Janus: Fine-grained and efficient provenance querying for Taverna

Paolo Missier
Information Management Group
School of Computer Science, University of Manchester, UK

in collaboration with:
Norman Paton, Khalid Belhajjame

IMG Seminar, June 2009
Motivation: Taverna for rapid information integration

Fine-grained process provenance and its role in Taverna

Context and scope: forms and uses for process provenance

Technical challenges in querying provenance traces

A solution, and experimental results
Workflow as data integrator

QTL genomic regions

genes in QTL

metabolic pathways (KEGG)
Workflow as data integrator

QTL genomic regions

genes in QTL

metabolic pathways (KEGG)
The genes→pathways workflow in action

KEGG gene ids:
“mmu:20816 (g1)
mmu:26416 (g2)
mmu:328788 (g3)"

“p1 MAPK signaling pathway
p2 VEGF signaling pathway
... “
The genes→pathways workflow in action

KEGG gene ids:
“mmu:20816 (g1)
mmu:26416 (g2)
mmu:328788 (g3)”

“p1 MAPK signaling pathway
p2 VEGF signaling pathway
...”
The genes→pathways workflow in action

KEGG gene ids: “mmu:20816 (g1)
  mmu:26416 (g2)
  mmu:328788 (g3)”

“p1 MAPK signaling pathway
p2 VEGF signaling pathway
... “
The genes→pathways workflow in action

KEGG gene ids: “mmu:20816 (g1)
mmu:26416 (g2)
mmu:328788 (g3)”

KEGG DB lookups

goal:
- list all pathways that are mapped from both sets of genes
- substantial list manipulation involved in achieving this

“p1 MAPK signaling pathway
p2 VEGF signaling pathway...”
The genes→pathways workflow in action

**Goal:**
- List all pathways that are mapped from both sets of genes
- Substantial list manipulation involved in achieving this

KEGG gene ids:
“mmu:20816 (g1)
mmu:26416 (g2)
mmu:328788 (g3)”

```
[ [ g1, g2, g3],
 [ “g1 p1 p2 ...”, “g2 p1 p2 ...”, “g3 p1 p2 ...”]
[ g1, g2, g3] ]
```

“p1 MAPK signaling pathway
p2 VEGF signaling pathway
...”
List-structured KEGG gene ids:

```
[ [ mmu:26416 ],
  [ mmu:328788 ] ]
```

[ [ path:mmu04010 MAPK signaling, path:mmu04370 VEGF signaling ],
  [ path:mmu04210 Apoptosis, path:mmu04010 MAPK signaling, ...],
  [ path:mmu04010 MAPK signaling, path:mmu04620 Toll-like receptor, ...] ]
An alternative design

List-structured KEGG gene ids:

\[
[ [ \text{mmu:26416} ], \\
[ \text{mmu:328788} ] ]
\]

\[
[ \text{path:mmu04010 MAPK signaling}, \\
\text{path:mmu04370 VEGF signaling} ]
\]

\[
[ [ \text{path:mmu04210 Apoptosis}, \text{path:mmu04010 MAPK signaling}, ...], \\
[ \text{path:mmu04010 MAPK signaling}, \text{path:mmu04620 Toll-like receptor}, ... ] ]
\]
An alternative design

List-structured KEGG gene ids:

[ [ mmu:26416 ], [ mmu:328788 ] ]

[ path:mmu04010 MAPK signaling, path:mmu04370 VEGF signaling ]

[ [ path:mmu04210 Apoptosis, path:mmu04010 MAPK signaling, ...], [ path:mmu04010 MAPK signaling, path:mmu04620 Toll-like receptor, ... ] ]
Is either design better?

• Pros:
  – simpler to design and understand (hopefully)
  – (no shims!)
  – accepts multiple gene sets
    • returns list of pathways separately for each gene set
    • in addition to those shared by the union of all sets

• Cons:
  – no genes in output list:

    [ [ path:mmu04210 Apoptosis, path:mmu04010 MAPK signaling, ...],
    [ path:mmu04010 MAPK signaling, path:mmu04620 Toll-like receptor, ... ] ]

  – so the relationship between the gene set and the pathway set is lost...

...Or is it?
Provenance trace to the rescue

List-structured KEGG gene ids:
[[mmu:26416], [mmu:328788]]

[ path:mmu04010 MAPK signaling, path:mmu04370 VEGF signaling ]

[[path:mmu04210 Apoptosis, path:mmu04010 MAPK signaling, ...],
[ path:mmu04010 MAPK signaling, path:mmu04620 Toll-like receptor, ...]]
Provenance trace to the rescue

List-structured KEGG gene ids:

\[
\text{[\text{mmu:26416}, \text{mmu:328788}]}\]

\[
\text{[\text{path:mmu04010 MAPK signaling, path:mmu04370 VEGF signaling}]}\]

\[
\text{[\text{path:mmu04210 Apoptosis, path:mmu04010 MAPK signaling, ...]],}
\text{[\text{path:mmu04010 MAPK signaling, path:mmu04620 Toll-like receptor, ...}]}\]

Provenance trace to the rescue

List-structured KEGG gene ids:

```
[ [ mmu:26416 ], [ mmu:328788 ] ]
```

[ path:mmu04010 MAPK signaling, path:mmu04370 VEGF signaling ]

[ [ path:mmu04210 Apoptosis, path:mmu04010 MAPK signaling, ...], [ path:mmu04010 MAPK signaling, path:mmu04620 Toll-like receptor, ... ] ]
Provenance trace to the rescue

List-structured KEGG gene ids:

- \([ \{ \text{mmu:26416} \}, \{ \text{mmu:328788} \} ]\)
- \([ \text{path:mmu04010 MAPK signaling}, \text{path:mmu04370 VEGF signaling} ]\)
- \([ \text{path:mmu04010 MAPK signaling}, \text{path:mmu04620 Toll-like receptor} ]\)

\([ \text{path:mmu04210 Apoptosis}, \text{path:mmu04010 MAPK signaling}, \ldots \], \text{path:mmu04010 MAPK signaling}, \text{path:mmu04620 Toll-like receptor}, \ldots \] \)
Taverna + provenance

• Taverna type system: strings + nested lists
  – “cat”, [“cat”, “dog”], [ [“cat”, “dog”], [“large”, “small”] ]

• Taverna dataflow model: data-driven execution
  • services activate when input is ready

• Workflow provenance: a detailed trace of workflow execution
  – which services were executed
  – when
  – inputs used, outputs produced
• Taverna type system: strings + nested lists
  – “cat”, [“cat”, “dog”], [ [“cat”, “dog”], [“large”, “small”] ]

• Taverna dataflow model: data-driven execution
  • services activate when input is ready

• Workflow provenance: a detailed trace of workflow execution
  – which services were executed
  – when
  – inputs used, outputs produced

Taverna dataflow model + provenance traces can be a powerful combination
Some additional user questions

• Causal relations:
  - which pathway sets come from which gene sets?
  - which processes contributed to producing this image?
  - which process(es) caused this data to be incorrect?
  - which data caused this process to fail?

• Process and data analytics:
  - show me the variations in output in relation to an input parameter sweep (multiple process runs)
  - how often has my favourite service been executed?
    • on what inputs?
  - who produced this data?
  - how often does this pathway turn up when the input genes range over a certain set S?
Focus is on the data: the **observable outcomes from a process**

<table>
<thead>
<tr>
<th>raw provenance metadata</th>
<th>provenance metadata + interpretation framework</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>design</strong></td>
<td></td>
</tr>
<tr>
<td>• process structure (workflow graph)</td>
<td>• service annotations:</td>
</tr>
<tr>
<td>• history of process composition - reuse</td>
<td>• ex. get_pathways_by_genes</td>
</tr>
<tr>
<td>• process versions</td>
<td>• who created /edited: attribution</td>
</tr>
<tr>
<td></td>
<td>• why: purpose, intent</td>
</tr>
<tr>
<td><strong>execution</strong></td>
<td></td>
</tr>
<tr>
<td>process events:</td>
<td>- data annotations,</td>
</tr>
<tr>
<td>- service invocation</td>
<td>results interpretation in terms of</td>
</tr>
<tr>
<td>- data production / consumption</td>
<td>conceptual data model:</td>
</tr>
<tr>
<td>- causal dependency graphs</td>
<td>set of pathways $\rightarrow$ gene sets</td>
</tr>
<tr>
<td>ex.:</td>
<td></td>
</tr>
<tr>
<td>- list_of_geneIDList = [ a, b, c]</td>
<td></td>
</tr>
<tr>
<td>- paths_per_gene = [ [d,e,f], [g,h,j]]</td>
<td></td>
</tr>
<tr>
<td>- ... in run #32</td>
<td></td>
</tr>
</tbody>
</table>
...and their uses and associated challenges

<table>
<thead>
<tr>
<th>raw provenance metadata</th>
<th>provenance metadata + interpretation framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>design</td>
<td>design</td>
</tr>
<tr>
<td>• exploiting semantic properties of the process structure to improve provenance exploitation</td>
<td>• semantic-based search of process space</td>
</tr>
<tr>
<td>• exploring process space across versions and structural similarities</td>
<td></td>
</tr>
<tr>
<td>• graph matching</td>
<td></td>
</tr>
<tr>
<td>execution</td>
<td>execution</td>
</tr>
<tr>
<td>- enabling partial re-runs of resource-intensive workflows</td>
<td>- semantic-based query answering over annotated traces</td>
</tr>
<tr>
<td>- storing very large provenance traces that accumulate over time</td>
<td></td>
</tr>
<tr>
<td>- efficient query over large traces</td>
<td></td>
</tr>
<tr>
<td>- presentation of query answers</td>
<td></td>
</tr>
</tbody>
</table>
...and their uses and associated challenges

<table>
<thead>
<tr>
<th>raw provenance metadata</th>
<th>provenance metadata + interpretation framework</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>design</strong></td>
<td>• exploiting semantic properties of the process structure to improve provenance exploitation</td>
</tr>
<tr>
<td></td>
<td>• exploring process space across versions and structural similarities</td>
</tr>
<tr>
<td></td>
<td>• graph matching</td>
</tr>
<tr>
<td><strong>execution</strong></td>
<td>• semantic-based search of process space</td>
</tr>
<tr>
<td></td>
<td>- enabling partial re-runs of resource-intensive workflows</td>
</tr>
<tr>
<td></td>
<td>- storing very large provenance traces that accumulate over time</td>
</tr>
<tr>
<td></td>
<td>- efficient query over large traces</td>
</tr>
<tr>
<td></td>
<td>- presentation of query answers</td>
</tr>
</tbody>
</table>

The rest of this talk!
...and their uses and associated challenges

<table>
<thead>
<tr>
<th>raw provenance metadata</th>
<th>provenance metadata + interpretation framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>design</td>
<td>• exploiting semantic properties of the process structure to improve provenance exploitation</td>
</tr>
<tr>
<td></td>
<td>• exploring process space across versions and structural similarities</td>
</tr>
<tr>
<td></td>
<td>• graph matching</td>
</tr>
<tr>
<td>execution</td>
<td>• semantic-based search of process space</td>
</tr>
<tr>
<td></td>
<td>- enabling partial re-runs of resource-intensive workflows</td>
</tr>
<tr>
<td></td>
<td>- storing very large provenance traces that accumulate over time</td>
</tr>
<tr>
<td></td>
<td>- efficient query over large traces</td>
</tr>
<tr>
<td></td>
<td>- presentation of query answers</td>
</tr>
</tbody>
</table>

The rest of this talk!
• **Lineage queries** involve traversing a *provenance graph* from bottom to top.
Naive provenance trace queries

- In most approaches, the originating process are not used for querying
- consequence: query requires provenance graph traversal
  - large traces $\rightarrow$ computationally complex
  - view materialization used in practice to get around the computational complexity
Requirements for lineage queries - I

- Focusing:
  Not all processors are interesting:
  - report lineage only at specified nodes in the graph
Requirements for lineage queries - II

List-structured KEGG gene ids:

- [[ mmu:26416 ], [ mmu:328788 ]]
- path:mmu04010 MAPK signaling,
  path:mmu04370 VEGF signaling

II - Granularity:
Trace lineage for individual elements within collections - when possible!

[[ path:mmu04010 MAPK signaling, path:mmu04370 VEGF signaling ]]
[[ path:mmu04210 Apoptosis, path:mmu04010 MAPK signaling, ...],
  [ path:mmu04620 Toll-like receptor, ... ]]
Requirements for lineage queries - III

- Answer queries efficiently without special auxiliary data structures
- (and, please provide declarative query specification)

Example:

```
BACKTRACE
(paths_per_gene[3,4], paths_per_gene[1,2])
AT get_pathway_by_genes
AND
commonPathways[1]
AT TOP
```

```
[[ path:mmu04210 Apoptosis, path:mmu04010 MAPK signaling, ...],
  [ path:mmu04010 MAPK signaling, path:mmu04620 Toll-like receptor, ...]]
```
III - Answer queries efficiently without special auxiliary data structures

(and, please provide declarative query specification)

Example:

BACKTRACE
(paths_per_gene[3,4], paths_per_gene[1,2])
AT get_pathway_by_genes
AND
commonPathways[1]
AT TOP

[[ path:mmu04210 Apoptosis, path:mmu04010 MAPK signaling, ...],
[ path:mmu04010 MAPK signaling, path:mmu04620 Toll-like receptor, ...]]
• III - Answer queries efficiently without special auxiliary data structures
• (and, please provide declarative query specification)

Example:

```
BACKTRACE
(paths_per_gene[3,4], paths_per_gene[1,2])
AT get_pathway_by_genes
AND
commonPathways[1]
AT TOP
targets

[ [ path:mmu04210 Apoptosis, path:mmu04010 MAPK signaling, ...],
  [ path:mmu04010 MAPK signaling, path:mmu04620 Toll-like receptor, ...] ]
```
How: Implicit iteration in Taverna

\[ a = [a_1 \ldots a_n] \]

\[ b = [b_1 \ldots b_m] \]

\[ c = [c_1 \ldots c_k] \]

\[
\begin{array}{ccc}
X_1 & X_2 & X_3 \\
P & & \\
Y & & \\
\end{array}
\]
How: Implicit iteration in Taverna

Depth mismatch between declared / offered type:

- \( \text{depth}(P:X1) = 0 \) but \( \text{depth}(a) = 1 \)
- \( \text{depth}(P:X2) = \text{depth}(c) = 1 \)
- \( \text{depth}(P:X3) = 1 \) but \( \text{depth}(c) = 1 \)
How: Implicit iteration in Taverna

How \( \mathbf{y} \) is computed at \( P \):

let \( I = a \otimes b = \left[ \left[ \langle a_i, b_j \rangle \mid b_j \in b \right] \mid a_i \in a \right] \) // cross product

\[ I' = \left[ \left[ \langle a_i, c, b_j \rangle \mid b_j \in b \right] \mid a_i \in a \right] \] // same product but with \( c \) interleaved

\[
\mathbf{y} = (\text{map} \ (\text{map} \ P) \ I') = [(\text{map} \ P [ \langle a_1, c, b_1 \rangle \ldots \langle a_1, c, b_m \rangle ]), \ldots , \\
(\text{map} \ P [ \langle a_n, c, b_1 \rangle \ldots \langle a_n, c, b_m \rangle ]) = \\
[ [y_{11} \ldots y_{1n}], \ldots [y_{n1} \ldots y_{nm}] ]
\]

Depth mismatch between declared / offered type:

- \( \text{depth}(P:X_1) = 0 \) but \( \text{depth}(a) = 1 \)
- \( \text{depth}(P:X_2) = \text{depth}(c) = 1 \)
- \( \text{depth}(P:X_3) = 1 \) but \( \text{depth}(c) = 1 \)
How: Implicit iteration in Taverna

\[ a = [a_1 \ldots a_n] \]
\[ b = [b_1 \ldots b_m] \]
\[ c = [c_1 \ldots c_k] \]

(a, b) \times (c, b) = (a, b) \times (c, b)

\[ y = [ [y_{11} \ldots y_{1n}], \ldots, [y_{m1} \ldots y_{mn}] ] \]

**Depth mismatch between declared / offered type:**

- \( \text{depth}(P:X_1) = 0 \) but \( \text{depth}(a) = 1 \)
- \( \text{depth}(P:X_2) = \text{depth}(c) = 1 \)
- \( \text{depth}(P:X_3) = 1 \) but \( \text{depth}(c) = 1 \)

How \( y \) is computed at \( P \):

\[
\text{let } l = a \otimes b = [ [ <a_i, b_j> | b_j \in b ] | a_i \in a ] \quad /\!/ \text{cross product}
\]
\[
l' = [ [ <a_i, c, b_j> | b_j \in b ] | a_i \in a ] \quad /\!/ \text{same product but with } c \text{ interleaved}
\]
\[
y = (\text{map (map } P \text{) } l') = [(\text{map } P [ <a_1,c, b_1> \ldots <a_1,c, b_m>]), \ldots,
(\text{map } P [ <a_n,c, b_1> \ldots <a_n,c, b_m>] )] =
[ [y_{11} \ldots y_{1n}], \ldots, [y_{n1} \ldots y_{nm}] ]
\]
**How: Implicit iteration in Taverna**

Let 

\[
\begin{align*}
\mathbf{a} &= [a_1 \ldots a_n] \\
\mathbf{b} &= [b_1 \ldots b_m] \\
\mathbf{c} &= [c_1 \ldots c_k]
\end{align*}
\]

be the data structures. Suppose that the operations are performed in the order:

1. Perform \(a \otimes b\) to get a cross product \(I\),
2. Interleave \(c\) with \(I\) to get \(I'\),
3. Map \(P\) over \(I'\) to get \(y\).

Then, the computation of \(y\) at \(P\) can be written as:

\[
y = (\text{map} \ (\text{map} \ P) \ I') = (\text{map} \ P \ [a_1, b_1]) \ldots (\text{map} \ P \ [a_n, b_1]) \ldots (\text{map} \ P \ [a_1, b_m]) \ldots (\text{map} \ P \ [a_n, b_m])
\]

where \(\otimes\) denotes some form of cross product and \(\text{map} P\) applies \(P\) to each element of the structure.

**Bottom line:**

\(y_{ij}\) depends only on values \(a_i, c, b_j\).
Extensional vs intensional tracing

Workflow structure graph

\[ v = [v_1 \ldots v_n] \]

\[ w \]

\[ c = [c_1 \ldots c_k] \]

\[ a = [a_1 \ldots a_n] \]

\[ b = [b_1 \ldots b_m] \]

\[ y = [ [y_{11} \ldots y_{1n}], \ldots [y_{m1} \ldots y_{mn}] ] \]
Extensional vs intensional tracing

Workflow structure graph

\[ v = [v_1, \ldots, v_n] \]

\[ a = [a_1, \ldots, a_n] \]

\[ y = \begin{bmatrix} y_{11}, \ldots, y_{1n} \\ \vdots \\ y_{m1}, \ldots, y_{mn} \end{bmatrix} \]

Provenance graph

\[ w = [b_1, \ldots, b_m] \]

\[ v_1, \ldots, v_n \]

\[ a_1, \ldots, a_n \]

\[ b_1, \ldots, b_m \]

\[ y_{11}, \ldots, y_{mn} \]
Hypothesis: we can exploit the static workflow graph structure to avoid explicitly traversing the entire trace to answer a query.
1) In general the actual depth at the output is:

\[
\text{depth}(y) = \text{depth}(Y) + \sum \delta(X_i = x_i)
\]

where \( \delta(X_i = x_i) = \text{depth}(x_i) - \text{depth}(X_i) \)

Therefore:
\( \delta(X=x) \) can be computed statically on the workflow graph structure,
- given the declared depth(\( X \))
- using a simple propagation algorithm

\[
[Y[i,j] \rightarrow X1[i], X2[], X3[j]] = [i_1 . i_2 . . . . i_k] = \text{-----------------------------}
\]
The path projection rule

1) In general the actual depth at the output is:

\[
depth(y) = depth(Y) + \sum \delta(X_i = x_i)
\]

where \(\delta(X_i = x_i) = depth(x_i) - depth(X_i)\)

Therefore:

\(\delta(X=x)\) can be computed statically on the workflow graph structure,
- given the declared depth(X)
- using a simple propagation algorithm

\[
[i_1 \cdot i_2 \cdot \ldots \cdot i_k] = \ldots
\]
The path projection rule

In general the actual depth at the output is:

\[
\text{depth}(y) = \text{depth}(Y) + \sum \delta(X_i = x_i)
\]

where \( \delta(X_i = x_i) = \text{depth}(x_i) - \text{depth}(X_i) \)

Therefore:
\( \delta(X=x) \) can be computed statically on the workflow graph structure,
- given the declared depth(X)
- using a simple propagation algorithm

\[
[i_1 . i_2 . . . . i_k] = \delta(X_1 = x_1)
\]
1) In general the actual depth at the output is:

\[ \text{depth}(y) = \text{depth}(Y) + \sum \delta(X_i = x_i) \]

where \( \delta(X_i = x_i) = \text{depth}(x_i) - \text{depth}(X_i) \)

Therefore:

\( \delta(X=x) \) can be computed statically on the workflow graph structure,
- given the declared depth(\(X\))
- using a simple propagation algorithm
The path projection rule

1) In general the actual depth at the output is:

\[
\text{depth}(y) = \text{depth}(Y) + \sum \delta(X_i = x_i)
\]

where \( \delta(X_i = x_i) = \text{depth}(x_i) - \text{depth}(X_i) \)

Therefore:
\( \delta(X=x) \) can be computed statically on the workflow graph structure,
- given the declared depth(\(X\))
- using a simple propagation algorithm

\[
(i_1, i_2, \ldots, i_k) = \delta(X_2 = x_2)
\]

\[
(i_1)
\]
The path projection rule

1) In general the actual depth at the output is:

\[
\text{depth}(y) = \text{depth}(Y) + \sum \delta(X_i = x_i)
\]

where \( \delta(X_i = x_i) = \text{depth}(x_i) - \text{depth}(X_i) \)

Therefore:
\( \delta(X=x) \) can be computed statically on the workflow graph structure,
- given the declared depth(\(X\))
- using a simple propagation algorithm
The path projection rule

1) In general the actual depth at the output is:

\[
\text{depth}(y) = \text{depth}(Y) + \sum \delta(X_i = x_i)
\]

where \(\delta(X_i = x_i) = \text{depth}(x_i) - \text{depth}(X_i)\)

Therefore:

\(\delta(X=x)\) can be computed statically on the workflow graph structure,
- given the declared depth(X)
- using a simple propagation algorithm

\[
[i_1 \cdot i_2 \cdot \ldots \cdot i_k] = \\
\underline{X_1} \quad \underline{X_2}
\]

\(\delta(X_k = x_k)\)
1) In general the actual depth at the output is:

\[
\text{depth}(y) = \text{depth}(Y) + \sum \delta(X_i = x_i)
\]

where \(\delta(X_i = x_i) = \text{depth}(x_i) - \text{depth}(X_i)\)

Therefore:
\(\delta(X=x)\) can be computed statically on the workflow graph structure,
- given the declared depth(X)
- using a simple propagation algorithm

\[[i_1 \ . \ i_2 \ . \ . \ . \ . \ i_k] = \]

\[
\begin{array}{ccc}
X_1 & X_2 & X_k \\
\end{array}
\]
Extension to the entire workflow graph
Extension to the entire workflow graph

\[ Y = \left[ \left[ \ldots \right], \ldots \left[ \ldots \right] \right] \]
Extension to the entire workflow graph

\[
\begin{align*}
\text{lineage}(P:Y[3,4]) & \rightarrow \text{lineage}(P:X1[3]), \\
& \text{lineage}(P:X2[]), \\
& \text{lineage}(P:X2[4])
\end{align*}
\]

\[
Y = \left[ \left[ \ldots \right], \ldots \left[ \ldots \right] \right]
\]
Extension to the entire workflow graph

\[
\text{lineage}(P:Y[3,4]) \rightarrow \text{lineage}(P:X1[3]), \text{lineage}(P:X2[]), \text{lineage}(P:X2[4])
\]

\[
Y = [ [ ...], ... [... ] ]
\]
Extension to the entire workflow graph

\[
Y = \begin{bmatrix}
[\ldots], \ldots [\ldots]
\end{bmatrix}
\]

\[
\text{lineage}(P:Y[3,4]) \rightarrow \text{lineage}(P:X1[3]), \text{lineage}(P:X2[]), \text{lineage}(P:X2[4])
\]

\[
\text{lineage}(P:X1[3]) = \text{lineage}(Q:Y[3]) \rightarrow \text{lineage}(Q:X[3])
\]
Extension to the entire workflow graph

\[ \text{lineage}(P:Y[3,4]) \rightarrow \text{lineage}(P:X1[3]), \]
\[ \text{lineage}(P:X2[]), \]
\[ \text{lineage}(P:X2[4]) \]

\[ \text{lineage}(P:X1[3]) = \text{lineage}(Q:Y[3]) \rightarrow \text{lineage}(Q:X[3]) \]

\[ Y = \left[ \left[ \ldots, \ldots \left[ \ldots \right] \ldots \right] \right] \]
Extension to the entire workflow graph

\[ \text{lineage}(P:Y[3,4]) \rightarrow \text{lineage}(P:X1[3]), \]  
\[ \text{lineage}(P:X2[]), \text{lineage}(P:X2[4])) \]  
\[ \text{lineage}(P:X1[3]) = \text{lineage}(Q:Y[3]) \rightarrow \]  
\[ \text{lineage}(Q:X[3]) \]  
\[ \text{lineage}(P:X3[4]) = \text{lineage}(R:Y[4]) \rightarrow \]  
\[ \text{lineage}(R:X[]) \]  

\[ Y = \left[ \left[ \ldots \right], \ldots \left[ \ldots \right] \right] \]
Extension to the entire workflow graph

\[
\begin{align*}
Y &= \{ [ \ldots ], \ldots [\ldots] \} \\
\text{lineage}(P:Y[3,4]) &\rightarrow \text{lineage}(P:X1[3]), \\
&\rightarrow \text{lineage}(P:X2[]), \\
&\rightarrow \text{lineage}(P:X2[4]) \\
\text{lineage}(P:X1[3]) &= \text{lineage}(Q:Y[3]) \rightarrow \\
&\rightarrow \text{lineage}(Q:X[3]) \\
\text{lineage}(P:X3[4]) &= \text{lineage}(R:Y[4]) \rightarrow \\
&\rightarrow \text{lineage}(R:X[]) 
\end{align*}
\]
Query processing

- Query processing:
  - alternating sequence of xform and xfer steps
- apply path projection at each xform step

- A complete granular and focused query can be answered by traversing the workflow graph alone
  - starting from the target vars
  - one simple query for each selected processor input port
Advantages

• Scalability:
  – query time depends on size of workflow graph, not size of provenance graph
  – workflow graphs are small, fit in memory, can be indexed easily, etc.
  – search over a graph at least as large as the workflow graph is inevitable -- this is the baseline cost!

• Graceful degradation:
  – worst case is a completely unfocused query
  – one query to trace at each xform step
  – no worse than other approaches

• Fine-grain answers provided at no additional cost
• Performance evaluation performed on programmatically generated dataflows

– the “T-towers”

countrol:
- size of the lists involved
- length of the paths
- includes one cross product
 Experimental results - I

- query response time: naive vs. "path projection" approaches

\[ d=10 \]

\[ d=150 \]

path length \( l \)
• workflow search time by path length ("tower height")
  – common to all strategies!

![Graph showing workflow pre-processing time by graph size]

• performance degradation on fully unfocused queries

![Graph showing response times for PP on unfocused queries (l=150)]
Summary

• An original approach to lineage queries for Taverna that combines
  – efficiency and fine-granularity

• Relies on semantic properties of the Taverna dataflow model

• Further work:
  – visual specification of user query
  – visual presentation of query answer
  – space compression
  – semantic overlays, annotations

• To be bundled with some future version of Taverna...