From Scripts Towards Provenance Inference

Mohammad Rezwanul Huq, Peter M.G. Apers, Andreas Wombacher
Department of Computer Science, University of Twente
7522NB, Enschede, The Netherlands
Email: {m.r.huq and p.m.g.apers and a.wombacher}@utwente.nl

Yoshihide Wada, Ludovicus P. H. van Beek
Department of Physical Geography, Utrecht University
3584CS, Utrecht, The Netherlands
Email: {y.wada and r.vanbeek}@uu.nl

***Abstract*—Scientists require provenance information either to validate their model or to investigate the origin of an unexpected value. However, they do not maintain any provenance information and even designing the processing workflow is rare in practice. Therefore, in this paper, we propose a solution that can build the workflow provenance graph by interpreting the scripts used for actual processing. Further, scientists can request fine-grained provenance information facilitating the inferred workflow provenance. We also provide a guideline to customize the workflow provenance graph based on user preferences. Our evaluation shows that the proposed approach is relevant and suitable for scientists to manage provenance.**

***Keywords*-Data provenance, Inference, Workflow, Hydrology.**

I. Introduction

Scientists from different domains facilitate data intensive e-Science applications to study and better understand complex systems like physical, geological, environmental, biological etc. [1]. In most e-Science applications, scientists might often go to the field to collect in-situ data. They might also get sensor readings. Scientists use this data fitting into their model describing processes in the physical world. During the execution of the model, they might get occasionally imprecise or unexpected values due to the anomalies either in their data or in the model. To investigate the origin of the unexpected value, scientists need to debug through their scripts used for actual processing as well as to trace back values of the input data sources. Maintaining data provenance could help them in such a situation.

Data provenance refers to the derivation history of data starting from its input sources [2]. Provenance can be defined at different levels of granularity [3]. Fine-grained data provenance is defined at the value-level documenting the relationship among the input values, the output value and associated processes. Coarse-grained or workflow provenance is defined at the more higher level of granularity. Workflow provenance only captures association among different processes within the model.

Existing literatures discussing maintenance of fine-grained provenance, explicitly document the relationship among input values, associated processes and output values [4], [5]. Since e-Science applications involve massive amount of data, both sampled and streaming, it takes a considerable amount of storage to store fine-grained provenance data. Sometimes, the size of provenance data might become a multiple of the actual data. Since provenance data is 'just' metadata and less often used by the end users, this approach seems to be infeasible and too expensive [6].

Recently, a provenance inference mechanism has been proposed which can generate provenance information without explicitly documenting them based on a given workflow provenance [7]. In [8], authors extended the approach to infer provenance data for a complete processing workflow assuming that the workflow had been specified before executing the system.

However, in practice, scientists from other domains rarely design any processing workflow before executing their model. Usually, they write scripts and execute these scripts to process the collected data and to generate the output value. In such a situation, neither of the aforementioned techniques can be applied instantly because of the unavailability of workflow provenance information. It is possible to make a workflow provenance by analyzing the scripts used for processing. However, it demands in-depth understanding of the processes and the underlying domain in cases of complex models. Furthermore, creating workflow provenance manually requires a lot of time to handle each script in the model separately.

We propose to infer the workflow provenance information based on a given script which is used for actual processing. Later, this workflow provenance could be facilitated to infer fine-grained provenance information. Since there are many programming and scripting languages and each has its own set of programming constructs and syntax, we showcase our approach using python scripts. Python[1] is widely-used to handle spatial and temporal data in the scientific community as well as commercial products such as ArcGIS[2] which has inspired us to make this choice.

Our main contribution is to infer workflow provenance information based on a given script which is used for actual processing. Moreover, we provide a guideline to further customize the workflow provenance graph based on user preferences. Users can also request fine-grained provenance information based on the achieved workflow provenance. We evaluate our proposed approach in a use case that estimates global water demand. Our evaluation demonstrates that the proposed approach can handle varieties of python scripts as well as it is relevant and suitable for scientists validating and investigating their model.

[1] http://www.python.org/
[2] http://www.esri.com/software/arcgis

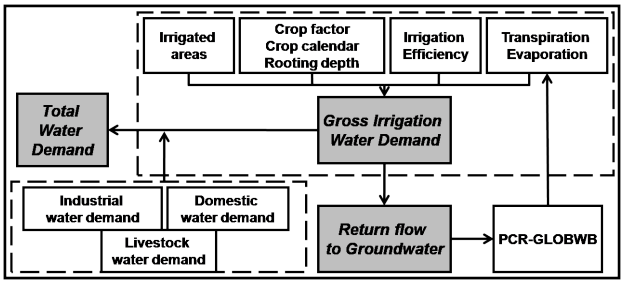


Fig. 1. Different types of data used in the use case

II. Use Case: Estimating Global Water Demand

Freshwater is one of the most important resources for various human activities and food production. During the past decades, use of water has been increased rapidly, yet available freshwater resources are finite. Therefore, estimating water demand and availability on a global level is necessary to assess the current situation as well as to make policies for future. In this use case, we focus on the script that estimates the total water demand from the year 1960 to 2000 at a monthly resolution.

A. Model Inputs

Source data are collected from different existing datasets. Irrigated areas are prescribed by the MIRCA2000 dataset [9] and the FAOSTAT database[3]. Crop factors, growing season lengths, and rooting depth are obtained from GCWM [10]. The irrigated areas are representative for the period 1960-2000 at a yearly temporal resolution, i.e. remains constant over each year, while the crop-related data sets are representative for the year 2000 at a monthly temporal resolution. A map of country-specific irrigation efficiency factors is also obtained from [11]. In addition, daily potential and actual bare soil evaporation and transpiration are prescribed from the simulation results from the global hydrological and water resources model PCR-GLOBWB [12]. Fig. 1 shows the input and output data and the dependences between them. The rectangles represent input data collected from various sources and the shaded ones represent output data. The edges from source to target rectangles represent dependences of target data on source data. All the data are PCRaster[4] maps containing 360×720 cells.

B. Computing Processes

The process begins with reading the annual and monthly input maps described above. First, using irrigated areas, crop factors, growing season lengths and potential transpiration, we calculate potential crop transpiration. Then, we calculate actual crop transpiration and determine the difference between potential and actual crop transpiration. In addition, we compute the difference between potential and actual bare soil evaporation for the top soil layer. Net irrigation water demand thus equals the sum of the differences between the potential and actual crop transpiration and between the potential and actual bare soil evaporation [13]. However, much of this water is lost to evaporation and percolation during the transport and application. Therefore, we calculate irrigation loss and add this to the net irrigation demand. At last, we use country-specific irrigation efficiency factors and multiply these with the net irrigation water demand to yield gross irrigation water demand.

The estimated gross irrigation water demand is then added to other sectoral water demands, i.e. industrial, domestic and livestock water demand, that are directly read from maps. Furthermore, we use gross irrigation water demand to calculate return flow to groundwater.

C. Model Outputs

Finally, the resulted total water demand, gross irrigation water demand, and irrigation return flow are reported as output maps (shaded boxes in Fig. 1) for each year from 1960 to 2000 at a monthly temporal resolution.

III. Basic Concepts

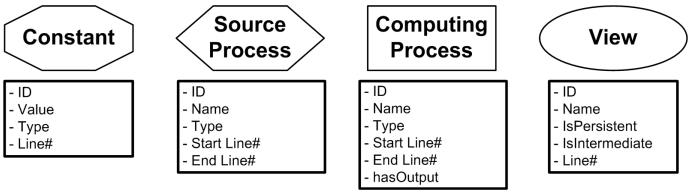


Fig. 2. Properties of different nodes

After introducing the use case, we describe the model to represent the output we are aiming to achieve, i.e. workflow provenance. Workflow provenance could be represented as a graph, known as *workflow provenance graph*. A workflow provenance graph G_{wp} is a set of (V, E) where V denotes the set of vertices or nodes and E denotes the set of directed edges. We introduce a graph model to distinguish different types of nodes. In our graph model, there are four different types of nodes. These are:

- **Constant**: represents any constant value taking part in an operation.
- **Source Process**: represents any operation that either assigns a constant or reads data from the disk.
- **Computing Process**: represents any operation that either computes a value based on its parameters or writes data into the disk.
- **View**: represents either any variable defined in the script or intermediate result generated by a process.

A directed edge connecting two nodes represents the data flow. In our provenance model, every source and computing process generates a view. Further, a view or constant node can be used as an input for multiple source and computing processes.

Each type of nodes has different properties. Fig.2 shows them. A *constant* node has an id starting with 'C', a value,

[3]http://faostat.fao.org/
[4]http://pcraster.geo.uu.nl/

the type of the value (e.g. integer, string etc.) and a line# referring to the line number in the code where it is defined. All the nodes also have this line# property. Since source and computing processes could be defined over multiple lines, they have start and end line#.

A *view* node has an id prefixed with 'V' and a name (variable name). It has also two important boolean properties: i) isPersistent and ii) isIntermediate. When *isPersistent=true*, it means that the variable which corresponds to this view is read from the disk or is written into the disk and hence, persistent. Otherwise, the view is not persistent and thus *isPersistent* becomes *false*. The property *isIntermediate* is *true* when the view is produced by a process and contains an intermediate result. Otherwise, *isIntermediate* becomes *false* and it indicates that the view is created because of defining the corresponding variable in the script.

The set of properties of both *source* and *computing* processes are almost similar except one property, *hasOutput*. This property belongs to a computing process which indicates whether a computing process produces a result that is persistent, i.e. written into the disk. Since source processes only read data from the disk, *hasOutput* is not applicable for a source process node. Among the other properties, both source and computing processes have an id prefixed with 'SP' and 'P' respectively, a name and type of operation (e.g. binary, function call etc.).

IV. Overview of the Approach

First, we parse a given python script based on a combined grammar, containing parser and lexer rules. After parsing the script, it returns an abstract syntax tree (AST) for the given python script. Then, we traverse through this AST based on a tree grammar and for each node in the AST, an object of the appropriate class based on the object model is created. Then, we build the initial workflow provenance graph based on our provenance graph model (see Sec.III). Since the initial provenance graph captures all syntactical details of the code, the size of this graph becomes quite large. Therefore, we apply a set of graph re-write rules on the initial graph to reduce the number of nodes and edges and thus achieve our *workflow provenance graph*. We also provide options to customize the workflow provenance graph to further reduce the graph complexity. Eventually, we infer fine-grained provenance based on the workflow provenance graph.

We have used an off-the-shelf grammar[5] as a starting point and extend it according to our requirements. In this paper, we focus on the mechanism of creating an initial graph, building a workflow provenance graph from the initial graph, customizing it and inferring fine-grained provenance eventually.

V. Generating Initial Workflow Provenance Graph

Fig. 3 shows a sample code snippet from the actual script. Since the complete script is bigger having 120 lines of code,

```
1   import PCRaster as pcr
2   conv1= 100
3   for year in range(1960,2001,1):
4       irrArea= pcr.scalar('irrigation\\irra%04d.map' % (year))
5       for month in range(1,13,1):
6           esp= pcr.readmap('PCRGLOBWB\\esp\\esp0%04d.%03d' % (year,month))
7           esa= pcr.readmap('PCRGLOBWB\\esa\\esac%04d.%03d' % (year,month))
8           irlCrop = pcr.max(0.,esp - esa)*irrArea*30/conv1
9           pcr.report(irlCrop,'tempResults\\irlCrop%04d.%03d'%(year,month))
```

Fig. 3. Code snippet from the use case

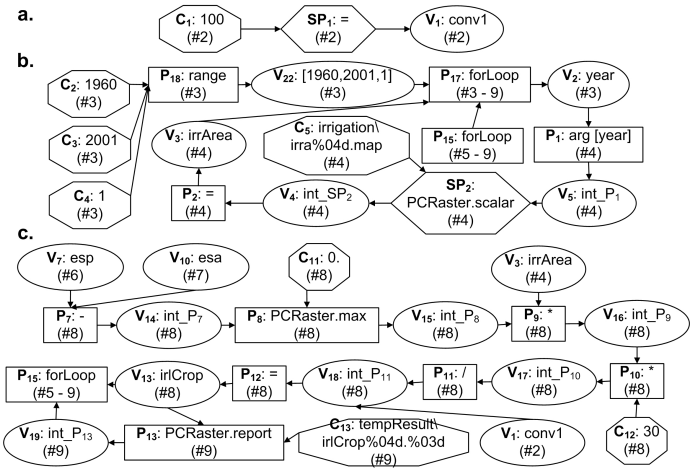


Fig. 4. Different sub-graphs in the initial workflow provenance graph

we use this code snippet as our running example. The code facilitates several functions from the PCRaster library to calculate irrigation loss for each year from 1960 to 2000 at a monthly resolution. The functions *scalar* and *readmap* are used to read input data from the disk. The *report* function writes the result into the disk. Another function *max* returns the maximum value among its parameters.

We generate the initial workflow provenance graph maintaining the flow of the program code of the script shown in Fig.3 by facilitating attributed graph grammar (AGG)[6] which is a graph writing engine.

We start by reading the first line where the PCRaster library is imported and is referenced as *pcr* through out the code. We maintain the mapping between library name and referenced name so that we can retrieve and use the actual library name to avoid any ambiguity.

Fig.4.a shows the sub-graph created for line#2 in the script where the value 100 has been assigned to the variable $conv1$. We create the following nodes: C_1 for the value 100 which is a *constant* node, SP_1 for the assign symbol which is a *source process* node since it assigns constant and V_1, the view node for the variable $conv1$. These nodes are also connected accordingly.

Fig.4.b shows the sub-graph created for line#3-4. In line 3-4, for each year starting from 1960 to 2000, data holding the irrigated areas value is read using the *scalar* function and is assigned into a variable, known as $irrArea$. We consider the *loop* as a computing process node (P_{17}) which takes the

[5]http://www.antlr.org/grammar/1200715779785/Python.g

[6]http://user.cs.tu-berlin.de/~gragra/agg/

range of $1960 - 2001$ as input and produces a loop control variable $year$ represented as a view (V_2). The $year$ variable is used as an argument for the *scalar* function to read irrigated area data of that corresponding year. Since the *scalar* function reads data from the disk, we represent this function as a source process node (SP_2) that takes the file name (C_5) and other arguments as input and produces a view (V_4), holding the intermediate result. Next, this intermediate result is assigned into the variable $irrArea$, represented as a view (V_3). The outgoing edge from $irrArea$ node (V_3) to the node created for the loop (P_{17}) indicates that $irrArea$ is defined within the scope of the loop.

Line#5-7 in the code read other data from disk nested around another loop. Since the mechanism of generating sub-graph for these lines is similar to the approach described for line#3-4, the sub-graph is not shown. Since the new loop in line#5 is defined within the scope of the old loop in line#3, in Fig.4.b, there is an outgoing edge from the computing process node created for the new loop (P_{15}) to the computing process node created for the old loop (P_{17}).

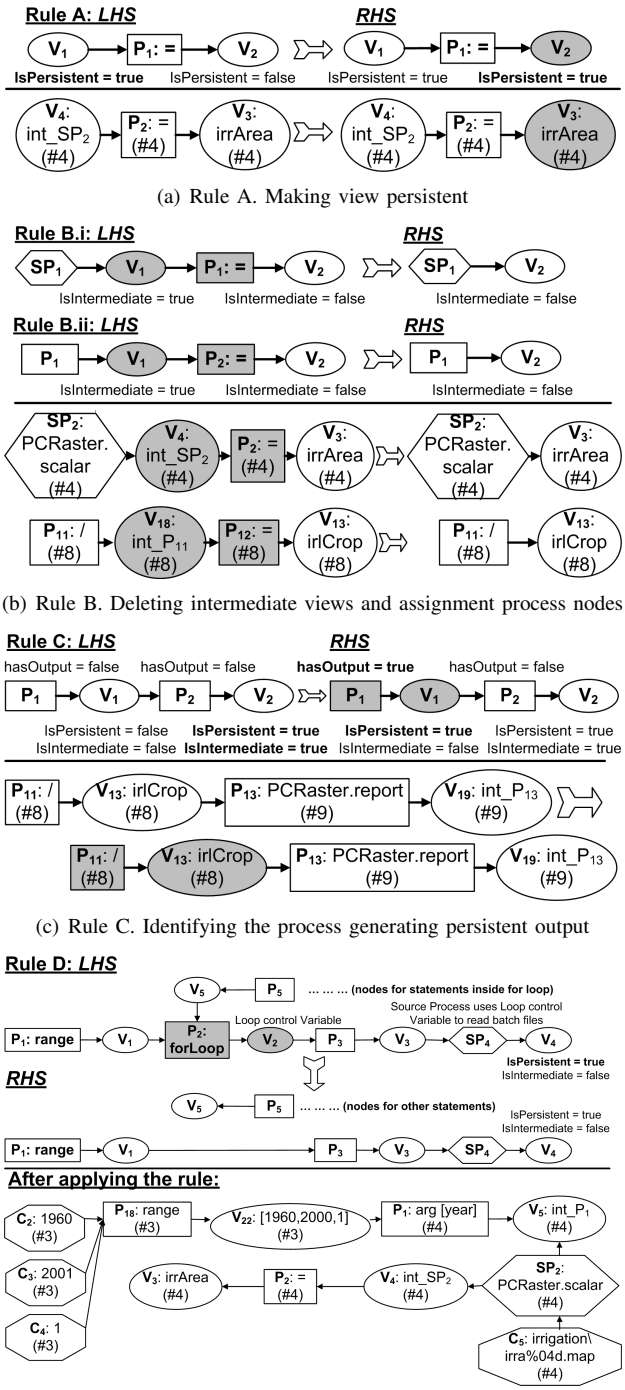
At last, we discuss the sub-graph generated for line#8-9. It is shown in Fig.4.c. Line#8 shows the formula to calculate $irlCrop$, the irrigation loss. We create a computing process (P_8), representing the *max* function, that takes its parameters as input and produces a view (V_{15}). Applying other operations result into creation of more computing process nodes (P_9, P_{10}, P_{11}) and eventually the result is assigned into the variable $irlCrop$, represented by the view V_{13}. Line#9 shows the use of the *report* function to write the values in $irlCrop$ into the disk. We create a computing process for the *report* function (P_{13}) and it is connected to its parameters accordingly.

VI. Building Workflow Provenance Graph

Since the initial graph captures all the syntactical details of the script, the number of nodes and edges are quite high. Many of them are intermediate nodes which could be deleted afterwards. Furthermore, identifying the process which generates final output and transforming any control-flow (e.g. loop) into data-flow dependences are also necessary to make the provenance graph more understandable. Therefore, we propose to use re-write rules to transform the initial workflow provenance graph. Each re-write rule has two parts: left-hand side (LHS) and right-hand side (RHS). Once a rule is defined and is executed, it searches for the pattern mentioned in the LHS of the rule. If the pattern is found, it is replaced by the sub-graph in the RHS of the rule.

A. Re-write Rules

Re-write rules are executed one after another. Rule A makes a view persistent *(IsPersistent=true)*, if another persistent view is assigned into it. The top portion of Fig.5(a) shows rule A. The bottom part of Fig.5(a) shows a sub-graph found in the initial graph which matches the pattern mentioned in LHS of rule A. The view V_4 is persistent and is assigned into the view V_3 via P_2. Therefore, executing this rule changes the *IsPersistent* property of V_3 from $false$ to $true$.



(a) Rule A. Making view persistent

(b) Rule B. Deleting intermediate views and assignment process nodes

(c) Rule C. Identifying the process generating persistent output

(d) Rule D. Identifying & eliminating 'forLoop' process

Fig. 5. Re-write rules

Rule B deletes all intermediate views *(IsIntermediate=true)* and subsequent assignment process nodes *(name='=')* if they are followed by a variable, thus a non-intermediate view *(IsIntermediate=false)*. It has two variants depending on the type of the node which produces the intermediate view (either a source process, SP_1 or a computing process, P_1) shown in the upper part of Fig.5(b). The lower part in Fig.5(b) shows

the patterns found in the initial workflow provenance graph and its rewritten versions for both variants. Executing this rule discards the shaded nodes from the initial graph and makes a connection between SP_2 and V_3 as well as between P_{11} and V_{13} for rules B.i and B.ii respectively.

Rule C identifies the computing process node which generates a persistent result, i.e. the result that is written into the disk. The top part in Fig.5(c) explicates the rule. V_2 is a persistent but intermediate view produced by the computing process P_2 that writes persistent data in the disk. P_2 has a non-persistent input view V_1 which is produced by the computing process P_1. Now, if this pattern matches to any of the sub-graphs in the initial workflow provenance graph, we change the value of a few properties of node P_1 and V_1 by following the given reasoning: since P_2 only writes data into the disk and do not change the data itself, the input view of P_2, V_1, is equivalent to the output view of P_2, V_2. Since V_2 is persistent, V_1 also becomes persistent *(IsPersistent=true)*. The aforesaid change leads us to make another change. Since V_1 is produced by P_1, P_1 must be the computing process which produces persistent and non-intermediate view V_1, referring to a variable defined in the script. Therefore, the value of *hasOutput* of P_1 becomes true. One may argue that since P_2 produces persistent view V_2, P_2 should have *hasOutput=true* also. However, it is not true since V_2 is an intermediate view which does not refer to any variable defined in the script unlike V_1. The shaded nodes in the bottom part of the Fig.5(c) show the nodes with the changed properties.

In any programming language, loops could be used for various purposes. In our example, shown in Fig.4.b, the computing process pointing to the loop in line#3 is P_{17} and it produces the view V_2 which refers to the loop control variable $year$. Later, V_2 is used to form the parameter for the source process SP_2 that reads files from disk. Furthermore, the $year$ variable represented as V_2 is not used as an input to any other operations within the script. Therefore, we conclude that the loop is used to iterate over data and does not manipulate any data structures (e.g. variable, array, list etc.). In this case, the computing processes referring to the loop are eliminated from the initial workflow provenance graph.

Rule D identifies and eliminates the loop mentioned in line#3 and 5 in the running example. The upper part in Fig.5(d) shows the LHS and RHS of the rule. The nodes P_{17}, V_2 and SP_2 found in the initial workflow provenance graph (see Fig.4.b) correspond to the nodes P_2, V_2 and SP_4 in the LHS of the rule. After getting the match to the pattern shown in the LHS, the nodes P_{17} and V_2 are deleted from the initial graph. The resulting sub-graph is shown in the lower part in Fig.5(d).

B. Graph Model Modification Rules

In the provenance graph model described in Sec.III, both source and computing processes have a view as an output. Therefore, the initial workflow provenance graph based on this model could be further reduced by discarding the views and also constants read by a source process. To ensure that no

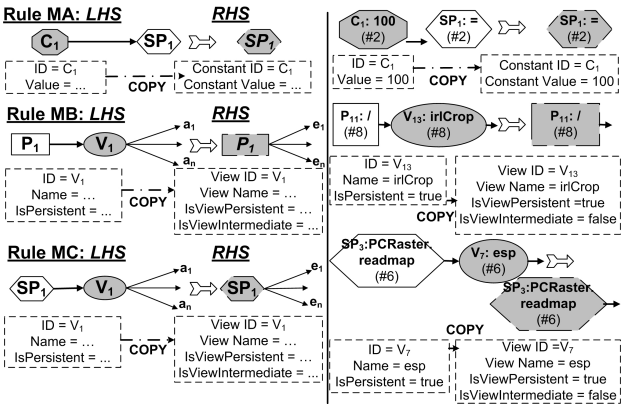


Fig. 6. Model modification rules and patterns found in the graph

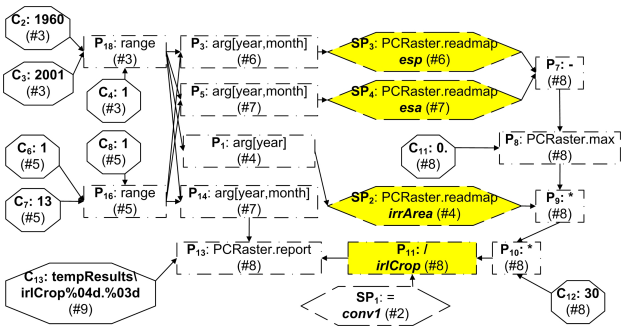
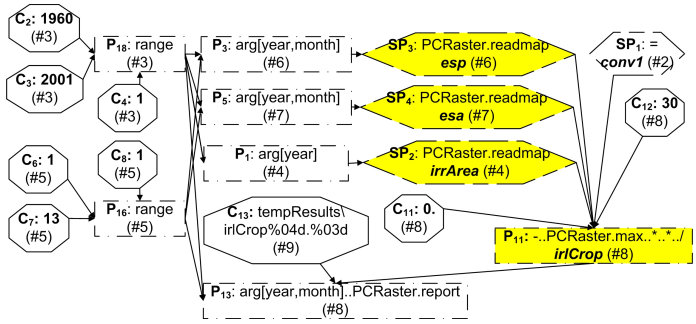


Fig. 7. Workflow provenance graph

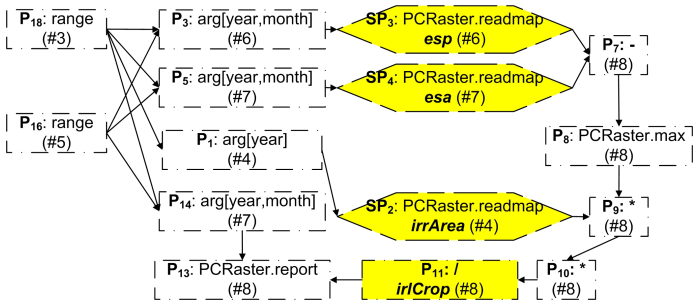
information is lost, we copy the value of a few distinguishing properties of the nodes to be deleted to the corresponding source or computing process nodes before the actual deletion takes place. Therefore, to apply these rules, we change our provenance graph model described in Sec.III. The new model has three types of nodes, except view nodes in the old model. The new model includes a few more properties for a source and a computing process. For a source process, we include the following properties with the existing ones: *i) Constant ID, ii) Constant Name, iii) View ID, iv) View Name v) IsViewPersistent* and *vi) IsViewIntermediate*. On the other hand, for a computing process, only the properties relevant to views are included. These are: *i) View ID, ii) View Name, iii)IsViewPersistent* and *iv) IsViewIntermediate*.

Left side of Fig.6 shows all three model modification rules. Rule MA unifies a constant node with the following source process node and deletes the constant node. If a match is found, the rule MA copies the value of *ID* and *value* of constant node C_1 to the property *Constant ID* and *Constant Value* of the source process node SP_1 and delete the constant node C_1 eventually. The dotted line in SP_1 refers to the source process node based on new modified model.

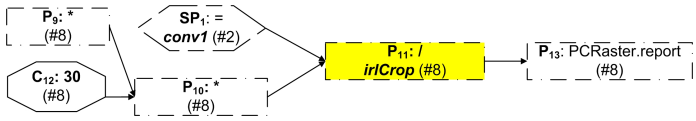
The other two rules, MB and MC, unifies a view node with the preceding computing process and source process node respectively and discard the view node. Rule MB and MC also ensure that the outgoing edges from the view node, i.e. $e_1, ..., e_n$, are now connecting from the computing process and



(a) Customized Workflow: Grouping intermediate computing processes



(b) Customized Workflow: Discarding constants



(c) Customized Workflow: Slicing process P_{11}

Fig. 8. Customized workflows

the source process node respectively. The right hand side of Fig.6 shows patterns for all three rules described above found in the initial graph and the sub-graphs which replace these found patterns.

After applying all these re-write rules and model modification rules, we achieve our workflow provenance graph. Fig.7 shows the workflow provenance graph for the running example (see Fig.3). The graph consists of three types of nodes: i) constant that is connected to a computing process only, ii) source process and iii) computing process. However, several source and computing processes are highlighted with light shade which means that the views produced by these processes are persistent *(IsViewPersistent=true)* and refer to the variables defined in the script *(IsViewIntermediate=false)*.

VII. Customizing Workflow Provenance Graph

The workflow provenance graph can be used to satisfy users with different level of understanding and objectives. To allow users to have more insight to the workflow provenance graph based on their choice, we provide a handful options to customize the workflow provenance graph.

We have observed that there might be several processing steps involved to produce a persistent view. As for example, line#8 in the running example involves several operations and eventually assign the result into a view that is persistent.

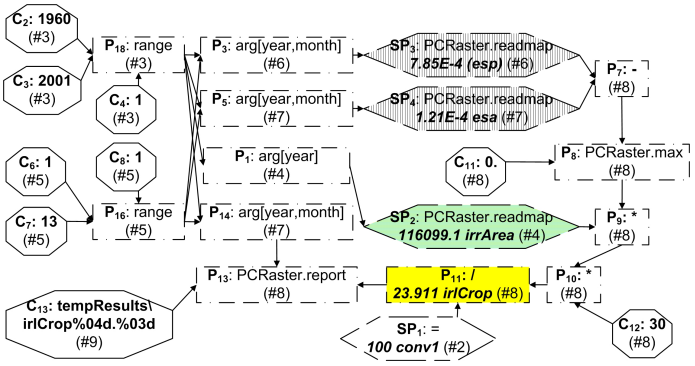


Fig. 9. Fine-grained provenance graph

Based on this observation, we decide to group intermediate computing processes together until we reach a computing process that produces a persistent view. This method of customization is termed as ***grouping process***. The resultant customized workflow provenance graph is shown in Fig.8(a).

The next customization of the workflow provenance graph is achieved by discarding the constant nodes from the graph, known as ***discarding constants***. This representation contains no information about the constants. Fig.8(b) shows the customized graph after eliminating all constant nodes.

We provide another option which allows users to put emphasize around a particular process node. We call this method of customization as ***slicing process***. In this technique, the user can select a process to visualize the nodes connected to the selected process with varying *radius* parameter. The *radius* refers to the highest level of ancestors and successors displayed around the selected process. Fig.8(c) shows the customized graph for P_{11} with $radius = 2$. Here, P_9 and C_{12} are 2nd level ancestors of P_{11}. There are no 2nd level successors of P_{11}. This customization method is analogous to the generic zooming in/out feature.

VIII. Inferring Fine-grained Provenance

Fine-grained provenance information could be inferred based on the workflow provenance graph and timestamps of contributed input values. In this use case, there are more than 3000 PCRaster maps containing input data. We create a SQLite[7] database that contains tables for each persistent view found in the workflow provenance graph and then populate these tables with the values transformed from the map files. Further, we attach a timestamp to every value based on the data collection time. The size of the database for the use case (see Sec.II) is around 40GB.

The inference phase is quite straightforward. First, users choose a particular value for which they want to have fine-grained provenance from any of the persistent output views. Each value is characterized by it's data collection time (year, month) and cell position in the (x,y) co-ordinates. Having this input from users, we apply the basic provenance inference algorithm [14]. This inference method is applicable to static

[7] http://www.sqlite.org/

data which perfectly suits to our use case. The method infers input values contributed to produce the chosen output value based on the given characteristics.

Fig.9 shows the fine-grained provenance graph based on the workflow provenance. The source computing processes are highlighted with different shades based on their data collection frequency. Suppose, both SP_3 and SP_4 read maps that vary over each month in every year, i.e. yearly-monthly variable map. The node SP_2 is highlighted in a different shade than those because it reads a map that varies only over the year, i.e. yearly variable map. In the fine-grained provenance graph, the actual data values are visible which help scientists to understand the origin of an unexpected value.

IX. Evaluation

We build a workflow provenance graph based on the python script having 116 lines of code. There are 438 nodes in the initial workflow provenance graph. After applying the re-write rules and model modification rules (see Sec.VI), the workflow provenance graph consists of 139 nodes which shows a significant reduction in the graph size by more than 300%.

We had several meetings with two scientists who are working in this use case. In the first meeting, we presented our approach of inferring provenance information and collected related data and scripts. Later, we developed our prototype and tested it with the given script as well as other python scripts.

After finalizing the prototype, we had another interview with the scientists to ask them several open-ended questions. We evaluate the proposed approach on the basis of four features: i) extensibility, ii) customization, iii) debugging-friendliness and iv) reproducibility.

A. Extensibility

Extensibility refers to the ability to handle different python scripts and building workflow provenance graph out of them. Our prototype can handle varieties of python scripts using different libraries. However, a user has to provide few basic information on each method call at the very first run. These includes whether the function reads persistent data or not (e.g. true/false) and whether the function writes persistent data or not (e.g. true/false).

Question: To what extent do you think that the **extensibility** of the proposed approach is helpful?

Feedback: The proposed approach is generic in the sense that it can handle varieties of python scripts and builds workflow provenance graph out of those. However, at the very first run, the user has to enter method-specific information which might be time-consuming and also requires some training for users.

B. Customization

Customization refers to the ability to adapt the workflow provenance graph based on user preferences. In Sec.VII, we discuss different customization techniques on the workflow provenance graph in detail.

Question: To what extent do you think that the **customization** on the workflow provenance graph is important?

Feedback: It is an important feature provided to users where users could customize the graph based on their objectives. The few basic options chosen for customizations are relevant. However, grouping process and discarding constants options could be only useful for representation, not for the modeling purpose. Moreover, it would be nice to add customization feature in other dimensions of the graph as well (e.g. node representation, graph layout etc.) to allow users to modify the layout or color scheme of the graph.

C. Debugging-friendliness

Both workflow and fine-grained provenance graph could be used for debugging purpose. The workflow provenance graph shows the flow of the program thus could be used for code-level debugging. On the contrary, fine-grained provenance graph refers to the input values and hence, could be used for value-level debugging.

Question: Have you ever experienced the need for a graph-based debugging tool? To what extent do you think that the provenance graphs are useful as a **debugging** tool?

Feedback: Usually, the scientists use the debugging tool which comes with the development environment. However, they appreciate the idea of debugging their code and the model using provenance graphs. Code-level debugging could be useful to determine the efficiency of the code, i.e. finding out code repetition. It is also useful to compare two different versions of the code expected to produce the same value. On the contrary, value-level debugging provides easy access to the actual data. It proves also beneficial when tracing back for identifying missing values in the file.

D. Reproducibility

Reproducibility refers to the ability to produce the same result using the same set of input values, irrespective of the time of the execution of the involved operations.

Question: To what extent do you think that fine-grained provenance graph is useful to achieve **reproducibility**? How do you use your reproducible results?

Feedback: Fine-grained provenance graph shows original data values contributed to produce the result which helps to achieves reproducibility. In practice, reproducible results might be useful to explain the mechanism of the model to one of the other scientists from the same group.

E. Discussion

The different features of the proposed approach makes it more practical for scientists to manage provenance data with limited knowledge of scientific workflows and database. The extensibility feature ensures that our solution can handle python scripts using different libraries. However, the first time entry of method-specific information must be done only once. Since it requires only few information, training phase for users should not be prolonged. The customization feature allows users to tailor the workflow provenance graph based on their

choice which has been appreciated. However, one of our future plans is to add further customization on different dimensions. Scientists could also see the use of provenance graphs for different levels of debugging. However, this feature provides static debugging only and cannot be used for debugging at each step execution. Eventually, scientists admit that the fine-grained provenance can achieve reproducibility to validate one's own model. Overall, the prototype satisfies the scientists with its simplicity and ease to use.

We demonstrate the mechanism of the proposed approach by facilitating a use case that involves static data. In case of a streaming scenario, the loop could be used to manipulate input data (e.g. array, list etc.) by implicitly defining trigger rate, i.e. how frequently the process should be executed and window size, i.e. the boundary over input data considered in a process. In this case, we could build workflow provenance graph by adding few properties to a process in the graph model based on [15].

X. Related work

There are several existing methods which maintain fine-grained provenance data explicitly. LIVE [4] is a complete DBMS which preserves explicitly the lineage of derived data items in form of boolean algebra. In sensornet republishing [5], authors used an annotation-based approach to represent data provenance explicitly which is expensive in terms of storage. These techniques work on the top of a relational database system with a specific workflow. Therefore, neither of these methods are applicable in our use case.

Recently, researchers have paid a lot of attention to make provenance-aware workflow engine. A provenance model described in [16] can collect provenance automatically during runtime. This model is an extension of Kepler[8] workflow engine. A layered model to represent workflow provenance is introduced in [17] which facilitates windows workflow foundation[9] as workflow engine. A relational DBMS has been used to store captured provenance data. These techniques assume the presence of workflow provenance before the execution starts and thus applicable for a closed system. Since our focus is to build the workflow provenance graph automatically from the given script, these methods cannot offer any help to us.

In [18], authors proposed an approach that can reconstruct provenance of the manipulations done over the data in an open system like excel sheet or a programming tool like R. This approach used a library of basic transformations to infer and reconstruct provenance for a particular value. Since it requires workflow of transformations prior to the reconstruction of provenance, this approach is not extensible enough.

Another work is proposed in [19] to document provenance by modifying the source code of a program automatically. It provides fine-grained data provenance after executing the script. However, one distinguishing factor is that our approach provides both high-level workflow provenance and fine-grained data provenance.

[8]https://kepler-project.org/

[9]http://www.windowsworkflowfoundation.eu/

XI. Conclusion and Future Work

Scientists feel the importance of provenance data. However, provenance data have been rarely maintained due to the lack of proper training to use workflow engines and other tools. Therefore, in this paper, we propose an approach which can build workflow provenance graph automatically based on a given python script and eventually can infer fine-grained provenance information. The approach is generally applicable to any procedural languages. We build a prototype of our system and is demonstrated to the scientists working in the use case. In future, we plan to improve user interface of the prototype as well as to add new functionalities. Overall, our proposed approach could help scientists to manage provenance with minimal training.

References

[1] H. B. Newman, M. H. Ellisman, and J. A. Orcutt, "Data-intensive e-science frontier research," *Commun. ACM*, vol. 46, no. 11, pp. 68–77.

[2] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *SIGMOD Rec.*, vol. 34, no. 3, pp. 31–36, 2005.

[3] P. Buneman and W. C. Tan, "Provenance in databases," in *SIGMOD 2007*. New York, NY, USA: ACM, 2007, pp. 1171–1173.

[4] A. Sarma, M. Theobald, and J. Widom, "LIVE: A Lineage-Supported Versioned DBMS," in *SSDBM 2010*, LNCS, vol. 6187, pp. 416–433.

[5] U. Park and J. Heidemann, "Provenance in sensornet republishing," *Provenance and Annotation of Data and Processes*, pp. 280–292, 2008.

[6] M. R. Huq, A. Wombacher, and P. M. G. Apers, "Facilitating fine grained data provenance using temporal data model," in *DMSN 2010*, ACM International Conference Proceeding Series, pp. 8–13.

[7] M. R. Huq, A. Wombacher, and P. M. G. Apers, "Adaptive inference of fine-grained data provenance to achieve high accuracy at lower storage costs," in *e-Science 2011*, IEEE Computer Society Press, pp. 202–209.

[8] M. R. Huq, P. M. G. Apers, and A. Wombacher, "Fine-grained provenance inference for a large processing chain with non-materialized intermediate views," in *SSDBM 2012*, LNCS, vol. 7338, pp. 397–405.

[9] F. Portmann, S. Siebert, C. Bauer, and P. Dll, "Mirca2000 - global monthly irrigated and rainfed crop areas around the year 2000: a new high-resolution data set for agricultural and hydrological modelling," *Global Biogeo. Cyc*, vol. 24, 2010.

[10] S. Siebert and P. Dll, "Quantifying blue and green virtual water contents in global crop production as well as potential production losses without irrigation," *Journal of Hydrology*, vol. 384, pp. 198–217, 2010.

[11] J. Rohwer, D. Gerten, and W. Lucht, "Development of functional types of irrigation for improved global crop modelling," *PIK Report 104, Potsdam Institute for Climate Impact Research*, 2007.

[12] L. P. H. van Beek, Y. Wada, and M. F. P. Bierkens, "Global monthly water stress: I. water balance and water availability," *Water Resources Research*, vol. 47 2011.

[13] Y. Wada, L. P. H. van Beek, D. Viviroli, H. H. Drr, R. Weingartner, and M. F. P. Bierkens, "Global monthly water stress: II. water demand and severity of water," *Wtare Resources Research*, vol. 47, 2011.

[14] M. R. Huq, A. Wombacher, and P. M. G. Apers, "Inferring fine-grained data provenance in stream data processing: Reduced storage cost, high accuracy," in *DEXA 2011*, LNCS, vol. 6861, pp. 118–127.

[15] A. Wombacher, "Data workflow - a workflow model for continuous data processing," CTIT, University of Twente, Enschede, Technical Report TR-CTIT-10-12, 2010.

[16] S. Bowers, T. M. McPhillips, and B. Ludäscher, "Provenance in collection-oriented scientific workflows," *Concurrency and Computation: Practice and Experience*, vol.20, no.5, pp. 519–529.

[17] R. Barga and L. Digiampietri, "Automatic capture and efficient storage of e-science experiment provenance," *Concurrency and Computation: Practice and Experience*, vol.20, no.5, pp. 419–429.

[18] P. Groth, Y. Gil, and S. Magliacane, "Automatic metadata annotation through reconstructing provenance," in *Semantic Web in Provenance Management*, CEUR Workshop Proceedings, vol. 856, 2012.

[19] S. Miles, "Automatically adapting source code to document provenance," in *Provenance and Annotation of Data and Processes*, LNCS, vol. 6378, pp. 102–110, 2010.