MANCHESTER 1824

The University of Manchester

myGrid

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

Paolo Missier, Norman Paton, Khalid Belhajjame

Information Management Group

School of Computer Science, University of Manchester, UK

EDBT Conference

Lausanne, Switzerland, March 2010

Problem statement and outline

- Setting:
  - *Black box* provenance of workflow data products
- Fine-grained provenance:
  - tracking provenance through collections: motivation
  - functional model of collection-oriented workflow processing
- Efficient query processing:
  - leveraging the functional model to achieve efficient processing for a simple query model
- Experimental evaluation

Computing provenance through a graph

- Provenance graph is an unfolding of the workflow graph structure
  - large: grows with size of input
  - lineage queries involve graph traversal

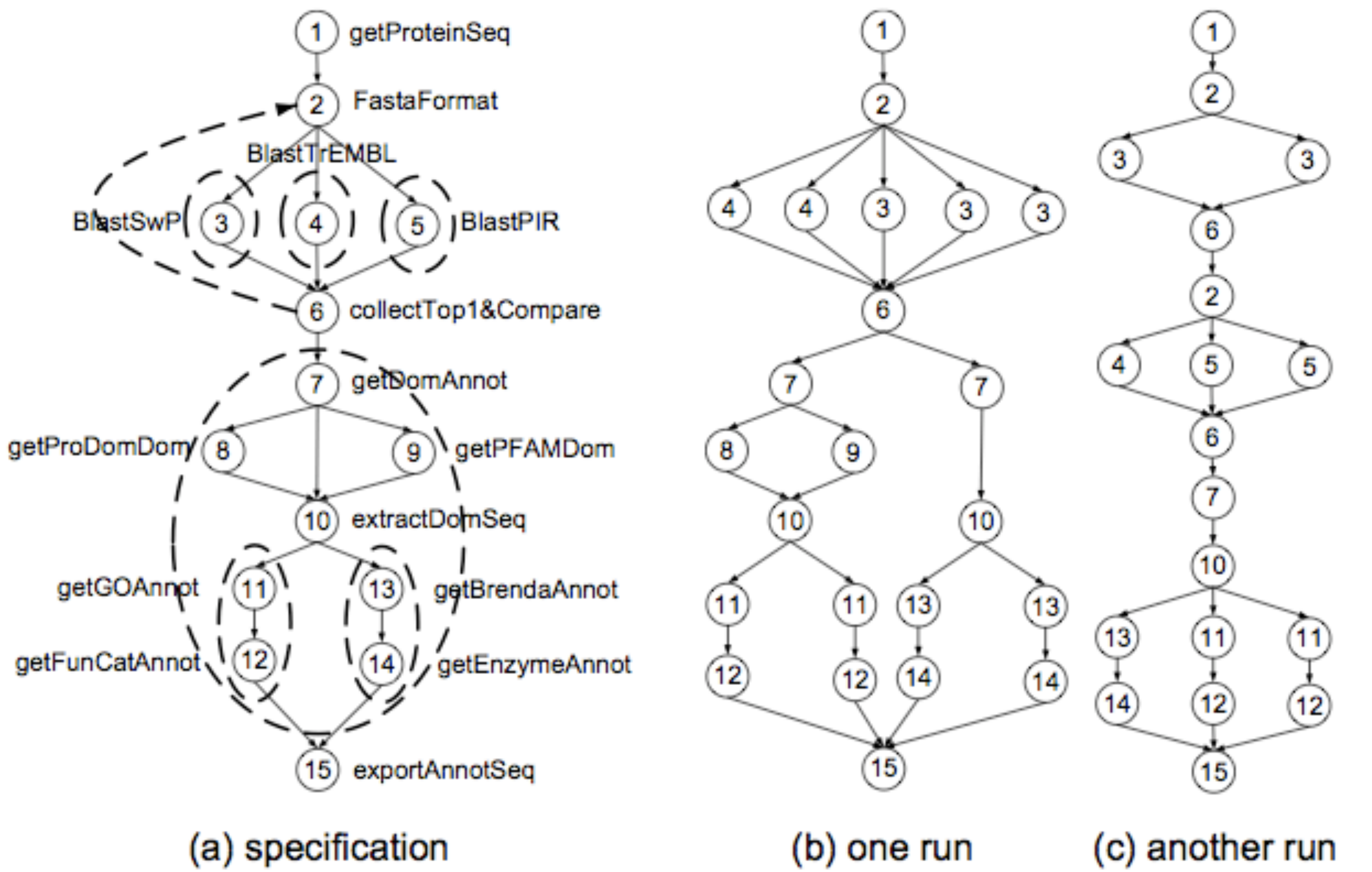
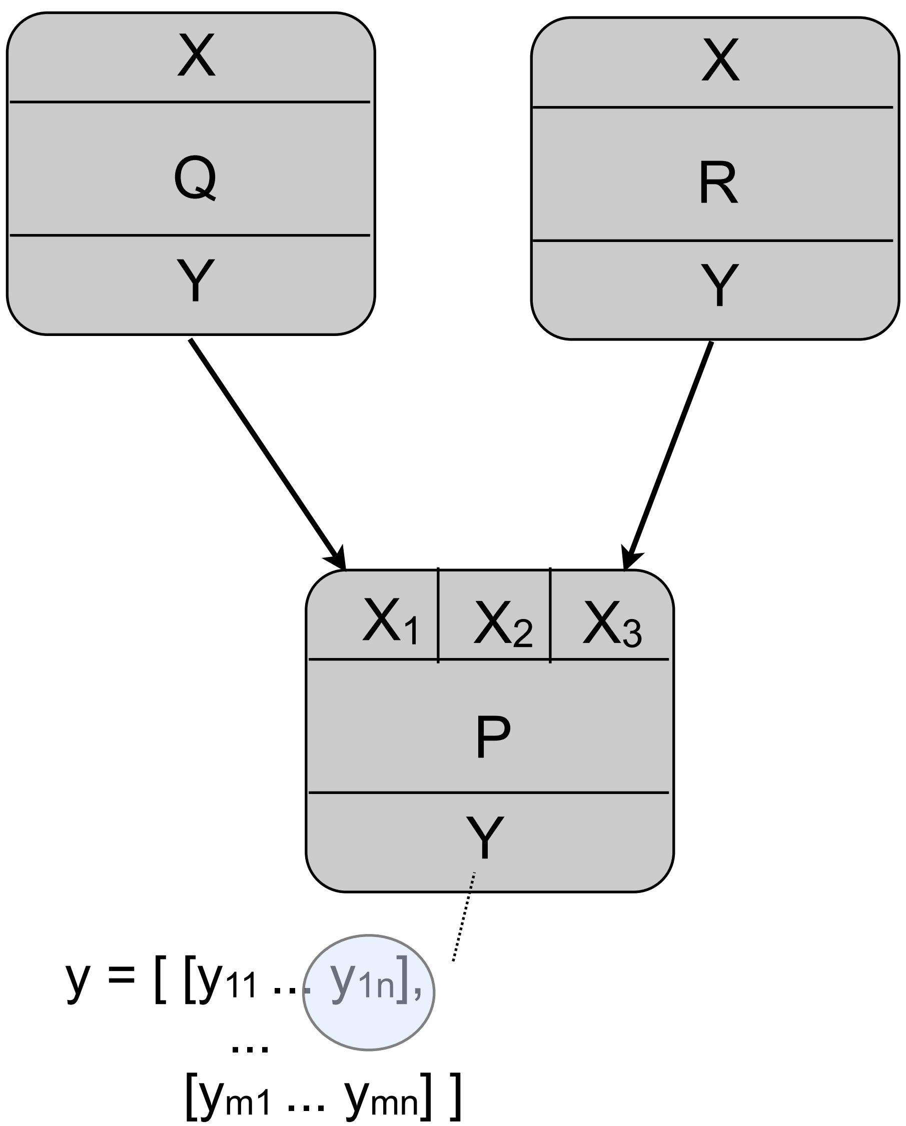


Fig. 1. Protein annotation workflow specification and runs

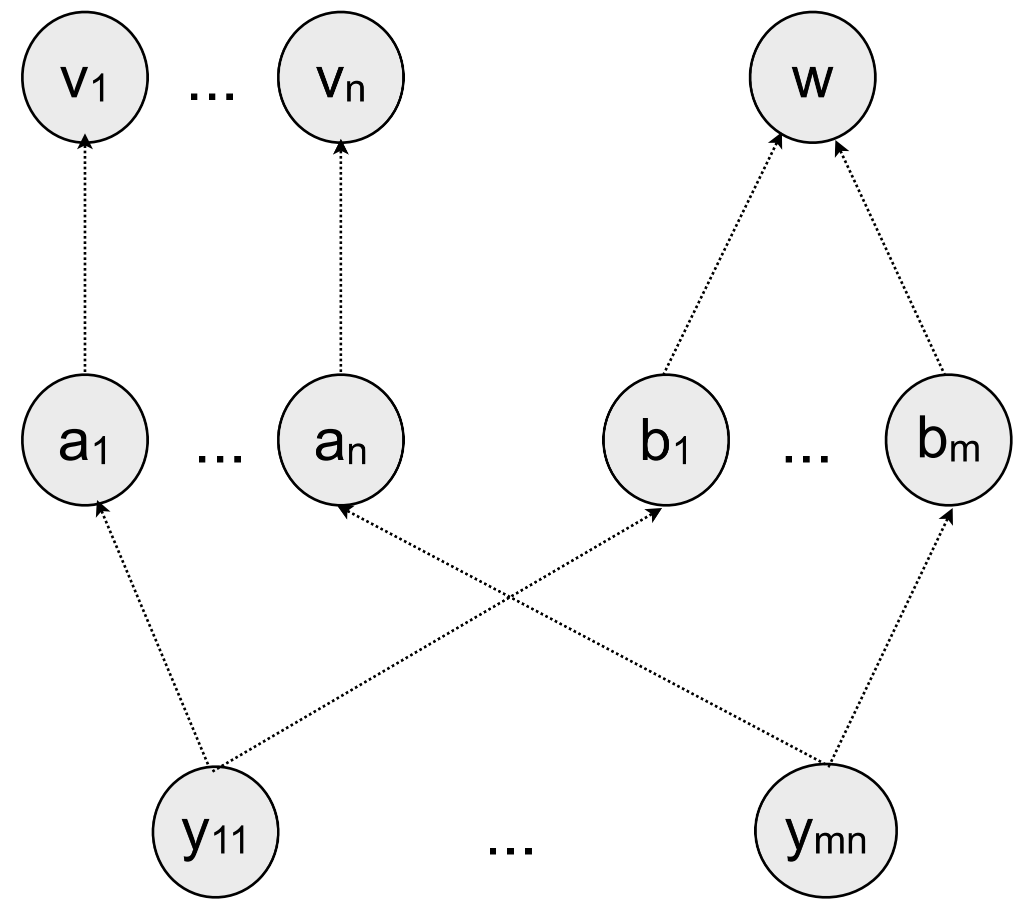
From:
Z. Bao, S. Cohen-Boulakia, S. Davidson, A. Eyal, and S. Khanna, "Differencing Provenance in Scientific Workflows," Procs. ICDE, 2009.

Main result

Workflow graph



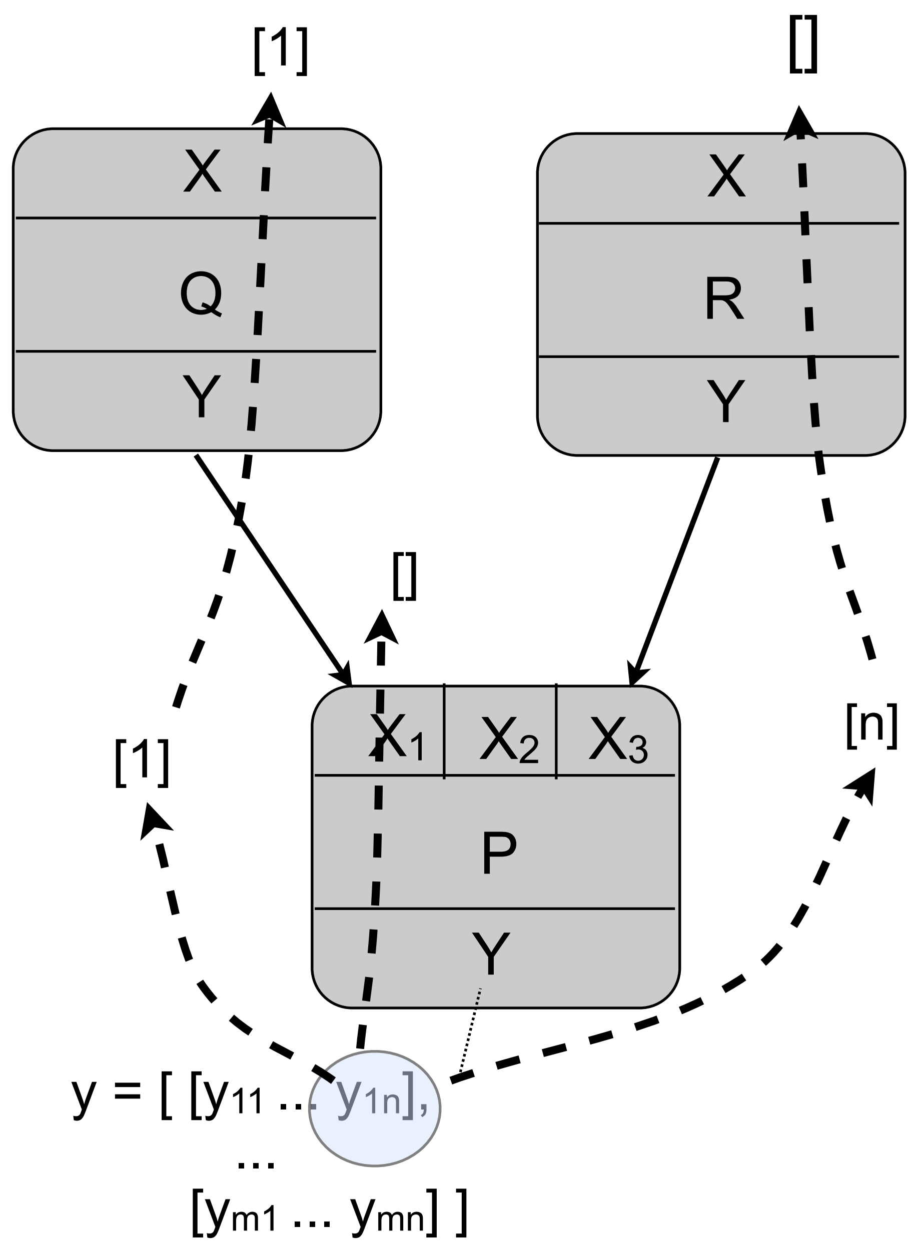
Provenance graph



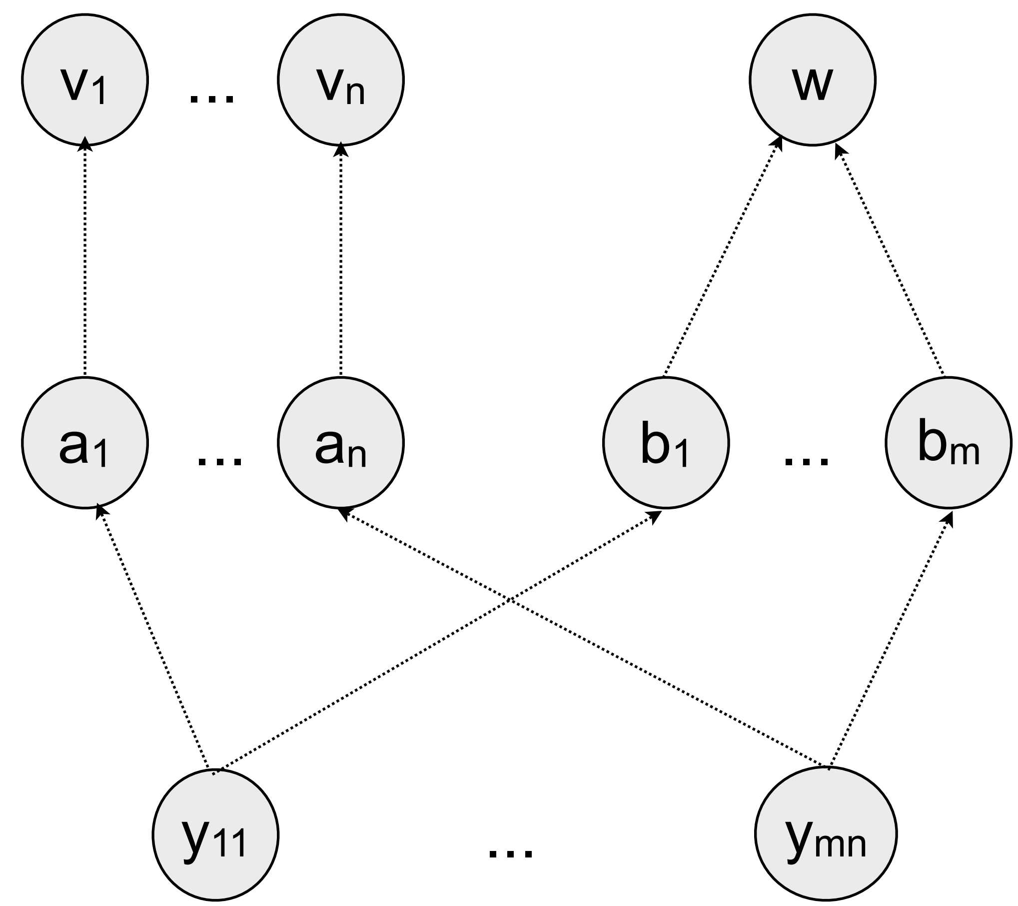
- Query the provenance of individual collections elements
- But, avoid computing transitive closures on the provenance graph
  - potentially very large
- Traverse the workflow graph instead -- much smaller
- This results in substantial performance improvement for typical queries

Main result

Workflow graph



Provenance graph

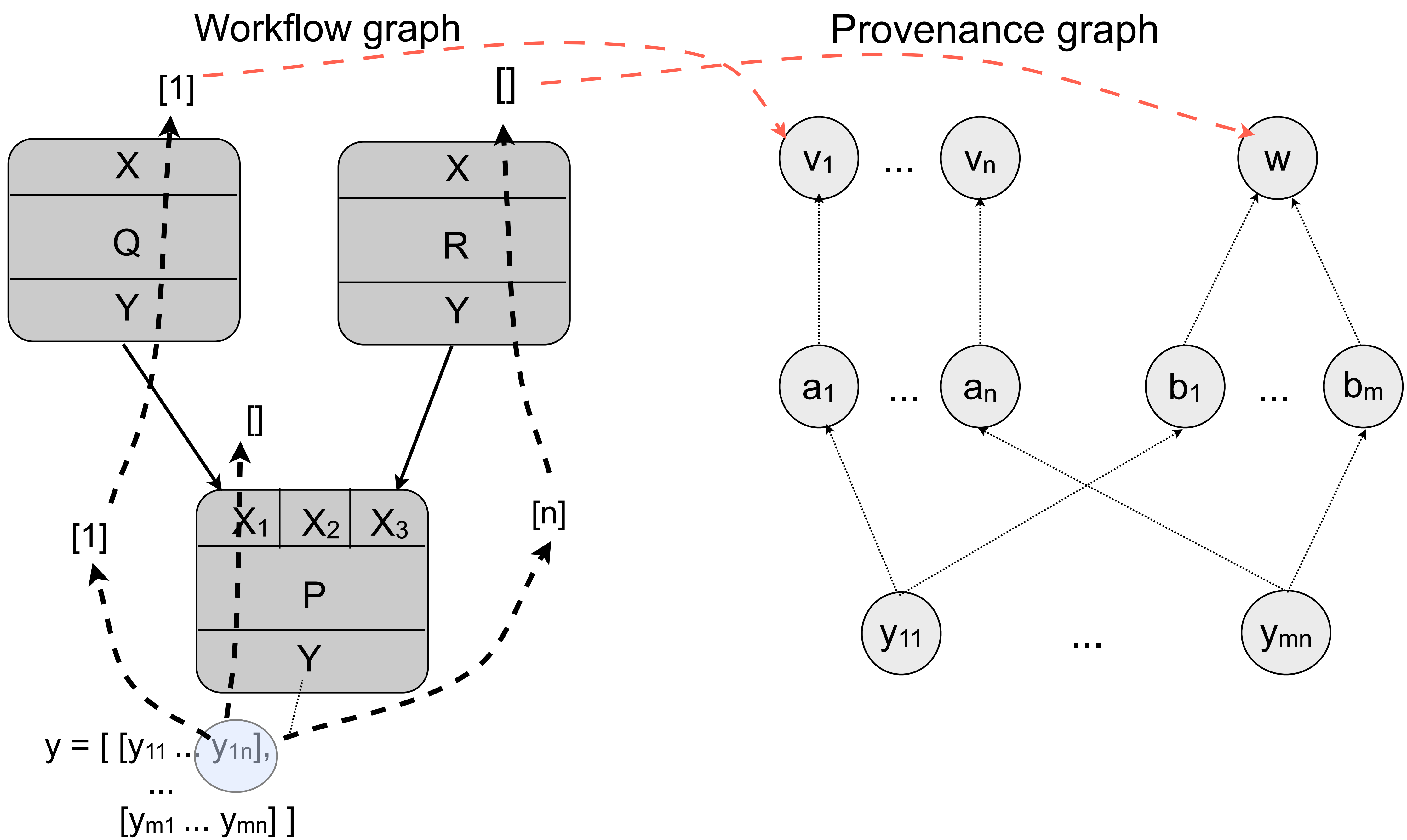


- Query the provenance of individual collections elements
- But, avoid computing transitive closures on the provenance graph
  - potentially very large
- Traverse the workflow graph instead -- much smaller
- This results in substantial performance improvement for typical queries

MANCHESTER 1824

The University of Manchester

Main result

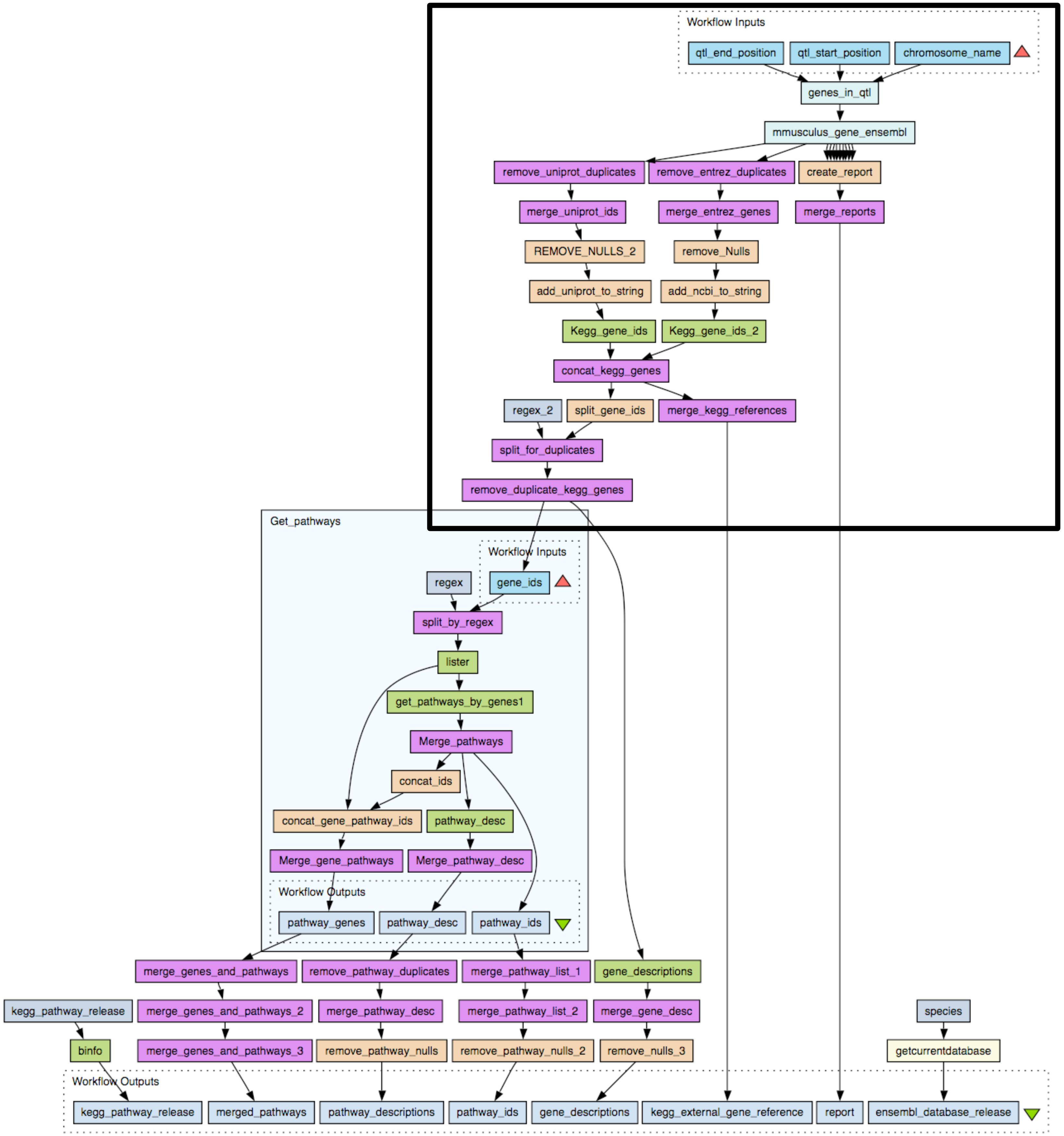


- Query the provenance of individual collections elements
- But, avoid computing transitive closures on the provenance graph
  - potentially very large
- Traverse the workflow graph instead -- much smaller
- This results in substantial performance improvement for typical queries

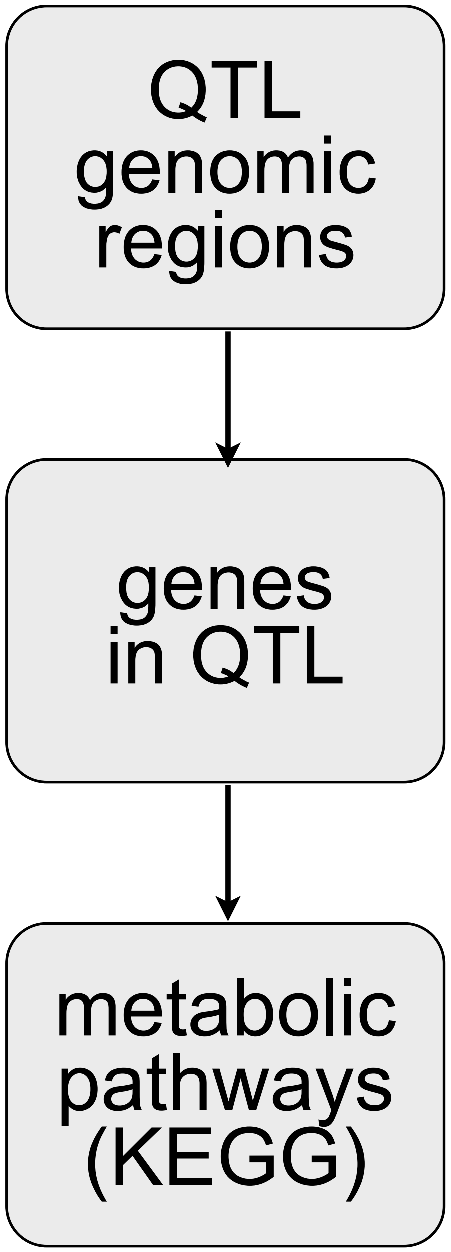
MANCHESTER
1824

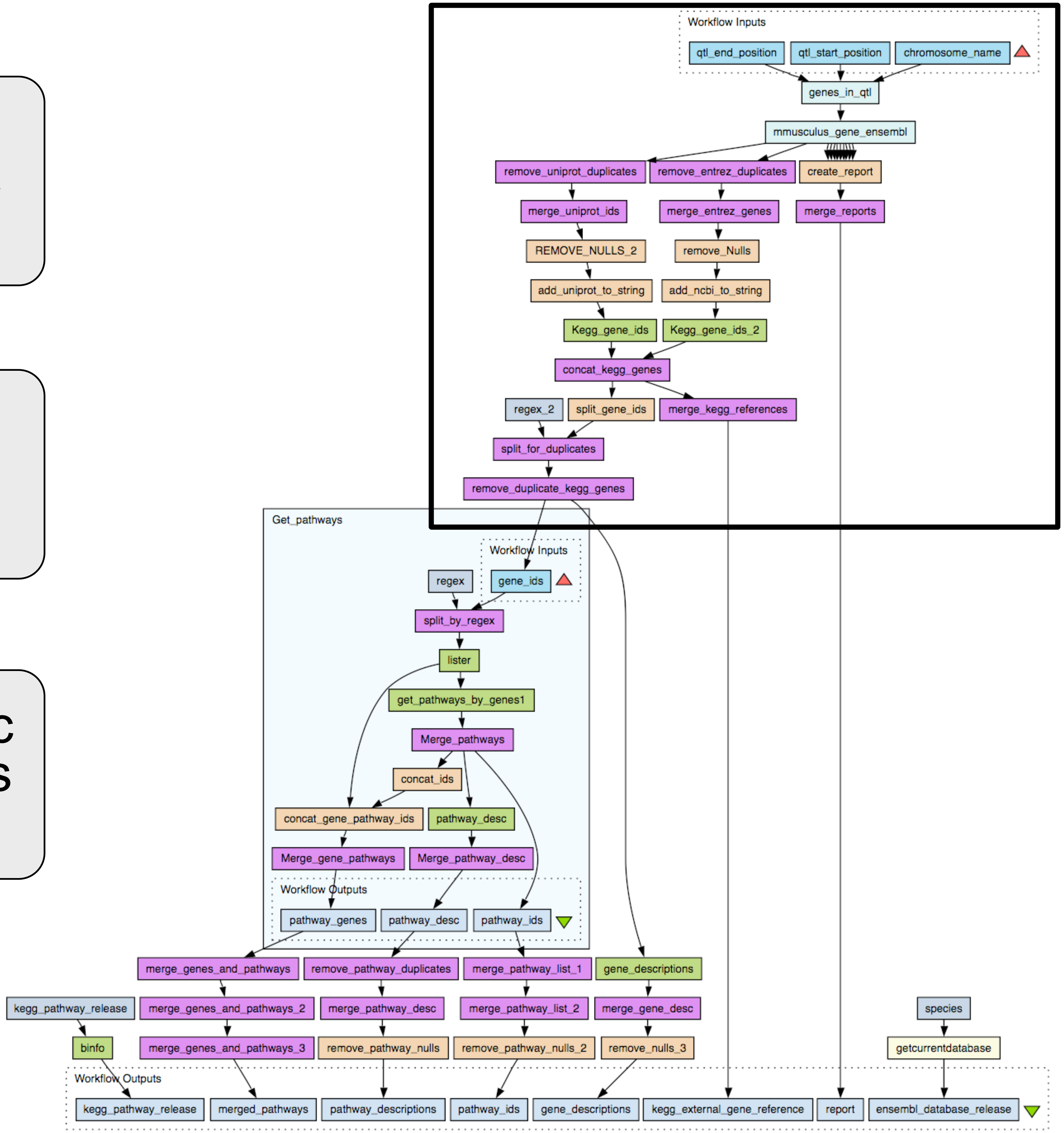
The University of Manchester

Workflow as data integrator



Workflow as data integrator





Workflow as data integrator

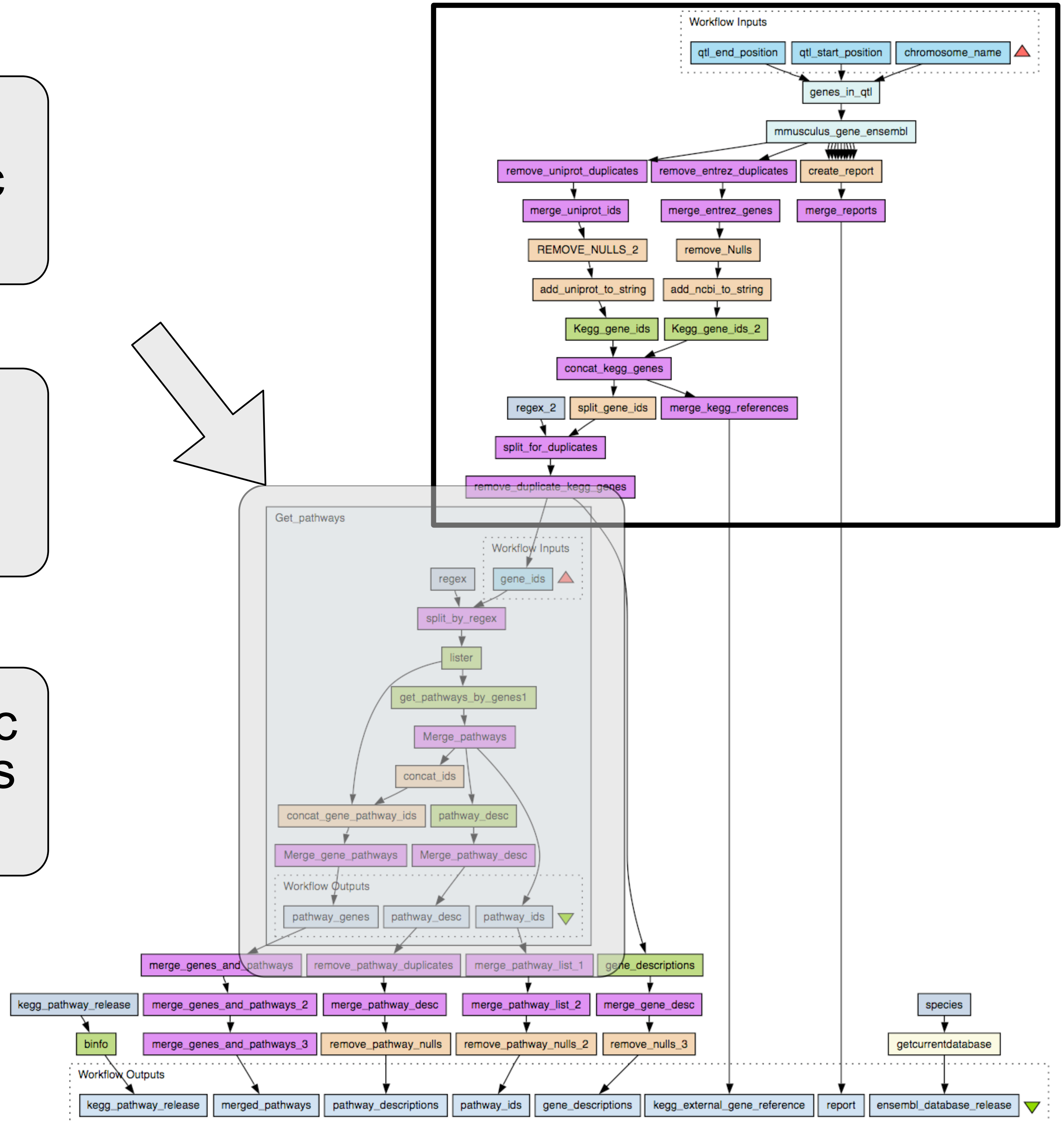
QTL genomic regions

↓

genes in QTL

↓

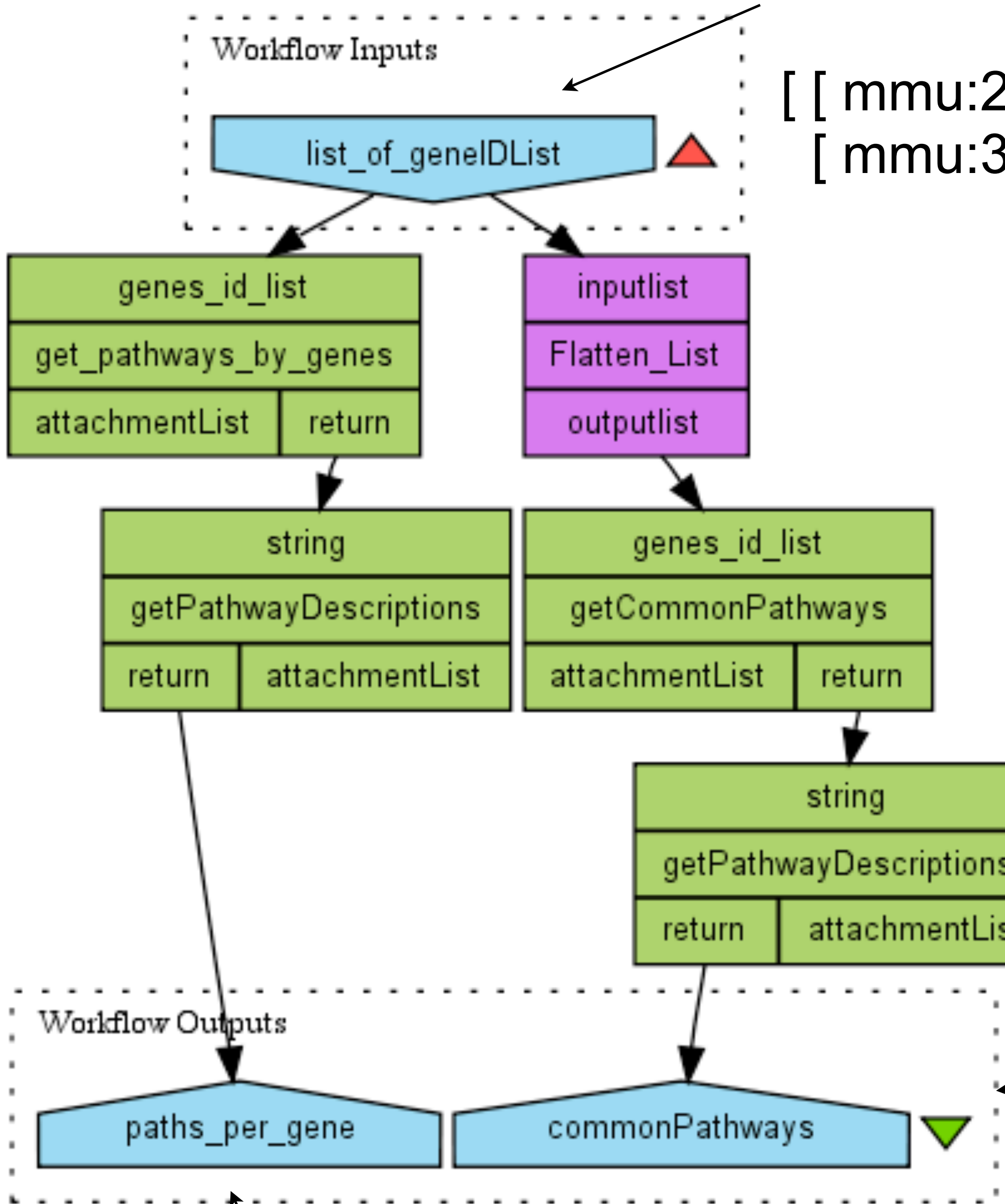
metabolic pathways (KEGG)



Motivation for fine-grained provenance

List-structured
KEGG gene ids:

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



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

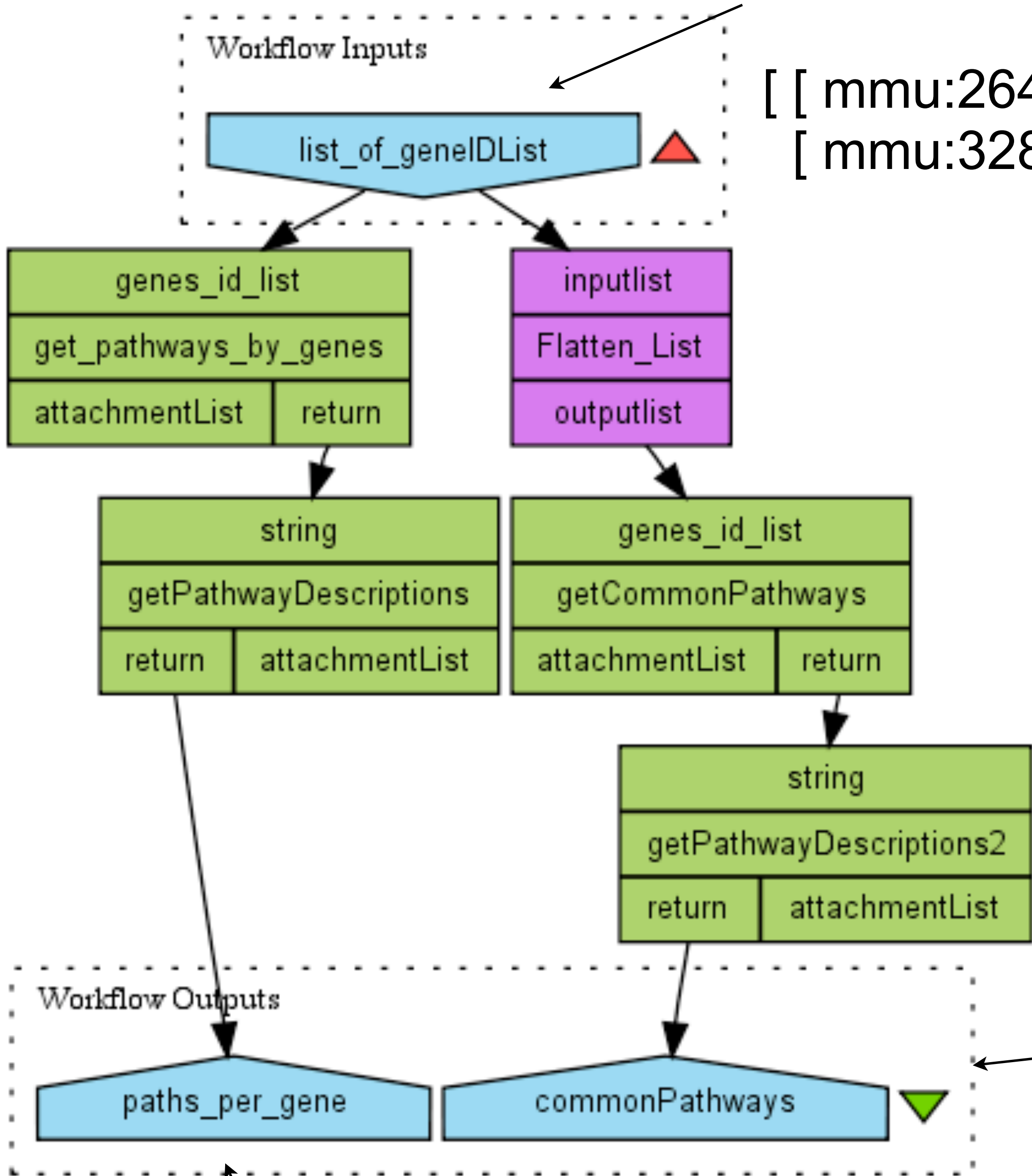
[[path:mmu04210 Apoptosis, path:mmu04010 MAPK signaling, ...],
  [path:mmu04010 MAPK signaling , path:mmu04620 Toll-like receptor, ...]]

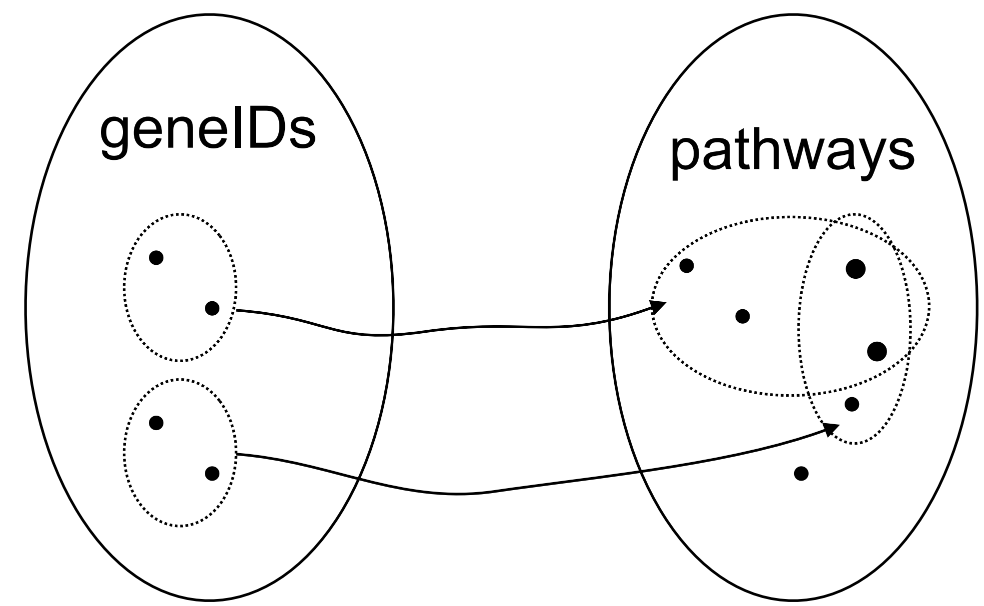
The University
of Manchester

Motivation for fine-grained provenance

List-structured
KEGG gene ids:

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





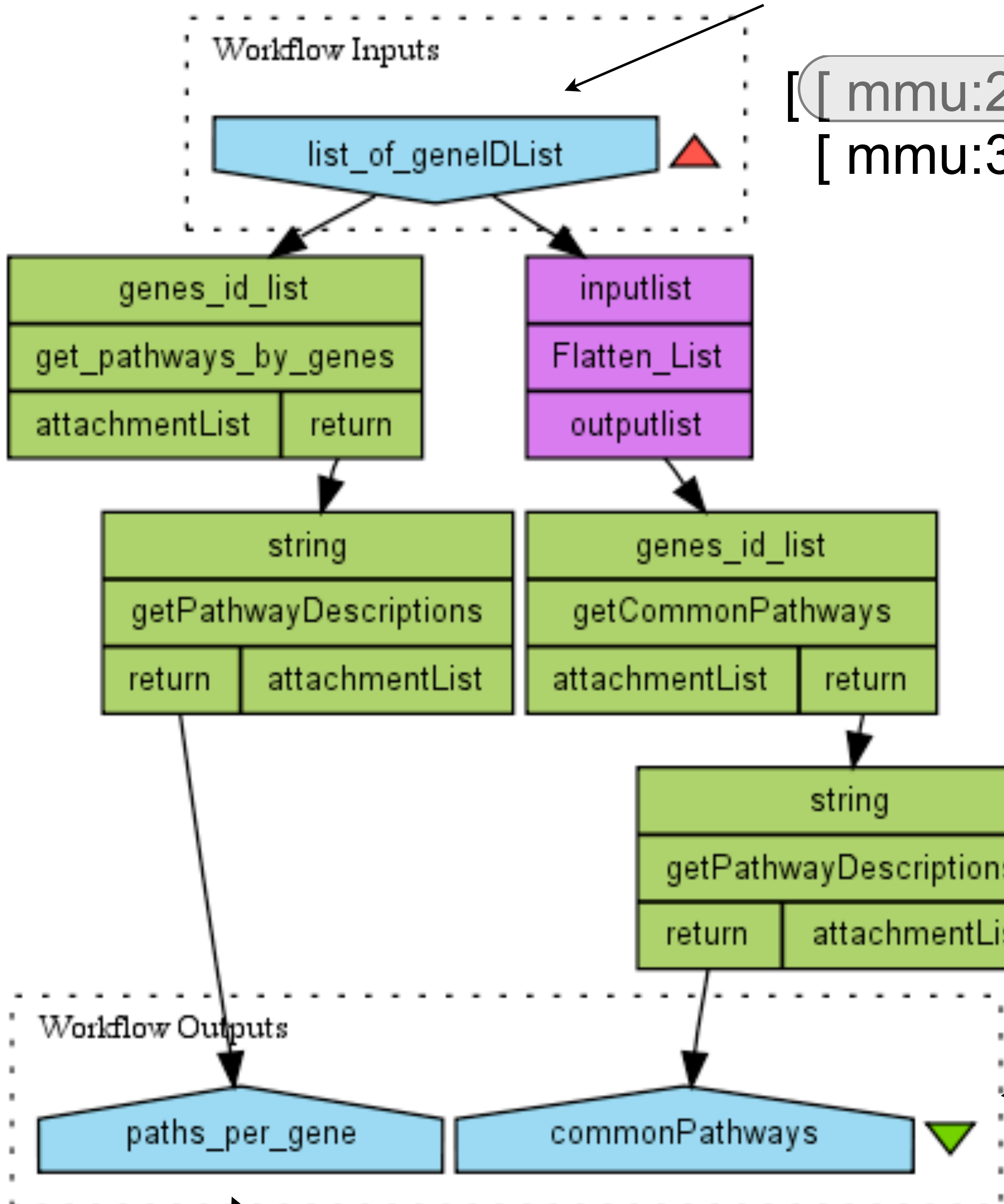
[path:mmu04010 MAPK signaling,
path:mmu04370 VEGF signaling]

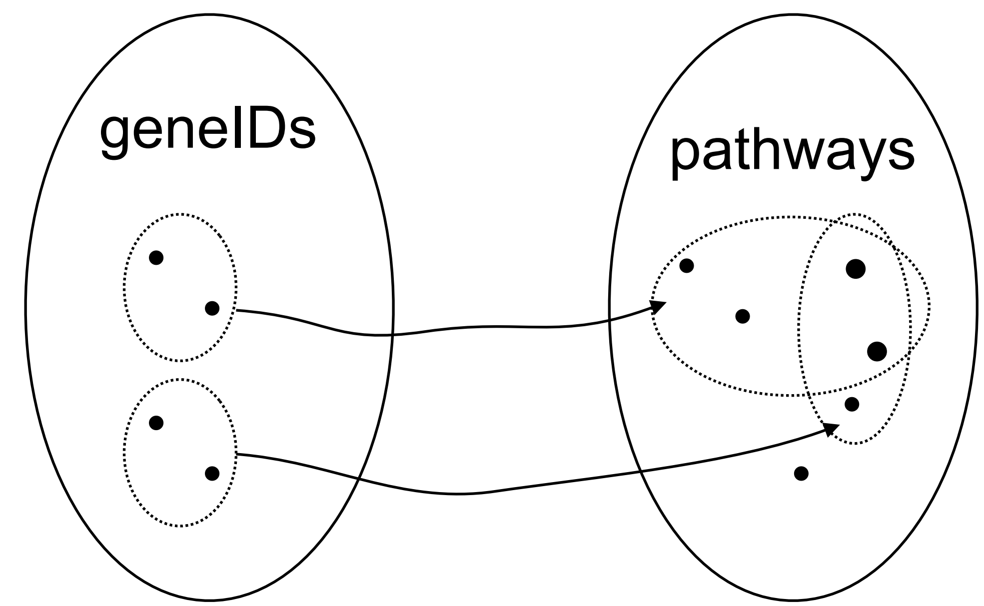
[[path:mmu04210 Apoptosis, path:mmu04010 MAPK signaling, ...],
[path:mmu04010 MAPK signaling , path:mmu04620 Toll-like receptor, ...]]

Motivation for fine-grained provenance

List-structured
KEGG gene ids:

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





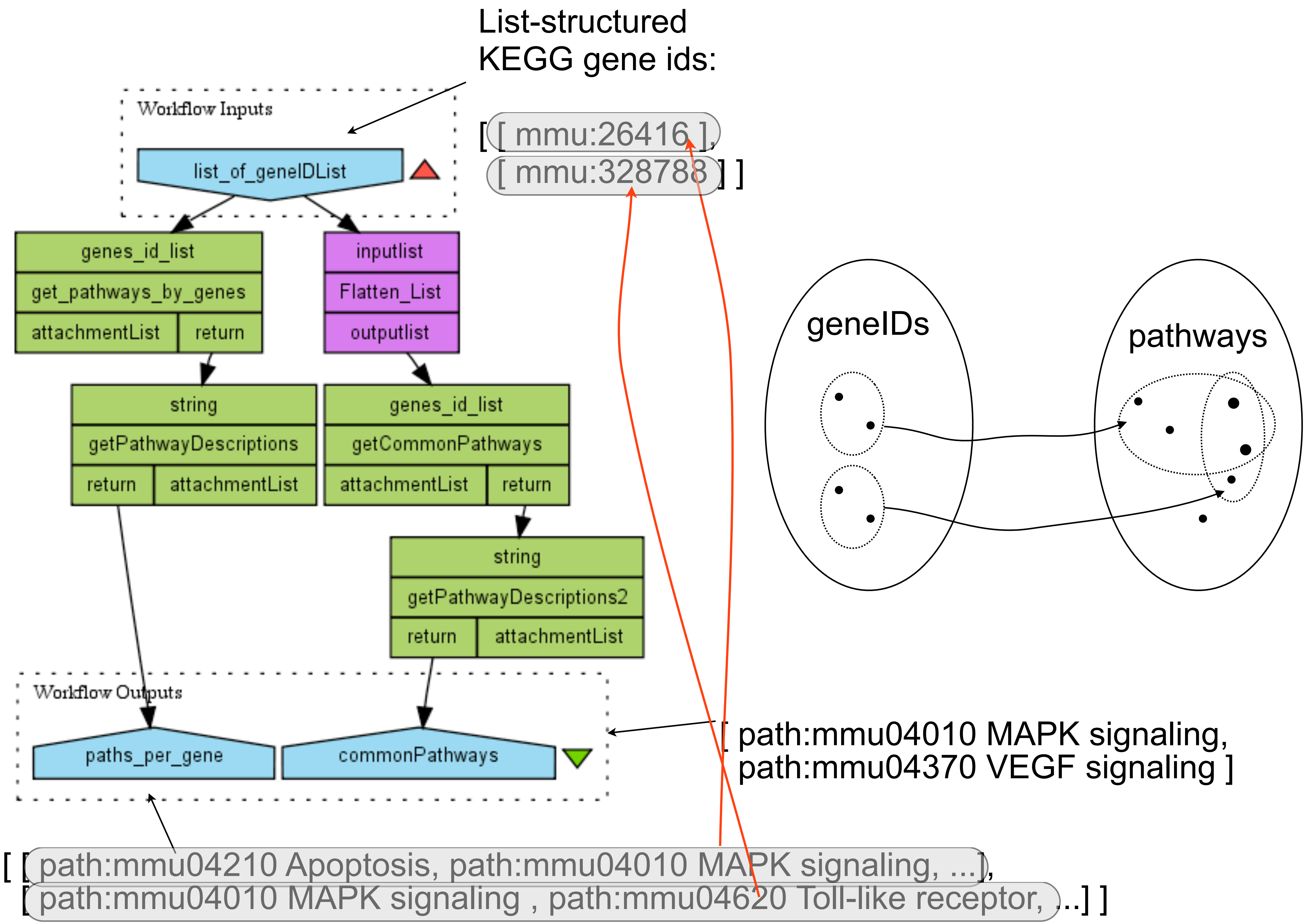
[path:mmu04010 MAPK signaling,
path:mmu04370 VEGF signaling]

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

MANCHESTER 1824

The University of Manchester

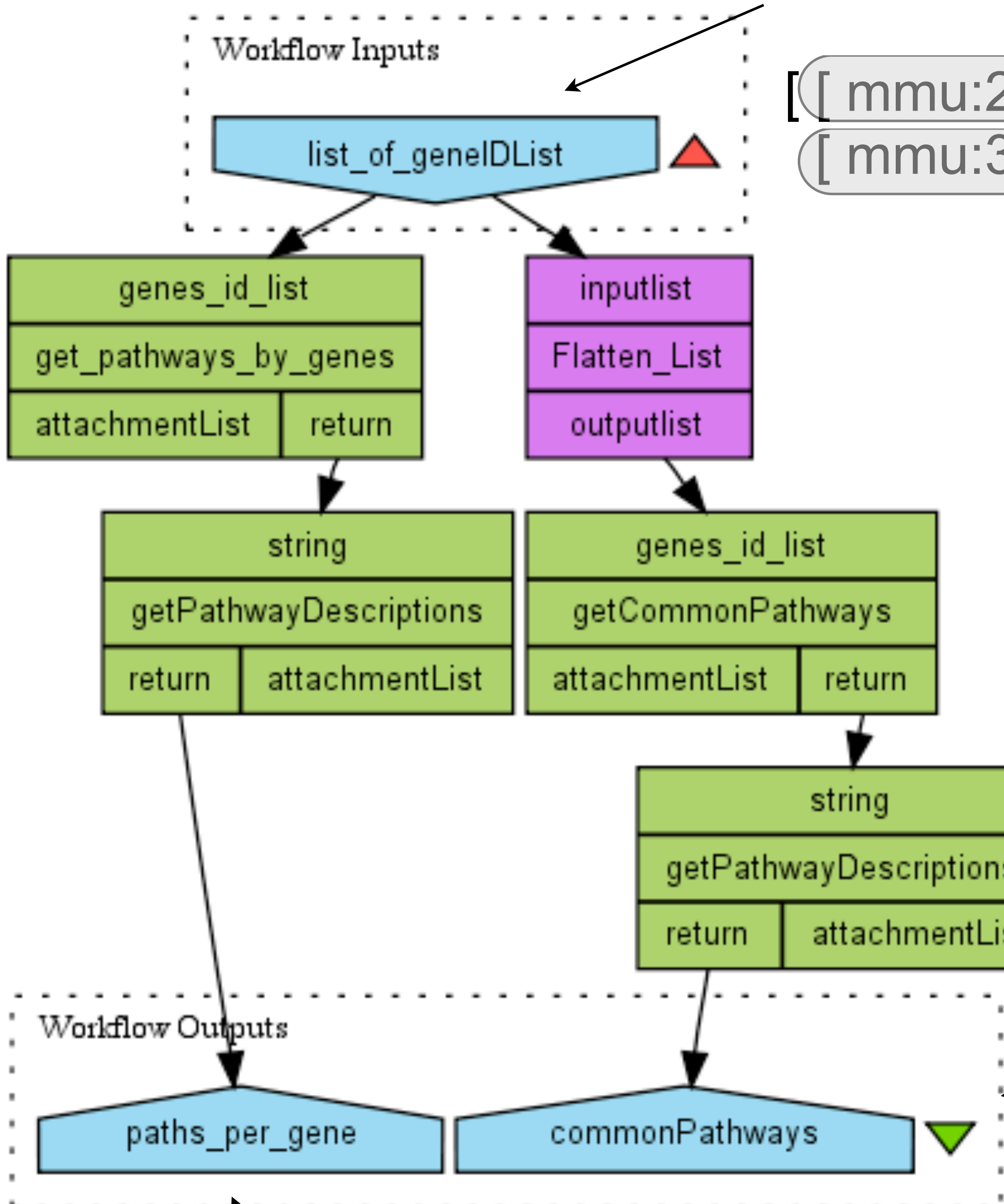
Motivation for fine-grained provenance

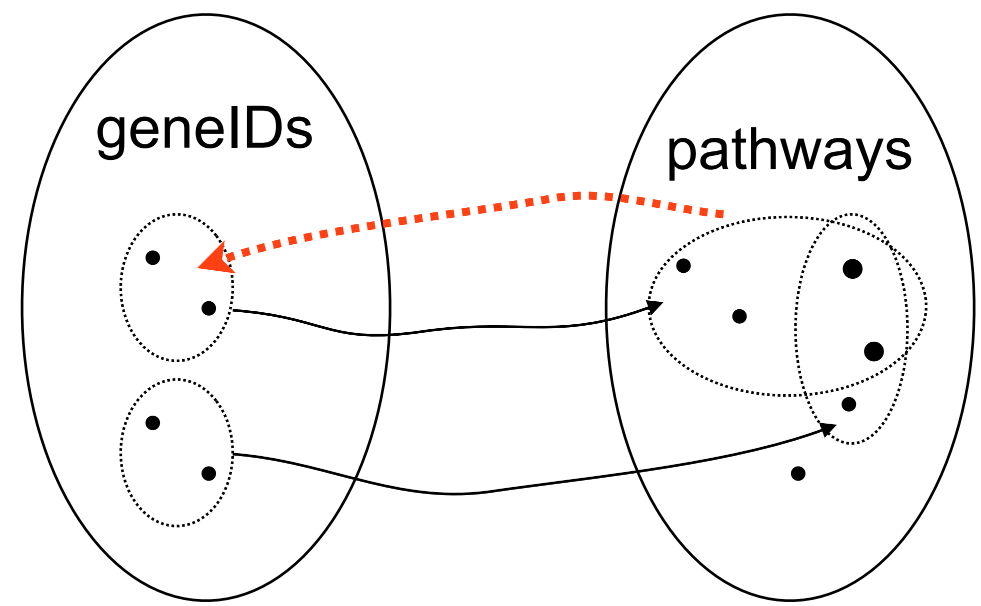


Motivation for fine-grained provenance

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, ...]]

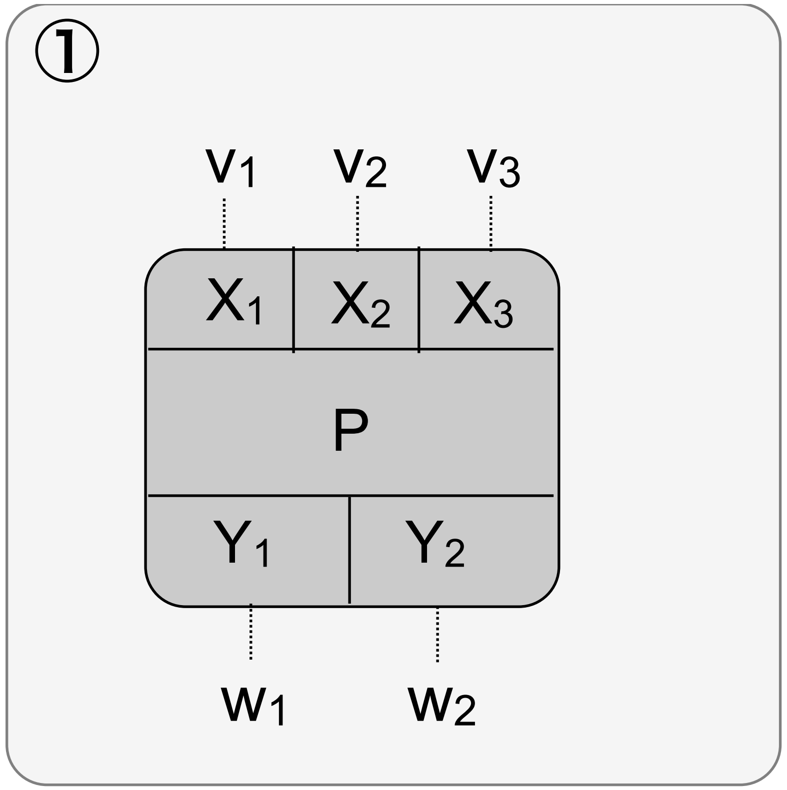
Problem statement and outline

- Setting:
  - *Black box* provenance of workflow data products

- Fine-grained provenance:
  - tracking provenance through collection elements
  - motivation
  - ➡ functional model of collection-oriented workflow processing

Functional model for collection processing /1

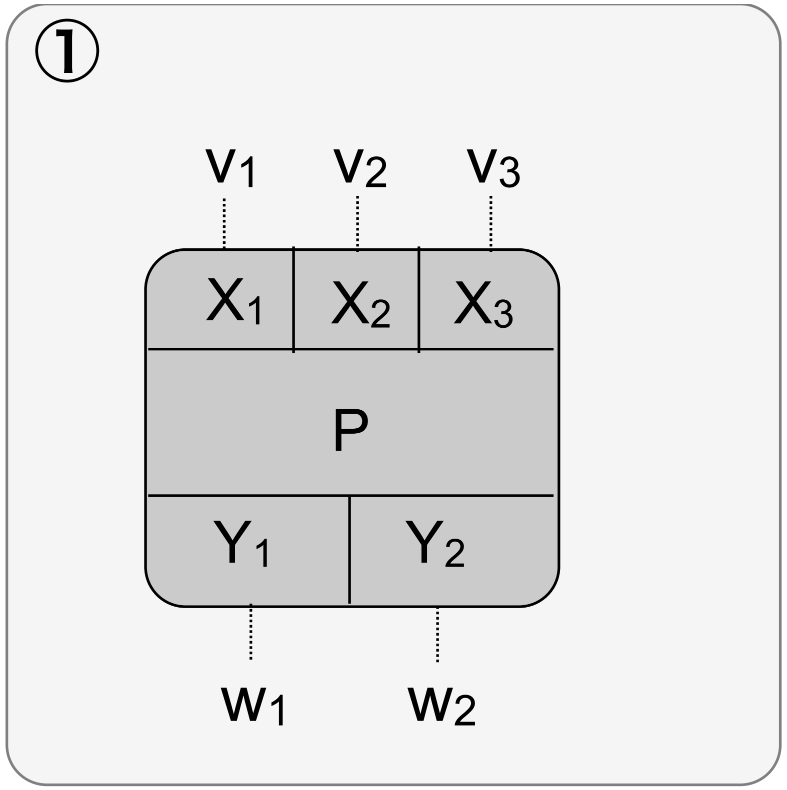
Simple processing:
service expects atomic values,
receives atomic values

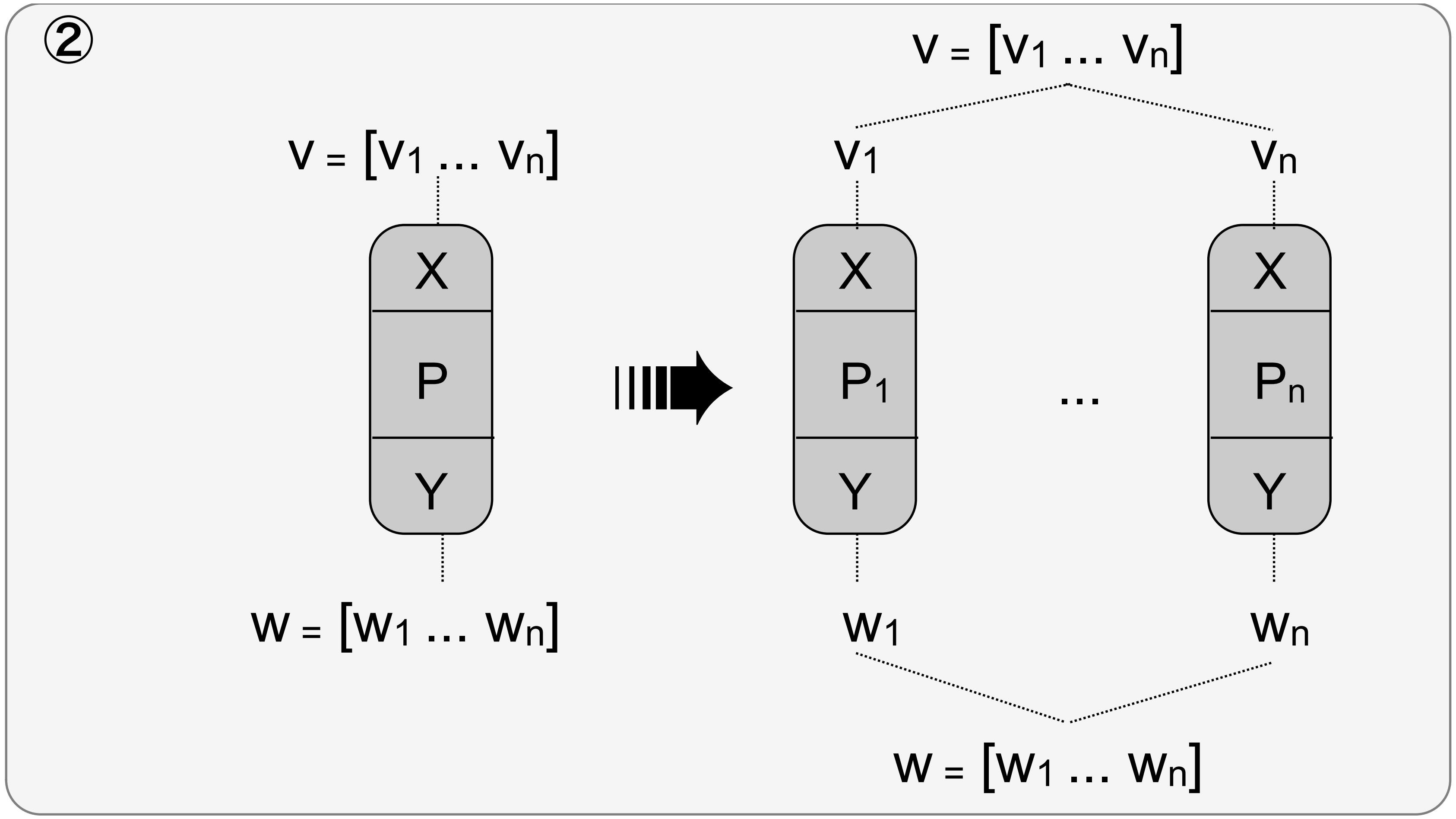


Functional model for collection processing /1

Simple processing:
service expects atomic values,
receives atomic values

Simple iteration:
service expects atomic values,
receives input **list**

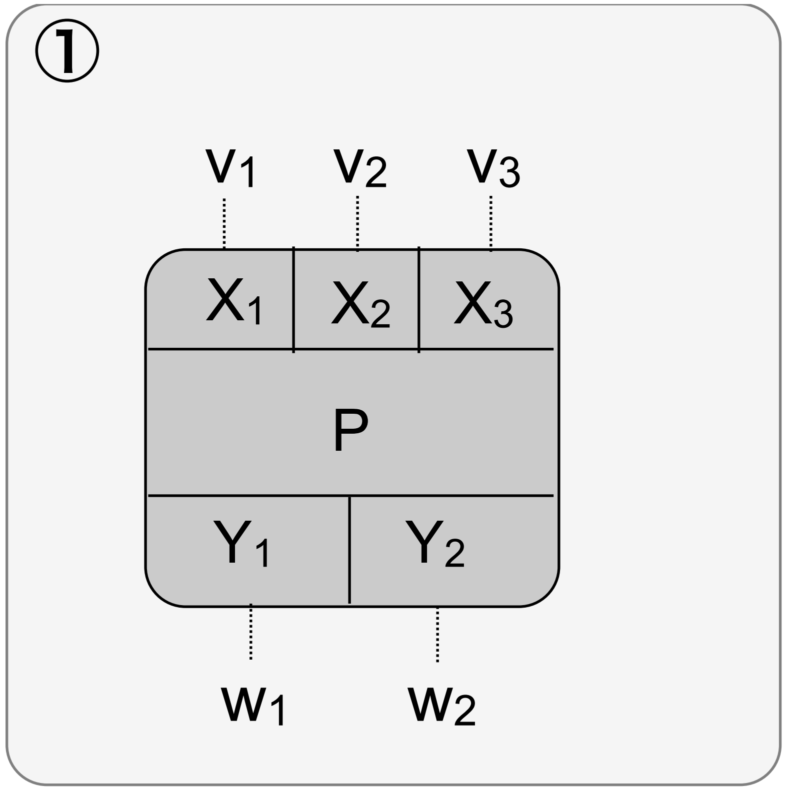


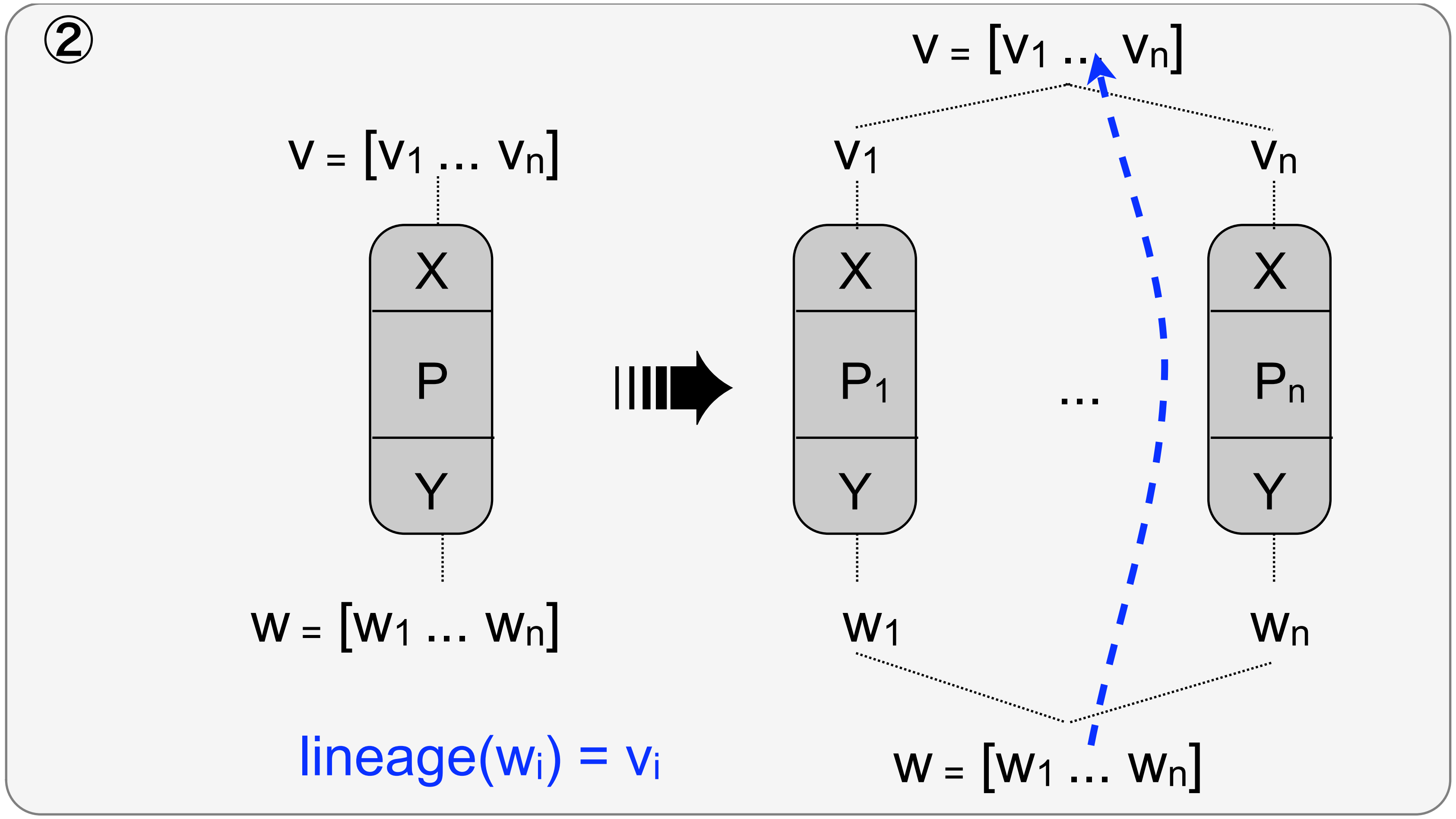


Functional model for collection processing /1

Simple processing:
service expects atomic values,
receives atomic values

Simple iteration:
service expects atomic values,
receives input **list**

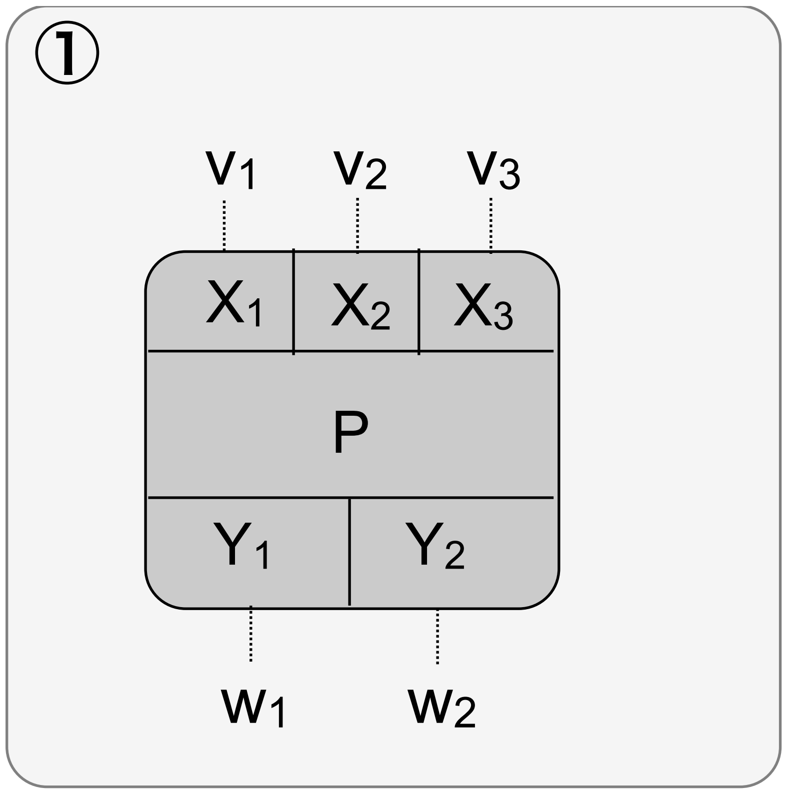


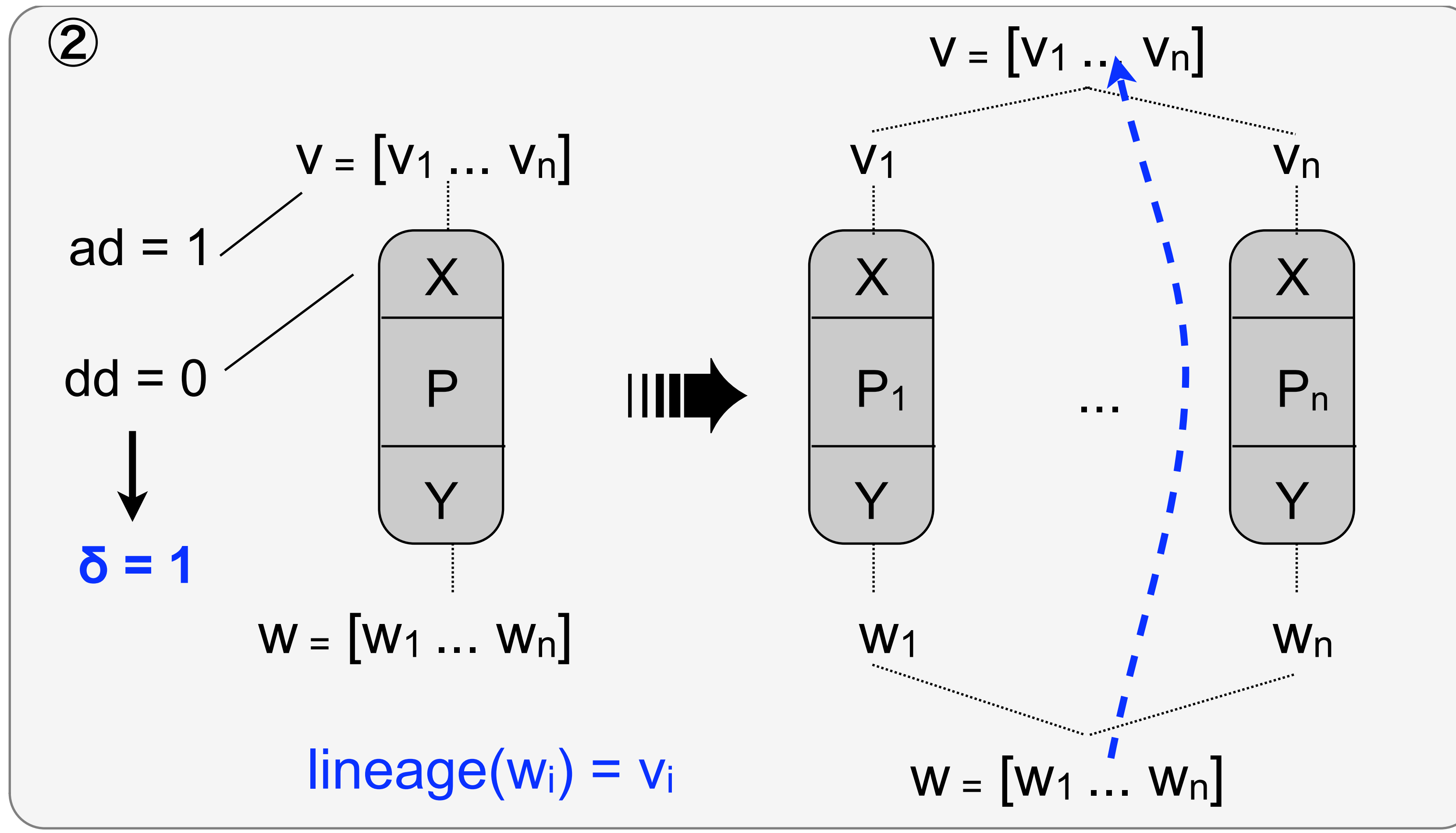


Functional model for collection processing /1

Simple processing:
service expects atomic values,
receives atomic values

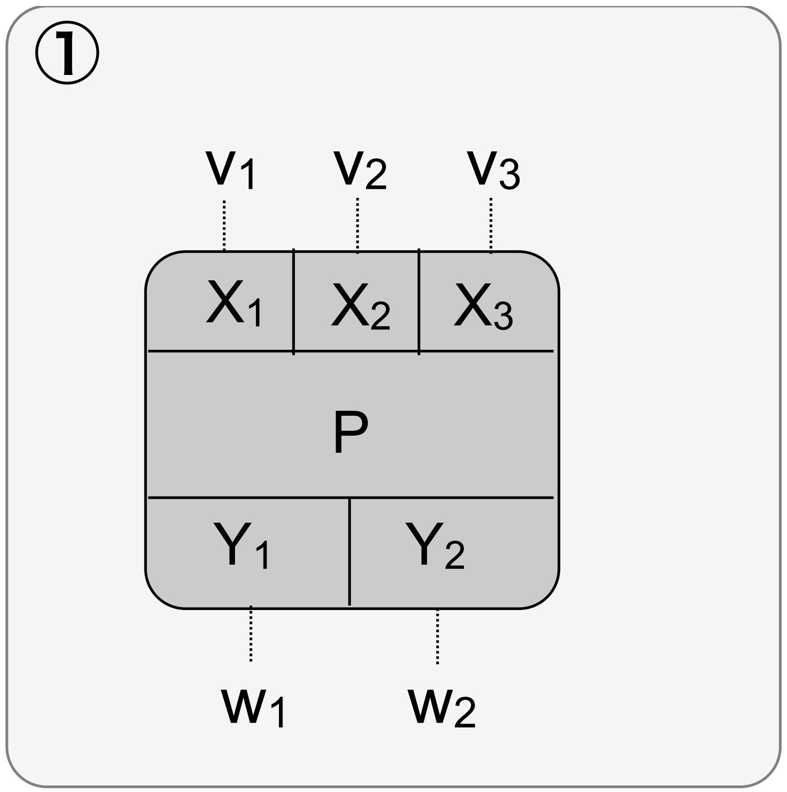
Simple iteration:
service expects atomic values,
receives input **list**



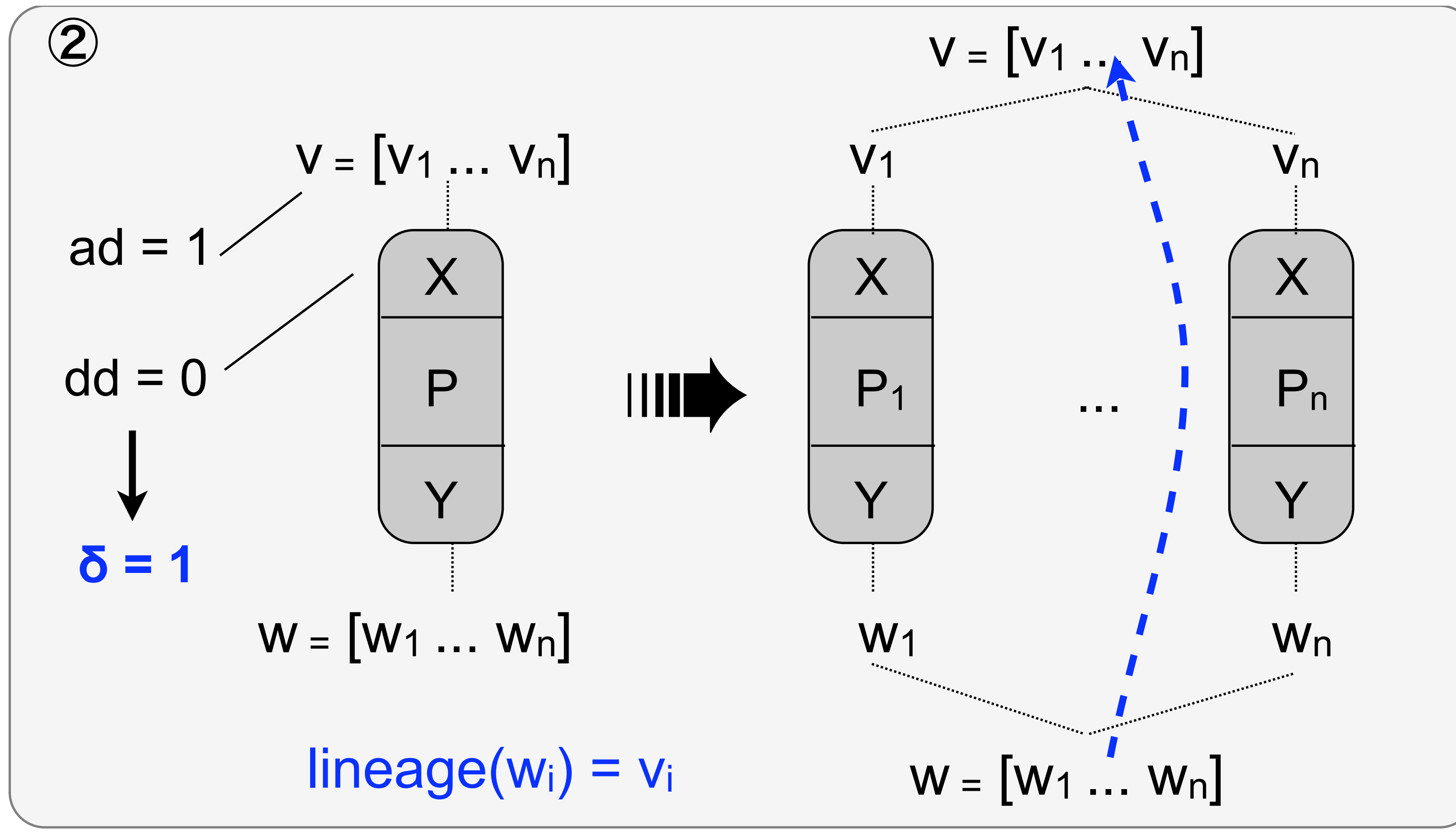


Functional model for collection processing /1

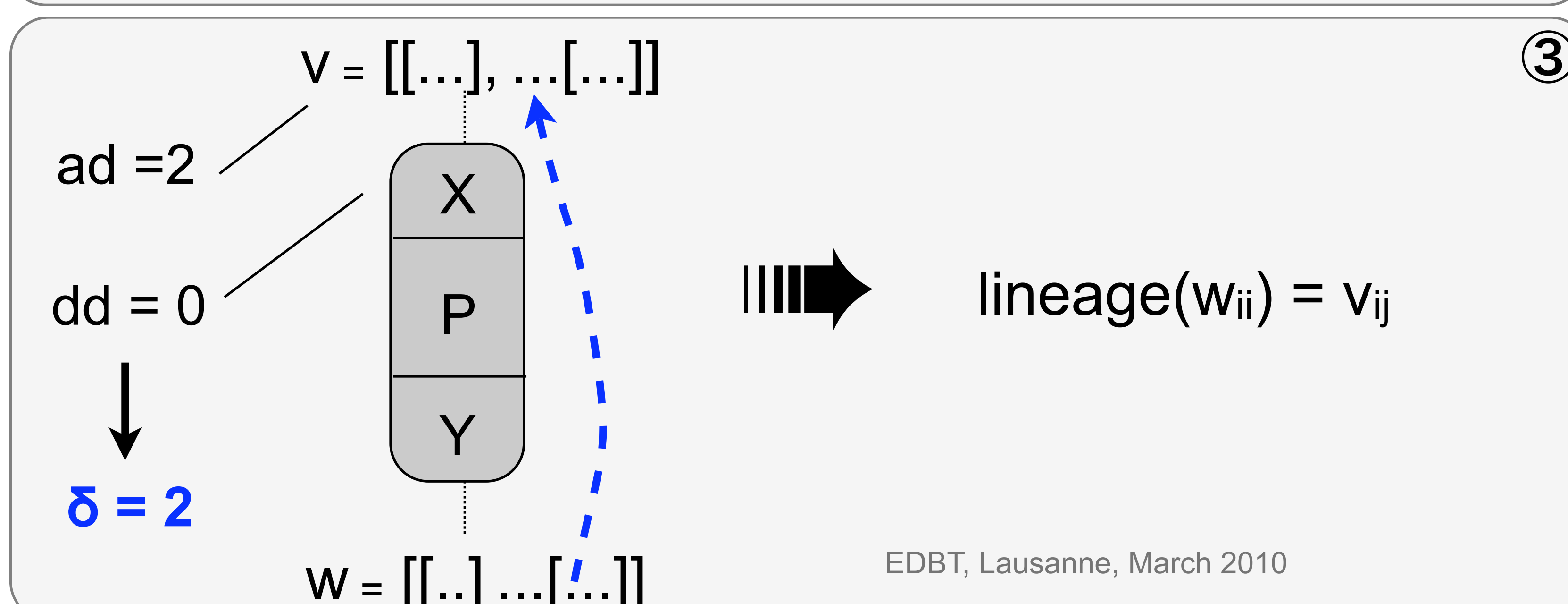
Simple processing:
service expects atomic values,
receives atomic values



Simple iteration:
service expects atomic values,
receives input **list**

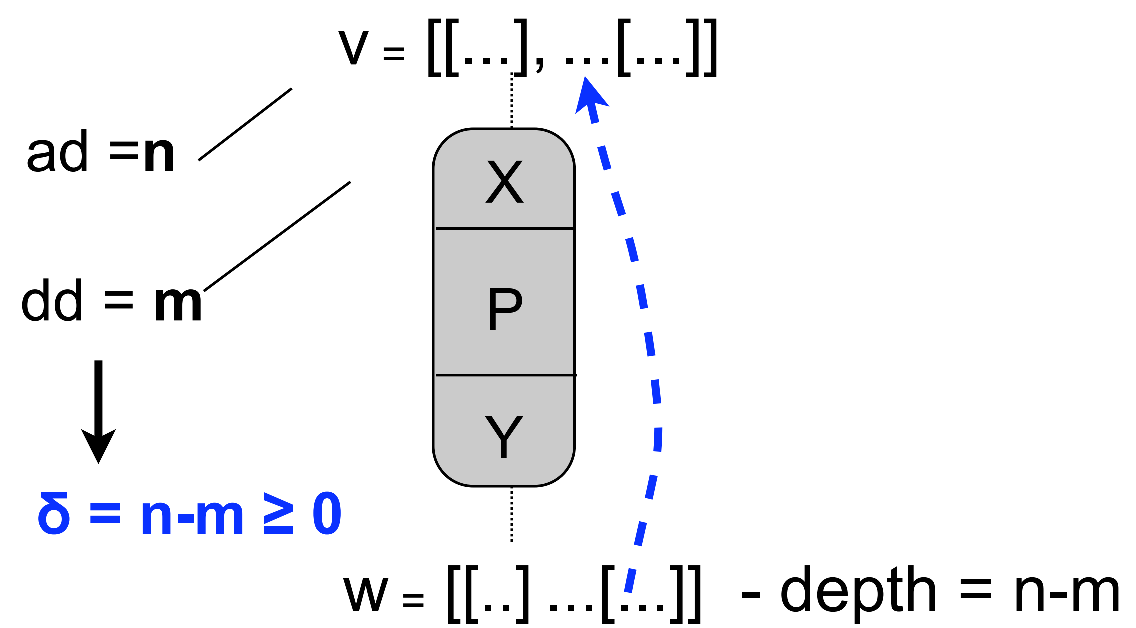


Extension:
service expects atomic values,
receives input **nested list**



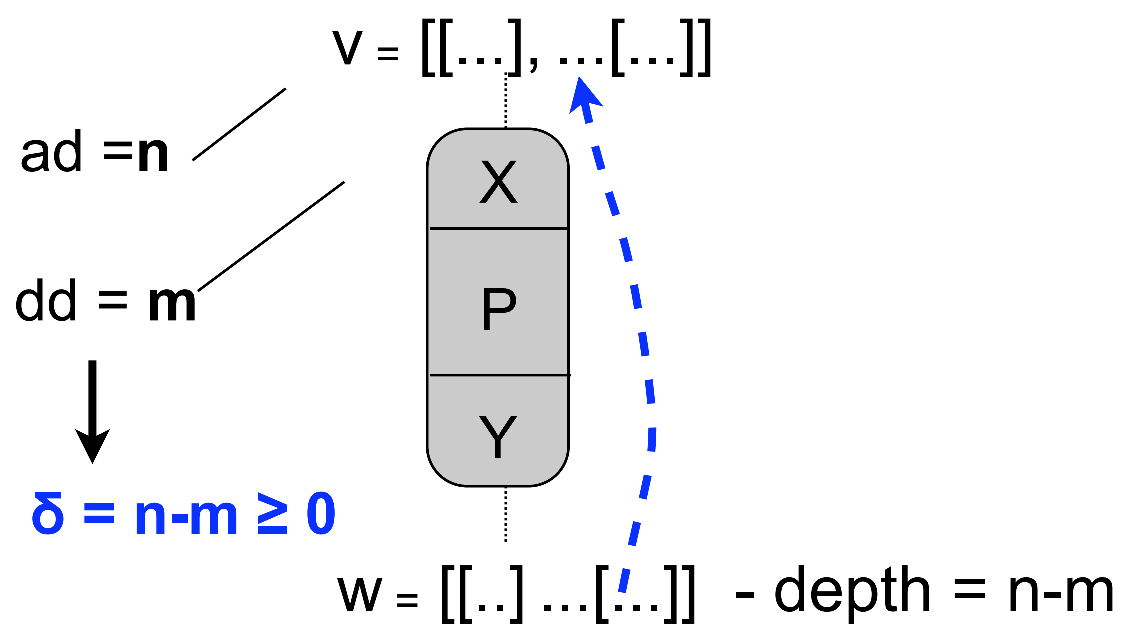
Functional model /2

The simple iteration model generalises by induction to a generic **δ=n-m**



Functional model /2

The simple iteration model generalises by induction to a generic **δ=n-m**



This leads to a recursive functional formulation for simple collection processing:

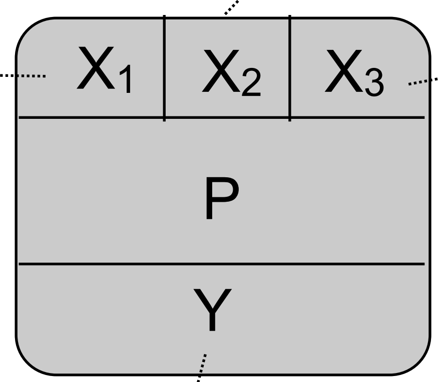
$$v = \langle a_1 \ldots a_n \rangle$$

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

Functional model - multiple inputs /3

$v_2 = [v_{21} \ldots v_{2k}]$

$v_1 = [v_{11} \ldots v_{in}]$



$v_3 = [v_{31} \ldots v_{3m}]$

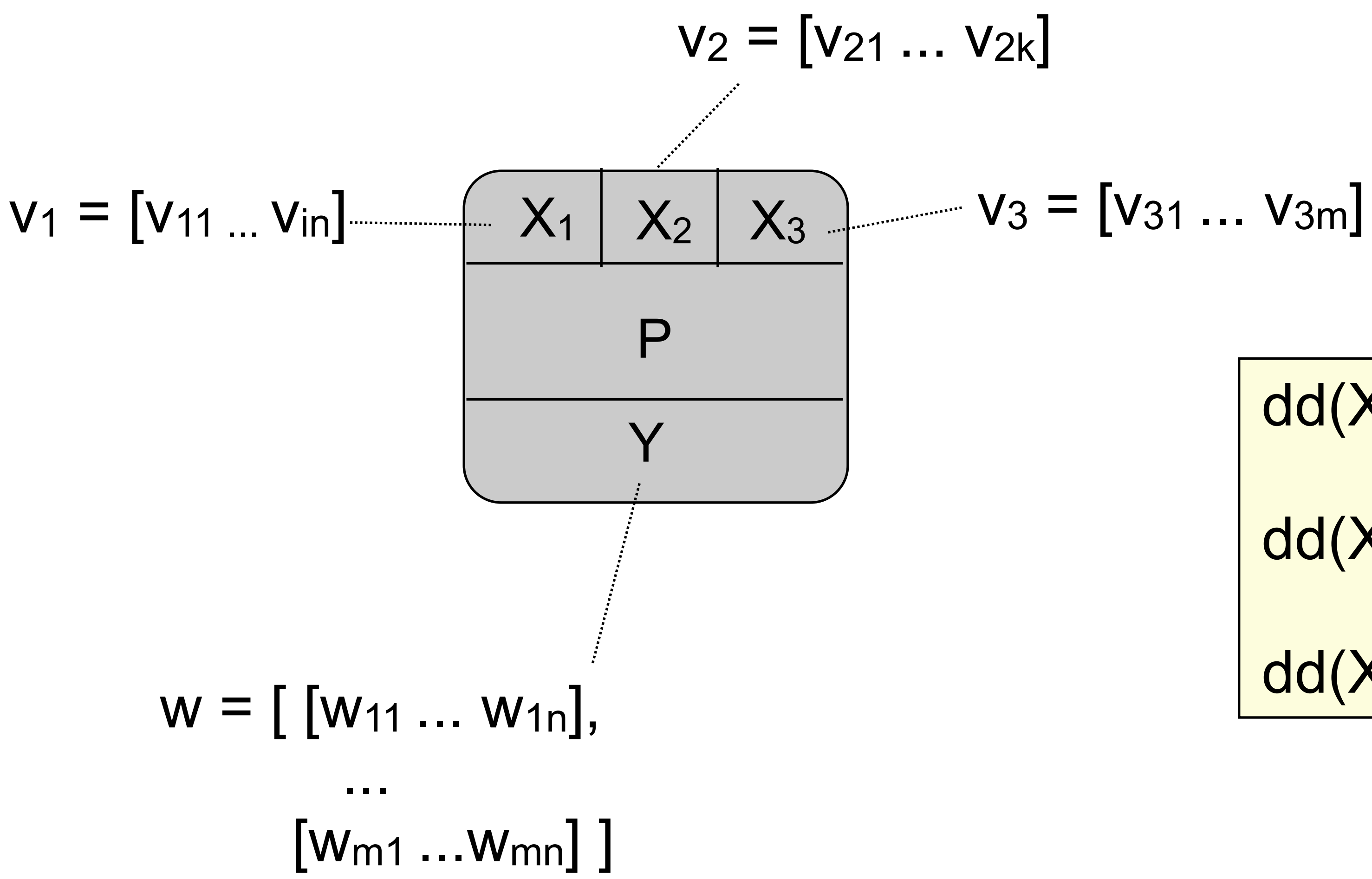
$w = [\,[w_{11} \ldots w_{1n}],$
$\ldots$
$[w_{m1} \ldots w_{mn}]\,]$

$dd(X_1) = 0,\ ad(v_1) = 1 \Rightarrow \boldsymbol{\delta_1 = 1}$

$dd(X_2) = 1,\ ad(v_2) = 1 \Rightarrow \boldsymbol{\delta_2 = 0}$

$dd(X_3) = 0,\ ad(v_3) = 1 \Rightarrow \boldsymbol{\delta_3 = 1}$

Functional model - multiple inputs /3



$dd(X_1) = 0,\ ad(v_1) = 1 \Rightarrow \boldsymbol{\delta_1 = 1}$

$dd(X_2) = 1,\ ad(v_2) = 1 \Rightarrow \boldsymbol{\delta_2 = 0}$

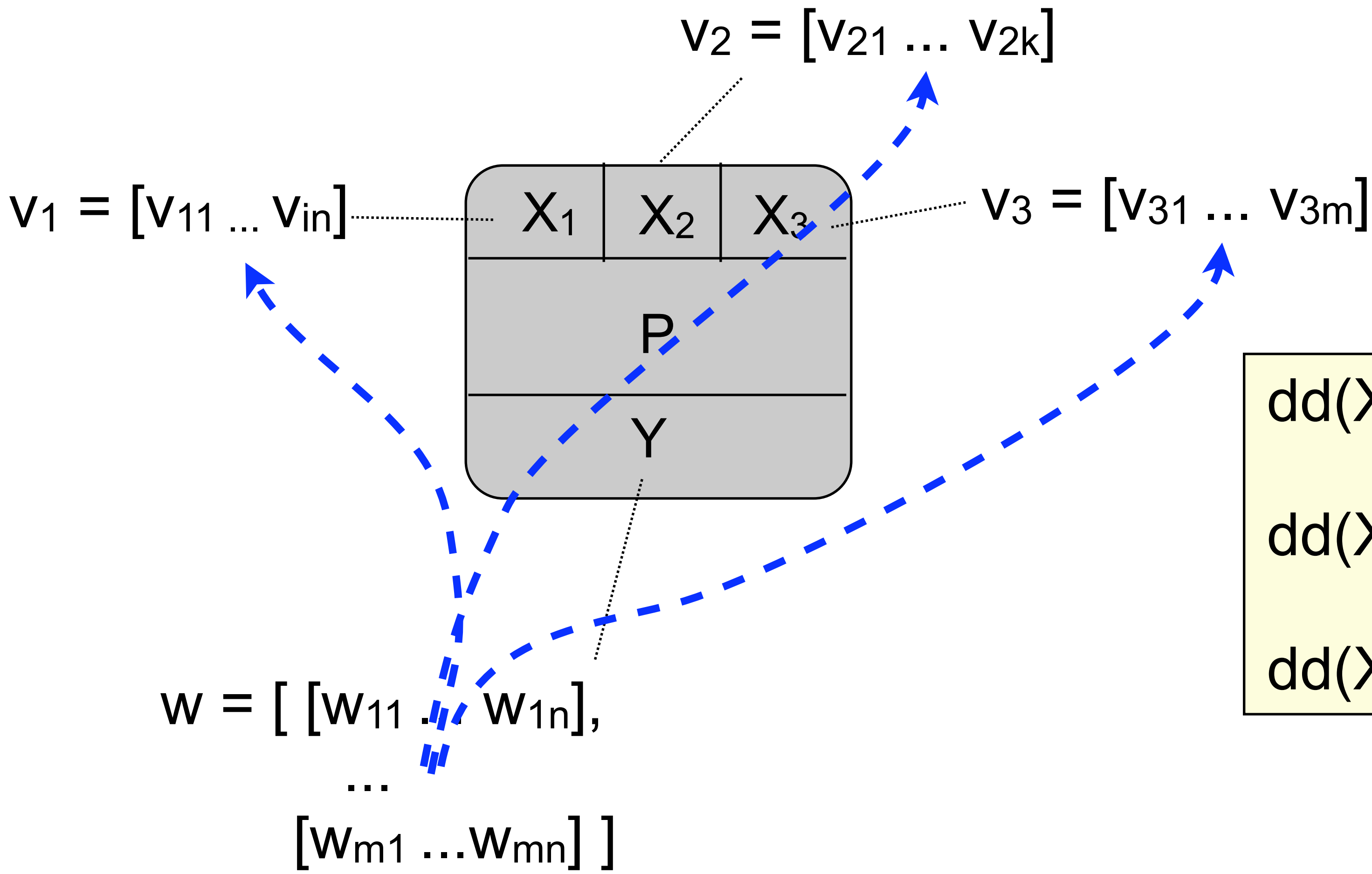
$dd(X_3) = 0,\ ad(v_3) = 1 \Rightarrow \boldsymbol{\delta_3 = 1}$

Cross-product involving v_1 and v_2 (but not v_3):

$v_1 \otimes v_3 = [\,[\,<v_{1i}, v_{3j}> \mid j{:}1..m\,] \mid i{:}1..n\,]$ // cross product

and including v_2: $[\,[\,<v_{1i}, v_2, v_{3j}> \mid j{:}1..m\,] \mid i{:}1..n\,]$

Functional model - multiple inputs /3



$dd(X_1) = 0,\ \ ad(v_1) = 1 \Rightarrow$ $\boldsymbol{\delta_1 = 1}$

$dd(X_2) = 1,\ \ ad(v_2) = 1 \Rightarrow$ $\boldsymbol{\delta_2 = 0}$

$dd(X_3) = 0,\ \ ad(v_3) = 1 \Rightarrow$ $\boldsymbol{\delta_3 = 1}$

lineage(w_{ij}) = $< v_{1i}, v_2, v_{3j} >$

Cross-product involving v_1 and v_2 (but not v_3):

$v_1 \otimes v_3 = [\ [\ <v_{1i}, v_{3j}> \mid j{:}1..m\ ] \mid i{:}1..n\ ]$ // cross product

and including v_2: $[\ [\ <v_{1i}, v_2, v_{3j}> \mid j{:}1..m\ ] \mid i{:}1..n\ ]$

Generalised cross product

Binary product, δ = 1: $a \times b = [[\langle a_i, b_j \rangle] | b_j \leftarrow b] | a_i \leftarrow a]$

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

Generalised cross product

Binary product, δ = 1: $a \times b = [[\langle a_i, b_j \rangle] | b_j \leftarrow b] | a_i \leftarrow a]$

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

Generalized to arbitrary depths:

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

...and to n operands: $\bigotimes_{i:1 \ldots n} (v_i, d_i)$

Generalised cross product

Binary product, δ = 1:

$$a \times b = [[\langle a_i, b_j \rangle] | b_j \leftarrow b] | a_i \leftarrow a]$$

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

Generalized to arbitrary depths:

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

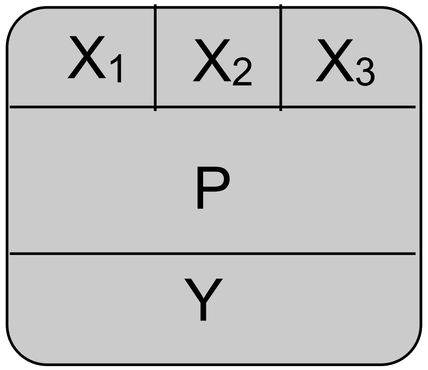
...and to n operands: $\bigotimes_{i:1\ldots n}(v_i, d_i)$

Finally: general functional semantics for collection-based processing

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

Static mapping of output to input values

The iteration structure can be determined statically



$dd(X_1) = 0, \; ad(v_1) = 1 \Rightarrow \boldsymbol{\delta_1 = 1}$

$dd(X_2) = 1, \; ad(v_2) = 1 \Rightarrow \boldsymbol{\delta_2 = 0}$

$dd(X_3) = 0, \; ad(v_3) = 1 \Rightarrow \boldsymbol{\delta_3 = 1}$

this leads to a simple mapping rule:

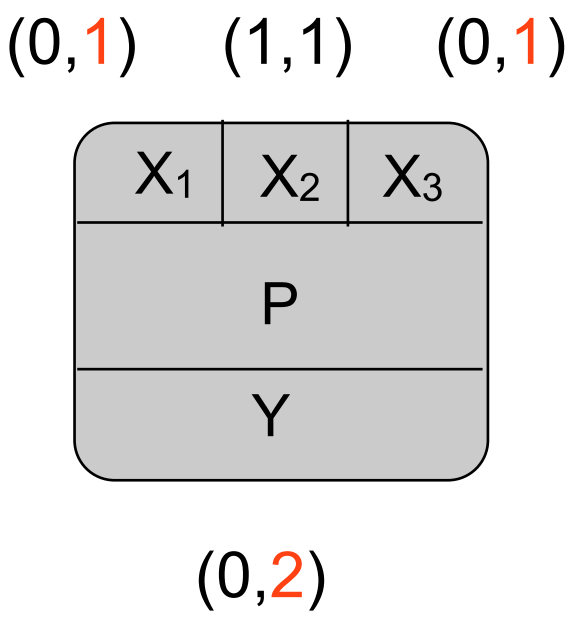
index of an output list value ➔ {index of input values}

$$Y[i.j] \rightarrow X1[i], X2[], X3[j]$$

$[i_1 . i_2 . \ldots . i_k] = \underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$

Static mapping of output to input values

The iteration structure can be determined statically



$dd(X_1) = 0,\ \ ad(v_1) = 1 \Rightarrow \boldsymbol{\delta_1 = 1}$

$dd(X_2) = 1,\ \ ad(v_2) = 1 \Rightarrow \boldsymbol{\delta_2 = 0}$

$dd(X_3) = 0,\ \ ad(v_3) = 1 \Rightarrow \boldsymbol{\delta_3 = 1}$

this leads to a simple mapping rule:

index of an output list value ➔ {index of input values}

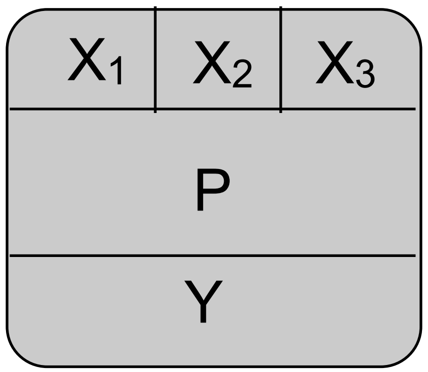
$$Y[i.j] \rightarrow X1[i], X2[], X3[j]$$

$[i_1 \,.\, i_2 \,.\, \ldots \,.\, i_k] =$ ______________________________

Static mapping of output to input values

The iteration structure can be determined statically

(0,1) (1,1) (0,1)



(0,2)

$dd(X_1) = 0, \; ad(v_1) = 1 \Rightarrow$ **$\delta_1 = 1$**
$dd(X_2) = 1, \; ad(v_2) = 1 \Rightarrow$ **$\delta_2 = 0$**
$dd(X_3) = 0, \; ad(v_3) = 1 \Rightarrow$ **$\delta_3 = 1$**

this leads to a simple mapping rule:

index of an output list value ➔ {index of input values}

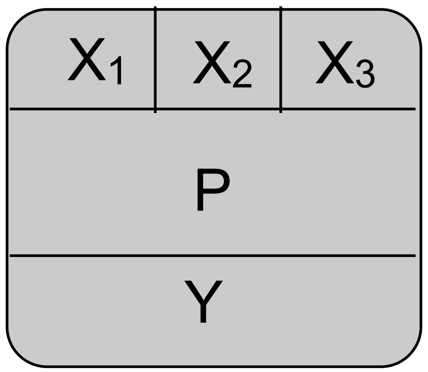
$$Y[i.j] \rightarrow X1[i], X2[], X3[j]$$

$$[i_1 \,.\, i_2 \,.\, \ldots \,.\, i_k] = \frac{\delta_1}{}$$

Static mapping of output to input values

The iteration structure can be determined statically

(0,1) (1,1) (0,1)



(0,2)

$dd(X_1) = 0,\ ad(v_1) = 1 \Rightarrow$ **$\delta_1 = 1$**

$dd(X_2) = 1,\ ad(v_2) = 1 \Rightarrow$ **$\delta_2 = 0$**

$dd(X_3) = 0,\ ad(v_3) = 1 \Rightarrow$ **$\delta_3 = 1$**

this leads to a simple mapping rule:

index of an output list value ➔ {index of input values}

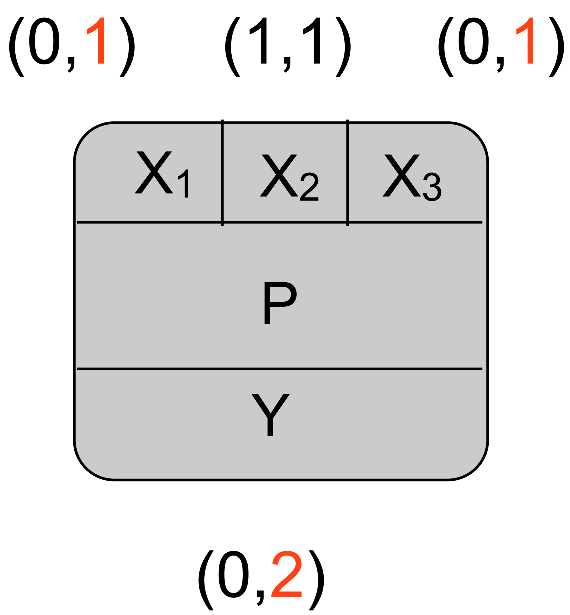
$$Y[i.j] \rightarrow X1[i], X2[], X3[j]$$

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

X_1

Static mapping of output to input values

The iteration structure can be determined statically



$dd(X_1) = 0,\ ad(v_1) = 1 \Rightarrow \delta_1 = 1$

$dd(X_2) = 1,\ ad(v_2) = 1 \Rightarrow \delta_2 = 0$

$dd(X_3) = 0,\ ad(v_3) = 1 \Rightarrow \delta_3 = 1$

this leads to a simple mapping rule:

index of an output list value ➔ {index of input values}

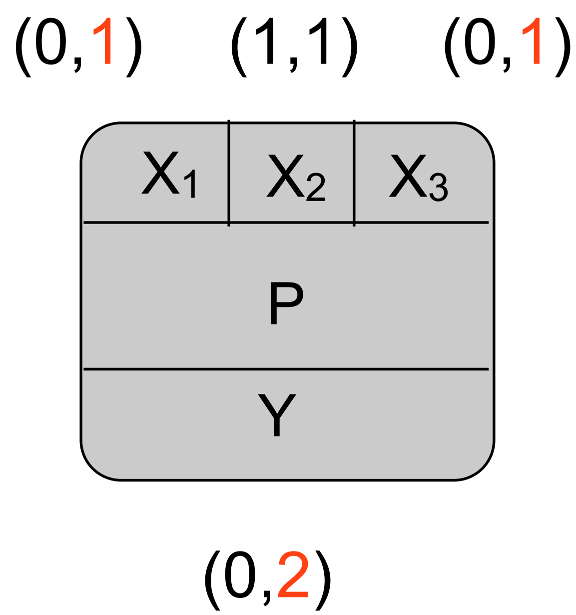
$$Y[i.j] \rightarrow X1[i], X2[], X3[j]$$

$[i_1 . i_2 . \ldots . i_k] =$ δ_2

X_1

Static mapping of output to input values

The iteration structure can be determined statically



$dd(X_1) = 0,\ ad(v_1) = 1 \Rightarrow$ **$\delta_1 = 1$**

$dd(X_2) = 1,\ ad(v_2) = 1 \Rightarrow$ **$\delta_2 = 0$**

$dd(X_3) = 0,\ ad(v_3) = 1 \Rightarrow$ **$\delta_3 = 1$**

this leads to a simple mapping rule:

index of an output list value ➔ {index of input values}

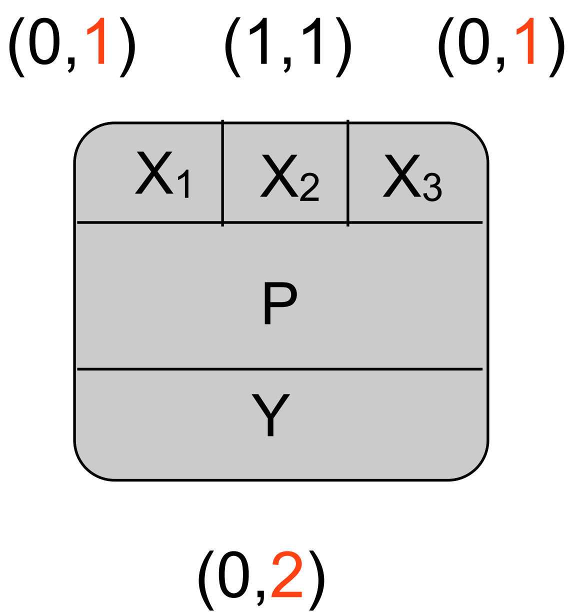
$$Y[i.j] \rightarrow X1[i], X2[], X3[j]$$

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

X_1 X_2

Static mapping of output to input values

The iteration structure can be determined statically



$dd(X_1) = 0, \; ad(v_1) = 1 \Rightarrow$ **$\delta_1 = 1$**

$dd(X_2) = 1, \; ad(v_2) = 1 \Rightarrow$ **$\delta_2 = 0$**

$dd(X_3) = 0, \; ad(v_3) = 1 \Rightarrow$ **$\delta_3 = 1$**

this leads to a simple mapping rule:

index of an output list value ➔ {index of input values}

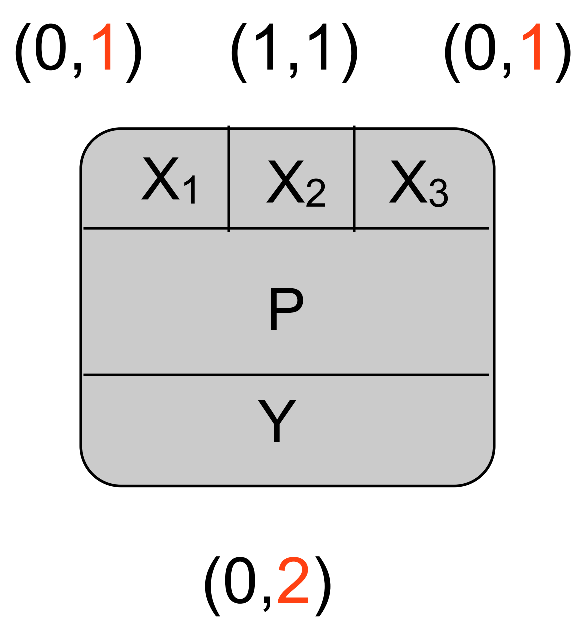
$$Y[i.j] \rightarrow X1[i], X2[], X3[j]$$

$[i_1 \,.\, i_2 \,.\, \dots \,.\, i_k] =$ $\underline{\quad\quad}$ $\underline{\quad\quad}$ $\underline{\delta_k \quad\quad}$

X_1 X_2

Static mapping of output to input values

The iteration structure can be determined statically



$dd(X_1) = 0$, $ad(v_1) = 1$ ➠ $\delta_1 = 1$

$dd(X_2) = 1$, $ad(v_2) = 1$ ➠ $\delta_2 = 0$

$dd(X_3) = 0$, $ad(v_3) = 1$ ➠ $\delta_3 = 1$

this leads to a simple mapping rule:

index of an output list value ➔ {index of input values}

$$Y[i.j] \rightarrow X1[i], X2[], X3[j]$$

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

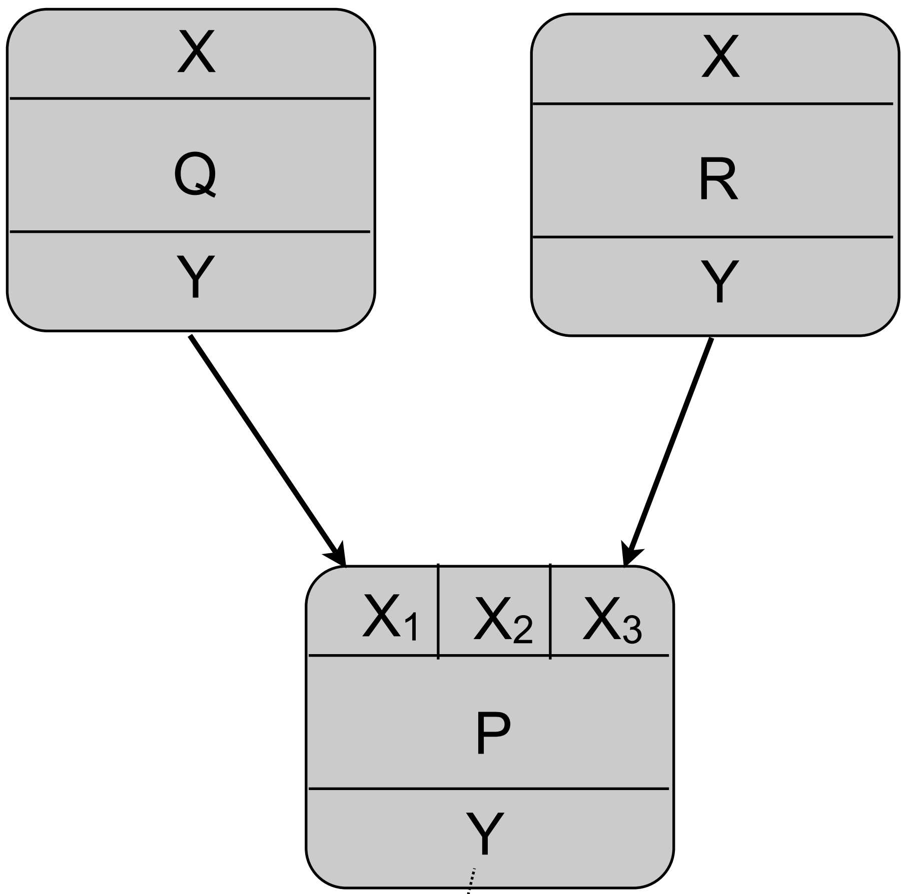
$\underline{\qquad\qquad}$ $\underline{\qquad\qquad}$ $\underline{\qquad\qquad}$

X_1 X_2 X_k

Workflow graph as index for query processing

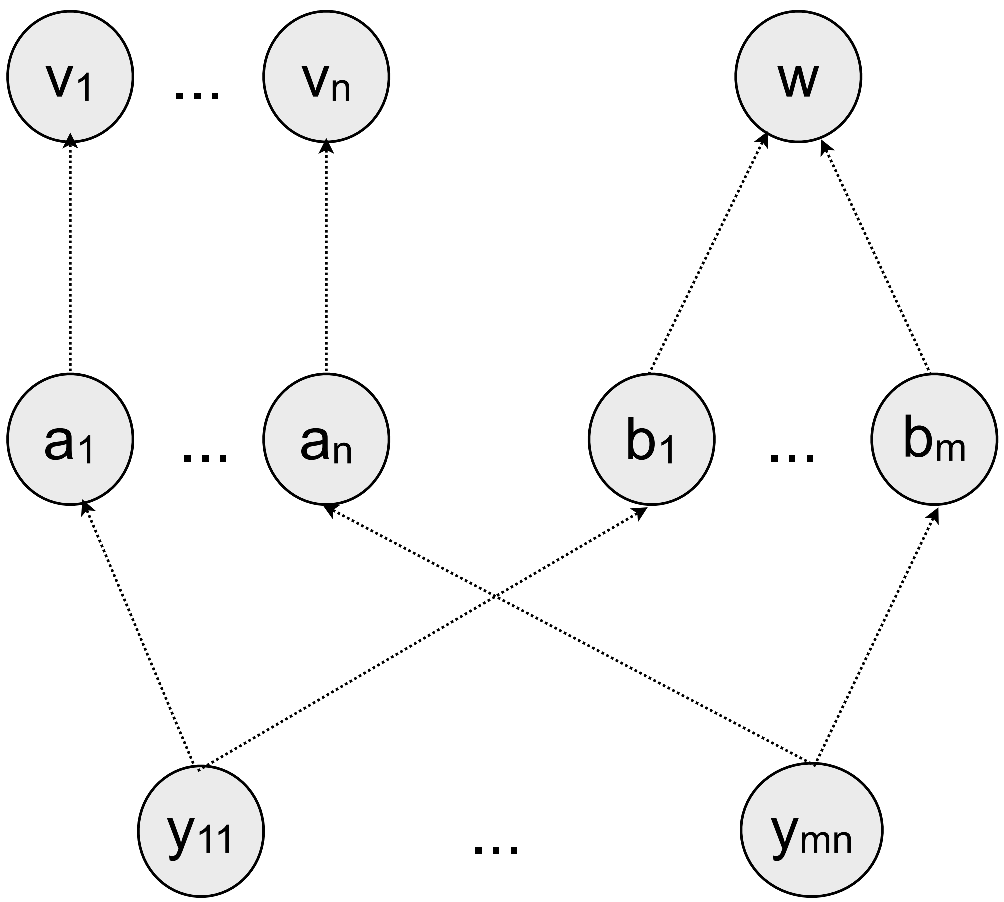
1. traverse the workflow graph (small) rather than the provenance trace (large)
2. use static prediction of iterations to trace through collection elements
3. "parachute" into the actual trace only at the end

Workflow graph



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

Provenance graph



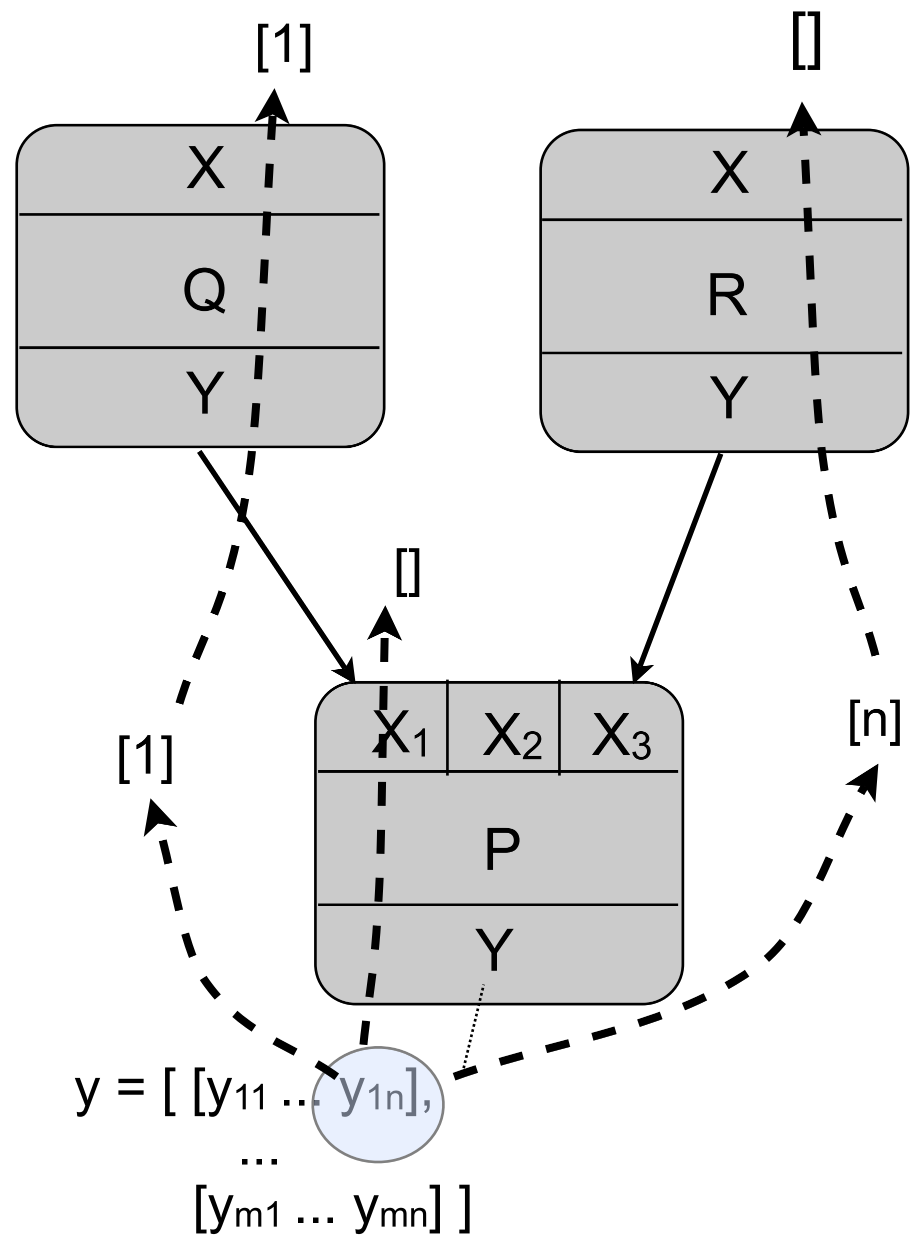
MANCHESTER 1824

The University of Manchester

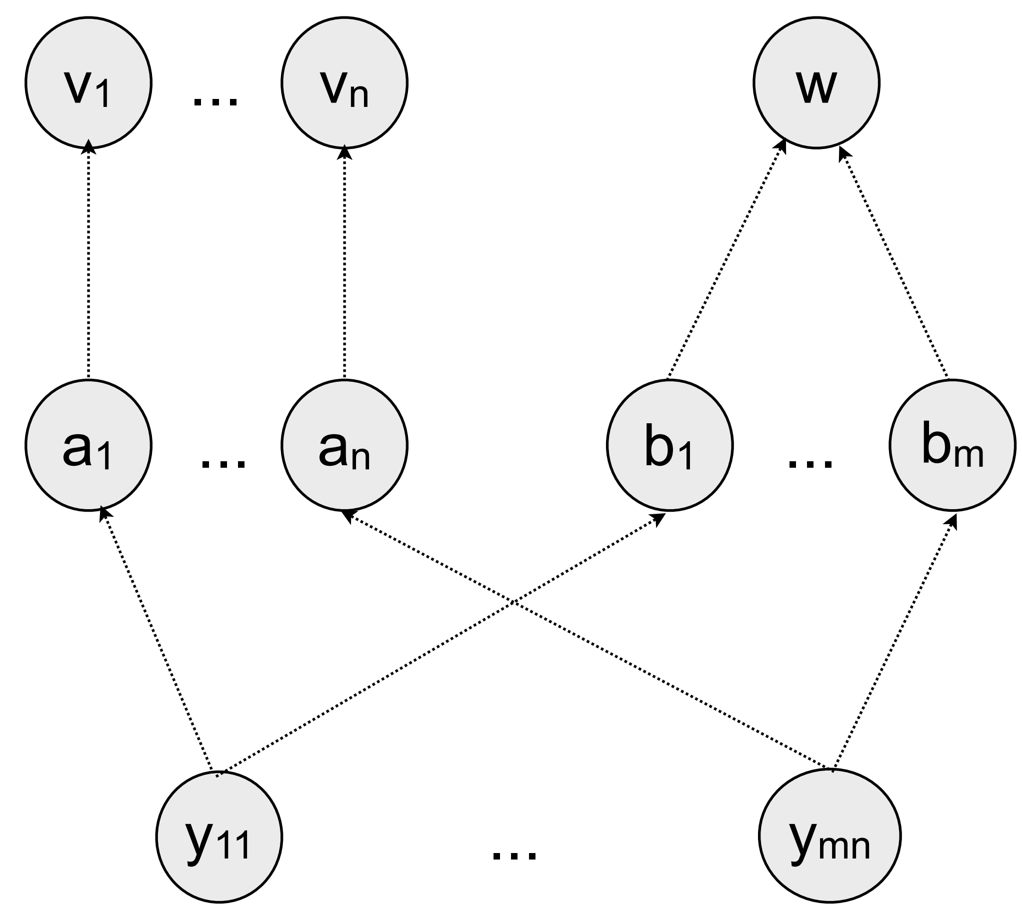
Workflow graph as index for query processing

1. traverse the workflow graph (small) rather than the provenance trace (large)
2. use static prediction of iterations to trace through collection elements
3. “parachute” into the actual trace only at the end

Workflow graph

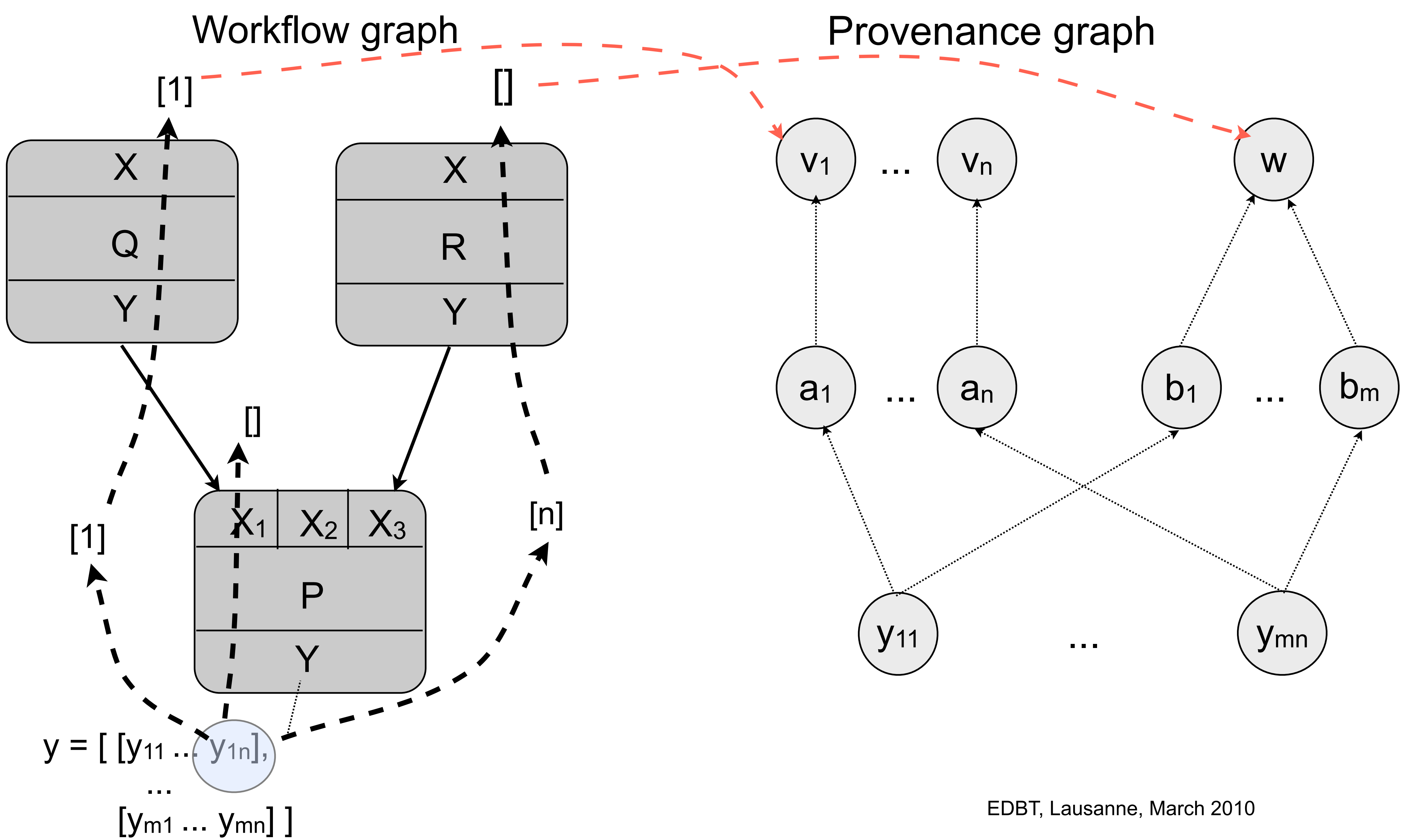


Provenance graph



Workflow graph as index for query processing

1. traverse the workflow graph (small) rather than the provenance trace (large)
2. use static prediction of iterations to trace through collection elements
3. “parachute” into the actual trace only at the end



From granularity to efficient query processing

Summary so far:

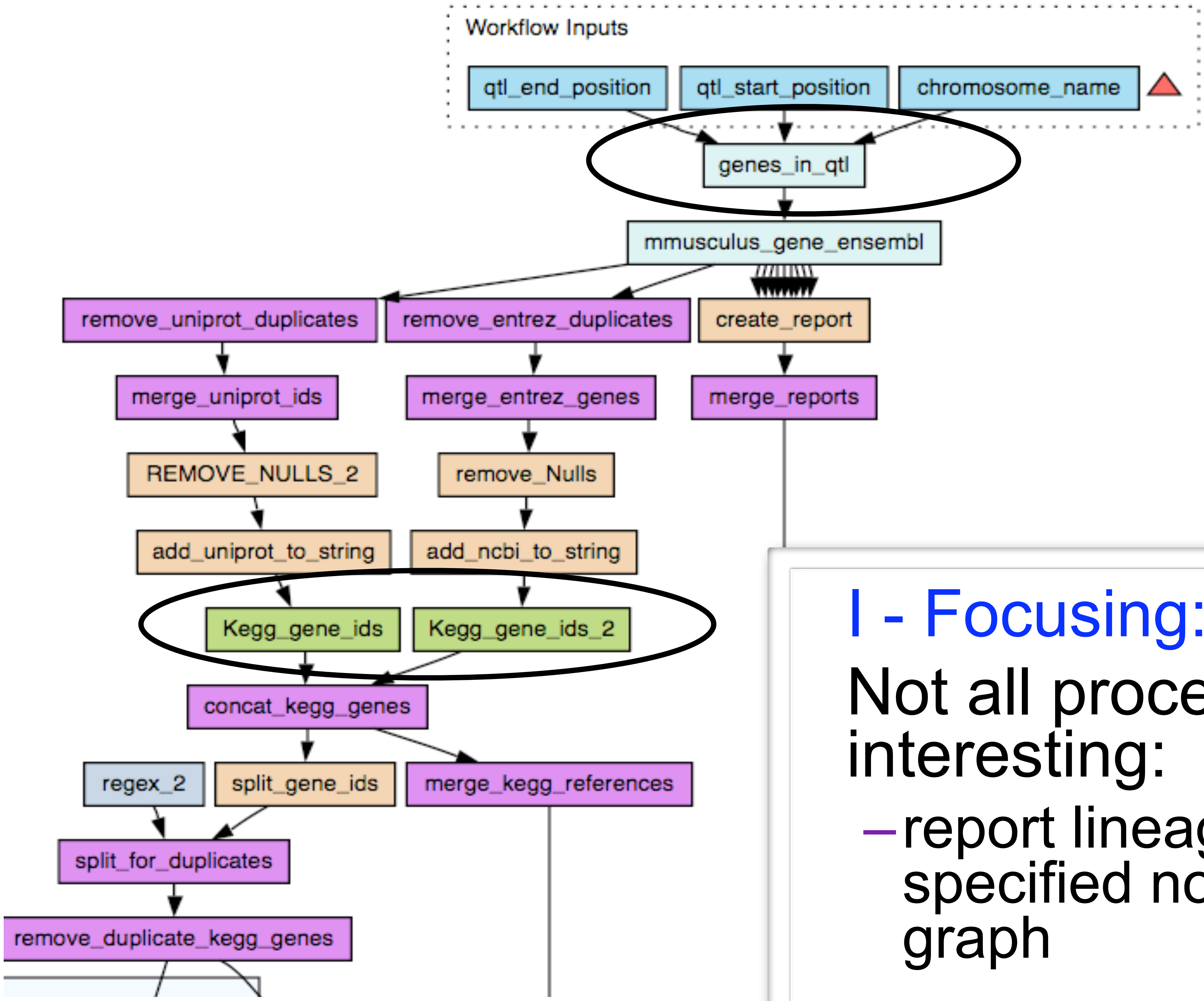
- whenever iterations are involved, we can trace the provenance of individual elements of a processor's output
- iterations are explained in terms of a functional model and based on list depth discrepancies
- The relationships between output and input indexes are derived using the workflow specification graph (statically)

How about expressivity and efficient processing of lineage queries?

MANCHESTER 1824

The University of Manchester

Lineage query model /1



I - Focusing:

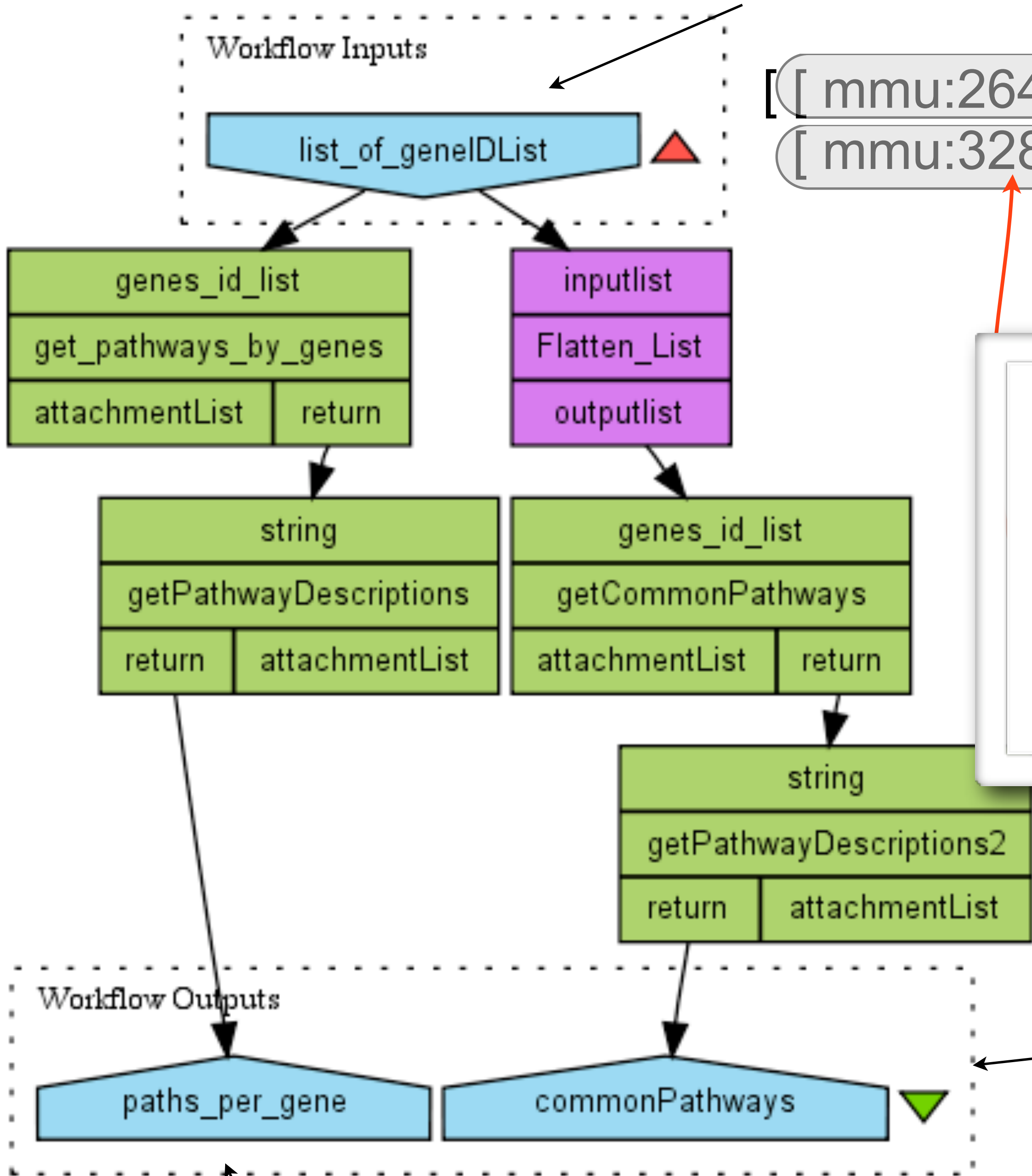
Not all processors are interesting:

– report lineage only at specified nodes in the graph

Lineage query model /2

List-structured KEGG gene ids:

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



II - Granularity:

Trace lineage for individual elements within collections

- when possible!

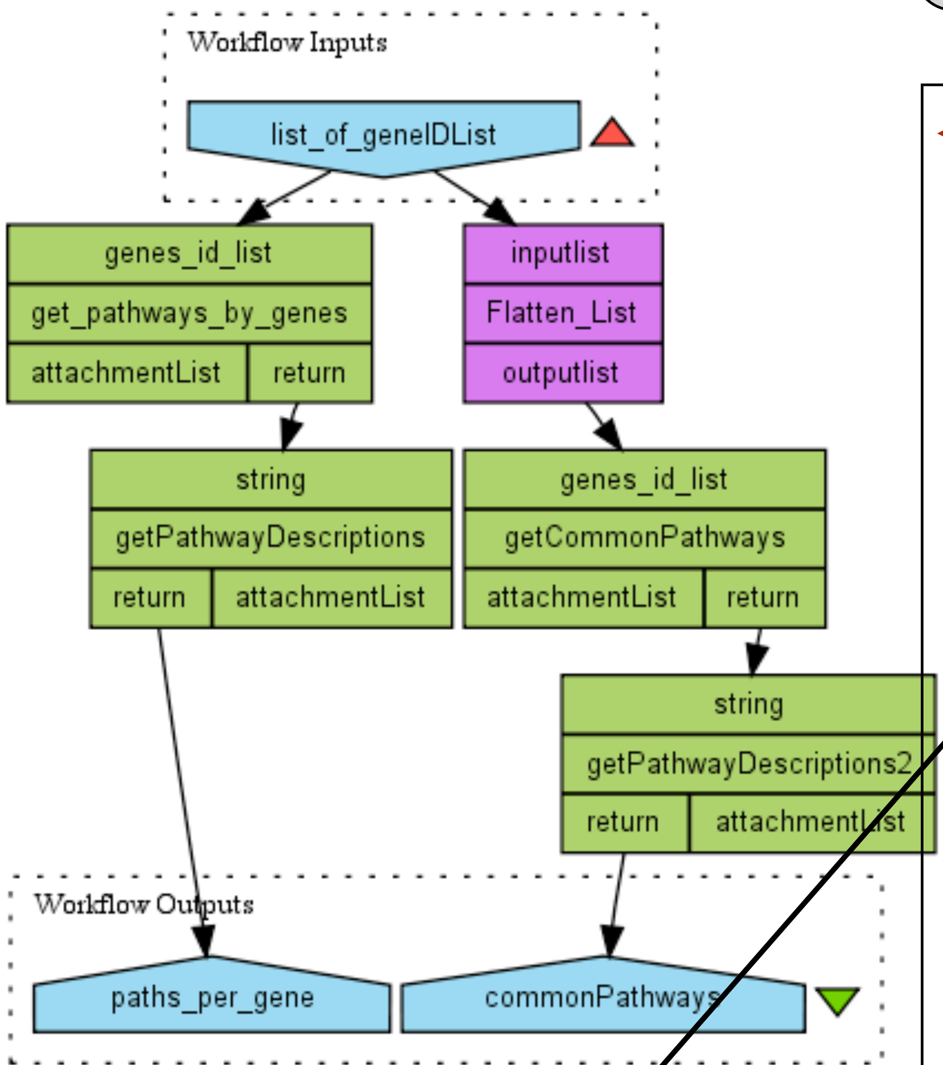
[path:mmu04010 MAPK signaling,
path:mmu04370 VEGF signaling]

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

Lineage query model and language

workflow scope defaults to latest run

optionally specifies one or more runs for the target workflow




```
<pquery>
  <scope workflow="keggPathways">
    <run id="ae1e2b6b-3bc5-4c93-a250-c4dd0210c3b3"/>
  </scope>
  <select>
    <outputPort name="paths_per_gene" index="[1,2]"/>
    <outputPort name="paths_per_gene" index="[3,4]"/>
    <outputPort name="commonPathways" index="[1]"/>
    <processor name="getPathwayDescriptions">
      <outputPort name="return"/>
    </processor>
  </select>
  <focus>
    <processor name="get_pathway_by_genes" />
  </focus>
</pquery>
```

port values for which lineage is sought: global outputs or processor-qualified

processors where lineage is to be reported - possibly workflow-qualified

Fine granularity + efficient processing

- 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
- Graceful degradation:
  - worst case is a completely unfocused query
  - no worse than other approaches
- Fine-grain answers provided at the same time

Outline

- Assumption:
  - *Black box* provenance of workflow data products

✓ Fine-grained provenance:
  - tracking provenance through collection elements
  - motivation, functional model of collection-oriented workflow processing

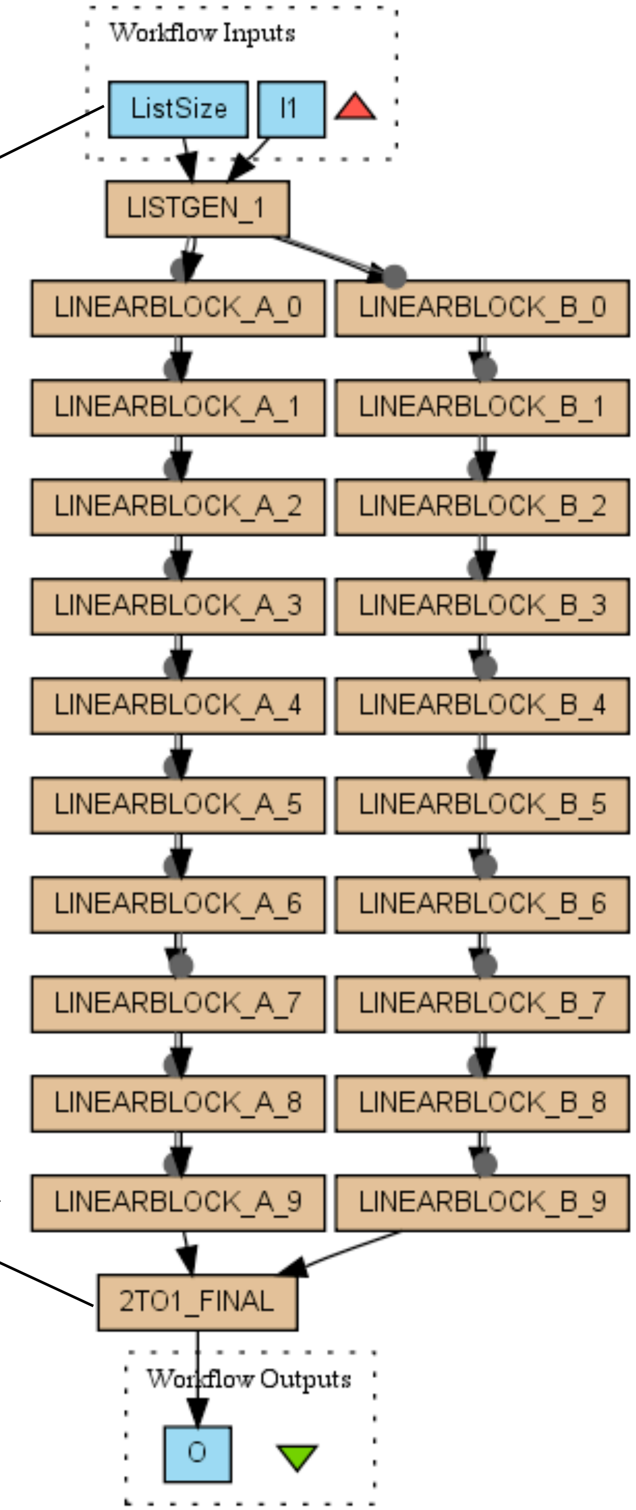
✓ Efficient query processing:
  - ✓ leveraging the functional model to achieve efficient processing for a simple query model

➡ Experimental evaluation

Experimental setup - I

- Performance evaluation performed on programmatically generated dataflows
  - the “T-towers”

parameters:
- size of the lists involved
- length of the paths
- includes one cross product

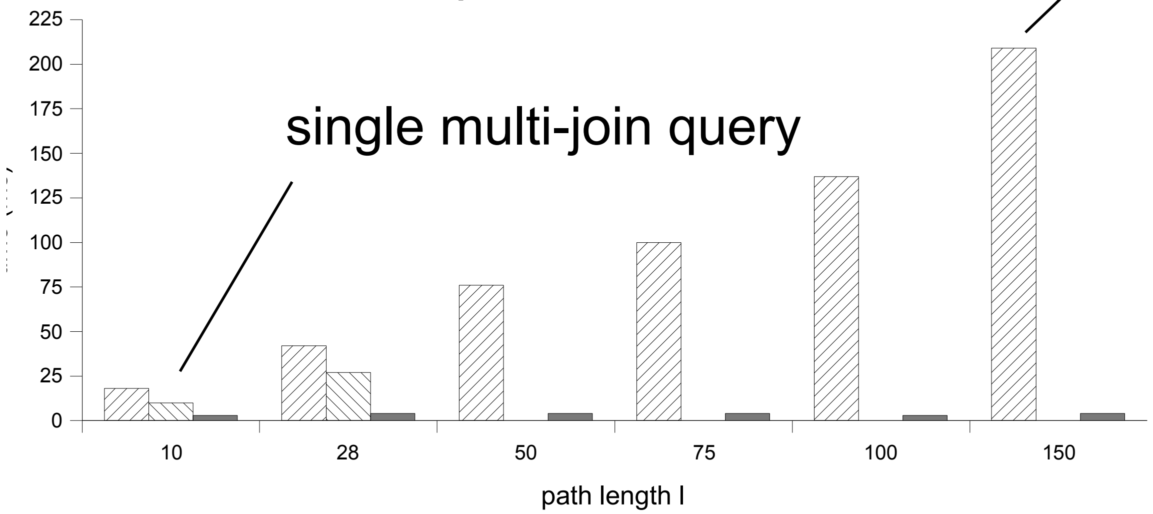


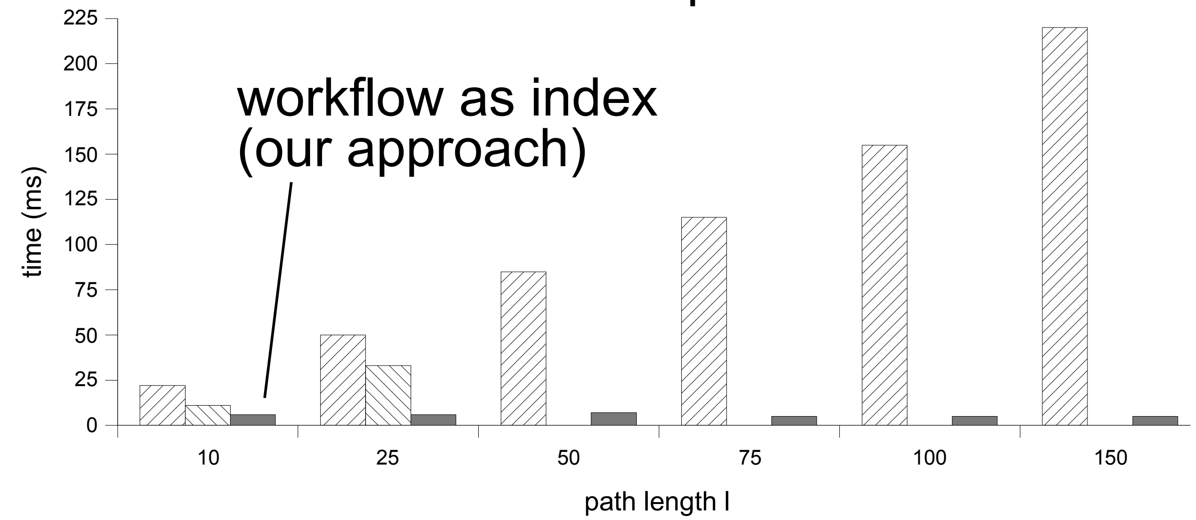
Experimental results - II

10 elements in input list

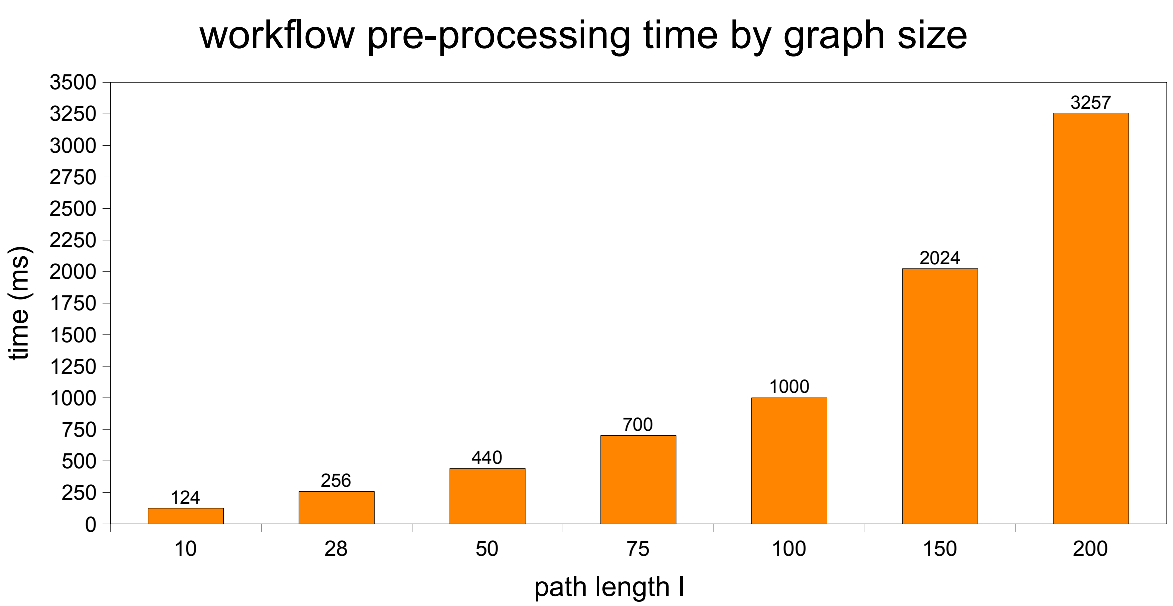
Naive traversal of provenance graph

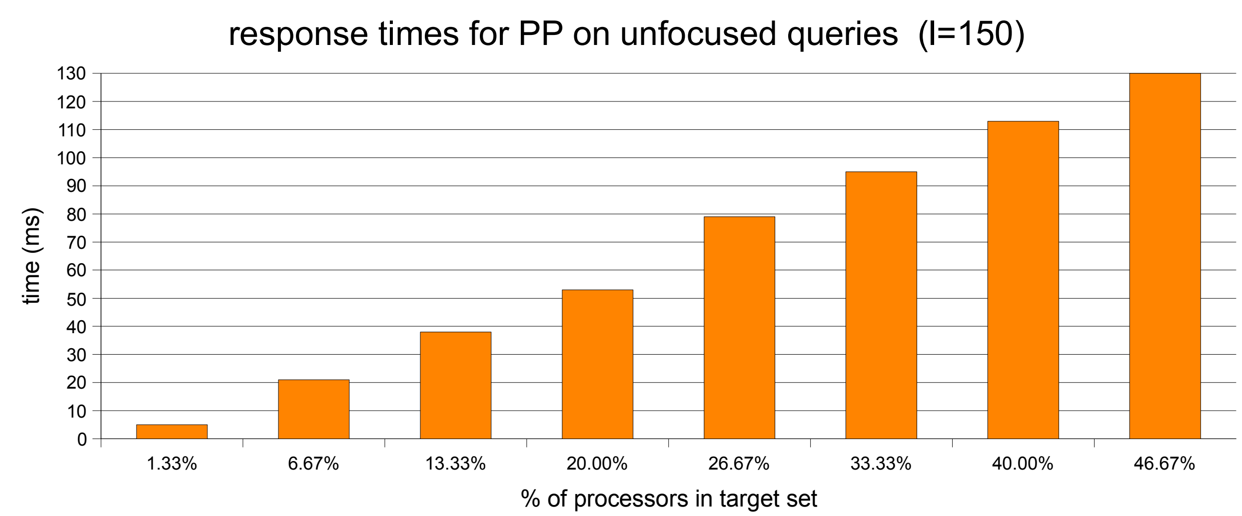
150 elements in input list





performance degradation on fully unfocused queries





MANCHESTER 1824

The University of Manchester

Summary

- A simple lineage query model for Taverna
  - grounded in the semantics of collection-oriented processing
  - combines fine-grain answers with efficient query processing

- Ongoing work:
  - space compression, indexing
  - QLP?
  - semantic provenance (initial paper submitted)

- Currently part of the Taverna 2.1 release