storage strategy used PostgreSQL to store provenance information, and our QLP parser was implemented in Java using JDBC to communicate with the provenance database.

We compare query response time and storage size using synthetic traces ranging from 100 to 6000 nodes, \( 5 \times 10^2 \) to \( 10^5 \) immediate dependencies, \( 10^3 \) to \( 10^8 \) transitive dependencies, and lineage paths of length 10 to 100, respectively. The synthetic traces were taken from [ABML09a], and represent typical lineage patterns of common scientific workflows [ABML09a, BMR+08, MLA+08]. Figure 6.10(a) shows the complexity of dependencies (immediate and transitive) as the nodes in synthetic traces increase, and Figure 6.10(b) shows the storage size of these traces under the different storage strategies. As shown, the pointer-based approach used to store immediate
and transitive dependencies in a reduced form leads to considerably smaller storage size [ABML09a].

We also evaluated our approaches using the following real traces from scientific workflows implemented within Kepler: the PTP, GBL, CGR, and PLC workflows [BMR+08] use different approaches to infer phylogenetic trees from protein and morphological sequence data; the PC1 workflow was used in the first provenance challenge [MLA+08] (and is similar to the example shown in Figure 6.1); the STP, STM, and CYC workflows are used in characterizing microbial communities by clustering and identifying DNA sequences of 16S ribosomal RNA; the WAT workflow characterizes microbial populations by producing phylogenetic trees from a list of sequence libraries; and the PC3 workflow was used within the third provenance challenge\(^1\).

The dependency annotations of these traces range from $10^2$ to $10^4$ with $2 \times 10^2$ to $2 \times 10^4$ transitive dependency edges.

We consider the following QLP queries for evaluating query response time.

\(^1\)see http://twiki.ipaw.info/bin/view/Challenge/
6.4. Evaluation of Query Optimization Techniques

Figure 6.11: Evaluating $\ast \ldots n$ over storage strategies.

\begin{align*}
\ast \ldots n & \quad \text{(Q1)} \\
n_1 \ldots n_2 & \quad \text{(Q2)} \\
n_1 \ldots n_2 \ldots n_3 & \quad \text{(Q3)} \\
\ast \ldots N & \quad \text{(Q4)} \\
N_1 \ldots N_2 & \quad \text{(Q5)} \\
N_1 \ldots N_2 \ldots N_3 \ldots N_m & \quad \text{(Q6)}
\end{align*}

These queries return (Q1) lineage relations denoting paths that lead to a node $n$ (e.g., to return the full lineage of $n$); (Q2) lineage relations denoting paths starting at node
n_1 and ending at node n_2; (Q3) lineage relations denoting paths starting at node n_1 and ending at node n_3 that pass through node n_2; (Q4) lineage relations denoting paths that lead to nodes in a set of nodes N; (Q5) lineage relations denoting paths starting at nodes in a set N_1 and ending at nodes in a set N_2; and (Q6) lineage relations denoting paths that satisfying path expression of varying length m (e.g., in our experiments, m varies from 2 to 25).

These queries are evaluated over various storage strategies (I, IC, and ICP, described in Section 6.3). We implemented strategy I using XSB (a logic programming
6.4. Evaluation of Query Optimization Techniques

Figure 6.13: Evaluating $n_1 .. n_2 .. n_3$ over storage strategies.

and deductive database system that supports in-memory queries\(^2\) whereas the implementation of $IC, ICP(N)$, and $ICP(S)$ uses PostgreSQL for storing and querying traces.

**Paths with Single Node Identifiers.** In queries Q1–Q3 each step evaluates to a single node identifier. Figs. 6.11–6.13 show the time to evaluate Q1, Q2, and Q3, respectively, over $I$, $IC$, and $ICP$. In each case, we show: (a) the total query

\(^2\)see [http://xsb.sourceforge.net/](http://xsb.sourceforge.net/)
Figure 6.14: Comparing node-based $ICP(N)$ and set-based $ICP(S)$ approaches for evaluating $N_1 .. N_2$.

(a) execution time for computing lineage relations, (b) the portion of the total time used to compute the nodes that lie on the lineage path, and (c) the portion of the total
6.4. Evaluation of Query Optimization Techniques

Figure 6.15: Evaluating $N_1 .. N_2 .. N_3 \cdot \cdot \cdot N_m$ using ICP($S$).

time used to compute the lineage edges from the nodes returned in (b).

Figure 6.11(a) shows that Q1 scales for ICP($N$) but does not scale for IC and I. I scales when returning set of nodes (Figure 6.11(b)), but I does not scale for computing edges from these set of nodes (Figure 6.11(c)). Similar results were observed for Q2 and Q3, which have more complex path expressions. We also observe that as path expressions increase in length, query time scales for ICP($N$) but not for I and IC. For example, for a trace containing 6000 nodes, Figure 6.11 shows that ICP($N$) takes 500 ms, 600 ms, and 700 ms to evaluate Q1, Q2, and Q3, respectively, representing only a marginal increase in query response time as the length of paths increases.

Paths with Sets of Node Identifiers. In query Q5, each step evaluates to a set of node identifiers. Figure 6.14 shows the time to evaluate Q5 over the storage strategies. Figure 6.14 shows that when the node-based approach ICP($N$) is used
6.4. Evaluation of Query Optimization Techniques

Figures 6.16: (a) and (b) show size for storing real traces in various storage strategies. (c) and (d) show time for evaluating constructs \( \ast \ldots N \) and \( N_1 \ldots N_2 \) over these storage strategies.

to evaluate Q5, the query time is proportional to the product of the number of nodes in each path set. This is due to the number of node-based subqueries that ICP\((N)\) must evaluate. However, for the set-based approach ICP\((S)\), the query response time remains constant as the size of sets increase. As shown, ICP\((S)\) outperforms ICP\((N)\) when the number of nodes in each set is greater than 2.

**Evaluating Path Expressions of Increasing Length.** Figure 6.15 shows that using the optimization techniques of ICP\((S)\), query response time scales linearly as the length of path expressions increase. In particular, we consider path expressions involving between 1 and 25 steps in Figure 6.15. As shown, ICP\((S)\) can be used to efficiently and scalably evaluate complex QLP path expressions.

**Storing and Querying Real Traces.** Figure 8.8(a) and (b) show the storage size for representing lineage graphs and transitive dependencies for the real traces under
our storage strategies. Figure 8.8(a) shows the storage size of “smaller” traces (trace nodes ranging from 80 to 200) and Figure 8.8(b) shows the storage size of “bigger” traces (trace nodes ranging from 1000 to 10,000). Figure 8.8(a) and Figure 8.8(b) show that the space requirement for storing lineage relations in ICP is considerably less than in I and IC. Note that I stores only lineage edges, while ICP stores both lineage edges and transitive closure of dependency nodes using our reduction techniques, and in most of the cases (except PTP, PC1, and PC3), the storage size for ICP is less than that of I (implying that many nodes have similar dependencies). Also, although ICP stores the same information as IC, the storage size for ICP is less than that for IC.

Figure 8.8(c) and (d) show the query time for evaluating Q4 and Q5 over real traces (both small and big) using IC and ICP. To consider complex queries, each step in the path expressions evaluate to all data nodes of the trace. As shown, path queries are evaluated using ICP(S) in less than $10^2$ ms, which is significantly less than using IC which can take up to $10^6$ ms. This is because IC has to perform a self-join over the transitive closure table together with a join over the dependency table, both containing a large number of tuples, whereas ICP(S) performs three joins over the reduced tables. If we compare this result with the results for the synthetic traces, we see that as the number of dependencies increases, IC will perform even worse.

**Analysis.** Our detailed experimental evaluation demonstrates that ICP not only requires less storage, but evaluating QLP queries over ICP is faster and scales with increasing complexity of traces. In particular, our optimized set-based approach ICP(S) for evaluating QLP path expressions is both efficient and scalable.
6.5 Related Work

Approaches for querying stored provenance information are largely based on physical data representations [DF08] (e.g., relational, XML, or RDF schemas), where users express provenance queries through corresponding query languages (i.e., SQL, XQuery, or SPARQL). Provenance queries often require computing transitive closures over dependency relations, and expressing such queries using standard approaches is typically done using recursion or stored procedures [HA08, CJR08, ABML09a]. Expressing such queries is both cumbersome and error-prone, and requires considerable user expertise. Instead, high-level languages such as QLP provide a separation between the logical provenance model and its underlying physical representation, which allows for the use of different representation schemes and additional optimization techniques. Our approach, in particular, automatically translates QLP queries to equivalent relational queries expressed against the provenance storage schemes described in [ABML09a].

Standard approaches for querying provenance information (e.g., [Car08, HBM+08, ZL08, CFL+07, BBDH08]) return sets of nodes (either sets of data items or process invocations) as the query result. This approach requires additional steps (queries) to reconstruct causal relations among nodes within a query answer. Instead, QLP is closed under lineage relations, where answers to lineage queries are sets of lineage dependencies (edges) forming provenance subgraphs, and thus query results are “provenance preserving.” This approach has a number of advantages, e.g., for supporting provenance views, incremental querying, and for supporting visualization applications [DF08, BMR+08].

Our approach is similar to a number of approaches for querying graph structures. In [HS08], a general query language based on graph grammars is presented in which
6.5. Related Work

queries return subgraphs. However, whereas our implementation uses a relational database system to store and query provenance graphs, the approach in [HS08] requires entire graph structures to be stored in main memory, and recursive patterns over graphs (e.g., similar to our ‘..’) are not supported. Similarly, semi-structured query languages (e.g., Lorel [AQM+97] and Florid [LHL+98]) often include so-called “path variables”, which can be used to query over and return paths (i.e., the nodes) between nodes. However, while a part of the query language of these systems, path variables are generally not implemented (e.g., see [AQM+97, LHL+98]). Additionally, the approach used in QLP differs from that of path variables in that query results are sets of lineage relations that implicitly denote paths between nodes as opposed to directly returning the paths themselves. This in turn offers advantages for query composition and efficient evaluation (as described in Section 6.3).

Similar to QLP, the VisTrails provenance query language (vtPQL) [Car08] defines provenance-specific query constructs. However, vtPQL provides only a limited set of constructs for accessing provenance relations (e.g., transitive queries are not supported). vtPQL defines upstream(\(x\)) to return all modules (i.e., actors) that precede \(x\) in the workflow definition, which is the primary construct for answering lineage queries related to invocations. However, vtPQL assumes only a single lineage path between two such modules, and thus would return incorrect results in the case where multiple paths exist between nodes.

The QLP construct \texttt{exists}(n_1 .. n_2) corresponds to a reachability query [JXRW08, WHY+06] over lineage graphs. Besides simple paths, our implementation can efficiently answer more complex reachability queries for paths \(p = s_1 .. s_2 .. s_3 .. · · · s_m\) of length \(m > 2\) in which each step \(s\) can evaluate to a set of nodes. Thus, our implementation can also efficiently determine whether a lineage graph satisfies relatively
6.6. Conclusion

In this work, we have presented a general approach for querying provenance using a novel query language (QLP) that provides specialized constructs for querying over lineage (i.e., data and process dependency) and structural (i.e., parent-child) relations introduced during scientific workflow runs. In QLP, answers to lineage queries are sets of lineage dependencies (edges) forming provenance subgraphs. This approach simplifies the expression of complex queries and allows query results to be easily queried and visualized. The work presented here extends our prior work on efficiently storing [ABML09a] and querying [ABML09b] scientific workflow provenance by (i) developing a formal semantics for QLP constructs; (i) presenting a set of novel query optimization techniques that leverage the lineage-graph reduction approaches presented in [ABML09a]; and (iii) by presenting a detailed experimental evaluation based on our QLP implementation that demonstrates the efficiency and scalability of our approach. We have shown that our implementation can simultaneously reduce the amount of storage required to represent scientific workflow provenance without negatively affecting query performance. We have also shown that unlike using standard approaches (which are often based on in-memory implementations, e.g., [ZL08, HS08, LHL+98]), answering QLP queries with our optimization techniques over a relational database system is both feasible and scales with the size of provenance information and query complexity. Our optimizations can also be used in more general settings to efficiently answer a broad range of path queries over labeled, acyclic digraphs.
Chapter 7

A Navigation Model for Provenance

7.1 Exploring Large Provenance Graphs

Most scientific workflow systems provide mechanisms for recording workflow provenance, i.e., the details of a workflow run including data and process dependencies [DF08, MLA+08, SPG05]. This provenance information is often displayed to users visually as one or more dependency graphs [MFF+08, BMR+08], e.g., where nodes denote data items or processes, and edges denote causal relationships between nodes. Displaying such graphs is especially useful for small workflows, involving only a few data sets and processes, since users can quickly see every data product, process, and dependency associated with a run. However, for many real-world scientific workflows provenance graphs may be large (e.g., thousands of nodes and edges) due to the complexity of the workflow, the size of input data sets, and the number of intermediate data sets produced [CJR08, ABML09a, DF08]. For large provenance graphs,
7.1. Exploring Large Provenance Graphs

Figure 7.1: Different provenance graphs displayed using the provenance browser of [6]: (a) and (b) shows a provenance graph for relatively small traces, whereas (c) shows a provenance graph for a much larger trace.

understanding and exploring provenance information becomes a significant challenge for users.

For example, Figure 7.1 shows three different provenance graphs generated from real-world workflows: Figure 7.1a shows a run of a workflow that infers phylogenetic trees from protein sequence data and then determining the maximum-likelihood tree using RAxML [?]; Figure 7.1b shows a run of standard phylogenetic tree inference workflow; and Figure 7.1c shows a run of a workflow that infers phylogenetic trees from morphological sequence data [BMR+08]. While the provenance graph of Figure 7.1a and Figure 7.1b can be quickly understood (since it contains only a few nodes and dependencies), the graph of Figure 7.1c is much larger and requires considerable effort to fully understand the associated provenance information.

The goal of the work described here is to allow users to specify and navigate between different abstractions (or views) of provenance graphs such that by displaying
these views, users can obtain the same benefits of quickly understanding and exploring provenance information as for complete (but small) provenance graphs. Specifically, we present a navigation model for scientific workflow provenance that consists of operations for creating, refining, and switching (or navigating) between different views of workflow provenance. We consider three main levels of granularity. An actor dependency graph consists of the types of processes (or actors) used in a workflow run and the flow of data between them. An invocation dependency graph consists of individual processes (or invocations) of the workflow run and the corresponding flow of data. And a flow dependency graph consists of the detailed data items input to and produced by the workflow run and their causal dependencies. In addition, the model supports operations that allow all or a portion of each provenance view to be expanded or collapsed, grouped into composite structures, and filtered or exposed using a high-level provenance query language. The navigation model is also based on a generic model of provenance that subsumes conventional approaches for representing workflow provenance while supporting more advanced workflow computation models that permit structured data and update semantics [ABML09a].

7.2 Navigation Model

Provenance model. As described in Section 3.6, our model of provenance subsumes conventional approaches for representing workflow provenance while supporting more advanced workflow computation models that permit multiple invocations of processes (e.g., for pipelining and loops), structured data, and update semantics [ABML09a]. Our approach naturally extends the conventional provenance model by adding support for nested data and for accurately capturing detailed lineage information of
processes employing update semantics. Please see Section 3.6 and Section 3.7 for details on our provenance model.

**Querying Provenance with QLP.** Our provenance model supports a high-level query language for provenance (QLP) [ABML09b], that allows users to easily express complex provenance queries. QLP queries can be posed against flow graphs through a number of different QLP constructs, some of which are described below. These constructs are used to query distinct dimensions of the flow graph representing: (i) *dependency paths* over nodes and invocations; (ii) *flow relations* among input and output structures of invocations; and (iii) *structural relations* among nodes within and across data structures. For a detailed description of QLP, please refer to Section 5.3.

While provenance query languages such as QLP can help users manage the complexity of large provenance graphs, they require knowledge of the provenance graph before queries can be issued. These languages also provide limited support for navigating provenance graphs (namely, by repeatedly issuing different queries). Thus, additional techniques are required to help users explore large provenance graphs who *a priori* do not know which portions are relevant, who want to display the graph in an aggregated or summarized form, or who wish to quickly navigate between different provenance views. This section describes a *provenance navigation model* that is designed to help address these issues.

The navigation model provides an integrated approach for exploring, summarizing, and querying all or select portions of provenance graphs through a set of *navigation operators* (see Figure 7.4). These operators allow users to: (i) explore and navigate various provenance views at different levels of granularity; (ii) summarize (or abstract) portions of views through grouping; and (iii) filter (or query) provenance views using QLP. We first describe the different views (i.e., levels of granularity)
of flow graphs supported by the navigation model, we then describe the navigation
approach and operators supported by the model, and end this section by describing
architectural issues associated with implementing the model.

7.2.1 Provenance Views

A workflow specification is composed of actors together with inter-actor connections.
These connections specify the desired flow of data between actors. During workflow
execution, actors are executed such that data flow is constrained by the given actor
connections. Each actor may be invoked multiple times during workflow execution,
where each invocation receives specific data items and produces new data items that
are dependent on some or all of the given input data. Thus, we consider three separate
levels of granularity for viewing flow graphs in the navigation model, corresponding
(from highest to lowest granularity) to the actor level, the invocation level, and the
data-dependency level.

Specifically, an actor dependency graph (ADG) is a high-level view of a flow graph
that consists of actors and their dataflow connections. An ADG consists of only those
actors (and corresponding connections) that were used during workflow execution.
Figure 7.2a shows an example ADG in which nodes $a$ and $b$ are actors, and edges
represent the flow of data of the workflow run. An invocation dependency graph (IDG)
is the next lower-level view of a flow graph that consists of invocations and their
dataflow connections. Figure 7.2b shows an example IDG in which $a:1$, $a:2$, $b:1$, and
$b:2$ denote invocations of actors $a$ and $b$, respectively. The lowest level of granularity
is a flow dependency graph (FDG), which is a view of a flow graph that contains node-
level data dependencies between input and output structures received and produced
by invocations. Figure 7.2c shows an example FDG containing different versions
Figure 7.2: Three separate levels of granularity for viewing flow graphs (from highest to lowest granularity) at: (a) the actor level; (b) the invocation level; and (c) the data-dependency level.

of the tree structure rooted at node 1 together with the corresponding dependencies (labeled with invocations). Note that an IDG is similar to the invocation-dependency graph of Figure 3.4d, whereas an FDG is similar to the data-dependency graph of Figure 3.4b.

In addition, we also consider three distinct representations of an FDG (see Figure 7.3) that can further simplify the display of dependencies of a flow graph. An expanded flow dependency view (EDEP) shows only those nodes in an FDG that participate in a dependency relationship together with the descendents of these nodes (for
Figure 7.3: Three representations of a flow dependency graph: (a) Expanded Flow Dependency (EDEP) view; (b) Collapsed Flow Dependency (CDEP) view; and (c) Data Flow Dependency (DDEP) view.

dependency nodes that are collections). A collapsed flow dependency view (CDEP) is similar to an EDEP, but does not show the corresponding descendant nodes of collections. A data flow dependency view (DDEP) shows only dependencies of data nodes. Figure 7.3 shows each of these three views for the FDG of Figure 7.2c. We note that the DDEP view can be used to construct a standard data-dependency graph as in Figure 3.4c. In general, the CDEP view (collapsed at the collection level) will produce the smallest flow-dependency graph and the EDEP will produce the largest flow-dependency graph of the three sub-views (CDEP, EDEP, and DDEP). Each of these, however, will be smaller than the FDG, which displays all nodes input to and output by each invocation and not just those that were used to derive new data items.

Within the navigation model, a user can switch to any of these provenance views and sub-views (ADG, IDG, FDG, EDEP, CDEP, and DDEP) from their current provenance view. That is, each view can be used as a simple form of a navigation operator (see Figure 7.4) that replaces the current view with the corresponding view. This allows users to bring all elements in the provenance view to the same level of granularity.

Although not shown in Figure 7.4, the navigation model also supports a select
operator that allows users to pick items within a view to display various details of the item. Selecting an item, however, does not modify the current provenance view. For example, by selecting an invocation, a user can determine the parameter values passed to the invocation, the duration of the invocation, and so on. Similarly, by selecting a connection between two invocations, a user can see the details of the data structure passed between them. For example, in Figure 7.2b, if a user selects the edge between invocation $a:1$ and $b:1$, (shown as a QLP expression), a new pop-up window would open in a browser and display the structure produced by invocation $a:1$. Selecting an edge in an actor dependency graph would similarly display the structures of the data passed between the invocations of each of the two actors. For example, in Figure 7.2a, if a user selects the output edge of actor $a$ (shown as a QLP expression), then all outputs of invocations of $a$ that were provided to invocations of actor $b$ are displayed (referred to as a “combined” view of the data structure).

### 7.2.2 Navigating Provenance Views

Besides navigating to specific, pre-defined provenance views, the navigation model also provides the collapse, expand, group, ungroup, and filter operators for constructing new views (see Figure 7.4). Given a set of these navigation operators, a new view is constructed using the `navigate` operation. If $v_i$ is the current provenance view (i.e., a provenance graph), $t$ is the trace, and \{op$_1$, op$_2$, ... \} is a set of operators,

\[
\text{navigate}(t, v_i, \{op_1, op_2, \ldots \}) = v_{i+1}
\]

returns the new provenance view $v_{i+1}$ that results from applying the navigation operators to $v_i$ over trace $t$. 
### 7.2. Navigation Model

<table>
<thead>
<tr>
<th>Operator</th>
<th>Versions</th>
<th>Effect on Current Provenance View</th>
</tr>
</thead>
<tbody>
<tr>
<td>expand</td>
<td>expand : $T \times A \rightarrow \text{Set}(I)$, expand : $T \times I \rightarrow \text{Set}(D)$</td>
<td>Replace an actor with its invocations, and an invocation with its dependencies</td>
</tr>
<tr>
<td>collapse</td>
<td>collapse : $T \times \text{Set}(I) \rightarrow A$, collapse : $T \times \text{Set}(D) \rightarrow I$</td>
<td>Replace invocations with their actor, and dependencies with their invocation</td>
</tr>
<tr>
<td>group</td>
<td>group : $T \times \text{Set}(A) \rightarrow G_A$, group : $T \times \text{Set}(I) \rightarrow G_I$</td>
<td>Replace actors with a composite actor, and invocations with a composite invocation</td>
</tr>
<tr>
<td>ungroup</td>
<td>ungroup : $T \times G_A \rightarrow \text{Set}(A)$, ungroup : $T \times G_I \rightarrow \text{Set}(I)$</td>
<td>Replace a composite actor with its actors, and a composite invocation with its invocations</td>
</tr>
<tr>
<td>filter</td>
<td>filter : $T \times Q \rightarrow \text{Set}(D)$</td>
<td>Filter flow graph according to a given query $Q$</td>
</tr>
<tr>
<td>navigate</td>
<td>navigate : $T \times V \times \text{Set}(Op) \rightarrow V$</td>
<td>Apply a set of operations to the current provenance view</td>
</tr>
<tr>
<td>standard views</td>
<td>ADG : $T \times V \rightarrow V$, IDG : $T \times V \rightarrow V$, FDG : $T \times V \rightarrow V$</td>
<td>Replace current view with actor, invocation, or flow graph view, respectively</td>
</tr>
<tr>
<td>flow-graph views</td>
<td>EDEP : $T \times V \rightarrow V$, CDEP : $T \times V \rightarrow V$, DDEP : $T \times V \rightarrow V$</td>
<td>Replace current view with expanded, collapsed, or data flow dependency view, respectively</td>
</tr>
</tbody>
</table>

Figure 7.4: Navigation model operators, where $T$, $A$, $I$, $D$, $G_A$, $G_I$, $Op$, $Q$, and $V$ are the set of traces (flow graphs), actors, invocations, dependencies, grouped actors, grouped invocations, navigation operations, QLP queries, and views, respectively.

We describe each of the navigation operators below. We assume that $w$ is a workflow consisting of actors $A$, and that $t$ is a trace (i.e., a flow graph) of a run of $w$ that contains invocations $I$, dependencies $D$, and data structures $S$. Note that a dependency $\langle n_1, i, n_2 \rangle \in D$ states that node $n_1$ was used by invocation $i$ to produce node $n_2$ where $n_1$ and $n_2$ are each part of structures $s_1$ and $s_2$ input to and output by $i$, respectively.
Expand and Collapse

Instead of displaying the entire provenance graph at the same level of granularity, the expand and collapse operators allow users to explore separate portions of the graph at different levels of detail. We consider two versions of the expand operator. Given an actor $a \in A$ of trace $t$,

$$\text{expand}(t, a) = \{i_1, i_2, \ldots\}$$
7.2. Navigation Model

returns the set of invocations \( i_1, i_2, \ldots \in I \) of a in \( t \). Alternatively, given an invocation \( i \in I \) of trace \( t \),

\[
\text{expand}(t, i) = \{d_1, d_2, \ldots \}
\]

returns the set of dependencies \( d_1, d_2, \ldots \in D \) introduced by \( i \) in \( t \), where \( d = \langle x, i, y \rangle \) for nodes \( x \) and \( y \).

The collapse operator acts as the inverse of expand. Namely, given a set of dependencies \( \{d_1, d_2, \ldots \} \in \text{Set}(D) \) generated by an invocation \( i \), where \( \text{Set}(D) \) denotes the powerset of \( D \),

\[
\text{collapse}(t, \{d_1, d_2, \ldots \}) = i
\]

returns invocation \( i \). Note that a user will typically select a single dependency to collapse, which will result in all such dependencies of the same invocation to also collapse.

Similarly, given a set of invocations \( \{i_1, i_2, \ldots \} \in \text{Set}(I) \) of an actor \( a \),

\[
\text{collapse}(t, \{i_1, i_2, \ldots \}) = a
\]

returns actor \( a \). Note that when a user selects only a single invocation to collapse, this operation will result in all such invocations of the same actor to also collapse.

To illustrate, Figure 7.2 shows the result of applying the expand operator to the actor dependency graph in Figure 7.2a, resulting in the new graph view shown in Figure 7.2b. In this example, all actors are expanded using the operator expression \( \text{expand}(\ast) \). Here we use the wildcard symbol ‘\( \ast \)’ to denote the set of all actors in the view. When expand is applied to each invocation of Figure 7.2b the view in
Figure 7.2c is returned. Alternatively, Figure 7.5 shows the result of applying the expand operator to only a portion of the corresponding dependency graph. As shown, only actor \( a \) is expanded in the actor dependency graph of Figure 7.5a, which results in the (mixed) view of Figure 7.5b. Expanding invocation \( a:1 \) in Figure 7.5b results in the provenance view shown in Figure 7.5c, which contains each level of granularity—actors, invocations, and flow dependencies—within a single graph.

**Group and Ungroup**

The *group* and *ungroup* navigation operators allow actors and invocations to be combined into composite structures. Unlike other approaches [DF08] that infer groups based on a user’s selection of “relevant” actors, the navigation model explicitly allows users to control which items should be grouped and supports both actor and invocation granularity.

We consider two versions of the group and ungroup operators. Given a set of actors \( \{a_1, a_2, \ldots \} \),

\[
\text{group}(t, \{a_1, a_2, \ldots \}) = g_{\{a_1, a_2, \ldots \}}
\]

returns a composite actor \( g_{\{a_1, a_2, \ldots \}} \) over the given set of actors. Similarly, given a composite actor \( g_{\{a_1, a_2, \ldots \}} \),

\[
\text{ungroup}(t, g_{\{a_1, a_2, \ldots \}}) = \{a_1, a_2, \ldots \}
\]

returns the set of actors corresponding to the group (i.e., ungroup is the inverse of group). Similarly, for a set of invocations \( \{i_1, i_2, \ldots \} \),

\[
\text{group}(t, \{i_1, i_2, \ldots \}) = g_{\{i_1, i_2, \ldots \}}
\]
returns a composite invocation $g_{\{i_1,i_2,\ldots\}}$; and given a composite invocation $g_{\{i_1,i_2,\ldots\}}$,

$$ungroup(t,g_{\{i_1,i_2,\ldots\}}) = \{i_1,i_2,\ldots\}$$

returns the set of invocations that comprise the group.

Figure 7.6 shows three examples of using the group operator. In Figure 7.6a, invocations of the same actor are grouped, i.e., $a:1$ and $a:2$ form one group and
invocations $b$:1 and $b$:2 form a different group. Here we use the shorthand notation $\ast$:1 and $\ast$:2 to construct these groups. In Figure 7.6b, invocations with the same invocation number are grouped, i.e., $a$:1 and $b$:1 form one group and invocations $a$:2 and $b$:2 form a different group. Similar to Figure 7.6a, we use the shorthand notation $a:\ast$ and $b:\ast$ to form the groups in Figure 7.6b. In Figure 7.6c, actors $a$ and $b$ are grouped, resulting in a composite actor with two distinct invocations. Unlike in Figure 7.6b, these invocations are of the same actor group and have different invocation numbers, whereas in Figure 7.6b two distinct groups are created. In general, forming invocation groups explicitly, as opposed to first forming actor groups and then expanding actor groups, supports grouping at a finer-level of granularity by allowing various patterns of composite invocations that are not possible to express at the actor level.

As shown in Figure 7.6, composites created by the group operator are assigned new identifiers. In addition, the inputs, outputs, and dependencies associated with grouped items are inferred from the underlying inputs, outputs, and dependencies of the invocations of the groups. For dependencies in particular, this often requires computing the transitive closure of dependencies associated with invocations of the group, e.g., as in Figure 7.6b-c.

When a group is created at the actor level, expanding the group results in a correspondingly grouped set of invocations, e.g., as shown in Figure 7.6c. These invocations are constructed based on the invocation dependency graph. In particular, each invocation group of the actor group contains a set of connected invocations, and no invocation within an invocation group is connected to any other invocation in a different invocation group. Thus, the portion of the invocation graph associated with the actor group is partitioned into connected subgraphs, and each such subgraph
forms a distinct invocation group of the actor group. Similarly, when an invocation group is expanded, this composite invocation is used in the flow dependency graph, resulting in a provenance view where dependencies are established between output and input data, without intermediate data in between. This approach allows scientists to continue to explore dependencies for grouped invocations (since the dependencies are maintained through groups).

We limit the use of the group operator such that the resulting actor and invocation dependency graphs with composites remain cycle free. Thus, grouping a set of invocations or actors should not introduce cycles in the ADG or IDG. For example, Figure 7.7 shows invocations $a:1$ and $b:2$ grouped as $G1:1$ and $a:2$ and $b:1$ grouped as $G2:1$. The invocation dependency graph that results has the output of $G1:1$ connected to the input of $G2:1$ and vice-versa, thereby resulting in a cycle between $G1:1$ and $G2:1$. The navigate operator checks to ensure that a given group operation will not result in cyclic dependency graphs.

Filter

Besides grouping and expanding provenance views, the navigation model also allows provenance views to be queried using QLP. Issuing a query results in the portion of the provenance view to be displayed that corresponds to query answer. A provenance graph is refined in this way using the filter operation of Figure 7.4. Given a QLP
7.2. Navigation Model

Figure 7.8: Exploring and managing provenance graphs: visualizing, summarizing, filtering, and navigating relevant sections of provenance views.

query \( q \),

\[
\text{filter}(t, q) = \{d_1, d_2, \ldots\}
\]

returns the set of dependencies that result from applying the query to the flow graph. The navigate operation uses these dependencies to either remove or add items to the current view. Items are added to a view if the current view is based on a more selective query.
Combining Operators

Here we describe a simple example that illustrates how the integrated environment provided by the navigation model enables users to explore provenance information by visualizing, summarizing, and filtering relevant sections of provenance views. Consider a user who is interested in exploring the provenance of an execution of the fMRI workflow of Section 3.7. Using the navigation model, the user begins by displaying the provenance of the run at the highest level of granularity, i.e., by visualizing the actor dependency graph (ADG) as shown in Figure 7.8a.

Assume that the user first decides to view the outputs generated by the workflow, which is performed by issuing the select operation over the outgoing edge of the actor Convert in Figure 7.8a. After examining the output structure, assume that the user notices that one of the data products—namely, the AtlasXGraphic—is not the expected output, i.e., the data product seems to be incorrect.

To check whether the product was generated correctly, the user decides to display the lineage of the AtlasXGraphic in more detail by navigating from the current actor dependency view to the invocation dependency view for this data item. The user creates this view by configuring a filter operator using the QLP query

\[ * \text{derived} //\text{AtlasXGraphic}, \]

which selects only those items that share dependency relationships with the data item AtlasXGraphic. The user also applies the operator expand(*) to display the invocation dependency graph (relative to the filter). After specifying these operators, the user applies the navigate function to generate and display the new provenance view, which is shown in Figure 7.8b.
While analyzing this new provenance view, the user suspects that the first invocation of each displayed actor (\(*:1\)) might have led to the incorrect \textit{AtlasXGraphic} output (e.g., based on inspecting the parameters or intermediate products produced by one or more of the invocations). To explore whether the problem is due to these invocations, the user further refines the provenance view by displaying only the first invocation of actors that share a dependency relationship with the \textit{AtlasXGraphic} data product by issuing the \textit{filter} operator using the QLP query

\begin{verbatim}
AlignWarp:1 through //AtlasXGraphic,
\end{verbatim}

which selects the lineage items that start from \textit{AlignWarp:1} and end in \textit{AtlasXGraphic}. After applying the navigate function, the resulting provenance view is shown in Figure 7.8c.

Before looking at the detailed data items and dependency relationships, the user decides to further simplify the invocation dependency graph by grouping the \textit{Reslice}, \textit{Softmean}, and \textit{Slicer} invocations into a single composite invocation. In particular, this group restricts the data items in the view to the output of \textit{AlignWarp} and the input to \textit{Convert}. The summarization operation is performed by specifying a group over the three invocations, resulting in the provenance view of Figure 7.8d.

The user is now ready to analyze the fine-grained data dependencies among the relevant portions of the provenance graph. The user first expands the resulting invocations (producing a flow dependency graph), and then applies the \textit{DDEP} view operator, as shown in Figure 7.8e. The user can now analyze the dependency relationships to verify that the relevant intermediate data products are correct and that they were correctly derived based on the dependencies (e.g., to check whether the correct warping parameters were used and that the slice was correctly generated).
Note that the dashed edge between node 11 and 18 denotes that the dependency is the result of a composite invocation. If the user wishes to further explore the dependencies represented by the composite invocation, the edge can be ungrouped by operator ungroup(G1:1) to expose the detailed dependency information, which in this case would result in the data dependency graph of Figure 3.4b.

This example demonstrates how the operations of the navigation model can provide a flexible approach for summarizing, refining, and navigating different provenance views, which is essential for users needing to explore and manage large provenance graphs.

### 7.2.3 System Architecture

The navigation model allows users to navigate from the current provenance view $v_i$ to another provenance view $v_{i+1}$ by applying a set of navigation operators $Op_{i+1}$. Once a user issues a set of navigation operators $Op_{i+1}$, the navigation system must compute a new provenance view $v_{i+1}$, which then replaces the display of the current view $v_i$. While exploring large provenance graphs, we expect users to repeatedly issue navigation operations, which would result in frequent changes to the current provenance view. Especially for large provenance graphs, it is important to ensure fast response time so that new views can be computed and displayed quickly. Here we compare different architectures and approaches for implementing the navigation model over large provenance graphs, as shown in Figure 7.9.

Each of the alternative architectures of Figure 7.9 have the following common components: a server-side Provenance Store that acts as a database repository for provenance traces; a current provenance view $v_i$ that is displayed by the Current View component; and a Navigation Engine that stores the current view, receives a new set
of navigation operations \( O_{p_{i+1}} \) from the user, computes the new provenance view \( v_{i+1} \) from user operations, and updates the current view \( v_i \) with the new computed view \( v_{i+1} \) via the \textit{Current View} component.

In the architecture of Figure 7.9a, the Navigation Engine is located at the client. To compute the initial provenance view \( v_1 \) for a workflow run, all related provenance information stored in the Provenance Store is initially loaded into a client-side Trace Information component. The Navigation Engine uses the trace stored at the Trace Information component to compute the initial view \( v_1 \) (i.e., the actor dependency graph), and this view is displayed by the Current View component. When a user issues a new set of navigation operations \( O_{p_{i+1}} \) against the current view \( v_i \), the Navigation Engine computes a new view \( v_{i+1} \) by applying the set of operators \( O_{p_{i+1}} \) to the recently stored view \( v_i \), and then updates the current view \( v_i \) with new computed view \( v_{i+1} \). This approach is largely an in-memory-based approach, i.e., the related provenance information is first loaded from the server (via the Provenance Store) to the client (via the Trace Information component). This in-memory approach can
speed up view computation, as the required provenance information is temporarily stored at the client. However, depending on the size of the trace, this approach may require significant CPU and memory resources [ABML09a] for answering queries and for storing the initial trace, data products, query results, and views. This cost also includes the time required to transfer the trace information from the Provenance Store to the client-side application, which for larger traces can take a considerable amount of time.

Figure 7.9b suggests a different architecture in which the Navigation Engine is split between the client and server. In particular, when the client-side Navigation Engine receives a new set of navigation operators $O_{P_{i+1}}$ from the user, these operations are sent to the server-side Navigation Engine (which includes the Provenance Store). The server-side Navigation Engine computes and stores the new view $v_{i+1}$ on the server and then sends the new view to the client. Thus, the architecture of Figure 7.9b differs from Figure 7.9a by storing the trace and computing and storing new views on the server, thus avoiding the storage and computation costs needed for the client in Figure 7.9a.

We can extend the approach of Figure 7.9b in Figure 7.9c by additionally minimizing the amount of information sent to construct a new view, thus (for many cases) reducing the communication costs between the client and server. In particular, in addition to computing the new view $v_{i+1}$ on the server, we also compute a set of changes (or “diffs”) $\Delta : V \rightarrow V$ between the current view $v_i$ and the new view $v_{i+1}$ such that $\Delta(v_i) = v_{i+1}$. In the normal way, each $\Delta$ consists of a set of nodes and edges that should be added to and removed from $v_i$ to generate $v_{i+1}$. Instead of returning $v_{i+1}$ (which will typically overlap with $v_i$), we only return the set of changes $\Delta$, which are used by the Current View component to display the new view. Both of the approaches
7.2. Navigation Model

in Figure 7.9b and Figure 7.9c can help to reduce the storage and processing costs of the client application in terms of computing new views for large-sized provenance graphs, since these approaches push view processing and provenance storage to the server. Also, computing and sending only the changes between views (for the cases when changes to views are small) as in Figure 7.9c can further speed up navigation response time compared to sending the entire view as in Figure 7.9b. This can be seen in even the small example of Figure 7.8, where each navigation step generally only makes small changes to the previous view. We can also further extend Figure 7.9c by sending the smaller of (i) the changes ∆ between views and (ii) the new view. In particular, if the new view is smaller than the corresponding ∆, then the server-side Navigation Engine would return the new view, whereas if the size of the changes ∆ is smaller, then the set of changes are sent.

In our prior work [BMR+08] we describe an interactive tool for browsing provenance that can display different provenance views for scientific workflow traces. The provenance browser has been integrated with the Kepler Scientific Workflow System [LAB+06], and can also be run as a standalone application. Using the provenance browser, a user can connect to a provenance store to display various provenance views of the execution trace: the dependency history view (Figure 7.1) combines data dependency and process invocation graphs (where data nodes are denoted as circles and invocations as squares); the collection history view shows the data structures input and output by invocations; and the invocation graph view 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. In 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
7.3. Related Work

Automatically recording provenance information during workflow execution is one of the important added values of scientific workflow systems over more conventional script-based approaches [MBZL09, SFC07]. However, providing techniques for effectively representing, managing, and accessing the large amounts of provenance information generated by workflow systems presents a number of technical challenges [DF08, SFC07]. The navigation model presented here helps to address a number of these challenges by providing users with an approach for exploring and viewing relevant portions of provenance information using intuitive and natural graph-based provenance representations. Our navigation model is based on a generic provenance representation scheme [ABML09a] that extends conventional approaches to support a wide range of workflow systems. Our model also combines approaches for querying (i.e., using QLP [ABML09b]) and summarizing workflow graphs (based on composites), while offering additional abstract views of provenance information (actor, invocation, and flow dependency graphs) and the ability to navigate between views (e.g., using expand and collapse). The remainder of this section describes related work and compares our work to existing approaches.

Scientific workflow systems are being used in many scientific domains, and many approaches have been proposed recently for representing workflow provenance (e.g.,
Most existing approaches for representing provenance do not consider workflow computation models that work over structured data, including XML. Standard provenance representation schemes (e.g., [MFF+08, HBM+08, BD08, BBDH08] among others) largely assume that workflow models are based on transformation semantics in which each workflow step consumes all input data and produces entirely new output data. Alternatively, workflow models that work over structured data (e.g., [BMR+08, Slo07, MBZG08, Tom06]) often employ update semantics, where only a portion of an incoming XML stream is modified by each workflow step. Our navigation model is based on a generic model of provenance that subsumes conventional approaches for representing workflow provenance while supporting more advanced workflow computation models permitting multiple invocations of processes (e.g., for pipelining and loops), structured data, and update semantics [ABML09a].

Current approaches for exploring workflow provenance are based on visualizing entire provenance graphs or specific views of these graphs, such as data and invocation dependency graphs [BMR+08, SFC07, HC07, MFF+08, MLA+08]. In these approaches, provenance graphs are typically displayed at the lowest level of granularity. Some systems further divide provenance information into distinct layers. For example, myGrid [ZGST08] divides provenance into data, process, organisational, and knowledge levels; VisTrails [CFS+06, SFC07, BCS+05] divides provenance information into workflow evolution, workflow, and execution layers; Redux [BD08] divides provenance into runtime execution, data instantiation, abstract service, and service instantiation layers; and the Provenance Aware Storage System (PASS) [MRBH+09] divides provenance into data and process layers. In all of these approaches, however, these levels are largely either orthogonal or hierarchical, whereas the provenance views...
7.3. Related Work

supported by our navigation model (i) combine both hierarchical abstractions (i.e., ADGs, IDGs, and FDGs) with (ii) the ability to seamlessly navigate between these different levels of granularity, while (iii) allowing users to summarize, group, and filter portions of these views to create new views for further exploration of relevant provenance information.

Unlike standard provenance approaches, the Zoom*UserViews system [CBD06, BBDH08] provides a mechanism for defining composite actors to abstract away non-relevant provenance information. The basic approach is to allow users to select one or more “relevant” actors from a workflow specification graph, and based on these selections, the system creates associated composite actors that contain at most one relevant actor. The composite actors are constructed in such a way as to maintain certain dataflow connections, thereby generating a workflow over the composites that is similar (in terms of dataflow) to the original. However, unlike in our approach, users of the Zoom*UserViews system cannot explicitly define their own composites, and composition is defined only at the actor level (where each actor is assumed to have at most one invocation). Our approach also maintains grouping across views (including the ability to ungroup composites within these views), maintains the original data dependencies (i.e., dependencies within composites are maintained, unlike in general within the Zoom*UserViews approach), and we support a more general provenance model that explicitly handles structured data.

Our navigation approach is inspired by and has similarities to those proposed previously for exploring object-oriented and XML databases, where graphical environments were developed that allow users to “drill-down” from schema to instances and navigate relationships among data. For example, PESTO [CHMW96] provides an integrated browsing and querying environment that allows users to employ a “query-
7.3. Related Work

in-place” paradigm for exploring the contents of object databases. In particular, PESTO allows users to mix navigation and query in which queries can be issued relative to a position in the database reached through navigating object relationships. Similarly, in Blended Browsing and Querying (BBQ) [MP00], a graphical user interface is provided that supports both browsing and querying of XML data. Like PESTO, querying in BBQ is schema driven, requiring users to know the details of the (Object-Oriented or XML) schema prior to issuing queries. Also, a standard XML-based web-based navigation and visualization approach tailored to clinical provenance information is proposed in [DCR06]. In contrast, provenance information is largely schema-free, i.e., the information contained within an ADG, IDG, and FDG is not constrained by an explicit schema, and queries in our model are posed directly against the items contained within these views (or generally the flow graph). In addition, our provenance model is considerably more specialized than the more generic data models supported by PESTO and BBQ, resulting in navigation operators (such as expand and group) that are tailored specifically to provenance information.

Finally, our navigation model is the first approach that we are aware of that combines navigation, abstraction (through composition), and query capabilities. A number of systems allow users to query provenance information, however, these approaches largely rely on physical representations of provenance information [DF08] (e.g., relational, XML, or RDF schemas), where users express provenance queries against these schemas using corresponding query languages (i.e., SQL, XQuery, or SPARQL). Provenance queries, however, often require computing transitive closures over dependency relations, and expressing such queries using standard approaches is typically done using recursion or stored procedures [HA08, CJR08, ABML09a]. Expressing such queries is both cumbersome and error-prone, and requires considerable
user expertise. Instead, high-level languages such as QLP provide a separation between the logical provenance model and its underlying physical representation, which allows for the use of different representation schemes and additional optimization techniques. Our approach, in particular, automatically translates QLP queries to equivalent relational queries expressed against the provenance storage schemes described in [ABML09a]. Standard approaches for querying provenance information (e.g., [Car08, HBM+08, ZL08, BBDH08]) return sets of nodes (either sets of data items or process invocations) as the query result. This approach requires additional steps (queries) to reconstruct causal relations among nodes within a query answer. Instead, QLP is closed under lineage relations, where answers to lineage queries are sets of lineage dependencies (edges) forming provenance subgraphs, and thus query results are “provenance preserving”. This approach has a number of advantages, e.g., for supporting provenance views, incremental querying, and for supporting visualization [DF08, BMR+08].

7.4 Conclusion

We propose an approach to address a number of open issues in effectively exploring large provenance information generated from complex scientific workflows. Specifically, we define a novel navigation model for scientific workflow provenance that consists of operations for creating, refining, and switching between different provenance views. The navigation model provides an integrated approach to: (i) create abstract views of provenance information (actor, invocation, and flow dependency graphs); (ii) seamlessly navigate between these views; (iii) summarize portions of views through grouping actors or invocations; and (iv) filter (or query) provenance views using QLP, a high-level provenance query language. We also present different architectures
7.4. Conclusion

for efficiently navigating large provenance graphs against an underlying provenance database. Our navigation model is the first approach that we are aware of that provides capabilities of visualizing, navigating, summarizing, and querying provenance views in an integrated environment. The approaches described here extend our prior work on browsing and query provenance information by providing explicit navigation operations and views for abstracting and summarizing provenance graphs. Combined with the provenance browser, these approaches can provide a powerful environment for scientists to explore and validate the results of scientific workflows, especially those that involve large and complex data sets and large numbers of interconnected processes.
Chapter 8

Provenance Browser

A key advantage of scientific workflow systems over traditional scripting approaches is their ability to automatically record data and process dependencies introduced during workflow runs. Scientific workflow provenance is typically represented using data and process dependency (i.e., causal) graphs [MFF+08, DF08], which can be used by scientists to better understand, reproduce, and verify scientific results. Provenance graphs may be large due to the complexity of the workflow, the size of input data sets, and the number of intermediate data sets produced, which can make it difficult for users to effectively find and view relevant information; and the ability to effectively store, query, and visualize provenance graphs has been identified as a significant challenge [DF08, HA08, ABML09a].

We address these challenges through a provenance browser that combines visualization, navigation, and high-level graph queries. Typical provenance queries are exploratory and involve finding some or all of the data and process dependencies that led to the creation of one or more data products [MLA+08]. Posed over dependency relations, these queries often require transitive closures and selection conditions on lineage paths. Current approaches for querying provenance information are
8.1 Provenance Browser Architecture

largely based on physical data representations [DF08] (e.g., relational, XML, or RDF schemas) in which users express provenance queries through corresponding query languages (SQL, XQuery, or SPARQL). For most users, expressing these queries against physical provenance schemas requires considerable expertise and is cumbersome even for simple queries [DF08]. The complexity of these queries also creates challenges for efficient query evaluation [HS08].

The Provenance Browser can display different provenance views for scientific workflow traces and that supports provenance queries through a novel Query Language for Provenance (QLP) [ABML09b]. QLP provides specialized operators for querying provenance graphs that, unlike other languages which return sets of nodes (e.g., [Car08, HBM+08]), are closed under lineage relationships (i.e., QLP lineage queries return sets of lineage dependencies forming provenance subgraphs). The browser provides an integrated environment where users can issue QLP queries against workflow traces, and query results can be displayed, navigated, and further queried. We also employ novel query and storage optimization techniques that makes querying real-world provenance information within the browser practical and efficient.

The provenance browser provides users with an interactive application for accessing provenance information generated by scientific workflow runs. We describe the main features of the browser below.

8.1 Provenance Browser Architecture

The basic architecture of the provenance browser is shown in Figure 8.1, and the basic user interface is shown in Figure 8.7. The provenance browser has been integrated with the Kepler Scientific Workflow System [LAB+06, BMR+08], and can also be run as a standalone application. The browser works over workflow traces represented in
Figure 8.1: Basic provenance browser architecture.

a generic XML format. Given a trace file, a set of pre-processing steps are applied to the trace prior to storage in a relational database (in our current implementation, PostgreSQL). These pre-processing steps perform storage reduction techniques (based on factorization) over the data lineage graph of the workflow trace as described in Section 4. Using the provenance browser, a user can connect to a provenance store to select traces to view, issue QLP queries against the trace, and then display, navigate, and further query these results. As shown in Figure 8.1, QLP queries are parsed, optimized, and rewritten to corresponding SQL queries expressed against the provenance database (see Section 6). Optimized and translated SQL queries return sets of lineage edges as query results from which the browser constructs and displays the corresponding lineage graph.

Constructing Provenance Views. Two types of provenance graphs are computed for displaying provenance information within provenance browser.

An actor invocation graph consists of nodes representing actor invocations and
8.2. Displaying Provenance Views

Directed edges representing the (immediate) invocation dependencies.\footnote{We assume that invocation graphs are acyclic.} An edge $B:1 \rightarrow A:1$ in an invocation graph states that the first invocation of actor $B$ was (immediately) invoked after the first invocation of $A$. These edges typically imply that $A:1$ produced an item used by $B:1$, or more generally, $A:1$ modified a collection that an output of $B:1$ depended on. Note that depending on how actor read-scopes are configured within a workflow, it is possible that certain actors during a run will be executed simultaneously (i.e., in parallel) such that no invocation dependencies exist between them, resulting in a partially ordered set of invocations.

A data dependency graph consists of nodes representing data items and directed edges representing (immediate) insertion dependencies. Each edge is additionally labeled by the corresponding insertion invocation. An edge $D_3 \xrightarrow{A:1} \{D_1, D_2\}$ states that data item $D_3$ was produced by the first invocation of actor $A$ from data items $D_1$ and $D_2$. Thus, $D_1$ and $D_2$ were “input” to $A:1$, i.e., they were within the read scope of the invocation. Note that unlike other approaches, data dependency graphs in our provenance model can distinguish items that depend only on a subset of the data input to an invocation. This is often the case, e.g., for actors implemented by non-strict (i.e., “streamable”) functions.

8.2 Displaying Provenance Views

As shown at the top of Figure 8.2, 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 (Figure 8.2) combines data depen-
Figure 8.2: A phylogenetics workflow in Kepler with dependency view shown in the provenance browser.

dency and process invocation graphs (where data nodes are denoted as circles and invocations as squares); the *collection history* view (top of Figure 8.3) shows the data structures input and output by invocations; and the invocation graph view (bottom of Figure 8.3) shows process dependencies. Users can select and display the details of each item in a view (including the underlying data represented by a token, e.g., the particular sequence alignment or phylogenetic tree), and each of the views are synchronized. For example, the selection of a data item in the dependency history also selects the corresponding item in the collection history.
8.3 Navigating Provenance Views

We have also implemented an approach within the browser that allows users to incrementally step forward and backward through execution history, incrementally displaying (i.e., revealing or hiding elements, depending on navigation direction) the corresponding portion of the collection and data-depenency history. This feature allows users to start from the input of the workflow and incrementally move forward through actor invocations to the final output. Similarly, it is possible to start at the output and navigate to the input, as well as move forward or backward at any point in between. Figure 8.5 shows both the data collections and invocations for a workflow at two distinct execution points, where the views on the right of the figure were created from those on the left by moving one step forward in the execution history. The
views in Figure 8.5 are especially useful for analyzing how the structure of collections evolved throughout a workflow run; whereas the view of Figure 8.4 more explicitly shows the steps and dependencies involved in generating data products.

We construct the navigation history for a trace from the invocation graph by creating a sequence of snapshots and “wavefronts”, which induce an ordered partition of the graph. Each wavefront denotes a set of invocations that could have been executed simultaneously (e.g., under an optimal workflow schedule). Thus, the invocations within a wavefront are independent of each other. The wavefronts are constructed
8.3. Navigating Provenance Views

Kepler/pPOD

- Collection and invocation view
- Incrementally step through execution history
- Actor invocation graph shows pipelining, implicit branches

Figure 8.5: The collection structure and invocation graph for a run of the workflow displayed in the provenance browser: The resulting collection structure after the first invocation of an actor is shown on the left; and the collection structure and invocation graph resulting from advancing one step through the execution history, i.e., after the first invocation of the next actor is shown on the right.

From an invocation graph as follows. The initial wavefront is created first by selecting all source invocations (i.e., which do not depend on other invocations). These source invocations are then removed with their incident edges from the graph (we can now view these invocations as completed) to obtain the next wavefront. This process is repeated until all invocations have been reached by a wavefront. Thus, moving forward (backward) through the trace implies moving between wavefronts/snapshots. The current wavefront represents the set of “executing” (most-recently invoked) actors. These invocations are then used to filter the collection- and dependency-history views (Figure 8.5). This feature allow users to step forward and backward (“VCR-style”) through the execution history to display corresponding portions of the XML structures and data dependencies.
8.4 Incremental Querying

The provenance browser allows users to issue QLP queries against lineage graphs. The provenance browser provides a separate query window, as in Figure 8.6, for users to issue QLP queries. The users can select any traces stored in the database to visualize the complete trace view traces stored in the database. The users can visualize,
Figure 8.7: The data-dependency view and navigation panels of the provenance browser (top), and QLP query results displayed in the browser (bottom).
navigate, and issue new queries over the query results. The browser automatically captures the sequence of queries that resulted into the current view, and is displayed in the query window. Also, users can navigate and visualize the queries, and query results that were issued since the browser was launched or started. All of these features are demonstrated in Figure 8.6, which shows the query window along with the query result displayed as dependency view.

In addition, the users can issue incremental queries over provenance views. For example, the top of Figure 8.7 shows the result of evaluating query Q1 "* derived *" returning the set of lineage relations shared between all nodes. Users can further execute queries via the QLP shell, which are executed over the current portion of the graph (alternatively, prior queries can be modified and rerun). For example, the bottom left of Figure 8.7 shows the result of query Q2 "* derived //AtlasXGraphic" issued over the result of query Q1. Similarly, the result of Q2 is further queried in Q3 "#Softmean through #Convert derived //AtlasXGraphic" returning the subgraph in the bottom-right of Figure 8.7. By incrementally querying provenance graphs in this way, users can more easily inspect and explore relevant portions of large provenance graphs—which contrasts with more static approaches that display entire lineage graphs containing, e.g., hundreds or thousands of nodes and edges.

8.5 Storage and Query Optimization

The left side of Figure 8.8 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 “s₁.. s₂”, 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). Thus, to make the provenance browser feasible, we have developed novel storage [ABML09a] and query optimization techniques [ABL10b], which are exploited by the provenance browser.

Efficient evaluation of QLP path queries is closely tied to how lineage relations are stored. In the ICP approach, immediate edges and their transitive closure are stored using “pointer-based” reduction techniques [ABML09a]. As shown in Figure 8.8, the space required for storing lineage relations in ICP is considerably less than for IC, and in many cases even less than I (due to the reduction techniques). By reducing storage size, evaluating lineage queries in two steps (computing nodes and then edges), and through pruning and temporary materialization of views and subquery results [ABL10b], query execution time can also be significantly reduced. As shown in Figure 8.8, employing these techniques reduces query execution time to less than 100 ms, making common (but relatively complex) graph queries practical within the provenance browser.
8.6 Conclusion

Provenance browser is an interactive application for visualizing and querying data dependency (lineage) graphs produced by scientific workflow runs. The browser allows users to explore different views of provenance as well as to express complex and recursive graph queries through a high-level query language (QLP). Answers to QLP queries are lineage preserving in that queries return sets of lineage dependencies (denoting provenance graphs), which can be further queried and visually displayed (as graphs) in the browser. By combining provenance visualization, navigation, and query, the provenance browser can enable scientists to more easily access and explore scientific workflow provenance information. To the best of my knowledge, the provenance browser is the only application of its kind that combines multiple views of provenance information, support for navigation, and a high-level, closed graph query language.
Chapter 9

Conclusions

This work proposes a framework that addresses a number of open issues in managing provenance of nested data in scientific workflows: from efficient representation (model), to efficient storage, to efficient querying, to visualization, and summarization of relevant provenance information. This conclusion presents a summary of completed contributions.

9.1 Summary of Contributions

Broadly stating, the contributions of this work include (1) a collection-oriented approach to model provenance of nested data, along with its formalism; (2) a set of storage strategies and reduction techniques to decrease the size of provenance store, update time, and query time; (3) a novel high-level query language of provenance (QLP) that allows users to express complex provenance queries easily and concisely; (4) a set of optimization techniques to efficiently evaluate provenance queries; (5) a novel navigational model for provenance that provides an integrated approach to visualize, navigate, summarize (or abstract), and query provenance views; and (6)
9.1. Summary of Contributions

development of the Provenance Browser, a provenance visualization tool that allows
users to store provenance traces in relational database systems; and users can view,
navigate, and query various provenance views.

In particular, the contributions of this work are as follows:

1. **A unified provenance model.** A logical model of provenance that subsumes
   conventional approaches for representing workflow provenance is described. The
   provenance model supports more advanced workflow computation models that
   permit multiple invocations of processes (e.g., for pipelining and loops), struc-
   tured data, and update semantics. A formal description of the provenance
   model is given in Chapter 3 and published in SSDBM09 [ABML09b].

2. **A set of storage strategies, and reduction algorithms.** The main contribu-
   tions in this area are the development and comparison of different strategies
   for efficiently storing, updating, and querying provenance graphs expressed in
   our unified model using a relational database system. We define: (i) inference
   techniques for *expanding* and *collapsing* the implied data and process dependen-
   cies within a trace; (ii) *reduction techniques* for minimizing the size of prove-
   nance storage for direct and transitive data and process dependencies (these
   techniques are generic and applicable to both hierarchical and non-hierarchical
   data structures); and (iii) *relational views* for recovering all implied and re-
   duced dependencies generated by the inference and reduction techniques. We
   compare trade-offs in terms of provenance storage size, update time, and query
   response time, and show using real-world and synthetic provenance traces that
   our techniques can improve each of these. Storage strategies, and reduction
   algorithms are presented in Chapter 4 and have been published in EDBT09
   [ABML09a].
3. **A specialized provenance query language.** A high-level Query Language for Provenance (QLP) is proposed, which provides specialized constructs for expressing both structural (i.e., parent-child) and lineage (i.e., data dependency) queries, and their combination (“hybrid” queries). QLP allow users to formulate provenance queries concisely by using simple query expressions. QLP is a closed query language, in the sense that results are provenance preserving, i.e., result of a provenance lineage graph is a sub-graph that can be queried further. QLP is presented in Chapter 5 and has been published in SSDBM09 [ABML09b].

4. **Optimization techniques for efficiently evaluating provenance queries.** A formal semantics for QLP constructs is described, and a set of novel query optimization techniques are developed that leverage the lineage-graph reduction approaches described in Chapter 4. The approach simultaneously reduces the amount of storage required to represent scientific workflow provenance without negatively affecting query performance. The optimizations can also be used in more general settings to efficiently answer a broad range of path queries over labeled, acyclic digraphs. Unlike using standard approaches (which are often based on in-memory implementations, e.g., [ZL08, HS08, LHL+98]), answering QLP queries with our optimization techniques over a relational database system is both practically feasible and scales linearly with the size of provenance information and query complexity. The semantics of QLP and our query optimization techniques are presented in Chapter 6 and have been published in EDBT10 [ABL10b].

5. **A provenance navigation model.** A novel navigation model is proposed that allows users to explore and view relevant portions of provenance information
9.1. Summary of Contributions

using intuitive and natural graph-based provenance representations. The navigation model provides an integrated approach to: (i) abstract views of provenance information (actor, invocation, and flow dependency graphs), (ii) seamlessly navigate between these views, (iii) summarize portions of views through the grouping actors or invocations; and (iv) filter (or query) provenance views using QLP, a high-level provenance query language. This navigation model is the first approach to the best of our knowledge that combines navigation, abstraction (through composition), and query capabilities. The navigation model is described in Chapter 7 and has been published in WORKS09 [ABL09].

6. Provenance Browser. The relational schema for various storage strategies, reduction algorithms, relational views over different schemas to query provenance information, a parser for converting QLP queries into SQL queries over relational storages, and optimization techniques to efficiently evaluate provenance queries have all been implemented and integrated into the Provenance Browser, a provenance visualization and querying tool. The Provenance Browser has been successfully integrated into the Kepler scientific workflow system. From the integrated environment of Kepler, the users can now not only develop and execute scientific workflows, but also efficiently store the provenance information into a relational database, browse the provenance information via meaningful provenance views, and query the these views through concise QLP expressions. The Provenance Browser is described in Chapter 8 and has been published in ICDE10 [ABL10a].
9.2 Future Work

I plan to continue my research and development activity in the area of provenance. Managing large provenance information is a rich field with the potential to be exploited for numerous applications. While a number of scientific workflow systems support data provenance, they primarily focus on collecting and querying provenance for single workflow runs. Scientific research projects, however, typically involve many interrelated workflows where data from one or more workflow runs are selected and used as input to subsequent runs. Managing provenance of multiple interconnected workflow runs is still an unexplored area. I am working to extend my doctoral results to build a provenance framework that would allow scientists to easily store, interconnect, organize, query, visualize, and explore comprehensive provenance information for multiple workflow runs, together with input and derived data products used within research projects.

Moreover, as business workflows have been existing for quite a time, and now with the substantial work on provenance management in the scientific workflows community, analyzing provenance systems can enable the re-use of existing techniques and also shed light on the necessary extensions. I feel that my industrial experience in developing e-business applications at Oracle, and the recent research and development in scientific workflows, provides me unique opportunities to analyze business and scientific workflows.

I am also interested in integrating workflow and database provenance, because workflows can operate on data inside a database, and can have their own provenance. We need a model to capture the interaction between database provenance and workflow provenance.
9.3 In Conclusion

This work is largely motivated by the need for managing provenance of scientific workflow systems. Provenance of scientific workflows is of paramount importance to support reproducibility, sharing and validation of scientific experimental results. The provenance management problem concerns about the efficiency and effectiveness of the recording, representing, storing, querying, visualizing, and summarizing large provenance graphs. Specifically, this work addresses research problems to efficiently represent multi-version nested data structures that include labeled node-level lineage relationships; efficiently store direct and transitive dependency relationships; allow users to easily express path-based lineage query expressions; optimize evaluation of path-based lineage queries; and explore and summarize relevant portions of directed acyclic graphs. This work addresses these generic research problems in the context of scientific workflow provenance, and our approaches and results are not dependent on any particular workflow system.
Bibliography


