Fine-grained and efficient lineage querying of collection-based workflow provenance

Paolo Missier
University of Manchester
School of Computer Science
Manchester, UK
pmissier@cs.man.ac.uk

Norman W. Paton
University of Manchester
School of Computer Science
Manchester, UK
norm@cs.man.ac.uk

Khalid Belhajjame
University of Manchester
School of Computer Science
Manchester, UK
khalidb@cs.man.ac.uk

ABSTRACT

The management and querying of workflow provenance data underpins a collection of activities, including the analysis of workflow results, and the debugging of workflows or services. Such activities require efficient evaluation of lineage queries over potentially complex and voluminous provenance logs. Näive implementations of lineage queries navigate provenance logs by joining tables that represent the flow of data between connected processors invoked from workflows. In this paper we provide an approach to provenance querying that: (i) avoids joins over provenance logs by using information about the workflow definition to inform the construction of queries that directly target relevant lineage results; (ii) provides fine grained provenance querying, even for workflows that create and consume collections; and (iii) scales effectively to address complex workflows, workflows with large intermediate data sets, and queries over multiple workflows.

1. INTRODUCTION

In many areas of science, the use of partially or fully automated workflows has been shown to accellerate scientific investigation, often leading to interesting new results that are only possible thanks to large-scale, automated data retrieval and analysis [4, 15]. However, the very ability of these workflows to automatically process large volumes of scientific data, combined with the complexity of the workflow structure, makes it increasingly difficult for scientists to fully understand the results of workflow executions. This, in turn, limits the usefulness of the process itself, as only experimental results that can be shown to be correct can be used to support scientific claims.

The provenance of a workflow [30] is a trace of all the data transformations that occur during one or more of its executions, or *runs*, and can be used to conduct *post mortem* analysis of the workflow results. Provenance traces can be used to answer queries on the *lineage* of a data product, that is, the input data and the sequences of transformations that led to that product during a run. Such queries are a stepping stone towards helping users understand and justify complex workflow results. Typical questions that users would like to pose to provenance traces include those proposed for the first three *provenance challenges*[1]. For example: a workflow loads data from files into a database, and then performs some processing on the data. It turns out that the database contains unexpected values. Provenance questions include, among others, whether the appropriate checks were performed by the workflow, what results they produced, and which input files were used for the loading. Traces have also been shown to be useful for other types of analysis, for example to determine whether parts of a workflow can be reused [29], to debug errors in the results, and to compare the effect of different versions of the same workflow [12].

Efficiently storing provenance graphs and answering lineage queries can be challenging, however. As workflows are specified using graphs, provenance traces are themselves naturally modelled as graphs, in which nodes represent instances of a processor invocation, and arcs represent data dependencies between processor invocations. Queries on the traces therefore naturally involve recursive traversals of the graph. Furthermore, since a single provenance trace keeps track of all intermediate data products of a workflow run, and traces accumulate over many runs, provenance databases can be large [13]. The problem of automatically capturing provenance traces, and efficiently storing them, has been addressed in recent research [14, 26, 11]. In particular, Heines and Alonso [17] propose an encoding of provenance graphs that is both space- and query-efficient, at the cost of an initial, offline graph transformation.

The work presented in this paper stems from two observations. Firstly, in many practical cases, users are only interested in selected elements out of the entire available lineage data, which is often noisy because it captures the many format and other trivial transformations undertaken by the workflow. Indeed, recent research [6] in this area investigated ways to support abstractions and modularisation of complex workflows, in order to hide some of the uninteresting details of complex workflows from the users.

Secondly, the use of collections of values, for example tree-structured documents or lists, is increasingly widely used to structure complex relationships among related pieces of information that are processed together by the workflow. Indeed, the use of collections as a first-class data type in sci-

[1]The provenance challenge wiki is at `http://twiki.ipaw.info/bin/view/Challenge/` (last accessed June 17, 2009).

entific workflows has been advocated as a way to better meet the needs of non-expert users to model e-science data [25]. Several scientific workflow systems, notably Kepler [22, 24] and Taverna [27, 31, 18], incorporate primitives for the manipulation of collections.

The following, real-life example illustrates both these points. Consider the Taverna workflow in Fig. 1[2]. This bioinformatics workflow takes one or more user-supplied collections of gene IDs, and uses as input the KEGG database[3] to retrieve the list of all metabolic pathways in which all of the genes *in each of the lists* are involved. It also produces a list of pathways in which *all* the genes in the input are involved. In this use case, provenance can be used to answer the natural question, "why is this particular pathway in the output?". In other words: "which of the input lists of genes is involved in this pathway"? In this case, thanks to the disciplined use of collections, the lineage query returns a fine-grained answer, namely "the pathways in sub-list i in `paths_per_gene` depend only on the genes in the corresponding sub-list i in `list_of_geneIDList`, while all pathways in `commonPathways` depend on all input genes". This information is particularly useful to users of the workflow *other than* the original designers, as it provides them with insight into the processing logic, that would otherwise be difficult to obtain. It also enables different views over the results to be constructed: the workflow output can be complemented by a query over the provenance log.

To answer this type of questions, the provenance management system must be able to keep track of data dependencies at the finest level available. Although desirable, however, fine-grained data dependencies are not always available (quite simply, processors that transform lists into new lists destroy the granularity of provenance), unless we assume that processors expose some of their internal behaviour. While others have studied the consequences of this assumption [1], in our work we assume we have no knowledge regarding the semantics of the data transformations performed by specific processors, which we treat equally as *black boxes* [8].

1.1 Scope and contributions

In the previous example, we have assumed that the user is only interested in selecting the inputs that are relevant for one specific output, rather than in looking at all the intermediate results computed by the workflow. We refer to these lineage queries as *focused*. In this paper we address the problem of answering focused and fine-grained lineage queries on large traces efficiently and in a scalable way. The specific contributions of this paper are: (i) a data model for fine-grained provenance that takes into account collection-based workflow data processing; (ii) a functional formulation of data-driven, collection-oriented workflow computation, which reflects the semantics of the Taverna workflow model; (iii) an efficient and scalable algorithm for answering focused lineage queries, which exploits the semantics of the computational model (previous point) to replace the repeated and costly traversals of the provenance trace with a less expensive traversal of the workflow specification graph. The algorithm also exploits the fine granularity of the provenance trace, when available as in the example above, to provide precise answers without sacrificing query performance. Finally, (iv) we offer an experimental evaluation of the approach, by comparing it with a baseline graph traversal algorithm, and we assess its benefits in a variety of practical settings. The main consequence of our approach is that our query response times scale with the size of the workflow specification graph, rather than with the size of the provenance traces. Furthermore, this result extends to queries that span multiple workflow runs, i.e., multiple traces.

[2]The workflow is available from the myExperiment website, please see `http://www.myexperiment.org/workflows/778`.
[3]KEGG is available at `http://www.genome.jp/kegg/`.

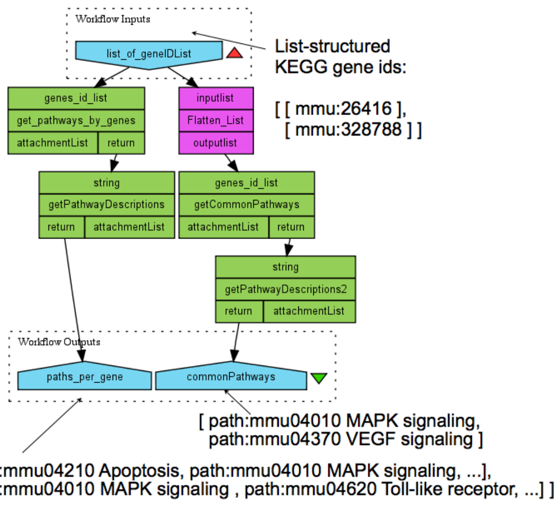


Figure 1: Example Taverna workflow for bioinformatics

Note that we are making two claims here, the first of fine granularity of provenance information, and the second of efficiency. As we show later in the paper, they are tightly connected, as the latter depends on the semantics of the workflow model that gives us the former. We note explicitly, however, that in this paper we are not concerned with the optimization of the space required to store a query trace. Also, we are going to focus on the common case of *structural* lineage queries, i.e., queries that predicate on attributes found in the workflow graph. This means that a query that explicitly predicates on the presence of a specific value on the trace, for example, can still be answered using a standard graph traversal technique, but would not benefit from our approach.

The algorithm presented in this paper is fully implemented and is soon to be released as part of the provenance management component of the Taverna workflow system[4]. The idea of using the workflow specification as an index into the provenance trace, however, can be generalized to all those systems, for example the Kepler system mentioned earlier, for which intensional rules that describe inverse transformations through a processor, i.e., from the outputs to the corresponding inputs, can be formulated.

1.2 Related work

The problem of tracing fine-grained data lineage has been studied in the past. Woodruff and Stonebraker [33] noted that, for many scientific applications, the space cost of stor-

[4]Taverna is freely available at `http://www.taverna.org.uk/`.

ing the metadata required to trace lineage at a fine grain, for example in imaging applications, is often prohibitive in practice. As a remedy, they propose a general model for the intensional, but approximate inversion of the processing steps, where a *weakly invertible* function f has an associated weak inverse f^{-w} which maps the output of f to the input of f, but is not always accurate. They apply the model to the problem of tracing granular lineage of images processed by workflows. Albeit applied to generic collections values, our algorithm is based on a similar fundamental intuition, i.e., of inverting processor transformations using intensional rules. However, in our case the inverse is accurate.

The importance of tracking lineage over collection values has been pointed out in the context of phylogenetics, prompting an extension to the Kepler workflow system, called pPOD [10]. The extension makes use of the COMAD model described by Bowers *et al.* [9] which, among other features, supports nested data collections. Although the data model for collection-oriented provenance is similar to the one we present in this paper, provenance queries are computed by recursion directly on the model (they are implemented in Prolog), in a way similar to what we call here the naïve approach, and they are aimed more at demonstrating the feasibility of the approach than its scalability to very large traces. Similar query models based on recursive queries on the lineage graph are used in Trio [5] and GridDB [21].

Anand et al. [1] have recently proposed an efficient storage model for dependencies that include nested data collections. The main connection with our work is the reliance of the model on some form of intensional knowledge to achieve efficiency, in their case on explicit declarations of element-level dependencies between input and output collections. This is where the similarity ends, however. Indeed, the paper investigates the effect of removing the *black box* assumption, which we retain, regarding the fine-grained transformations performed by workflow processors.

Our notion of focused lineage queries is related to, but takes a more simplified approach than, the *Zoom* model proposed by Biton et al. [6, 7]. The Zoom*UserView system allows the specification of additional levels of abstraction over raw provenance, known as *User views*, in order to reduce the complexity of provenance information. These are user-selected aggregations of adjacent processors in the graph, which form a new virtual workflow. In contrast, in our model we rely on the workflow nesting model offered natively by Taverna, and to this we add the option for users to focus their queries by selecting a subset of relevant processors. We regard all of [1], [6], and [7] as complementary to our work, in the sense that our approach can be combined with additional, explicit annotations and further abstraction models to deliver precise and selective lineage data to users.

The distinction between multiple *layers*, of views, over a workflow definition has been proposed in other contexts, namely by Scheidegger et al. [28] in the context of the VisTrail workflow manager, where a distinction is made between the execution, specification, and evolution of a workflow definition, as well as by Barga et al. [3]. In this paper we make use of a similar distinction to define specific conditions that the workflow model must meet in order to exploit the specification layer for answering provenance queries. In addition, we show how these conditions are met by the Taverna model, and show experimental evidence of the corresponding savings in query execution times.

The scheme for encoding provenance graphs, proposed by Heinis et al. [17] and already mentioned, is interesting in that it avoids recursive queries over the dependency graph, at the cost of an initial graph transformation process. While the encoding scheme is indeed applicable to our dependency graphs, i.e., as a way of compressing the traces, we observe that in our approach recursive queries are only applied to the static workflow graph, which is much smaller than any realistic dependency graph.

Finally, two recently proposed, more general types of query languages are relevant in our context. Firstly, the *GraphQL* [16] language uses graph patterns as their way of expressing general graph queries. Although pattern matching over graphs reduces to graph isomorphism, a known NP-complete problem, [16] show promising performance results for an optimized implementation of graph pattern matching. We plan to test *GraphQL* to our lineage query problem in the future. Secondly, [23] shows good query processing times for *QLP*, a query language for provenance graphs that includes transitive closure operators. The language implementation, however, is specific to the provenance architecture described in [1] and thus not readily applicable to our data model.

1.3 Paper organisation

We begin in Section 2 by introducing the definition of our reference Taverna workflow model, the notation for fine-grained provenance traces, illustrated with the help of the example of Fig. 1, and our definition of lineage queries in this setting. In Section 3 we present our original formulation of the Taverna list processing model, and use it to describe our main result on efficient lineage query processing. Our experimental evaluation is in Section 4, followed by conclusions (Sec. 5).

2. DATAFLOW AND LINEAGE MODELS

In this section we introduce the models that underpin our approach: the Taverna dataflow computation model, a model for provenance traces, and a lineage query model. In particular, provenance traces can be described in terms of a provenance graph, along the lines of previous work [5, 17], which we extend to capture transformations that involve individual elements within nested data collections. Lineage queries are defined in terms of traversals of the provenance graph.

2.1 Dataflow model

An informal description of the computational model for Taverna dataflows is sufficient for the purposes of this paper. A complete and formal description of the Taverna semantics can be found in [31] and in [18].

A dataflow specification is a directed graph $D = (N, E)$ where a node $\langle P, I_P, O_P \rangle \in N$ represents a processor P, i.e., a software component, for instance a service, with a set of *ordered* input ports I_P and output ports O_P. To avoid ambiguities, we will denote the ports $X \in I_P \cup O_P$ as $P : X$. Note that a processor can also be a dataflow itself, i.e., the model accounts for a hierarchy of nested dataflows.

Every port X has a declared type, denoted $\mathit{type}(X)$, which is either one of a set S of basic types, or a list $\mathit{list}(\tau)$ where $\tau \in S$ or $\tau = \mathit{list}(\tau')$ for some type τ'. Thus, a value can be an arbitrarily nested list, for example $\mathit{type}([[\text{"foo"}, \text{"bar"}], [\text{"red"}, \text{"fox"}]]) = \mathit{list}(\mathit{list}(\textbf{string}))$. To refer to an element within a nested list v, we adopt the standard element accessor notation for k-dimensional arrays, $v[p_1 \dots p_k]$. A *binding* $\langle P : X[p_1 \dots p_k], v\rangle$ denotes element $v[p_1 \dots p_k]$ of a value v that is bound to port $P : X$, for instance $\langle P : X[1, 2], [[\text{"foo"}, \text{"bar"}], [\text{"red"}, \text{"fox"}]]\rangle = \text{"bar"}$. We will denote $[p_1, \dots, p_k]$ by $\mathbf{p}$ for conciseness, for example $\mathbf{p} = [1, 2]$ in the example above.

An arc $(P : Y \to P' : X) \in E$ specifies a data dependency from $P : Y$ to $P' : X$. Note that not all of the inputs I_P of P need to be the destination of an arc – indeed it is common for processors to have optional inputs. Ports with no incoming arcs are bound to default values that are specified during workflow design[5].

Workflow execution follows a pure dataflow model [19]: a processor P can execute (is *fireable*) as soon as all of its input ports that are destinations of an arc are bound to values. When it executes, a processor consumes its inputs and computes new bindings for the output ports. When some of the inputs are lists, in some cases its elements are consumed by different instances of the same processor. This behaviour is described informally in the following example, and formalized in Section 3. We assume that a data dependency $(P : Y \to P' : X) \in E$ specifies that $P' : X$ is bound to value v as soon as the binding $\langle P : Y, v\rangle$ is established. This makes for a data-driven computational model, whereby data is transferred along the arcs as soon as it is produced by the originating processor, and fireable processors execute as soon as possible[6].

2.2 Example

We use a simple but concrete example to illustrate the use of lists in the workflow model. Consider again the Taverna bioinformatics workflow of Fig. 1. More specifically, the workflow maps a nested input list of the form $v = [[20816, 26416], [328788]]$, containing gene IDs, into corresponding lists of metabolic pathways, retrieved from the KEGG database. Specifically, consider an input value v as above, bound to input port `list_of_geneIDList`. On output port `commonPathways`, the workflow produces a list of pathways, for instance [`path:04010 MAPK signaling`, `path:04370 VEGF signaling`], such that each of the input genes is known to participate in each of those pathways. Additionally, the workflow takes each input sub-list separately, and retrieves the set of pathways that involve genes in that list, independently from the others. Thus, output port `paths_per_gene` consists of two sub-lists, of the form [[`path:04210 Apoptosis`, `path:04010 MAPK signaling`,...], [`path:04010 MAPK signaling`, `path:04620 Toll-like receptor`,...]], where the first sub-list corresponds to the set of genes in the first input sub-list, and the second to the genes in the second input sub-list.

The workflow illustrates a simple case of collection-handling capabilities in Taverna. Indeed, suppose that, in the left branch of the workflow, *type*(`get_pathways_by_genes` : `genes_id_list`) = *list*(`string`), while the input provided by the user is of type *list*(*list*(`string`)). Taverna handles this mismatch between the depth of the declared port type and the type of the input value bound to the port, by invoking a separate instance of `get_pathways_by_genes` on each element of the input list independently, namely on [`mmu:20816`, `mmu:26416`] and on [`mmu:328788`] (note that each of these two inputs is now of the expected input type, *list*(`string`)). Each of the two executions generates one value, a list of pathway IDs, on output port `return`. The next processor, `getpathwayDescriptions`, retrieves a human-readable description of the pathway given a list of pathway IDs. Again, two instances of the processor are executed, one for each list value bound to input port `string`, and coming from the `return` output port through an arc. The overall effect is that the final output `paths_per_gene` consists of a list of lists, as shown in Fig. 1.

At the same time, the right branch of the workflow includes an initial step where the two input lists are flattened into one, by removing the nesting. The resulting list is then used as input to the same sequence of services as in the left branch. The result is a single output flat list, on port `commonPathways`, containing the pathways that involve all the genes in each of the initial input lists.

2.3 Provenance trace

We describe the provenance trace associated with an execution using two types of relations. The first accounts for the data transformations computed by processors, and is of the form:

$$\begin{aligned}\langle P : X_1[\mathbf{p_1}], v_1\rangle \dots \langle P : X_n[\mathbf{p_n}], v_n\rangle \to \\ \langle P : Y_1[\mathbf{q_1}], w_1\rangle \dots \langle P : Y_m[\mathbf{q_m}], w_m\rangle\end{aligned} \tag{1}$$

Following our earlier convention, we write $\langle P : X_i[\mathbf{p_j}], v\rangle$ to denote that v is the value at position $\mathbf{p_j}$ within the collection bound to port X_i or P. The empty index [] denotes the entire port X_i. An example of partial and fine-grained trace with data transformation is shown in Fig. 2.

$$\begin{aligned}
&\langle \texttt{get\_pathways\_by\_genes} : \texttt{genes\_id\_list}[1], v_1\rangle \\
&\quad \to \langle \texttt{get\_pathways\_by\_genes} : \texttt{return}[1], v'_1\rangle \\
&\langle \texttt{get\_pathways\_by\_genes} : \texttt{genes\_id\_list}[2], v_2\rangle \\
&\quad \to \langle \texttt{get\_pathways\_by\_genes} : \texttt{return}[2], v'_2\rangle \\
&\langle \texttt{getPathwayDescriptions} : \texttt{string}[1], v'_1\rangle \\
&\quad \to \langle \texttt{getPathwayDescriptions} : \texttt{return}[1], w_1\rangle \\
&\langle \texttt{getPathwayDescriptions} : \texttt{string}[2], v'_2\rangle \\
&\quad \to \langle \texttt{getPathwayDescriptions} : \texttt{return}[2], w_2\rangle \\
&\quad \dots \\
&\langle \texttt{getPathwayDescriptions} : \texttt{return}[1], w_1\rangle \\
&\quad \to \langle \texttt{workflow} : \texttt{paths\_per\_gene}[1], z_1\rangle \\
&\langle \texttt{getPathwayDescriptions} : \texttt{return}[2], w_2\rangle \\
&\quad \to \langle \texttt{workflow} : \texttt{paths\_per\_gene}[2], z_2\rangle
\end{aligned}$$

Figure 2: Partial provenance trace for the workflow of Fig. 1

As a shorthand for (1) we may write $InB_P \to OutB_P$. where InB_P and $OutB_P$ denote the sets of input and output bindings, respectively.

The second type of record accounts for the transfer of elements of a value v along an arc with source $P_1 : Y$ and

[5]If no defaults are specified, the services that represent the processors may produce unpredictable results if they rely on the values of those ports.

[6]In particular, the computation is triggered by binding input ports defined on the top-most workflow, which act as roots of the workflow graph, to user-supplied values. The computation terminates when the values reach the sink nodes in the graph.

sink $P_2 : X$:

$$\langle P : X[\mathbf{p}], v\rangle \rightarrow \langle P' : Y[\mathbf{p}'], v\rangle \quad (2)$$

We refer to (1) as the *xform* relations, and to (2) as the *xfer* relations. These relations describe the *observable* events that occur during the workflow execution, namely the computation of each instance of each processor (as we have seen in the previous example, multiple instances can be activated on elements of an input collection), and the propagation of values along the arcs. We assume, realistically, that processors are black boxes and thus that their internal behaviour cannot be observed. The *trace* T_{E_D} of one execution E (a "run") of dataflow D, is the collection of all observable *xform* and *xfer* events during the execution. Note that the observation of provenance events does not require any knowledge of the specific computational model that generates the data transformations, and in particular of the rules that govern multiple processor invocations when lists are present on the input ports. Consider, for example, processor `P` in Fig. 3. The computational model specifies that multiple instances of `P` be executed, each on a set of input bindings of the form $\langle \texttt{P : X1}[\mathbf{p_1}], a\rangle, \langle \texttt{P : X2}[\mathbf{p_2}], c\rangle, \langle \texttt{P : X3}[\mathbf{p_3}], b\rangle$ for some $\mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}$, each resulting in a binding for `Y`: $\langle \texttt{P : Y}[\mathbf{q}], y\rangle$ for some $\mathbf{q}$. This is due to the depth mismatches that occur on the input ports, namely $\textit{type}(\texttt{Q:X}) = \texttt{string}$, but $\textit{type}(v) = \textit{list}(\texttt{string})$, $\textit{type}(\texttt{P:X1}) = \texttt{string}$ while $\textit{type}(a) = \textit{list}(\texttt{string})$, and $\textit{type}(\texttt{P:X3}) = \texttt{string}$, but $\textit{type}(b) = \textit{list}(\texttt{string})$.

The provenance trace, however, can be collected without any knowledge of the model that defines the relationship amongst the $\mathbf{p_i}$ and the $\mathbf{q}$. Indeed, such a relationship is not always trivial; as an example, here is a trace that can be generated from this workflow, using the Taverna model for implicit iterations over lists:

(1) $\langle \texttt{Q : X}[1], v\rangle \rightarrow \langle \texttt{Q : Y}[1], a\rangle$

...

$\langle \texttt{Q : X}[n], v\rangle \rightarrow \langle \texttt{Q : Y}[n], a\rangle$

(2) $\langle \texttt{R : X}[], w\rangle \rightarrow \langle \texttt{R : Y}[], b\rangle$

$\langle \texttt{P : X1}[1], a\rangle, \langle \texttt{P : X2}[], c\rangle, \langle \texttt{R : X3}[1], b\rangle \rightarrow \langle \texttt{R : Y}[1,1], y\rangle$

...

$\langle \texttt{P : X1}[1], a\rangle, \langle \texttt{P : X2}[], c\rangle, \langle \texttt{R : X3}[m], b\rangle \rightarrow \langle \texttt{R : Y}[1,m], y\rangle$

...

$\langle \texttt{P : X1}[n], a\rangle, \langle \texttt{P : X2}[], c\rangle, \langle \texttt{R : X3}[m], b\rangle \rightarrow \langle \texttt{R : Y}[n,m], y\rangle$

The trace events for `Q` in the figure show that each element of output list `Q:Y` is obtained by one instance of `Q` on an input element (with the same index). The events for `R` indicate that b is computed using the entire input value w in `R`, i.e., `R` takes a simple value and produces a list. Finally, the trace for `P` includes $|a| \cdot |b| = n \cdot m$ events, one for each instance of `P`. Each such instance consumes one element of a, one element of b, and the entire list c. The iteration model of Taverna that is responsible for this choice of indices is explained in the next section. We refer to events like (1) above as *fine-grained*, as we can use them to trace the lineage of individual elements wihtin lists. This is a desirable property of traces. For example, in our earlier workflow of Fig. 1, $\textit{lin}(\langle \texttt{workflow : paths\_per\_gene}[1], v\rangle) = [26416]$, i.e., a fine-grained trace can be used to map one of the output sets of pathways back to precisely the list of genes that is involved in that set of pathways, allowing the user to reconstruct a data relationship that is not explicitly maintained by the workflow.

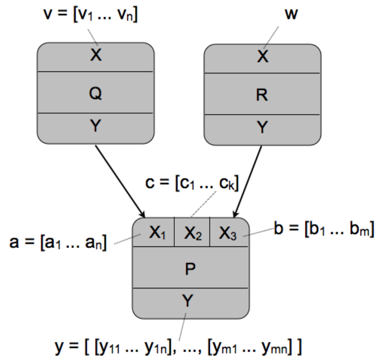


Figure 3: Abstract workflow, to illustrate traces with multiple processor instances

Such a fine level of granularity is not always available, however. Processor `R` in the execution sketched in Fig. 3, maps w to list c. Even if w were a list, event (2) in the trace above indicates that each element of b depends on the entire w, rather than on any of its elements. This situation is typical of "many-to-many" processors that map entire lists to new lists, or of "many-to-one" processors that compute aggregate values from lists, and is an intrinsic limitation on the granularity of the provenance mode, reflecting the fact that the processes called from the workflow may act on or read complete collections.

2.4 Lineage Queries

It is convenient to view a trace T_{E_D} as a directed acyclic graph, denoted the *provenance graph* of T_{E_D}, in which the nodes are all the bindings b_i that appear in the trace, and there is an arc from b_i to b_j if and only if (i) T_{E_D} contains an *xform* event $\textit{InB}_P \rightarrow \textit{OutB}_P$ such that $b_i \in \textit{InB}_P$ and $b_j \in \textit{OutB}_P$, or (ii) T_{E_D} contains an *xfer* event $b_i \rightarrow b_j$.

In this section we define the *lineage* of a data binding $\langle P : Y[\mathbf{p}], v\rangle$ that appears in a trace, as a function that returns the set of bindings found by traversing the provenance graph, upwards from the node that represents the binding.

Taking these considerations into account, we define a lineage query by mutual induction on the base cases of *xform*and *xfer*, as follows:

Definition 1. ***(Lineage query)***

1. *xform case.*
Let $\textit{InB}_P \rightarrow \textit{OutB}_P$ be an xform event, and let $\langle P : Y[\mathbf{p}], v\rangle \in \textit{OutB}_P$ for some index $\mathbf{p}$ within value v (recall that $\mathbf{p} = []$ if v is atomic).

$$\textit{lin}(\langle P : Y[\mathbf{p}], v\rangle, \mathcal{P}) = \begin{cases} \textit{In}_P \bigcup_{b \in \textit{InB}_P} \textit{lin}(b) \text{ if } P \in \mathcal{P} \\ \bigcup_{b \in \textit{InB}_P} \textit{lin}(b) \text{ otherwise} \end{cases}$$

2. *xfer case.*
Let $\langle P : Y[\mathbf{p}], v\rangle \rightarrow \langle P' : X[\mathbf{p}'], v\rangle$ be a xfer event for some indeces . Then $\mathbf{p}$ and $\mathbf{p}'$.

$$\textit{lin}(\langle P' : X[\mathbf{p}'], v\rangle) = \textit{lin}(\langle P : Y[\mathbf{p}], v\rangle)$$

Each of the two cases accounts for one step of a recursive traversal of the provenance graph, starting from value v. Case (1) accounts for the traversal of one processor, from its output ports to its input ports. The lineage of an output value v at position $\mathbf{p}$ of port Y of P is computed recursively as the lineage of the values on P's input ports, and it only includes the values In_P if $P \in \mathcal{P}$, the set of "interesting" processors. Case (2) accounts for the traversal of an arc in the provenance graph. Thus, definition (1) reads as a straightforward algorithm for computing the lineage of a binding $b = \langle P : Y[\mathbf{p}], v\rangle$, by traversing the provenance graph starting at node b and collecting the bindings that involve any $P \in \mathcal{P}$ along each of the paths upwards from b. The answer to the query is potentially fine-grained, because the input binding includes a specific index $\mathbf{p}$, but only to the extent that all the arcs in a path are derived from fine-grained trace events.

For example, with reference to the trace shown earlier, the computation unfolds into an alternation of *xfer* and *xform* steps, as follows:

$$
\begin{aligned}
&lin(\langle \mathtt{P} : \mathtt{Y}[h, l], y\rangle, \{\mathtt{Q}, \mathtt{R}\}) = \\
&lin(\langle \mathtt{P} : \mathtt{X1}[h], a\rangle) \cup lin(\langle \mathtt{P} : \mathtt{X2}[], c\rangle) \cup lin(\langle \mathtt{P} : \mathtt{X3}[l], b\rangle) = \\
&\quad lin(\langle \mathtt{Q} : \mathtt{Y}[h], a\rangle) \cup lin(\langle \mathtt{R} : \mathtt{Y}[l], b\rangle) = \\
&\quad \{\langle \mathtt{Q} : \mathtt{X}[h], v\rangle, \langle \mathtt{R} : \mathtt{X}[], w\rangle\}
\end{aligned}
$$

It may be argued that, in some cases, fine-grained provenance is not always desirable, for example when the products of a nested workflow is rather viewed as a black box than in full detail. We can easily achieve this by asking a query that is focused only on the nested workflow's inputs, and should coarse granularity be desired, one can specify an empty index into the port value for the query input, as this will compute the lineage of the entire value, regardless of its internal list structure. For example:

$$
\begin{aligned}
&lin(\langle \mathtt{P} : \mathtt{Y}[], y\rangle, \{\mathtt{Q}, \mathtt{R}\}) = \\
&lin(\langle \mathtt{P} : \mathtt{X1}[], a\rangle) \cup lin(\langle \mathtt{P} : \mathtt{X2}[], c\rangle) \cup lin(\langle \mathtt{P} : \mathtt{X3}[], b\rangle) = \\
&\quad lin(\langle \mathtt{Q} : \mathtt{Y}[], a\rangle) \cup lin(\langle \mathtt{R} : \mathtt{Y}[], b\rangle) = \\
&\quad \{\langle \mathtt{Q} : \mathtt{X}[], v\rangle, \langle \mathtt{R} : \mathtt{X}[], w\rangle\}
\end{aligned}
$$

3. EFFICIENT FOCUSED QUERIES

It is easy to see that the algorithm just described performs a traversal of the provenance graph, which involves all nodes in each of the paths from a given node to the source nodes of the graph. Each step of such traversal involves retrieving one event from the provenance trace. This simple approach is potentially wasteful, however, because a full traversal is required even when the query is focused, that is, all the nodes along a path must be visited, although some of them contain information of no interest to the user. In the algorithm described in this section, we avoid visiting trace events that are not part of $\mathcal{P}$, and replace the traversal of the provenance graph with the traversal of the much smaller workflow specification graph. Indeed, note that in our simple algorithm we have effectively used the *xform* events to invert the data transformations computed by each P: given a tuple of outputs of P, the algorithm retrieves the corresponding inputs by finding a matching *xform* event in the provenance trace. In particular, when an implicit iteration over the input lists of P takes place, this matching cuts through all the available *xform* events for the transformation computed by P, thus retaining the fine grain of provenance.

Our observation is that we can save on the number of *xform* events to be visited, by replacing this *extensional* matching with an intensional rule that describes all possible inverse transformations in terms of the iteration semantics, and independently of the values bound to the ports. Then, we repeatedly apply the rule in combination with a traversal of the workflow specification graph, and only lookup the actual *xform* events when we need to retrieve the values, i.e., when we visit a processor $P \in \mathcal{P}$. More specifically, the rule maps each index $\mathbf{q}$, pointing to a value that appears in one output binding for P, to a set of indices $\mathbf{p_1} \dots \mathbf{p_l}$, one for each of the l input ports of P.

Elaborating on this approach requires (i) a formal description of the rules that define the semantics of iterations over collections, and (ii) using such rules to define the inverse transformation. In this section we show how the Taverna computational model satisfies both requirements, and as a consequence, the workflow specification graph can indeed be used as an index into the provenance graph, to answer lineage queries of the form given in Def.1. We present the necessary details on the iteration model in Sections 3.1 and 3.2, followed by our inversion rule in Sec. 3.3.

It should be clear that this intensional approach is most beneficial when applied to focused queries, for several reasons. Firstly, for a focused query $lin(\langle P : Y[\mathbf{p}], v\rangle, \mathcal{P})$, not all paths in the graph connect the binding $\langle P : Y[\mathbf{p}], v\rangle$ to processors in $\mathcal{P}$. Therefore, some of the accesses to the explicit trace, required by the simple algorithm in Section 2, would be wasted, as they explore regions of the graph where no interesting processors are eventually found. Secondly, since the workflow graph is generally much smaller than any provenance graph, it is feasible to cache the nodes visited in one query to speed up their access in subsequent queries, as all queries on a provenance trace share the same workflow structure. And finally, since the same workflow graph specification is also shared by all the traces that represent different runs of that workflow, one single traversal is sufficient to answer lineage queries that involve multiple runs. Such queries are common for a batch of runs that "sweep" the value range of one or more input parameters, a standard technique in scientific applications. In this case, a query such as "report the lineage of binding b at processor P, across a set of executions" only requires one traversal of the workflow specification graph, followed by one access to the trace of each run, for each interesting processor. The results presented in the experimental section support these intuitions.

3.1 Computing actual port list depths

As mentioned in Section 2, each input or output port X has a type $type(X)$. Since the model assumes that all elements in a list value are at the same depth, we can additionally associate a declared depth $dd(X)$ to X, regardless of its value. The values bound to X, however, need not be lists of depth exactly equal to $dd(X)$. Indeed, in several of our examples so far we have observed a *depth mismatch*, for instance in Fig. 1, port `string` of `getPathwayDescriptions` expects a string value but instead receives a list of strings. In Taverna, the difference $\delta(X, v) = depth(v) - dd(X)$ between the declared depth of X and the *actual* depth $depth(v)$ of a value v bound to X, determines how input lists are iterated upon, by assigning its elements to one instance of P at a time.

While $\delta(X, v)$ appears to depend on the value v, in reality it can be computed statically on the workflow graph specification, providing that the following assumptions hold:

1. Each processor assigns values of the declared type to its output variables. Thus, $dd(P : Y) = 1$ implies that Y is bound to a list of depth 1, *each time P is invoked*. The iteration structure wraps these partial results by building additional nesting levels, as described in the next subsection;

2. Top-level dataflow input variables are always assigned values of the declared type.

With these quite natural assumptions we conclude that, for any $\langle P, I_P, O_P\rangle$ in the workflow, $\delta(X, v)$ is independent of v for all $X \in I_P \cup O_P$, so we simply denote the mismatch with $\delta_s(X)$ (where the 's' is for "static"). This is an important property, that we can use to pre-compute all mismatches on the workflow graph, by propagating the declared depths and the depth mismatches along each node of the graph, starting from the initial inputs, and regardless of the actual values that are bound at runtime. At each node, two rules are applied:

1. For each input variable $P : X \in I_P$, $depth(P : X)$ is set to $dd(P : X)$ if P is a root processor in the graph, and is set to $depth(P' : Y)$ when there is an arc from $P' : Y$ to $P : X$:

$$depth(P : X) = \begin{cases} dd(P : X) & \text{if } pred(P) = \emptyset, \\ depth(P' : Y) & \text{if } (P' : Y \to P : X) \in E \end{cases}$$

for all $X \in I_P$, where $pred(P)$, the predecessors of P, is the set of processors with outgoing links into some of P's input variables.

2. For each output variable $P : Y \in O_P$:

$$depth(P : Y) = dd(Y) + \sum_{X_i \in I_P} \delta_s(X_i)$$

Note that rule (2) requires the depths for all input ports of P to have been computed, prior to computing the depths of P's output ports. A simple way to ensure that this condition holds is to sort the processors according to their data dependencies. To achieve this, we perform a topological sort of the graph prior to propagating the depths. Having the nodes sorted also ensures the correct propagation order in the case, which may occur in practice, where some of the input variables are not connected through links from upstream processors (and thus they will not be bound to any values at all). The pseudo-code for this algorithm is shown in Alg. 1. The algorithm is executed only once for every new workflow definition graph for which a trace is generated.

3.2 List iteration model

We now use the depth mismatches just described, to formalize the iteration behaviour of Taverna. Iterations over collections have been used in the past, for instance as part of a general model of process networks that operate on streams [20], and they can be given a clear semantics by using higher-order functions (i.e., `map`). Our definition of iteration semantics, presented in Sec. 3.3 uses a similar notation but is original; the formulation given here differs from earlier proposals (e.g. [31, 18]), which we found to be less amenable for describing the approach presented in this paper.

Algorithm 1 compute PROPAGATEDEPTHS(D), i.e., set $depth(P : X)$ for all variables

```
$D' = (N', E') = toposort(D)$
for all $\langle P, I_P, O_P\rangle \in N'$ do
    $tot\_d = 0$
    for all $X \in I_P$ do
        if $P' : Y \to P : X \in E$ then
            $depth(X) = depth(Y)$
        else
            $nlX = dd(X)$
        end if
        $tot\_d = tot\_d + depth(X)$
    end for
    for all $Y \in O_P$ do
        $depth(Y) = dd(Y) + tot\_d$
    end for
end for
return $D'$
```

Using a functional notation, given a tuple $\langle a_1 \dots a_n\rangle$ of input values, each bound to a corresponding input port of P, we write ($\texttt{eval}$ P $\langle a_1 \dots a_n\rangle$) to denote the evaluation of P on the input tuple. Let us first consider the simple case of a single input port X bound to value v, and let $l = \delta_s(X)$ be the statically determined level mismatch on the port. The evaluation of P on v is defined recursively, using l as an additional index, as follows:

$$(\texttt{eval}_l\ P\ v) = \begin{cases} (P\ v) \text{ if } l = 0 \\ (\texttt{map}\ (\texttt{eval}_{l-1}\ P)\ v) \text{ if } l > 0 \end{cases} \tag{3}$$

For example, let $v = [[a, b]]$, and $\delta_s(X) = 2$, i.e., $dd(X) = 0$, and let $(P\ x) = $ "x $\texttt{isNice}$". We have

$$(\texttt{eval}_2\ P\ [[a, b]]) = (\texttt{map}\ (\texttt{eval}_1\ P)\ [[a, b]]) =$$
$$[(\texttt{eval}_1\ P\ [a, b])] = (\texttt{map}\ (\texttt{eval}_0\ P)\ [a, b]) =$$
$$[[(\texttt{eval}_0\ P\ a), (\texttt{eval}_0\ P\ b)]] = [[(P\ a), (P\ b)]] =$$
$$[[\text{"}a\ \texttt{isNice}\text{"}, \text{"}b\ \texttt{isNice}\text{"}]]$$

When P has multiple inputs, different depths mismatches may occur on the different ports. For the sake of illustration, consider the simplest and most common case of iteration over multiple inputs. Let processor P have input ports X_1, X_2 bound to lists a and b, respectively. When $\delta_s(X_1) = \delta_s(X_2) = 1$, Taverna first computes the *cross product* $a \times b = [[\langle a_i, b_j\rangle] | b_j \leftarrow b] | a_i \leftarrow a]$, and then applies P to each element of $a \times b$:

$$y = (\texttt{eval}_2\ P\ \langle a, b\rangle) = (\texttt{map}\ (\texttt{eval}_1\ P)\ a \times b) \tag{4}$$

Note that, unlike the ordinary cartesian product, the cross product of two lists is a nested list of depth 2, the value used in the initial application of `eval`. The result y is itself a list of depth 2, where $y[i, j] = (P\ \langle a_i, b_j\rangle)$.

We generalize the binary cross product operator to account for varying levels of mismatches on the different ports, as follows:

DEFINITION 2. ***Generalized cross-product***

$$(v, d_1) \otimes (w, d_2) = \begin{cases} [[(v_i, w_j) | w_j \leftarrow w] | v_i \leftarrow v] & \text{if } d_1 > 0, d_2 > 0 \\ [(v_i, w) | v_i \leftarrow v] & \text{if } d_1 > 0, d_2 = 0 \\ [(v, w_j) | w_j \leftarrow w] & \text{if } d_1 = 0, d_2 > 0 \\ (v, w) & \text{if } d_1 = 0, d_2 = 0 \end{cases}$$

where we have used a standard list comprehension syntax to denote iterators over lists. The two parameters d_1 and d_2, associated to each of the inputs, determine whether the corresponding values v and w should be iterated upon. Their values reflect port depth mismatches, i.e., although both v and w can be lists, iterators are only applied if either $d_1 > 0$ or $d_2 > 0$. Note also that each of the v_i and w_j can potentially be an entire sub-list, rather than an atomic value. Thus, the top case in the definition corresponds to the binary cross product, in which iteration is applied to both operands, while the next three are used when only one, or none, of the iterators apply. The parameters will be instantiated to reflect the depth mismatch that arises when the operands v and w are bound to ports X_1 and X_2. $\otimes$ is naturally extended to n operands, written $\otimes_{i:1\dots n}(v_i, d_i)$ (the binary $\otimes$ is left associative). Note also that we have assumed $d_i \geq 0$. When $d_i < 0$, no iteration occurs at all. Instead, the mismatch is dealt with by nesting a value v within d_i new lists, creating a d_i-deep singleton.

Using $\otimes$, we can now define $\texttt{eval}_l$ in the general case of for n inputs:

Definition 3. $\texttt{eval}_l$

$$(\mathit{eval}_l\ P \quad \langle (v_1, d_1), \dots, (v_n, d_n) \rangle)$$
$$= \begin{cases} (P\ \langle v_1, \dots, v_n \rangle) & \mathit{if}\ l = 0 \\ (\mathit{map}\ (\mathit{eval}_{l-1}\ P) \otimes_{i:1\dots n} \langle v_i, d_i \rangle) & \mathit{if}\ l > 0 \end{cases}$$

where initially $l = \sum_{i:1\dots n} \delta_s(X_i)$, the sum of the statically determined mismatches, and $d_i = \delta(X_i)$.

As an example, consider processor P in Fig. 3, where $\delta_s(\texttt{X}_1) = 1$, $\delta_s(\texttt{X}_2) = 0$, and $\delta_s(\texttt{X}_3) = 1$. Note that c is not involved in the iteration. Our definition takes account of this, as follows:

$$(\texttt{eval}_2\ \texttt{P}\ \langle a, c, b \rangle) = (\texttt{map}\ (\texttt{eval}_1\ \texttt{P})\ \langle a, 1 \rangle\ \otimes\ \langle c, 0 \rangle \otimes \langle b, 1 \rangle)$$

where:

$$\langle a, 1 \rangle\ \otimes\ \langle c, 0 \rangle \otimes \langle b, 1 \rangle) = \langle [(a_1, c) \dots (a_n, c)], 1 \rangle \otimes \langle b, 1 \rangle =$$
$$[[(a_1, c, b_1) \dots (a_1, c, b_m)], \dots, [(a_n, c, b_1) \dots (a_n, c, b_m)]]$$

Therefore,

$$(\texttt{eval}_2\ \texttt{P}\ \langle a, c, b \rangle) = [(\texttt{eval}_1\ \texttt{P}\ [(a_1, c, b_1) \dots (a_1, c, b_m)]), \dots,$$
$$(\texttt{eval}_1\ \texttt{P}\ [(a_n, c, b_1) \dots (a_n, c, b_m)]]$$

and for each of the sub-lists:

$$(\texttt{eval}_1\ \texttt{P}\ [(a_1, c, b_1), \dots, (a_1, c, b_m)]) =$$
$$[(\texttt{eval}_0\ \texttt{P}\ (a_1, c, b_1)), \dots, (\texttt{eval}_0\ \texttt{P}\ (a_1, c, b_m))] =$$
$$[y_{11}, \dots, y_{1m}]$$

In summary:

$$(\texttt{eval}_l\ \texttt{P}\ \langle a, c, b \rangle) = [[y_{11} \dots y_{1m}] \dots [y_{n1} \dots y_{nm}]]$$

as expected[7].

[7]Taverna also offers one additional list combinator operator, namely a "zip" or dot product between lists of equal length, as well as constructors that allow these operators to be combined into complex expressions. The general formulation of the model, however, is beyond the scope of the paper, and would not add to the exposition of our results.

3.3 The index projection rule

Let $\langle P : X_1[\mathbf{p_1}], v_1 \rangle \dots \langle P : X_n[\mathbf{p_n}], v_n \rangle \rightarrow \langle P : Y[\mathbf{q}], w \rangle$ be an *xform* event in the provenance trace, resulting from one elementary execution of P, as part of a computation defined in Def. 3. Using the semantics of the `eval` function, the following proposition states that the relationship between the output index $\mathbf{q}$ and the input indices $\mathbf{p_1} \dots \mathbf{p_n}$ is independent of the values $v_1 \dots v_n, w$, and furthermore, that the p_i can be computed from q using the statically computed mismatches $\delta_s(X_i)$.

Proposition 1. ***(Index projection)*** *Let*

$$\langle P : X_1[\mathbf{p_1}], v_1 \rangle \dots \langle P : X_n[\mathbf{p_n}], v_n \rangle \rightarrow \langle P : Y[\mathbf{q}], w \rangle$$

be an xform event in the provenance trace. The following hold:

1. *For each input index p_i: $|p_i| = \delta_s(X_i)$, and*

2. *$q = p_1 \cdot \dots \cdot p_n$, i.e., q is the concatenation of all indices p_i, each of length equal to the depth mismatch on the corresponding input port.*

The proof, omitted for brevity, is by induction on the total level of mismatches $\sum_{i:1\dots n} \delta_s(X_i)$ (this is also the length of q). Intuitively, the argument exploits the recursive definition of $\texttt{eval}_l$, by showing that, when the two properties above hold for the case $\texttt{eval}_{k-1}$, then they also hold for $\texttt{eval}_k$.

As a simple example of application, the proposition guarantees that $[i, j] = [i] \cdot [] \cdot [j]$ in the previous example, for each valid index $[i, j]$ of the output y — regardless of the values a, c, b, and y. Thus, Prop. 1 confirms the intuition that an *xform* relation can be inverted simply by apportioning fragments of the output index to each input port, in order, according to the depth mismatch found on that port. More precisely, let $\langle P : Y[\mathbf{p}], v \rangle$ be an output binding, and let $X_i \in I_P$ be an input port at position i within I_P. Note that Prop. 1 implies that we can ignore v while computing $\mathit{lin}(\langle P : Y[\mathbf{p}], v \rangle, \mathcal{P})$. Let us define the projection $\Pi_{X_i}(\mathbf{p})$ of $\mathbf{p}$ onto X_i to be the fragment $\mathbf{p}(i : i + \delta_s(X_i) - 1)$ of $\mathbf{p}$ that begins at position i and ends at position $i + \delta_s(X_i) - 1$, if $\delta_s(X_i) > 0$, and the empty index otherwise:

Definition 4. ***Index projection***

$$\Pi_{X_i}(\mathbf{p}) = \begin{cases} \mathbf{p}(i : i + \delta_s(X_i) - 1) & \mathit{if}\ \ \delta_s(X_i) > 0 \\ [] & \mathit{otherwise} \end{cases} \tag{5}$$

Using Def. (4), $\mathit{lin}(\langle P : Y[\mathbf{q}], v \rangle, \mathcal{P})$ is computed by traversing the workflow graph, as follows. Suppose $Y \in O_P$. The index projection rule is applied to $\mathbf{q}$ to compute a vector of indices $(\mathbf{p_1}, \dots \mathbf{p_n})$, one for each $X_i \in I_P$. Each of the $\mathbf{p_i}$ is then propagated along all the arcs that have X_i as their sink node, reaching a new processor node where the rule is applied again, until the entire graph has been traversed. The pseudo-code for this algorithm is shown as Alg. 2. In the code, $Q(P, X_i, \mathbf{p}_i)$ denotes the query on the provenance trace, needed to retrieve the value associated to a binding when $P \in \mathcal{P}$.

3.4 Computing lineage across runs

So far we have only considered lineage queries within the scope of one single trace, i.e., for a single workflow run. We

Algorithm 2 compute IndexProj$(P, Y, \mathbf{p}, \mathcal{P})$. Returns a set of variable bindings.

```
$result = \emptyset$
if $Y \in O_P$ then
    for all $X_i \in I_P$ do
        $\mathbf{p}_i = \Pi_{X_i}(\mathbf{p})$
        if $P \in \mathcal{P}$ then
            $result = \{Q(P, X_i, \mathbf{p}_i)\}$
        end if
        $result = result \cup$ IndexProj$(P, X_i, \mathbf{p}_i, \mathcal{P})$
    end for
else
    for all arc $P' : Y' \to P : Y \in E$ do
        $result = result \cup$ IndexProj$(P', Y', \mathbf{p}, \mathcal{P})$
    end for
end if
return $result$
```

conclude the section with a note on lineage queries that involve more than one trace, a common requirement in provenance analysis. This generalised form of quey is useful for comparing data products across multiple runs of the *same* workflow, as well as across runs of different versions of a workflow. An analysis of provenance based on differencing data dependency graphs is beyond the scope of this paper, and research in this area has only recently been undertaken [2]. Queries across multiple runs of the same workflow, however, can be easily described as part of our framework, by extending the trace query $Q(P, X_i, \mathbf{p}_i)$ in Alg. 2 to include a set $\mathcal{T}$ of traces which define the scope of the query: $Q(P, X_i, \mathbf{p}_i, \mathcal{T})$. In terms of practical implications, broadening the scope of the query to multiple runs does not significantly increase its complexity, as trace IDs are key attributes in our relational implementation of the traces database.

4. EXPERIMENTAL EVALUATION

We begin our evaluation by presenting an assessment of the expected response times for typical lineage queries, performed on two real-life Taverna scientific dataflows. The selected workflows are meant to cover both ends of the spectrum of sizes for the workflow collection found in the myExperiment repository[8]. Specifically, we have used the workflow of Fig. 1, denoted `genes2Kegg` (GK) in the following, as an example of a typical short-paths design, and a longer workflow, denoted **`protein discovery`** (PD) that looks for protein terms in a set of article abstracts from PubMed[9]. For each of the two workflows, we have measured the query response time for both a fully focused and fully unfocused query, and for queries that range over a set of runs of each of the two workflows. The resulting chart, shown in Fig. 4 introduces and justifies the design of our synthetic experimental testbed, presented in the next section. The total query response time can be broken down into two components: (s1) traverse the workflow graph in order to compute the indexes for the input ports of each $P' \in \mathcal{P}$, generating $|\mathcal{P}|$ queries in time t_1, and (s2) execute each of the queries in time t_2. As the chart shows, in the case of a one-run query, for GK and PD the totals $t_1 + t_2$ are small and quite close to one another, as both workflows are quite small. As we expand the query to range over multiple runs, however,

[8]`http://www.myexperiment.org/`.

[9]The BIoAid Protein Discovery workflow is available at `http://www.myexperiment.org/workflows/74`.

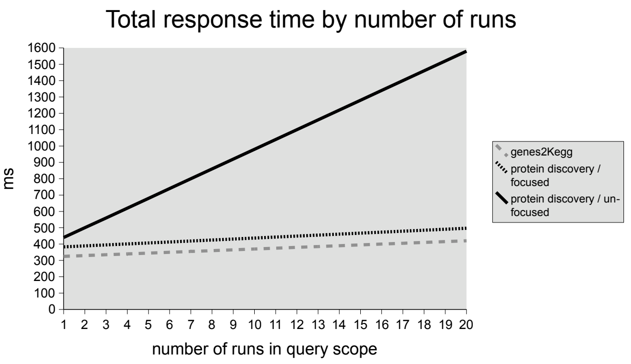


Figure 4: Query response time for focused/unfocused queries ranging over multiple runs

the IndexProj algorithm shares the initial traversal step (s1) amongst all the runs, and thus (s1) only needs to be performed once, while (s2) is performed once for each run (simply using the trace ID as a parameter). Thus, the times for the unfocused query only increase proportionally to t_2. As t_2 is typically around 6ms for our implementation, for both focused-PD and for GK, these scale well over multiple runs, while unfocused-PD takes 10 times longer to execute (s2), making multi-run queries proportionally slower.

In the rest of this section we compare the response times of the IndexProj algorithm with that of the baseline algorithm described informally in Section 2.4, denoted here `NI` (for *naïve* implementation). `NI` computes $lin(\langle P : X[\mathbf{p}], v\rangle, \mathcal{P})$ using an extensional representation of the provenance graph, implemented on a relational DBMS. Our comparison is based upon the observation that, as `NI` involves a full traversal of the provenance graph, it is equivalent to executing (s1) followed by (s2) for the fully unfocused case. This confirms the intuition that IndexProj never does worse than `NI`, and it only approaches `NI` on fully unfocused queries. It should also be clear that on multi-run queries `NI`, which does not rely upon the workflow graph specification, requires one full traversal of the provenance graph *for each run*, making its total response time proportional to $t_1 + t_2$, rather than to t_2 only.

The lineage query architecture described in this paper is fully implemented, in Java, as a plugin component of the Taverna 2 dataflow engine. Taverna 2, an open source project released under the LPGL license, is designed to run on desktop environments, such as users' laptops, with minimal third part software requirements and a small footprint. The configuration used in our experiments reflects this spirit: we have used a laptop installation (Intel CPU, 4GB RAM, MacOS 10.5), and a local mySQL 5.1 database.

4.1 Synthetic experimental testbed

The experiment described above helped us characterise the set of parameters that affect query performance, namely:

1. **Total number of nodes in the graph**. The overall graph size only affects the performance of the path generation step, denoted (s1) above. As we noted earlier, this step is common to all approaches, and can therefore be factored out in the comparison. Nevertheless, for completeness in Sec. 4.2 we report on t_1

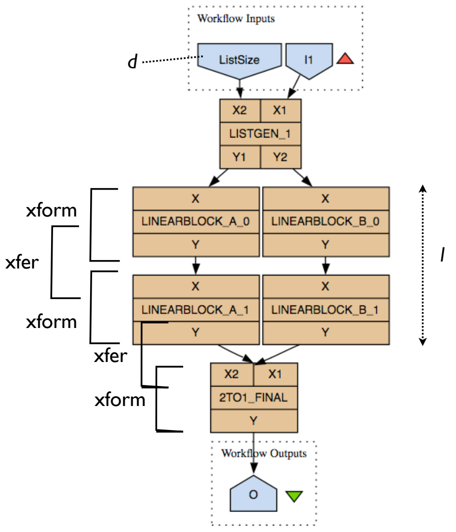


Figure 5: Example of generated testbed dataflow

times for various graph sizes.

2. **Length l of the path involved in the query**. This determines the number of queries in NI. Query times for IndexProj, however, are constant in the length of the path.

3. **Number d of elements in the input lists**. involved in the computation: large lists result in large traces, even for small graphs.

Using a Taverna workflow generator, we have designed an experimental testbed that allows us to explore this parameter space better than a more extensive collection of real workflows would have allowed. The synthetic dataflows share the common structure shown in Fig. 5, consisting of (i) the top processor `ListGen`, which generates a 1-deep list of d elements; (ii) two linear chains of processors, each of length l; and (iii) a final processor `2TO1FINAL` where the lists resulting from the linear chains are joined using a binary cross product. Fig. 5 shows a simple generated dataflow with $l = 2$. The choice of this family of workflows is motivated by the observation that the main factor that affects query response time is the length of the path to be traversed. While the "breadth" of a workflow does indeed affect the graph search phase of query processing, it does so equally for all approaches, and so we simply "factor it out" of our experiment space. Also, as we are not interested in workflow execution times, the family of workflows that we can experiment with by varying l and d is sufficient to describe common configurations that can be encountered in practice (including the total number of nodes in the graph, which affects t_1), at the same time providing a systematic approach to lineage query analysis. While this workflow pattern can be extended to multiple input processors and thus n-ary products, this family is adequate for our objective to simulate the propagation of lists through long paths, and the proliferation of nodes in lists through a product (at the end of the workflow).

Parameter l is set at dataflow generation time, while d is controlled by input port `ListSize`. In all of these dataflows, copies of the initial list simply propagates through each of

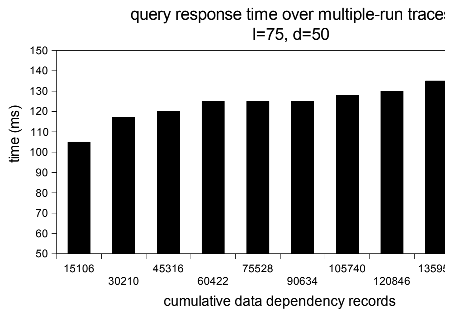


Figure 6: Lineage query response times for NI for varying trace size

the linear chains, until the final cross product is performed, causing lineage events to be generated along the way and the *xform* and *xfer* tables to be populated. As all processors are one-to-one, lineage precision is maintained throughout, making it possible to test fine-grained lineage queries of the form $lin(\langle \texttt{2TO1\_FINAL} : Y[\mathbf{p}], v\rangle, \{\texttt{LISTGEN\_1}\})$ while at the same time requiring a full traversal of each of the paths.

We have used the generator to explore a region of test parameters defined by $10 \le l \le 150$, $10 \le d \le 75$, which covers most real-life dataflow configurations that we have encountered in practice, e.g. in the myExperiment repository cited earlier.

4.2 Experimental results

The first set of results concern queries on single runs and on the synthetic workflows. Although the query scope is a single run, we have first determined whether the number of runs *stored* in the database, i.e., the sheer size of the database, affected query performance. Predictably, database size is not a dominant factor on single-run queries: since all of the queries on the traces involve the use of indexes, with none requiring full table scans, we did not expect this to be a dominant factor in practice. This is confirmed experimentally: as shown in the chart of Fig. 6 for the particular configuration $l = 75, d = 50$, the response times show a modest 20% increase, compared to a 10-fold increase in the number or records (from about 15,000 to 150,000) in the data dependency relations, obtained by accumulating traces for 10 dataflow runs. The results for other configurations, not shown for simplicity, are similar[10].

Having established the limited impact of the number of runs, we have chosen to compare the two algorithms by using a trace DB that only contains data for a single run. Table 1 reports the DB sizes in terms of number of records in the trace table, for our entire configuration space. The table can be used as a guideline for actual database sizes that reflect real operational settings where traces for any number of runs are stored.

We can further reduce the experimental space by assessing the impact of the list sizes on lineage query response times, i.e., by varying d for various configuration of l. Once again the results, plotted in Fig. 7, show a modest increase in response times for each choice of d and for each of the three

[10]All results, here and in the following, refer to the best response times over a sequence of five identical queries for all strategies, i.e., assuming the best case of a warm cache for all runs.

	l					
d	**10**	**28**	**50**	**75**	**100**	**150**
10	626	1346	2226	3226	4226	6226
25	2306	4106	6306	8806	11306	16306
50	7106	11000	15106	20106	25106	35106
75	14406	15479	26406	33906	41406	49561

Table 1: Number of trace database records for one run and one test dataflow

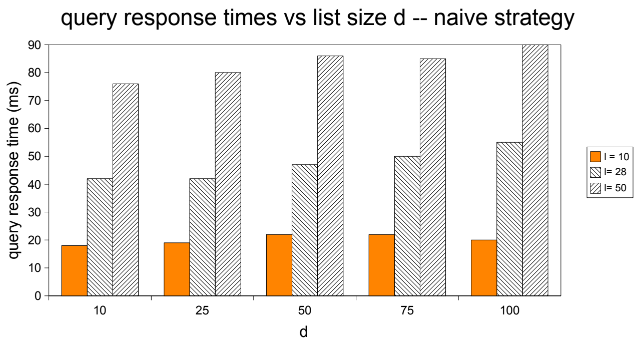


Figure 7: Lineage query response times for NI for varying input list size

values of l shown. This is due primarily to the increased size of the indexes (we use the standard indexing facilities offered by mySQL), and agree with the intuition that only d affects the size of the trace, but not the complexity of the query, as in the previous experiment.

Finally, we report our results for times t_1 and t_2 within our configuration space, for focused lineage queries of the form $lin(\langle \texttt{2TO1\_FINAL} : Y[\mathbf{p}], v\rangle, \{\texttt{LISTGEN\_1}\})$ for some specific path $\mathbf{p}$. The results for t_1 are shown in Fig. 8, where we have extended our configuration space to $l = 200$ for completeness. For dataflow graph with up to 100 nodes, pre-processing times are below 1 sec.

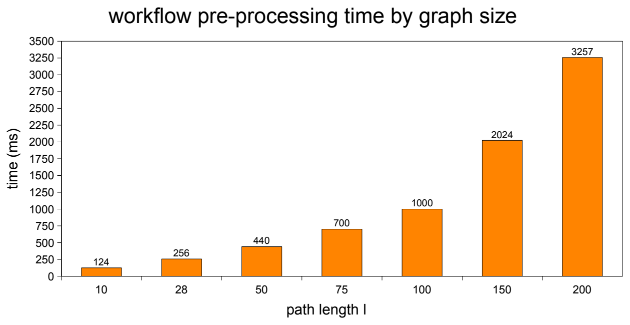
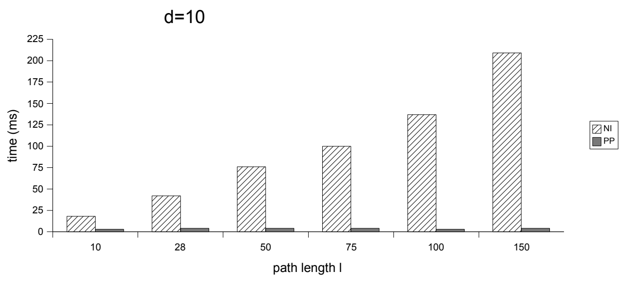


Figure 8: Pre-processing times vs. l

Regarding t_2, Fig. 9 shows the lineage query response times for our three strategies, for two extreme configurations with $d = 10$ and $d = 150$, respectively. As anticipated, the results look very similar, as we have established that response time is largely unaffected by d. As expected, IndexProj is constantly low, as t_2 reduces to one simple query on the trace when the queries are focused.

As discussed earlier, the performance of IndexProj for unfocused queries approaches that of NI. The plot in Fig. 10 shows the IndexProj response times for a target set of processors $\mathcal{P}$ that contains up to nearly 50% of the total.



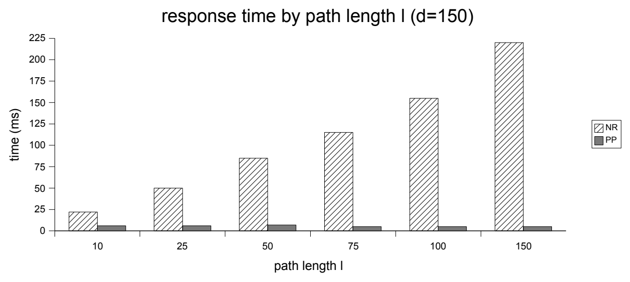


Figure 9: Lineage query response time across strategies as a function of l, for $d = 10$ and $d = 150$

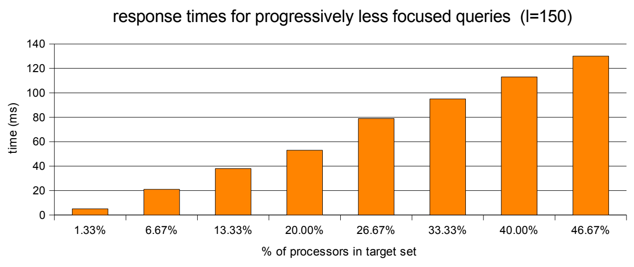


Figure 10: Lineage query response for IndexProj on partially unfocused queries

5. CONCLUSIONS

We have presented a lineage model for the Taverna collection-oriented workflow system, and an efficient algorithm for computing fine-grained and focused lineage queries over large provenance traces. The algorithm exploits the implicit iteration semantics of dataflow processors when it is available, and otherwise provides coarse-grained query answers. The efficiency is achieved by operating mostly on the structure of the dataflow graph, as opposed to the data dependency graph as is the case in most other approaches, by replacing the explicit traversal of the latter with the evaluation of intensional rules on the former. This results in manageable pre-processing times and a lineage query time that is essentially constant in the length of the provenance path, as well as in the size of the collections. As a result, the approach scales well on both those dimensions. The algorithm is well-suited to be combined with complementary approaches, such as the Zoom system for user-defined abstraction views on the workflow [6], and the explicit processor annotation model proposed in [1].

We have provided an implementation based on a standard RDBMS, with no need for auxiliary data structures, and have presented performance results over a large dataflow configuration space. The implementation is part of the provenance management component of the Taverna workflow system.

Acknowledgments

The authors are grateful to all members of the ^{my}Grid team, and in particular to Stuart Owen and Ian Dunlop, for their contribution to the design and implementation of the Taverna provenance architecture. We also acknowledge the financial support of the EPS Research Council (EPSRC) of the UK.

6. REFERENCES

[1] M. Anand, S. Bowers, T. McPhillips, and B. Ludaescher. Efficient provenance storage over nested data collections. In *Procs. EDBT*, March 2009.

[2] Z. Bao, S. Cohen-Boulakia, S. Davidson, A. Eyal, and S. Khanna. Differencing provenance in scientific workflows. In *Procs. ICDE*, March 2009.

[3] R. S. Barga and L. A. Digiampietri. Automatic capture and efficient storage of e-science experiment provenance. *Concurrency and Computation: Practice and Experience*, 20(8):419–429, 2008.

[4] A. Barker and J. van Hemert. *Scientific Workflow: A Survey and Research Directions*, volume 4967/2008 of *LNCS*. Springer, 2008.

[5] O. Benjelloun, A. Das Sarma, A. Y. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB J.*, 17(2):243–264, 2008.

[6] O. Biton, S. Cohen Boulakia, and S. B. Davidson. Zoom*userviews: Querying relevant provenance in workflow systems. In *VLDB*, pages 1366–1369, 2007.

[7] O. Biton, S. Cohen Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *ICDE*, pages 1072–1081, 2008.

[8] S. Bowers and B. Ludäscher. Actor-oriented design of scientific workflows. In *ER*, pages 369–384, 2005.

[9] S. Bowers, T. M. McPhillips, and B. Ludäscher. Provenance in collection-oriented scientific workflows. *Concurrency and Computation: Practice and Experience*, 20(5):519–529, 2008.

[10] S. Bowers, T. M. McPhillips, S. Riddle, M. Kumar Anand, and B.Ludäscher. Kepler/pPOD: Scientific workflow and provenance support for assembling the tree of life. In *IPAW*, pages 70–77, 2008.

[11] U. Braun, S. L. Garfinkel, D. A. Holland, K.K.Muniswamy-Reddy, and M. I. Seltzer. Issues in automatic provenance collection. In *IPAW*, pages 171–183, 2006.

[12] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, Cláudio T. Silva, and H. T. Vo. VisTrails: visualization meets data management. In *SIGMOD Conference*, pages 745–747, 2006.

[13] A. Chapman and H. V. Jagadish. Issues in building practical provenance systems. *IEEE Data Eng. Bull.*, 30(4):38–43, 2007.

[14] A. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In Wang [32], pages 993–1006.

[15] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *Computer*, 40(12):24–32, Dec. 2007.

[16] Huahai He and Ambuj K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Procs. SIGMOD Conference*, pages 405–418, Vancouver, BC, Canada, June 2008.

[17] T. Heinis and G. Alonso. Efficient lineage tracking for scientific workflows. In *SIGMOD Conference*, pages 1007–1018, 2008.

[18] J. Hidders and J. Sroka. Towards a calculus for collection-oriented scientific workflows with side effects. In *OTM Conferences (1)*, pages 374–391, 2008.

[19] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[20] E. A. Lee. Dataflow process networks. Memorandum UCB/ERL M94/53, UC Berkeley EECS Dept, 1994.

[21] D. T. Liu and M. J. Franklin. The design of GridDB: A data-centric overlay for the scientific grid. In *VLDB*, pages 600–611, 2004.

[22] B. Ludäscher, I. Altintas, and C. Berkley. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2005.

[23] Shawn Bowers Manish Anand and Bertram Ludaescher. Techniques for efficiently querying scientific workflow provenance graphs. In *Procs. EDBT*, Lausanne, Switzerland, March 2010.

[24] T. McPhillips, S. Bowers, and B. Ludäscher. Collection-oriented scientific workflows for integrating and analyzing biological data. In *Proceedings 3rd International Conference on Data Integration for the Life Sciences (DILS)*, LNCS/LNBI. Springer, 2006.

[25] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher. Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5):541 – 551, 2009.

[26] KK Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track*, pages 43–56, 2006.

[27] T. Oinn, M. Greenwood, M. Addis, and M. Nedim Alpdemir. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, August 2006.

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

[29] C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and re-using workflows with VisTrails. In *SIGMOD*, pages 1251–1254, New York, NY, USA, 2008. ACM.

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

[31] D. Turi, P. Missier, D. De Roure, C. Goble, and T. Oinn. Taverna Workflows: Syntax and Semantics. In *Proceedings of the 3rd e-Science conference*, Bangalore, India, December 2007.

[32] Jason Tsong-Li Wang, editor. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. ACM, 2008.

[33] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, pages 91–102, 1997.