Linking Prospective and Retrospective Provenance in Scripts

Saumen Dey
University of California
scdey@ucdavis.edu

Khalid Belhajjame
Université Paris-Dauphine
Khalid.Belhajjame@dauphine.fr

David Koop
University of Massachusetts Dartmouth
dkoop@umassd.edu

Meghan Raul
University of California
mtraul@ucdavis.edu

Bertram Ludäscher
University of Illinois UC
ludaesch@illinois.edu

Abstract

Scripting languages like Python, R, and MATLAB have seen significant use across a variety of scientific domains. To assist scientists in the analysis of script executions, a number of mechanisms, e.g., noWorkflow, have been recently proposed to capture the provenance of script executions. The provenance information recorded can be used, e.g., to trace the lineage of a particular result by identifying the data inputs and the processing steps that were used to produce it. By and large, the provenance information captured for scripts is fine-grained in the sense that it captures data dependencies at the level of script statement, and do so for every variable within the script. While useful, the amount of recorded provenance information can be overwhelming for users and cumbersome to use. This suggests the need for abstraction mechanisms that focus attention on specific parts of provenance relevant for analyses. Toward this goal, we propose that fine-grained provenance information recorded as the result of script execution can be abstracted using user-specified, workflow-like views. Specifically, we show how the provenance traces recorded by noWorkflow can be mapped to the workflow specifications generated by YesWorkflow from scripts based on user annotations. We examine the issues in constructing a successful mapping, provide an initial implementation of our solution, and present competency queries illustrating how a workflow view generated from the script can be used to explore the provenance recorded during script execution.

1. Introduction

Despite the popularity of scientific workflows as a means to specify and enact *in silico* experiments, the majority of scientists still process and analyze their datasets using scripts. This may be partly due to the fact scripting languages such as Python and R allow easier customization as they expose the processing steps at the level of a statement, as opposed to workflows where the designer is typically unable to access or modify the implementation of the modules that constitute the workflow.

To assist scientists in the analysis of script executions, retrospective provenance information specifying, amongst other things, the lineage of script results and the transformations they underwent, can be useful in, e.g., assessing the validity of the hypothesis the scientist is investigating. For example, a scientist may wish to know the data inputs and processing steps that have been used to produce a peculiar result. To this end, several mechanisms have been recently proposed to capture the provenance traces of script execution [4, 7, 11, 18]. Consider, for example, noWorkflow, a tool that we focus on in this paper, which was developed by Murta *et al.* to capture provenance information for Python scripts [18]. noWorkflow automatically captures data manipulation at the level of statement, thereby providing a fine-grained and rich account of the data dependencies between the data artifacts used and generated by the script execution. While useful, the amount of information such fine-grained traces contain can be overwhelming for end users. This demonstrates the need for abstraction techniques that focus the attention of the user on the provenance information that is relevant for her analysis. We stress here that recording fine-grained, statement-based provenance can be crucial for understanding the lineage of scripts results. However, we argue that in order for such provenance information to be readily accessible and useful for end users, it needs to be abstracted and filtered to match the questions users want answered. Indeed, end users are often interested in the lineage of a small subset of the data artifacts manipulated and generated by the script. Moreover, by and large, they are not keen on examining the transformations such data artifacts underwent at the statement level.

With the above observation in mind, we present a solution where the fine-grained provenance information captured by noWorkflow is overlaid with a workflow that focuses the attention of the user on a small subset of data artifacts of interest, and hides the complexity of statement-based provenance by aggregating script statements within activities. Both data artifacts and activities are tagged with informative domain annotations.

Our work is related to the Zoom system proposed by Biton *et al.* which utilizes user views as abstractions for focusing the user attention on a subset of the provenance traces of a possibly large workflow by composing together subsets of the activities that constitute the workflow [3]. Our work is also related to the proposal of Alper *et al.* for summarizing complex workflows. Given a workflow specification, they define reduction rules that exploit user-defined tags annotating the activities of the workflows and the input and output ports thereof, to produce a concise workflow specification containing only the steps necessary for understanding the *in silico* experiment implemented by the initial workflow [1, 2].

We adopt an approach that is similar to the above proposals focusing on scripts instead of workflows. We do not build our solution from scratch. Instead, we reuse an existing solution, YesWorkflow, which provides a workflow-like view of scripts. Given a script, YesWorkflow generates a workflow based on user annotations that reveal the computational modules and dataflows otherwise implicit in the script, i.e. the prospective provenance. We show how the workflow structure produced from YesWorkflow can be utilized to abstract the provenance information captured by noWorkflow. Our contributions are as follows: (i) we examine how the prove-

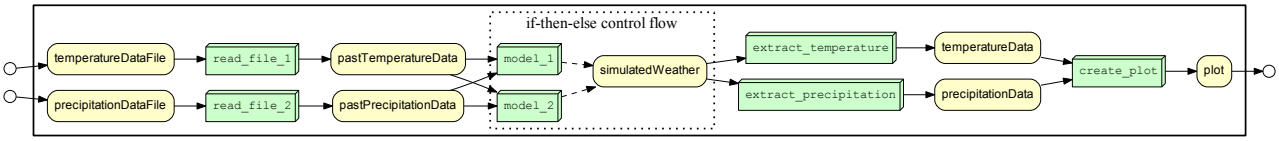

Figure 1. Weather prediction workflow

nance traces captured by noWorkflow can be mapped to the more abstract workflow view defined by YesWorkflow; (ii) we define a framework illustrating how this integration can be achieved; (iii) we present a prototypical implementation using a collection of Datalog rules that maps noWorkflow's provenance to YesWorkflow's provenance; and (iv) we show competency queries illustrating example lineage provenance queries that are can be answered.

Outline. Section 2 describes the related work and discusses the working details of YesWorkflow and noWorkflow. Section 3 introduces the weather simulation workflow. Section 4 analyzes gaps we have found and Section 5 presents our proposed framework with a prototypical implementation. Section 6 shows how to use this framework to query provenance data and Section 7 summarizes our contributions and discusses future work.

2. Background

The importance of provenance in computational science is well-recognized [10] and has led to provenance standards including the W3C PROV data model [17]. Both prospective provenance (i.e., the specification of the computation) and retrospective provenance (i.e., the exact steps followed during execution) contribute important information [22]. For workflows, the ProvONE model extends the PROV model to allow detail about the processes, ports, and data links [5].

While there has been much work in provenance relating to scientific workflows and at the system level, recently there has been some focus to capture the provenance of script executions. [4] relies on client libraries to capture traces of executions of scripts, whereas [20] make use of compilers that insert provenance capture within the script before and after function calls. Such approaches require major changes to the script to capture provenance. Moreover, they do not provide users with the means to understand the script itself (i.e., its prospective provenance).

noWorkflow adopts a non-intrusive approach to capturing retrospective provenance of scripts [18]. It is worth underlining here that noWorkflow also provides prospective view of the provenance. However, it uses the underlining script as is to derive such provenance. Because of this, the prospective (as well as the retrospective) provenance generated by noWorkflow depends on the level of abstraction in the script and can produce fine-grained provenance that becomes cumbersome.

YesWorkflow [15] uses an annotation language, which is similar to Javadoc* and Doxygen†, to generate workflow graphs that help users understand the building blocks (modules) that compose the script and their dataflow dependencies, a.k.a. prospective provenance. YesWorkflow is also related to the area of literate programming (e.g., Knitr [21] and IPython [19]). In literate programming a script is decomposed into snippets of macros, which are interspersed within documents that are written in natural language to explain the scripts and eventually analyze the results it generates upon execution. Dataflow dependencies in literate programming approaches are implicit, and it is up to the user to infer them by examining the details of the script snippets in the cells.

Previous work describes how causality can be inferred using both prospective and retrospective provenance and discuss the benefits of *user-defined information*, which is user annotation captured at different level of granularities [6, 10]. [12] extends the Open Provenance Model (OPM) to model both prospective and retrospective provenance. [22] argues that prospective and retrospective provenance together provide a complete understanding of the data, illustrates a virtual data approach to integrating prospective and retrospective provenance with semantic annotations, describes the powerful queries that can be performed on such an integrated base, and introduces an implementation that provides these benefits in a large-scale scientific computing environment. [16] extends W3C PROV to model both prospective and retrospective provenance. In our earlier work [9], we showed that prospective and retrospective provenance can be combined to provide unambigious lineage querying. In [8], we showed that prospective provenance can improve the precision of retrospective provenance by reducing the number of "false" dependencies and conversely, fine-grained execution provenance can be used to improve the precision of input-output dependencies of workflow actors.

Our work integrates noWorkflow and YesWorkflow provenances by mapping (i.e., translating) the retrospective provenance captured by noWorkflow into a model that is compatible with the prospective provenance generated by YesWorkflow.

2.1 noWorkflow

noWorkflow captures the retrospective provenance of an execution of a Python script by analyzing the abstract syntax tree of the script to identify the function calls and variables referenced in the script. noWorkflow produces an execution log containing function calls, the input values, and the returned values. Additionally, it captures information about the signatures of the functions, the variables within the script, and the execution environment details. In what follows, we will focus on the execution log captured by noWorkflow, which is modeled using the following predicates.

The `activation` predicate has arguments `run_id`, `id`, `name`, `start`, `finish`, and `caller_id`, and is used to capture the function calls. The `run_id` identifies the script run; the `id` uniquely identifies the function activation within the run; `name` is the name of a function; and `caller_id` is the identifier of the function activation that caused the activation of this function. `start` and `finish` are time-stamps denoting the beginning and end of the activation. The `access` predicate has arguments `run_id`, `id`, `name`, `mode`, `content_before`, `content_after`, `timestamp`, and `activation_id`, and is used to record read and write access to data files. The `id` identifies the access within the run; the `name` holds the file name being accessed; the `mode` is the mode of access, i.e., read or write; `content_before` (resp. `content_before`) a hash key used to locate the content of the file before (resp. after) the access took place; `timestamp`, representing the time when the access occurred; and `activation_id` identifying the activation within which the access took place. The `variable` predicate has arguments `run_id`, `v_id`, `name`, `line`, `value`, and `timestamp`,

* http://en.wikipedia.org/wiki/Javadoc

† http://en.wikipedia.org/wiki/Doxygen

and is used to record the values that a variable identified by `v_id` and named `name` within the script has been accessed within the run `run_id`. It also records the time stamp at which the access occurred. The `usage` predicate has arguments `run_id`, `id`, `vid`, `name`, and `line`, and is identified by an integer `id`. Given a run identified by `run_id`, it records the lines in the script where a variable identified by `v_id` and named `name`, was used. The `dependency` predicate has arguments `run_id`, `id`, `dependent`, and `supplier`, and is used to define data flow dependencies between variables, or more specifically with a run. `dependent` refers to the variable, the value of which depends on the variable identified by `supplier`.

2.2 YesWorkflow

YesWorkflow allow script developers (or annotators) to expose the prospective provenance of a script, i.e., the steps that compose the script and their data flow dependencies by providing mark ups within the script. YesWorkflow annotations are of the form `@tag, value`, where `@tag` is a keyword that is recognized by YesWorkflow and *value* represents value instance of the concept represented by the `@tag`. The script annotations are used by YesWorkflow to generate a workflow specification, which can be mapped to the following ProvONE model constructs.

A `process` in YesWorkflow corresponds to a chunk in the script that represents a meaningful computational step. It is defined and delimited within the script using the `@begin` and `@end` tags. The `InputPort` (resp. `OutputPort`) represents an input (resp. output) parameter of a `process`. They are designated using the tags `@in` and `@out` respectively. `Datalink` specifies the dependencies between output ports and input ports within the script. In YesWorkflow, `Datalinks` are not explicitly specified. Instead, they are implicitly inferred by matching the names of the output ports and inputs ports specified by the `@in` and `@out` tags in the script.

3. Running Example

To illustrate how we can leverage integrated YesWorkflow and noWorkflow provenances, we use a Python script that approximates a weather forecasting calculation (see Fig. 2). The script reads two files, `data1.dat` and `data2.dat`, which contain information about the past temperature readings and precipitation amounts, respectively (lines 29–39). The temperature data is used to select a model—`model1` or `model2`—which takes both input arrays and creates a matrix with future predictions for temperature and precipitation (lines 41–54). The script then extracts the temperature and precipitation columns (line 56–66), and creates a scatterplot of these two variables that is saved to the `output.png` file (lines 68–73).

Notice that the script is annotated with YesWorkflow comments and segmented into user-defined functions to make it easier to parse. Lines 29–31 show a YesWorkflow annotation that specifies the name of the process and the inputs and outputs, each tied to a more descriptive alias via the `as` keyword. These lines denote the beginning of a computational process, and Line 33 denotes the end of that block. In that block, the actual input filename is `data1.dat`—a rather generic name, but the YesWorkflow alias identifies it as past temperature data. Using these annotations, YesWorkflow can produce a workflow that encodes the prospective provenance of the script as depicted in Figure 1.

By executing the example script in noWorkflow, we can automatically collect a variety of provenance data ranging from prospective provenance (definition provenance) gathered from the abstract syntax tree to retrospective provenance that is fine-grained and captured by tracing execution steps. This provenance can be

```
1  import csv
2  import sys
3  import matplotlib.pyplot as plt
4  from simulate import model1, model2
5  import time
6
7  def read_file(data_fname):
8      reader = csv.reader(open(data_fname, 'rU'),
           delimiter=':')
9      arr = []
10     for row in reader:
11         arr.extend([float(x) for x in row])
12     return arr
13 def extract_column(data, idx):
14     c = []
15     for row in data:
16         c.append(row[idx])
17     return c
18 def create_plot(x, y, xlabel, ylabel, marker,
       out_fname):
19     plt.scatter(x, y, marker=marker)
20     plt.xlabel(xlabel)
21     plt.ylabel(ylabel)
22     plt.savefig(out_fname)
23
24 ## @begin main
25 # @in data1.dat @as temperatureDataFile @URI
       file:temp.dat
26 # @in data2.dat @as precipitationDataFile @URI
       file:precip.dat
27 # @out output.png @as plot
28
29 ## @begin read_file_1
30 # @in data1.dat @as temperatureDataFile @URI
       file:temp.dat
31 # @out a @as pastTemperatureData
32 a = read_file("data1.dat")
33 ## @end read_file_1
34
35 ## @begin read_file_2
36 # @in data2.dat @as precipitationDataFile @URI
       file:precip.dat
37 # @out b @as pastPrecipitationData
38 b = read_file("data2.dat")
39 ## @end read_file_2
40
41 if sum(a)/len(a) < 0:
42     ## @begin model_1
43     # @in a @as pastTemperatureData
44     # @in b @as pastPrecipitationData
45     # @out data @as simulatedWeather
46     data = model1(a, b)
47     ## @end model_1
48 else:
49     ## @begin model_2
50     # @in a @as pastTemperatureData
51     # @in b @as pastPrecipitationData
52     # @out data @as simulatedWeather
53     data = model2(a, b)
54     ## @end model_2
55
56 ## @begin extract_temperature
57 # @in data @as simulatedWeather
58 # @out t @as temperatureData
59 t = extract_column(data, 0)
60 ## @end extract_temperature
61
62 ## @begin extract_precipitation
63 # @in data @as simulatedWeather
64 # @out p @as precipitationData
65 p = extract_column(data, 1)
66 ## @end extract_precipitation
67
68 ## @begin create_plot
69 # @in t @as temperatureData
70 # @in p @as precipitationData
71 # @out output.png @as plot @URI file:ouput.png
72 create_plot(t, p, "Temperature", "Precipitation", 'o',
       "output.png")
73 ## @end create_precipitation
74
75 ## @end main
```

Figure 2. Simulation.py

exported as Prolog facts that identify variables and data dependencies. In the following facts:

```
variable(6, 93, 'row', 14, "['0.0']",
         1430231173.397779).
variable(6, 94, 'return', 15, 'None',
         1430231173.397797).
dependency(6, 34, 94, 93).
```

the first fact states that the variable identified by the integer `93` and named `row` took the value `['0.0']` in line `14` in the script. The second fact similarly identifies a variable, its value, and the line number of the code while the third fact states a data dependency between the aforementioned variables.

4. Analysis and Observations

As discussed in Section 2, the annotations from YesWorkflow and provenance from noWorkflow can be connected to permit a wider set of provenance queries. However, constructing the necessary links involves some modifications to the provenance generated by these tools. Using the weather simulation example (Fig. 2), we highlight some of the obstacles involved in defining these connections.

The example workflow was modified from an example included with noWorkflow source code. Changes include moving temperature and precipitation data to separate files and adding an if-then-else block to switch between two models based on the average temperature in the input. In addition, repeated code has been refactored into functions. Finally, we have added YesWorkflow annotations. The entire script is shown in Listing 2. Using noWorkflow, the workflow was then executed multiple times with different input datasets. The workflow generated from YesWorkflow annotations is shown in Fig. 1 and the provenance graph from one of the noWorkflow executions in Fig. 10.

We note that noWorkflow generates provenance using the script's variable names, and YesWorkflow binds (some of) the variables in the script to aliases that are used to name the input and output ports of the workflow. This allows us to connect the retrospective provenance generated by noWorkflow with the workflow extracted by YesWorkflow. There are, however, issues that need to be addressed, which we discuss in what follow.

4.1 Issues With Prospective Provenance

Control flow support. Consider lines 41–54 of Fig. 2. The if-then-else control flow construct is used to switch between two models. Intuitively, we might define each block separately with YesWorkflow annotations, thus using the `@out data @as simulatedWeather` annotation *twice*, once for each block. Currently, YesWorkflow does not support this. A solution to get around this issue and allow YesWorkflow to parse the script comments consists in having the annotation outside the if-then-else block as illustrated in Fig. 3.

It is worth underlining here that YesWorkflow is ongoing development, and that it will be possible in a future release to have the invocations of $model1$ and $model2$ in two different blocks with two different $@out$ annotations referring to the same succeeding input port. Fig. 1 illustrates the workflow that will be extracted when this feature becomes supported by YesWorkflow.

Identifier reuse. In a script, an identifier may be used multiple times for different purposes. For example, an identifier may be used in the scope of a function as well as in the main script. Even if the systems are able to resolve the scope, users may have trouble understanding provenance links. noWorkflow tracks line numbers to help. YesWorkflow does not. However, its does record the line numbers of the start and end of each block when requested. Therefore, when noWorkflow records that a given variable was updated on a particular line, it is possible to identify the block, and thus the variable, in the workflow that was subject to update.

```
1 # @begin model
2 # @in a @as pastTemperatureData
3 # @in b @as pastPrecipitationData
4 # @out data @as simulatedWeather
5 if (sum(a)/len(a)) < 0:
6         data = model1(a,b)
7 else:
8         data = model2(a,b)
9 # @end model
```

Figure 3. Annotating an if-then-else control block using YesWorkflow

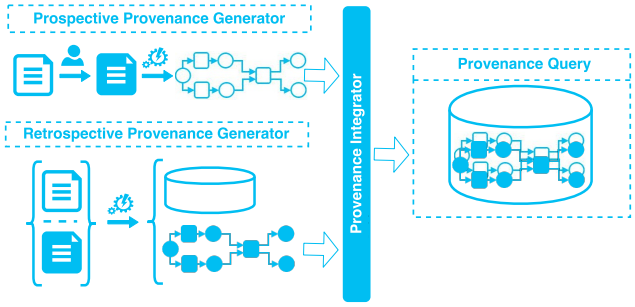

Figure 4. Logical architecture.

Linking identifiers. YesWorkflow generates provenance using identifiers from the user-defined annotations, but noWorkflow captures provenance using the script's variable names. While YesWorkflow does have methods to specify the link between the variable name and the user-defined identifier, it does not check that the annotation references an actual variable.

4.2 Issues With Retrospective Provenance

Data dependencies. There can be gaps in the noWorkflow dependency graph (when exported to prolog). Specifically, function returns do not always link back to correct variable. Furthermore, an object that is modified (e.g via a `list.append` call) is not always captured in the dependency graph.

Objects in objects. When an object has a field that is itself an object, the dependencies in noWorkflow are often handled at the highest level meaning that a change to a sub-object counts as a change to the parent object.

No script name. Same variable names may be used in different scripts. The script names along with the line numbers would be required to unambiguously identify a variable. Currently, noWorkflow does not capture the script name.

5. Framework and System

We introduce a new framework and provide a prototypical implementation. The logical architecture of our proposed framework is shown in Fig. 4. This framework has four primary components: (i) Prospective Provenance Generator, (ii) Retrospective Provenance Generator, (iii) Provenance Integrator, and (iv) Provenance Store. We now describe these components in detail.

```
For each invocation of a function
    Find return identifier (say X)
    Find variable V, whose value is returned
    Find line L, where V was modified last
    Find variable Z, which was assigned to V
    Add a new identifier Y with V and L
    Add dependency X to Y
    Add dependency Y to Z
```

Figure 5. Sketch of dependency repair for missing dependencies of function returns.

The **Prospective Provenance Generator** generates the prospective provenance using the user-specified names and the mapping between these identifiers and the variables used in the script. In our current implementation we used YesWorkflow for this component. YesWorkflow provides the prospective provenance using the user-specified annotations. Because YesWorkflow is language-agnostic and can be utilized for various scripting languages, e.g., Python, R, and MATLAB, it does not currently provide such a mapping.

The **Retrospective Provenance Generator** executes a script and captures its retrospective provenance. As this component needs to execute the workflow, it is difficult to have a technology-agnostic tool. We are using noWorkflow for Python scripts which captures the retrospective provenance using the variable names used in the script. Using this framework, tools for retrospective provenance can be integrated as they become available.

The **Provenance Integrator** integrates prospective and retrospective provenance and stores them in the provenance store. In our current prototypical implementation, this component (i) parses the script and generates the map between the annotations and the variable names using the relation $\texttt{map}(\texttt{X},\texttt{Y})$, where X is the variable name and Y is the user defined annotation, (ii) converts the prospective provenance into $\texttt{wDep}(\texttt{X}_\texttt{w},\texttt{Y}_\texttt{w})$, $\texttt{wData}(\texttt{D}_\texttt{w},\texttt{V}_\texttt{w})$, and $\texttt{wProcess}(\texttt{P}_\texttt{w},\texttt{N}_\texttt{w})$, where X_w depends on Y_w, D_w is data artifact identifier & V_w is its value, P_w is process identifier & N_w is process name, and X_w (similarly Y_w) is either a data artifact or a process, (iii) repairs the dependency gaps in the retrospective provenance (discussed in the next section) and converts retrospective provenance into $\texttt{iDep}(\texttt{X}_\texttt{i},\texttt{Y}_\texttt{i})$, $\texttt{iData}(\texttt{D}_\texttt{i},\texttt{V}_\texttt{i})$, and $\texttt{iProcess}(\texttt{P}_\texttt{i},\texttt{N}_\texttt{i})$, where the attributes are defined as they are in the prospective provenance, and (iv) abstracts retrospective provenance based on the prospective provenance and stores the dependencies into $\texttt{iDepAbs}(\texttt{X}_\texttt{i},\texttt{Y}_\texttt{i})$.

The **Provenance Store** stores D-PROV compliant prospective provenance, retrospective provenance, and the mapping between them. It also allows users to interactively query provenance data. An important feature is that users can query retrospective provenance based on the prospective provenance (i.e., using the annotations they provided). In our current prototypical implementation, we store the provenance data as Datalog facts and allow interactive queries using DLV Datalog.

5.1 Dependency Repair

The **Provenance Integrator** repairs the dependency gaps discussed in Section 4.2 in the retrospective provenance generated by noWorkflow and computes the abstract retrospective provenance. A sketch of the dependency repair algorithm for missing dependencies of function returns is shown in Fig. 5. Fig. 10 shows the modified retrospective provenance graph of the running example with dependency gaps fixed.

```
(1) nodeInt(X) :- map(X,_).
(2) iDepTC(X,Y) :- iDep(X,Y).
(3) iDepTC(X,Y) :- iDepTC(X,Z), iDep(Z,Y).
(4) pDep(X,Y)  :- iDepTC(X,Y), nodeInt(X),
                  nodeInt(Y).
(5) depExists(X,Y) :- pDep(X,Y), pDep(X,Z),
                      pDep(Z,Y), X != Z, Y != Z.
(6) iDepAbs(X,Y) :-  pDep(X,Y),
                     not depExists(X,Y).
```

Figure 6. Algorithm to abstract noWorkflow retrospective provenance based on YesWorkflow prospective provenance.

```
tFileAnnot(X) :- wDep(X,_), wData(X,V),
                 V="temperatureDataFile".
tFileName(V) :- tFileAnnot(X), map(X,Y),
                iDepAbs(Y,_), iData(Y,V).
```

Figure 8. Query to find actual temperature file used in an execution.

```
wID(X) :- wDep(X,_), wData(X,V),
                V="plot".
iID(V) :- wID(X), map(X,Y), iData(Y,V).
lineage(X,Y) :- iDepAbs(X,Y), iID(V).
lineage(X,Y) :- iDepAbs(X,Y), lineage(Y,_).
```

Figure 9. Query to find the actual temperature file used in an execution.

5.2 Abstract Retrospective Provenance

Fig. 6 presents the algorithm we developed to abstract noWorkflow retrospective provenance data based on the YesWorkflow prospective provenance data. In line 1, it selects all the variable names annotated in YesWorkflow. Lines 2 and 3 compute the transitive closure of all the dependencies in repaired retrospective provenance. Line 4 computes all the potential dependencies. Line 5 captures which redundant dependencies which should be removed and line 6 computes the final list of abstracted dependencies. Given the retrospective provenance shown in Fig. 10, this algorithm computes the abstracted retrospective provenance shown in Fig. 7.

Complexities. Consider prospective provenance with N_w nodes representing variables and E_w edges representing variable dependencies. Similarly, assume the retrospective provenance has N_i nodes representing variable invocations with values and E_i edges representing variable value dependencies. In general, $\mid N_w \mid \ll \mid N_i \mid$ and $\mid E_w \mid \ll \mid E_i \mid$. Thus, the space required to store both the prospective and retrospective provenance is $\mathrm{O}(N_i + E_i)$. In a worst case scenario, the **Dependency Repair** module would review all E_i edges and add another E_i so the space complexity remains $\mathrm{O}(N_i + E_i)$ and time complexity is $\mathrm{O}(E_i)$. As the **Abstract Retrospective Provenance** module "overlays" the E_w edges over E_i edges to find the relevant subgraph of E_i, the space requirement is same as prospective provenance and thus space complexity still remains to be $\mathrm{O}(N_i + E_i)$. Because we compute the transitive closure of the E_i edges, the time complexity would be $\mathrm{O}(E_i^2)$.

6. Query

To answer the query discussed in Section 1, we need to find out which temperature file was used in an execution and then compare the files among various runs. The query in Fig. 8 would find the temperature file name used in an execution. This illustrates how retrospective and prospective provenance are integrated to enable

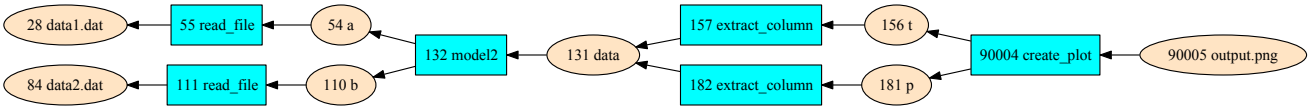

Figure 7. noWorkflow retrospective provenance graph after abstracting based on the yesWorkflow.

users to ask queries with the annotations used in YesWorkflow and compute answers using the provenance data captured by noWorkflow. In the sample workflow shown in Fig. 2, `tFileName(V)` from Fig. 8 will have value `data1.dat`.

Additionally, suppose a user wishes to find the retrospective lineage of the final product—the plot. The query shown in Fig. 9 could be used answer such query. This also illustrates the need for integrating both the prospective and retrospective provenances. `lineage(X, Y)` would contain the retrospective lineage for the final data product "plot".

7. Conclusion

We have showed in this paper how the retrospective provenance of script executions generated using noWorkflow can be mapped onto a high-level of abstraction by using the workflow specification generated by YesWorkflow. We have also showed how the mapping exercise enables answering provenance queries over script executions. The work reported is still ongoing development. We intend to assess the mapping specified in the context of other applications with larger scripts. We also note that the work reported in this paper is related to another ongoing (and complementary) work [14] that aims to capture the provenance of data artifacts used and generated within scripts and stored on the file system using YesWorkflow.

As part of future work, we want to (a) extend this framework to support other tools generating retrospective and prospective provenance for various scripting languages, (b) study the query performances using larger scientific workflows with bigger data sets, and (c) validate the links that this framework establish between retrospective and prospective provenance.

References

[1] P. Alper, K. Belhajjame, C. A. Goble, and P. Karagoz. Small is beautiful: Summarizing scientific workflows using semantic annotations. In *IEEE International Congress on Big Data, BigData Congress 2013, June 27 2013-July 2, 2013*, pages 318–325. IEEE, 2013.

[2] P. Alper, K. Belhajjame, C. A. Goble, and P. Karagoz. Labelflow: Exploiting workflow provenance to surface scientific data provenance. In Ludäscher and Plale [13], pages 84–96.

[3] O. Biton, S. C. Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In G. Alonso, J. A. Blakeley, and A. L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 1072–1081. IEEE, 2008.

[4] C. Bochner, R. Gude, and A. Schreiber. A python library for provenance recording and querying. In J. Freire, D. Koop, and L. Moreau, editors, *Provenance and Annotation of Data and Processes, Second International Provenance and Annotation Workshop, IPAW 2008, Salt Lake City, UT, USA, June 17-18, 2008. Revised Selected Papers*, volume 5272 of *Lecture Notes in Computer Science*, pages 229–240. Springer, 2008.

[5] V. Cuevas-Vicenttín, P. Kianmajd, B. Ludäscher, P. Missier, F. Chirigati, Y. Wei, D. Koop, and S. Dey. The pbase scientific workflow provenance repository. *International Journal of Digital Curation*, 9(2):28–38, 2014.

[6] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1345–1350. ACM, 2008.

[7] A. Davison. Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science and Engineering*, 14(4):48–56, 2012.

[8] S. Dey, K. Belhajjame, D. Koop, T. Song, P. Missier, and B. Ludäscher. Up & down: Improving provenance precision by combining workflow- and trace-level information.

[9] S. C. Dey, S. Köhler, S. Bowers, and B. Ludäscher. Datalog as a lingua franca for provenance querying and reasoning. In *TaPP*, 2012.

[10] J. Freire, D. Koop, E. Santos, and C. T. Silva. Provenance for computational tasks: A survey. *Computing in Science & Engineering*, 10(3):11–21, 2008.

[11] M. R. Huq, P. M. G. Apers, and A. Wombacher. Provenancecurious: a tool to infer data provenance from scripts. In G. Guerrini and N. W. Paton, editors, *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 765–768. ACM, 2013.

[12] C. Lim, S. Lu, A. Chebotko, and F. Fotouhi. Prospective and retrospective provenance collection in scientific workflow environments. In *Services Computing (SCC), 2010 IEEE International Conference on*, pages 449–456. IEEE, 2010.

[13] B. Ludäscher and B. Plale, editors. *Provenance and Annotation of Data and Processes - 5th International Provenance and Annotation Workshop, IPAW 2014, Cologne, Germany, June 9-13, 2014. Revised Selected Papers*, volume 8628 of *Lecture Notes in Computer Science*. Springer, 2015.

[14] T. M. McPhillips et al. Retrospective provenancewithout a run-time provenance recorder. In *Tapp*, 2015. Under review.

[15] T. M. McPhillips, T. Song, T. Kolisnik, S. Aulenbach, K. Belhajjame, K. Bocinsky, Y. Cao, F. Chirigati, S. C. Dey, J. Freire, D. N. Huntzinger, C. Jones, D. Koop, P. Missier, M. Schildhauer, C. R. Schwalm, Y. Wei, J. Cheney, M. Bieda, and B. Ludäscher. Yesworkflow: A user-oriented, language-independent tool for recovering workflow information from scripts. *CoRR*, abs/1502.02403, 2015.

[16] P. Missier, S. C. Dey, K. Belhajjame, V. Cuevas-Vicenttín, and B. Ludäscher. D-prov: extending the prov provenance model with workflow structure. In *TaPP*, 2013.

[17] L. Moreau and P. Missier. Prov-dm: The prov data model. 2013.

[18] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noworkflow: Capturing and analyzing provenance of scripts. In Ludäscher and Plale [13], pages 71–83.

[19] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[20] D. Tariq, M. Ali, and A. Gehani. Towards automated collection of application-level data provenance. In U. A. Acar and T. J. Green, editors, *4th Workshop on the Theory and Practice of Provenance, TaPP'12, Boston, MA, USA, June 14-15, 2012*. USENIX Association, 2012.

[21] Y. Xie. *Dynamic Documents with R and knitr*. CRC Press, 2013.

[22] Y. Zhao, M. Wilde, and I. Foster. Applying the virtual data provenance model. In *Provenance and Annotation of Data*, pages 148–161. Springer, 2006.

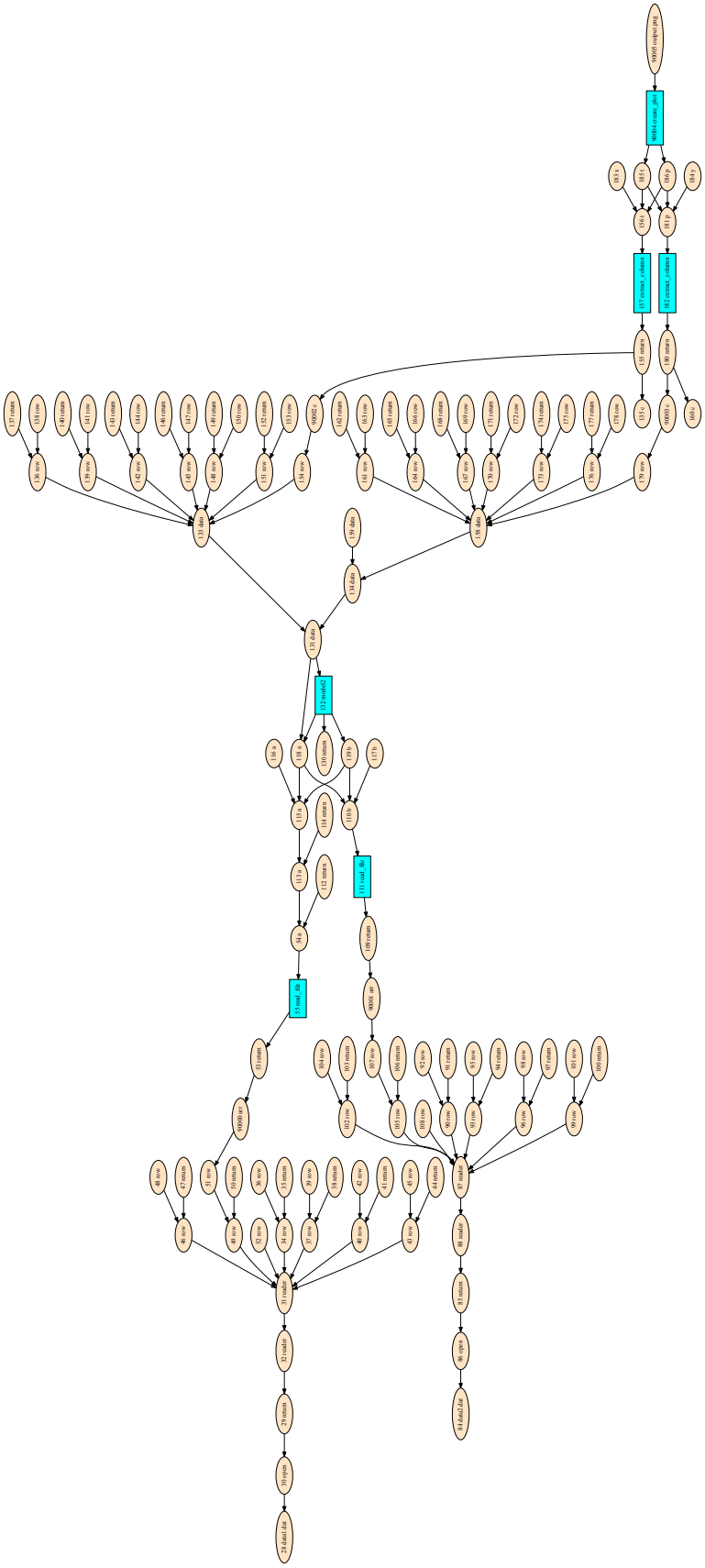

Figure 10. noWorkflow retrospective provenance graph after applying the dependency repair. This provenance graph has no dependency gaps.