Interactive Provenance Summaries for Reproducible Science

Xiang Li and Xiaoyang Xu
Computation Institute
University of Chicago
Chicago, Illinois, 60637
Email: xiaoyangx, lix1@uchicago.edu

Tanu Malik
School of Computing
DePaul University
Chicago, Illinois, 60604
Email: tanu@cdm.depaul.edu

***Abstract*—Recorded provenance facilitates reproducible science. Provenance metadata can help determine how data were possibly transformed, processed, and derived from original sources. While provenance is crucial for verification and validation, there remains the issue of the granularity—detail at which provenance data must be provided to a user, especially for conducting reproducible science. When data are reproduced successfully the need for detailed provenance is minimal and an essence of the recorded provenance suffices. However, when data are not reproduced correctly users want to quickly drill down into fine-grained provenance to understand causes for failure.**

In this paper, we describe a drill-up/drill-down method for exploring provenance traces. The drill-up method summarizes the trace by grouping nodes and edges of the trace that have same derivation histories. The method preserves provenance data flow semantics. The drill-down method compares summary groups and ranks groups that may have information about the errors. Both the methods are implemented in an efficient manner using light-weight data structures so as to be suitable for reproducible science. We conduct a thorough experimental analysis to show how the operators perform in compressing and expanding real provenance graphs.

I. Introduction

Reproducibility requirements often entail experiments to be repeated in different computational environments [10], [4]. Containerizing has become a *de facto* method for reproducing computational experiments. In this method, code, data, and environment, i.e., elements associated with an experiment are copied into a single container. The resulting containers are significantly smaller in size than virtual machine images and require minimal installation and configuration in the new environment.

Provenance associated with a computational experiment is considered a crucial element of these containers. Included provenance describes how resulting data were derived from original sources and processed using binaries. A correct and authorized trace can thus be useful for validation and verification of future runs of the experiment, either exactly repeated or by changing parameters or data[1]. While most tools for building containers, such as Smart Containers [6], Parrot [13], SciPackage [11], Light Virtualization [12] record provenance, interacting with the recorded provenance, especially for verification and validation is currently time-consuming.

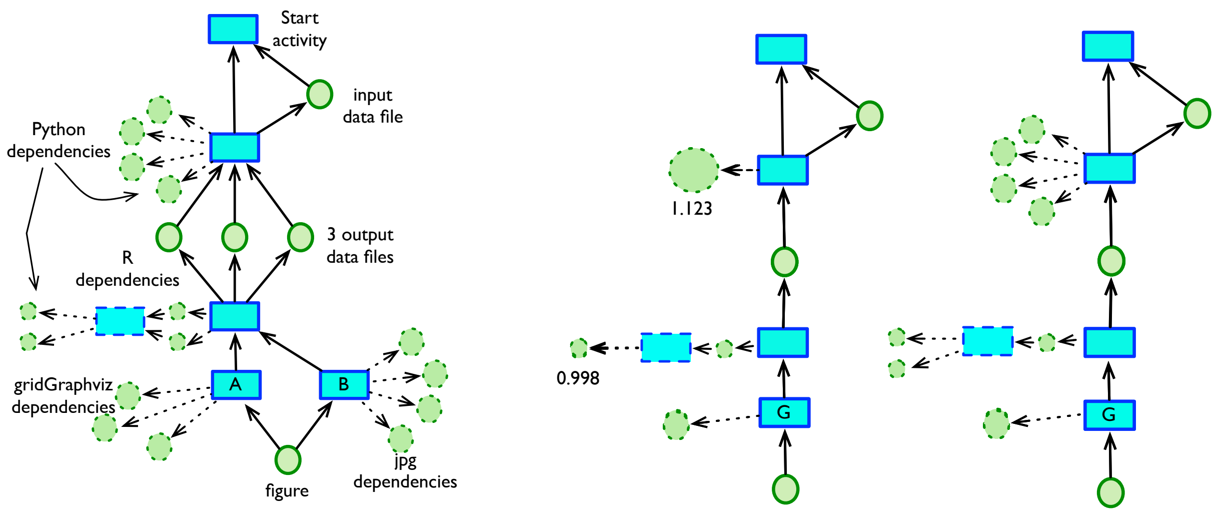
Owing to the light-weight property of the containers, the modest-size recorded provenance, is typically stored in light-weight databases such as SQLite or LevelDB. These databases, however, provide limited interfaces for querying and particularly so for exploring provenance, which is stored as directed acyclic graphs. Consequently, interacting with provenance data for conducting reproducible science becomes time consuming.

Current methods [2], [8], [1] for exploring provenance graphs may be included within the container, but most are either non-interactive or require significant user input; for example, provenance browsers [1] can be included for interacting with provenance, but require explicit queries to be formulated. Similarly provenance views [2] are based on the user explicitly knowing parts of the graph that are of relevance. Such knowledge is typically known for workflow systems but absent for ad hoc computation experiments that must be reproduced. Statistical and graph clustering methods [8] provide one-time summaries but are merely restricted to determining the patterns of very large provenance graphs.

Practically, the user is concerned with reproducing the computational experiment. If the experiment reproduced correctly (based on human judgement), then user simply wants to verify of the new provenance trace is similar to the old. In this case, low-level details of the provenance trace are not important. Instead, a summary is sufficient. However, in case the experiment is not reproduced correctly, the low-level details of the provenance trace become vital, and the user must be able to drill-down into such details relatively quickly.

In this paper, we describe a drill-up/drill-down method for interacting with provenance traces in containers. The drill-up method summarizes provenance so as to absorb its essence quickly; the drill-down method enables a user to quickly find anomalies. The former is based on grouping nodes and edges of the execution trace that have similar derivation histories. It produces a summary that is complete and preserves provenance data flow semantics. The drill-down method, instead of producing another summary, provides users with guidance on which summary grouping is more informative than others. General graph summarization methods [14] use resolution

[1]While there is no standard definition for reproducibility [3], in this paper we imply repeating to imply an exact run of the experiment, and reproducing to imply a similar run obtained by changing any of the code, data, and environment



(a) A Provenance trace.
Squares represent activities and circles represent entities.
Dashed entities and activities are low-level provenance details.

(b) Drill-up. Many dependencies and activities are grouped.

(c) Drill-down. Groups with high information are expanded.

Fig. 1: The drill-up and drill-down operators on a provenance trace

parameter to drill-up/down but the entire graph needs to be processed, which is inefficient for quickly drilling down to a few specific nodes in a provenance graph.

Figure 1 shows the use of the drill-up and drill-down methods for provenance traces. In Figure 1(a) the computational experiment is a simple Python analysis program that reads input data, transforms it into data for three variables that are plotted using R and visualized as a jpg picture. A explicit start node is added to show start of the experiment. Most container techniques being operating system specific, record provenance at the program and process level. The dashed nodes denote software dependency information (crucial for reproducing and therefore recorded), and are low-level details which in the case the program reproduces correctly can be ignored.

The drill-up method automatically ignores such low-level details by aggregating nodes that have the same *derivation histories* into a group. So for example, execution trace nodes, such as A and B in the graph, which have same derivation history and are similar files in a directory are grouped. Given a node, its derivation history is determined by node and edge types of its ancestors. If a node has two ancestors then it acquires the derivation history of both.

If the figure, however, fails to reproduce, then the user would immediately like to drill down. The failure might arise due to several reasons—a missing dependency, incorrect input files, or activity taking longer than required. We describe a method for scoring the nodes of the summary graph. The score indicates which nodes of the summary graph may contain the error. Figure 1(b) shows a scoring on the summary graph and a drill-down based on that scoring (Figure 1(c)).

The paper makes the following contributions:

- **Drill-up and drill-down methods for provenance graphs:** We summarize provenance graphs by grouping nodes based on their derivation history. We show that such a summary preserves data flow properties and is complete.
- **Efficient implementation within light-weight databases:** We show the summary method is light-weight and efficient, especially for conducting reproducible science in which the user wants to play with execution traces interactively, and for inclusion within light-weight databases used within reproducible packages.
- **Experimental evaluation:** We apply the methods to ad hoc workflows contained within reproducible packages and observing provenance at a fine granularity.

The rest of the paper is organized as follows. Section II describes a provenance model and drill-up/down requirements. The next two sections (Section III and Section IV describe the drill-up and drill-down methods. Section V evaluates the efficiency and quality of the operators. Related work related to provenance and graph summarization is described in Section VI and we conclude in Section VII.

II. Provenance Recording in Containers

Most containers use application virtualization (AV) mechanisms to create a package and record its provenance. AV mechanisms track application code using system call interposition methods such as *ptrace* (Unix), *dyld* (Mac OS X), etc. These methods, during interposition copy code, data, and environment and also track provenance dependencies. This tracking is done for all input files, including dependencies, system processes, network processes, and temporary data that the application interacts with, thus resulting in fairly large amounts of provenance. Provenance recording, depending upon the AV method, typically include additional attributes such as process name, owner, group, parent, host, creatine time, time of interaction, command line, environment variables and process binarys path. In the package, provenance is audited once by the author of the application and stored in the database as a reference for verification during re-execution time. For the recorded provenance, we assume that the following holds:

1) Recorded provenance defines a set of activity and entity types valid in for a domain.
2) Using recorded provenance it is straightforward to determine data dependencies that connect entities, e.g., a file written by a process p may depend on a file read by p.
3) The produced provenance can be represented in PROV.

Given the assumptions, we can define a model for recorded provenance generically as follows:

Definition 1 (Provenance Model)**.** *Let $\mathbb{L}$ be a domain of labels. A provenance model is a triple $\mathbb{P} = (\mathcal{A}, \mathcal{E}, \mathcal{L})$ where $\mathcal{A} \subseteq \mathbb{L}$ is a set of activity types and $\mathcal{E} \subseteq \mathbb{L}$ is a set of entity types. $\mathcal{L}$ is a subset of $\mathbb{L} \times (\mathcal{A} \cup \mathcal{E}) \times (\mathcal{A} \cup \mathcal{E})$. Each triple in $\mathcal{L}$ represents an edge type with an allowed start and end activity or entity type. We require that activity, entity, and edge labels be pairwise distinct.*

The recorded provenance can be defined in terms of a trace, formally defined as a graph in which the nodes are instances of the provenance model's activity and entity types and edges represent provenance dependencies.

Definition 2 (Provenance Trace)**.** *Let $\mathbb{P} = (\mathcal{A}, \mathcal{E}, \mathcal{L})$ be a provenance model. A provenance trace for $\mathbb{P}$ is a labeled directed graph $G = (V, E, T)$ with nodes V and edges $E \subseteq V \times V$. Each node must be of one of the activity and entity types specified in the provenance model and each edge must fulfill the type constraints specified by $\mathcal{L}$.*

Unlike the real provenance record in which each edge also records the time interval attribute, $T : E \to \mathbb{T} \times \mathbb{T}$ indicating when the two connected nodes interacted, in the provenance trace we ignore such interactions. In this paper we assume such time intervals to be negligible and interactions to be instantaneous. This is a safe assumption given that our current objective is summarization of the graph.

Given a provenance model $\mathbb{P}$ and its corresponding execution trace G, we consider two operators, the drill-up operator, $\ominus(G)$ which creates a summary ϕ of G and $\oplus(\phi)$ which given a summary, will create a information entropy based ranked order of groups to drill-down upon. Given the provenance model and the trace, the operators must satisfy the following three properties:

1) **The summary must be complete:** The resulting summary graph must be complete in that every node and edge in G can be mapped to a node and edge of the summary graph.
2) **The summary must preserve provenance data flows:** The summary must preserve the provenance data flows of G, i.e every path in the summary graph is a subset of the paths in G.
3) **The summary must guide to erroneous nodes or outliers:** The summary must not deeply hide nodes and edges that are useful for reproducing experiments.

III. Drill-up Operator

The drill-up operator is based on grouping nodes into groups that have the same type (*i.e.*, entity or activity) and have the same derivation history, where derivation history implies the same node and edge labels from the start node. Since G is a directed acyclic graph, a grouping consists of nodes that do not have a relationship between them. To identify nodes with the same derivation history first nodes of the same type are grouped, defined as

Definition 3 (Node Grouping)**.** *$\phi = \{G_1, G_2, \ldots, G_k\}$ is a node-grouping such that*
(1) $\forall G_i \in \phi, G_i \subseteq \mathcal{A}(\mathcal{G})$ or $G_i \subseteq \mathcal{E}(\mathcal{G})$, and $G_i \neq \emptyset$,
(2) $\cup_{G_i \in \phi} G_i = V(G)$,
(3) $\forall G_i, G_j \in \phi$ and $(i \neq j), G_i \cap G_j = \emptyset$.

For a given node grouping ϕ, the ancestor groups of a node v is the set $AncestorGroups_{\phi,E}(v) = \{Label(\phi(u)), Label \in \mathcal{L}$ and $(u, v) \in E\}$. Since, ancestor groups is a set, a node may have different ancestor groups. Nodes that do not have an ancestor are assigned the start node as an ancestor, with a start label edge. Now we define grouping nodes by derivation history.

Definition 4 (Derivation History Grouping)**.** *A grouping $\phi = \{G_1, G_2, \ldots, G_k\}$ has the same derivation history if it satisfies the following:*
(i) Node Grouping Definition 3,
(ii) $\forall u, v \in V(G)$, if $\phi(u) = \phi(v)$, then $\forall E_i \in \mathcal{L}$, $AncestorGroups_{\phi,E_i}(u) = AncestorGroups_{\phi,E_i}(v)$.

Finally, we define the drill-up operator $\ominus(G)$ in terms of the derivation history grouping.

Definition 5 (Drill-down)**.** *The drill-down operator $\ominus$ takes as input G that conforms to the provenance model $\mathbb{P}$ and outputs ϕ that satisfies Definition 4.*

The derivation history grouping definition differs from [14] in two distinct ways. First, the grouping in [14] is defined on both ancestors and descendants, which implies that nodes with the same types and same ancestors are separated into different groups if their descendants are different From a provenance perspective, such a separation is redundant. Second, to enable grouping by the same ancestors, we also consider edge labels as part of group definition. In [14] the neighbor group definition is only based on node labels. Thus the derivation history grouping is homogeneous in terms of node types and commonality of ancestors. We now show that $\ominus(G)$ as defined preserves the provenance data flow and is complete, which is not the case for the grouping defined in [14].

Theorem 1. *$\ominus(G)$ is complete and preserves the data flow.*

Proof. We first prove that $G' = \ominus(G)$ results in a complete graph. Let there be an edge $E = (u, v), E \in G$ that is not present in E', where E' is the set of edges in G'. This implies that nodes u and v are not in ϕ. However, this violates condition (2) for node grouping. Thus all nodes and edges in G.

We now prove that G' preserves the data flow i.e., every path in the summary graph is a subset of the paths in G. In other words there does not exist a path that is in G' but not in G. Let us assume there exists a path $P = u_{G_1} \rightarrow v_{G_2} \rightarrow w_{G_3}$ in G', where u_{G_1} is the node corresponding to group G_1 and Path P does not exist in G. This implies that there is a path from nodes in u_{G_1} to nodes in v_{G_2} and v_{G_2} to w_{G_3} but not from u_{G_1} to w_{G_3}. By this definition v_{G_2} must have at least two nodes v' and v'' grouped together such that there is a path u to v' and v'' to w. This implies u is an ancestor of v' but not of v''. Such a grouping violated grouping by derivation histories (condition (ii) of Definition 4).

□

The details of a fast implementation of $\ominus(G)$ based on topologically sorting G and when child nodes are combined checking if topological sort is violated are provided in the companion technical report [16]. The full pseudo code of the algorithm of $\ominus(G)$ is also provided in the report.

IV. Drill-Down Operator

The drill-down operator helps a user to flexibly zoom-in into groups of the summary graph which have more information about errors that arise while reproducing an experiment. In this section, we present a qualitative treatment about how to determine the subset of groups that must be drilled-down. The specific description of the types of errors is present in our experimental section (Section V)

To determine the groups to drill-down, we match the immutable $G = (V, E, T)$ available to us as part of the reproducible package with a new $G' = (V', E', T)$ generated when the package is run again. For nodes that are not present in G', i.e., $V - V'$, we need to determine specific ancestors and their corresponding groups in the summary graph. These groups are the potential groups in which to drill down. One way to identify such groups is to compute the edit distance between G and G' through a sequence of vertex and edge deletion operations and then check for isomorphism. However, it will still not identify the specific ancestors that are the cause of errors and their corresponding groups in the summary graph since there can be many such ancestors. In addition, checking for isomorphism is redundant since the graphs belong to the same computational experiment.

To determine specific ancestors (descendants), which need to be identified multiple times, instead of querying the graph repeatedly, we maintain a sketch of the provenance graph. This sketch is implemented as a matrix filter [9] in which the node u of an edge $(u \rightarrow v)$ is hashed into a row array of m bits, and the node v into a column array of m^2 bits. Using k independent hash functions $\{h_1, \ldots, h_k\}$, each of which has a range of $\{1, \ldots, m\}$, the i^{th} bit b_i of the row array is associated with the i^{th} set of m bits in the column array. A count of the number of ancestors(descendants) is maintained.

For nodes in $V - V'$, we determine the signature of the ancestors. All nodes and their corresponding groups that satisfy the signature are then identified. Obtaining the signature of ancestors is as $O(1)$ operation. Once a group is identified, we ignore checking for all other nodes in that group, thus identifying the groups an $O(G)$ operation.

While the above method will identify the potential groups to drill down, the number of groups to drill down may itself be several. To guide the user to relevant groups which might be more useful to drill down, we compute an inverse entropy measure of each group. Intuitively this measure informs how useful a group is for the purpose of drill-down in that what is the probability of finding a node that is directly leading to the cause of incorrect provenance graph. To determine useful groups, the nodes in a group are weighed with numerical measures. Given a weighted node group, an inverse entropy measure is defined as:

$$1/H(w_1, w_2, \ldots, w_n) = -1/(\sum_{i=1}^{n}(\frac{w_i}{W})\log_2(\frac{w_i}{W})) \quad (1)$$

where W denotes the sum of the reals, if $W > 0$, and 0 otherwise. We take $0\log_2(0)$ to be 0 in this computation. (Also, from here onward, we denote $\log_2$ by lg.) We define the entropy of a node-weighted, n-node group G_i with node weights $w_1, w_2, \ldots, w_n$, to be $H(w_1, w_2, \ldots, w_n)$. We will also use the shorthand notation $H(G_i)$ to denote $H(w_1, \ldots, w_n)$.

The justification for using an entropy follows from the fact that we wish to drill down into the group that provides the most information about the distribution of node weights to the viewer. It would not be very informative, for example, to drill down into a group in which all node weights are identical, which is equivalent to finding a group that has the maximum entropy. However, if in a group 99% of the weight is concentrated in a few nodes then the user will possibly be interested in drilling down in this group to see what properties of the nodes give the group a larger weight. The intuition to find the inverse of the entropy agrees with maximizing entropy, since entropy is maximized when all the weights are identical.

We describe two ways to assign node weights, in decreasing order of computational efficiency.

1) By node properties: Each activity and entity node recorded in G consists of several properties, such as file sizes, paths, access, process cpu and memory consumption, time it took to run the process, etc. The objective in node properties is to weight node based on the normalized values of the properties. For instance, entity nodes can be weighted by file sizes, and activity nodes by process memory consumption, since these two property values a good indication.

2) By number of descendents: We weight nodes based on the number of other nodes in whose lineage a given node appears. Intuitively this is weighing nodes that have a chance of stronger influence on their descendants. To determine this value we simply use the count value in the matrix filter.

Finally, the drill-down operator $\oplus$ is defined as

Definition 6. *The $\oplus$ operator takes as input ϕ and G' and outputs φ_{ranked}, a list of groups to expand ordered by Equation 1*

V. Experimental Analysis

A. Implementation

1) LevelDB implementation: LevelDB is a fast key-value storage library from Google that provides an ordered mapping from string keys to string values. Its API only supports `Put, Get, Delete, CreateSnapshot, WriteBatch, and RangeIteration` functions. Bloom filters are part of LevelDB. To store a graph in a LevelDB, we use LevelGraph which is a wrapper for storing graphs in LevelDB. LevelGraph considers each edge as an RDF triple and uses six indices for every triple in order to access them as fast as possible. The intuition for storing six triple is based on HexaStore [15] in which an edge is indexed in six possible ways, one for each possible ordering of the edge. The replicated indices provide upto five times speedup. We have implemented our drill-up and drill down methods as C++ routines in LevelGraph. All experiments were run on a 2.8GHz Pentium 4 machine running Ubuntu 14.04, and equipped with a 250GB SATA disk.

2) Provenance datasets in Containers: In general, we can build containers of all application programs. In this work, we consider applications for which we have established containers because those applications were as described by scientific communities either hard to reproduce, or in general required containerization, an in which we can test reproducibility scenarios. These containers are built with our SciPackage [11] in which provenance is audited using *ptrace*. We term these applications as **Athena**, which is a major experiment at the CERN Large Hadron Collider, **VIC**, which is a large-scale hydrologic model that applies water and energy balances to simulate terrestrial hydrology at a regional spatial scale [7], and **Swift**, where in we consider a raytracing workflow implemented and executed through the Swift parallel scripting language [5]. For details on these applications, we refer the reader to the companion technical report [16].

Table I shows the number of nodes and edges, average degree, and types of PROV edge relationships, and the number of properties that are recorded in these datasets.

TABLE I: Provenance graph analysis of analyzed datasets

	Number of Nodes	Number of Edges	Average Degree	PROV Edge Type	Number of Properties
Athena	7392	10237	3	11	25
VIC	3288	4936	2	6	12
RayTracing	1045	1284	6	5	4

3) Failures in reproducing: To evaluate the drill-down operator we must be able to simulate failures in workflows. The most common reproducibility issue in these workflows is missing dependency and here containerization solves that problem completely. Thus we must not be drilling down into dependency information. Thus instead we change input program and parameters for Athena and VIC workflows based on program semantics which lead to failures. Nodes and groups of interest that must be drilled-down into are manually identified, and tested against the drill-down operator. In these workflows, node weights are assigned based on number of descendants. In Swift no errors are introduced, but drill down is based simply on node properties.

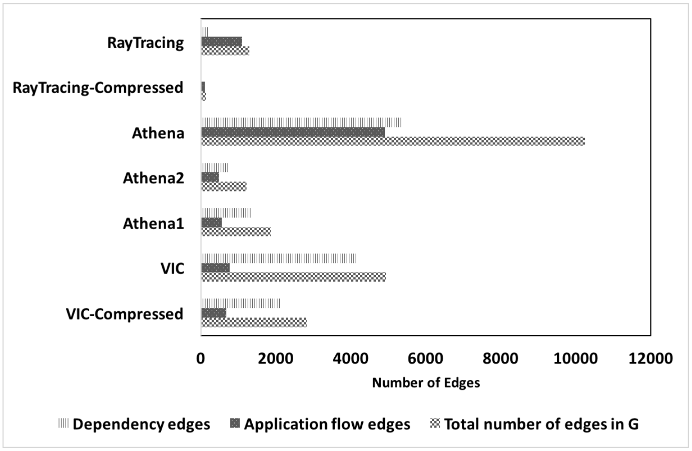


Fig. 2: Compression in Athena, VIC, and Swift

B. Effectiveness Evaluation

1) Is there a summarization?: We apply drill-up to the provenance traces generated from the workflows to see if there is effective summarization of the traces. The majority of edges in Athena and VIC are dependency edges, and so the measure for drill-up is to determine how good is the compression within dependency edges. To be able to compare the relative performance of drill-up on the different traces, we plot (i) the total number of edges, (ii) the number of edges corresponding to the application flow, (iii) the number of edges that are dependencies in the original graph and compare these values with that of the drilled-up graph (Figures 2).

We see a significant reduction in total number of edges compression ratio between 2.4 and 9.0 for all applications. Swift is a homogeneous workflow and therefore the compression is maximum. In Athena, we consider two runs Athena1, which is exactly repeating the workflow and Athena2 in which the input program is changed. While there is a factor of 8.5 reduction for Athena1, the repeatable workflow, the compression is reduced to 5.5 for the reproducible workflow. The variation is simply due to change in input parameters. For VIC there the compression is good with respect to dependencies, but not good with respect to main application flow. In reality there is compression within each sub-module of the hydrological parameter but barely any across them.

2) Drill-down performance: Identifying ancestors in groups: We examine if we are correctly able to identify ancestors, especially as the provenance graph grows. We measured the number of false positive answers to queries about whether a path between two vertices exists in a matrix filter. Since the sketch contains provenance metadata of an increasing number of nodes and edges it is expected to provide an increasing number of false positive responses. Figure 3 shows that the rate of such false positives is very low, ensuring that the sketches are robust as the provenance grows.

3) Drill-down performance: Does the drill-down guide to the most important groups?: For each workflow, the table (Table II) compares the number of groups that were identified as relevant to be drilled, and the number of groups which were

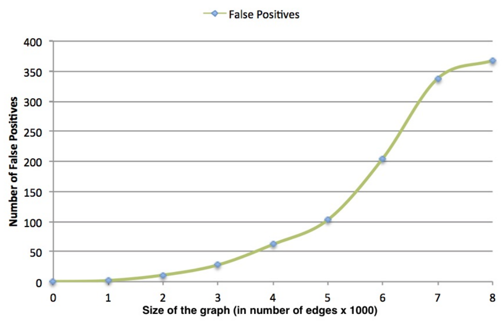


Fig. 3: Rate of False Positives

TABLE II: Group guidance

	Number of groups to be drilled	Number of groups not identified	Ranked Order
Athena	5	1	Yes
VIC	3	0	No
RayTracing	2	1	No

not identified for drilling. The table also shows the entropy-based ranked order and the order given to us by the workflow developers as the ones they would like to drill down.

VI. Related Work

ZOOM*userview [2] provides provenance graph abstractions to focus on the most relevant information. A user view is similar to drill-up and determines what level of sub-workflow the user can see, and thus what data and tasks are visible in provenance queries. While, we have focused on automatic creation of relevant views instead of they being user-defined like in ZOOM*userview, we also address the problem of selective drill-down, which is unaddressed in ZOOM*userview.

Macko et. al [8] summarize large system graphs us- ing statistical clustering methods and identify semantically meaningful tasks in an objects history. Our methods are non-statistical and based on pure aggregation of nodes and derivation histories. Thus they are easier to implement and use in light-weight databases embedded within reproducible packages than clustering based methods.

Tian et. al [14] define SNAP and k-SNAP operators on general graphs. These operators are based on user-selected node attributes and relationships and allow the user to control the resolution (i.e. the number of groups in a summary) based on the value of k. While SNAP works on both directed and undirected graphs, in this work, we have adapted SNAP for provenance graphs: grouping nodes with same ancestors but different descendants, and providing guidance to user about how to explore summary groups.

Typical relational databases come with query languages to support drill-up/drill down operations. In reproducible packages, provenance is however, stored in light-weight databases, such as SQLite and LevelDB. These databases support limited form of SQL and currently do not support the rollup operators.In this paper we have implemented the operators within LevelDB, a lightweight key-value database.

VII. Conclusion

In this paper we have presented an interactive method for exploring provenance that is included as part of the reproducible packages. The method provides two operators for interaction, a drill-up method that summarizes provenance graphs based on derivation histories, and a drill-down method that provides guidance into which summary group contains the nodes that are likely sources of error or anomalies. Through experimental evaluation we show that the methods are useful for summarizing large and complex provenance graphs obtained from reproducing both workflows and ad hoc data analysis experiments. The source code of our work is available at http://www.github.com/tanumalik/ProvSummary.

Acknowledgements

This work was supported in part by National Science Foundation grants ICER-1440327, SES-0951576, and ICER-1343816.

References

[1] M. K. Anand, S. Bowers, and B. Ludäscher. Provenance browser: Displaying and querying scientific workflow provenance graphs. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 1201–1204. IEEE, 2010.

[2] S. Cohen-Boulakia, O. Biton, S. Cohen, and S. Davidson. Addressing the provenance challenge using zoom. *Concurrency and Computation: Practice and Experience*, 20(5):497–506, 2008.

[3] Reproducibility of Data-Oriented Experiments in eScience. Dagstuhl Seminar 16041. http://www.dagstuhl.de/de/programm/kalender/semhp/?semnr=16041, 2016.

[4] J. Freire, P. Bonnet, and D. Shasha. Computational reproducibility: state-of-the-art, challenges, and database research opportunities. In *SIGMOD*, 2012.

[5] L. M. Gadelha Jr, B. Clifford, M. Mattoso, M. Wilde, and I. Foster. Provenance management in swift. *Future Generation Computer Systems*, 27(6):775–780, 2011.

[6] D. Huo, J. Nabrzyski, and C. F. Vardeman II. Smart containers: An ontology design pattern towards preservation of computational experiments. http://linkedscience.org/wp-content/uploads/2015/04/Linked-Science-ISWC-2015-Huo-et-al..pdf, 2016.

[7] X. Liang, E. F. Wood, and D. P. Lettenmaier. Surface soil moisture parameterization of the vic-2l model: Evaluation and modification. *Global and Planetary Change*, 13(1):195–206, 1996.

[8] P. Macko, D. Margo, and M. Seltzer. Local clustering in provenance graphs. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 835–840. ACM, 2013.

[9] T. Malik, L. Nistor, and A. Gehani. Tracking and sketching distributed data provenance. In *International Conference on eScience*, 2010.

[10] T. Malik, Q. Pham, and I. T. Foster. *SOLE: Towards Descriptive and Interactive Publications*. CRC Press, 2014.

[11] Q. Pham, T. Malik, and I. T. Foster. Using provenance for repeatability. In *USENIX NSDI Workshop on Theory and Practice of Provenance (TaPP)*, 2013.

[12] Q. Pham, T. Malik, B. Glavic, and I. Foster. LDV: Light-weight Database Virtualization. In *ICDE*, pages 1179–1190, 2015.

[13] D. Thain and M. Livny. Parrot: An application environment for data-intensive computing. *Scalable Computing: Practice and Experience*, 6(3):9–18, 2005.

[14] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 567–580. ACM, 2008.

[15] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[16] X. Xu, X. Li, and T. Malik. Technical Report. Interactive Provenance Summaries for Reproducible Science. http://dbgroup.cdm.depaul.edu/pubs/2016/ProvSummary, 2016.