Reorganizing Workflow Evolution Provenance

David Koop Juliana Freire

New York University

{dakoop, juliana.freire}@nyu.edu

Abstract

The provenance of related computations presents the opportunity to better understand and explore the differences and similarities of various approaches. As users design and refine workflows, evolution provenance captures the relationships between workflows as actions that mutate one workflow to another. However, such provenance may not always be the most compact or intuitive. This paper presents algorithms to update and transform workflow evolution provenance to achieve a representation that better exposes the correspondences between computations. We evaluate these algorithms based on the efficiency of the representation as well as the speed of the transformation.

1. Introduction

Data provenance is widely accepted as an important ingredient in ensuring the reproducibility of computational results. However, the usefulness of provenance extends beyond just data—there are benefits that derive from also capturing information about data sources, user interaction, and the development of the computation itself. How a computation evolves to produce a final result is important in understanding an analysis. In addition, exploring the differences between similar computations not only enables users better understand (and reason about) their results, but it also helps them in the development of new explorations. Finally, the ability to refer back to any past computation allows users to explore new directions without worrying about losing work.

Change-based provenance encodes user-driven actions in a tree structure [9]. It (1) captures the exact derivation of a particular specification from user actions and (2) stores related specifications in a compact form — instead of saving each specification individually, only the differences between them are saved. The first is arguably more important for provenance because it preserves the thought-process involved in creating or modifying a computation. However, the efficiency of change-based provenance allows *more* provenance to be stored, and it also enables useful operations over the provenance information, *e.g.,* visual differences and analogies [9, 12]. Note that the two are not mutually exclusive. With each change tagged by date, we can reorder change-based provenance to be more efficient while preserving chronological order.

For scientific workflows, change-based provenance captures the *evolution* of the specifications as users add and delete computational modules and change parameter values. Some systems, like VisTrails, allow users to navigate the provenance tree during their work, enabling the rapid recall and *use* of past versions as they branch off of previous work [8]. However, because the trees record a user's actions exactly, there is no guarantee that the provenance they record best captures meaningful relationships between versions.

For example, a user exploring New York City subway fare data might explore different subregions by plotting them with many different parameter combinations (*e.g.,* setting the latitude lower bound from 40.6 to 40.7 to 40.8). We can eliminate these intermediate changes to produce more compact provenance (see Figure 2a). Furthermore, consider a user who plots the station locations with labels and then later modifies that workflow to plot a subregion of the data with a different visualization. The latter workflow may be very different from the starting workflow, but because the user developed it by modifying the labeling workflow, the tree shows a relationship between them (see the **A** and **B** nodes in Figure 1a). We can move this workflow to branch from a more similar node elsewhere in the tree. For example, node **B** can be moved to branch from node **C** instead of **A** (see Figure 2b) because the number of actions to transform **C** to **B** (69) is fewer than the number of actions between **A** and **B** (95). Both types of reorganization—minimization and refactoring—create more compact version trees. At the same time, we must be careful to ensure that the reorganized provenance maintains the set of workflow specifications we are interested in.

In this paper, we formalize the problem of reordering workflow evolution provenance, present algorithms to accomplish this, and show preliminary results from applying these algorithms to real provenance traces.

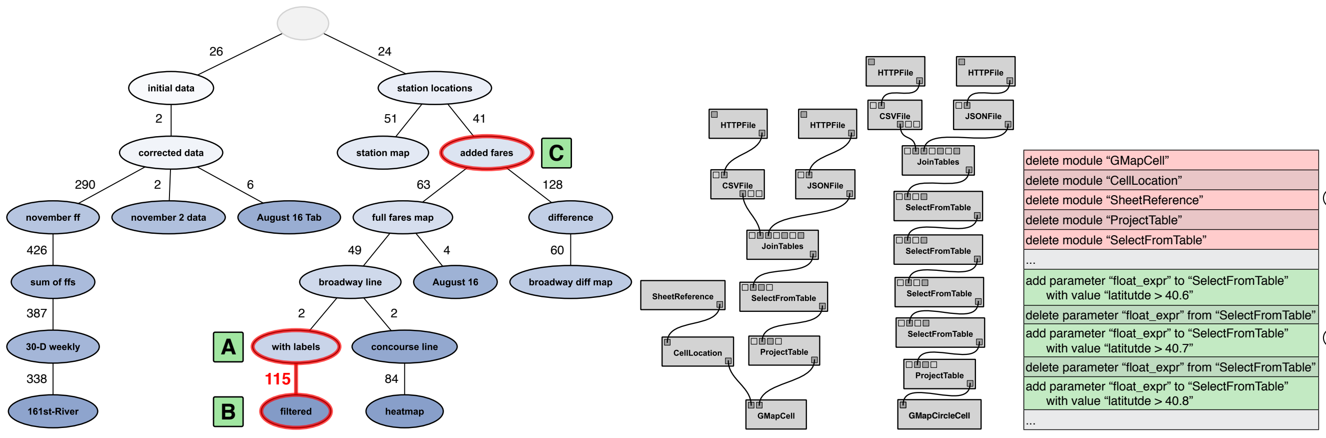
2. Background

Provenance captures how a particular result was derived. Computational tasks (*e.g.,* data analysis and visualization) often require a series of steps that may involve a variety of algorithms and tools. Scientific workflow systems provide infrastructure to design and execute workflows that encapsulate such complex computations. These systems also support provenance capture. Besides data provenance, which represents the lineage of a given data product, it is also useful to store the provenance of how the workflows were designed and how they evolved over time.

Workflows. A *scientific workflow* consists of a set of modules and connections that together define a computation. Each *module* has a set of input and output ports, and a *connection* links an output port from one module to an input port of another. Modules may also have literal *parameters* that serve as input values. A workflow is recursively executed by executing modules whose inputs are literals or are connected to modules that have been computed. An example of a workflow is shown in Figure 1b.

Change-based Provenance. Change-based provenance stores individual actions that can be composed to form full workflows. The idea is similar to version control systems: it efficiently stores a set of related provenance traces by keeping track of the changes between versions. However, in contrast to those systems, change-based provenance is driven by the actual actions rather than inferred by differences between files. Thus, it accurately captures the sequence of user-driven changes.

TaPP '14, June 12–13, 2014, Cologne, Germany.
Copyright is held by the owner/author(s).
ACM 978-1-nnnn-nnnn-n/yy/mm.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn



delete module "GMapCell"
delete module "CellLocation"
delete module "SheetReference"
delete module "ProjectTable"
delete module "SelectFromTable"
...
add parameter "float_expr" to "SelectFromTable" with value "latitutde > 40.6"
delete parameter "float_expr" from "SelectFromTable"
add parameter "float_expr" to "SelectFromTable" with value "latitutde > 40.7"
delete parameter "float_expr" from "SelectFromTable"
add parameter "float_expr" to "SelectFromTable" with value "latitutde > 40.8"
...

(a) Version Tree (b) Workflow **A** (c) Workflow **B** (d) **A** $\rightarrow$ **B** Actions

Figure 1: (a) A version tree captured during an exploration of New York City subway fare data. The tree shows only the vertices tagged by users. The number of vertices between two nodes is shown on the edges. (b) & (c) Two workflows from the exploration. (d) A subset of the actions that transform (b) into (c).

The set of changes (or actions) $\mathcal{A}$ are stored as a *version tree*, where vertices represent workflows and edges correspond to actions (or changes) applied to the workflows. The edges also capture the order in which the changes were applied. Given a node n, the trace for n is the path between the root of the tree and n, and corresponds to the composition of the changes at each node in the path. Thus, a path $\texttt{ROOT}, a_{i_1}, \ldots, a_{i_n}$ gives provenance $a_{i_n} \circ \cdots \circ a_{i_1}$. We define the *storage cost* of a version tree as the number of actions it stores (for all versions). Similarly, we can define the *materialization cost* of a specific version as the length of the path in the tree from the root to that version. Provenance is more efficiently represented when these costs are low. For example, the storage cost of the version tree in Figure 1a is 2101 actions and materialization cost of workflow **B** is 294 actions.

For workflows, changes can be divided into *add* and *delete* operations. (Note that changing a value can be accomplished by a delete value operation followed by an add value.) This means that not only do the changes encode what happened when the workflow was constructed, but they also provide a recipe for recreating the workflow in exactly the same fashion. Thus, change-based provenance also provides efficient representations for collections of workflows or other objects that can be created and modified using add/delete operations. Figure 1d shows a subset of the actions used to transform workflow **A** to workflow **B**.

3. Reordering Provenance

Problem Definition. Given a version tree $T(V, E)$ with vertices V and directed edges E and a subset of focus vertices $V^* \subseteq V$, we wish to construct a new version tree $T'(V', E')$ such that each workflow represented by $v \in V^*$ has a corresponding node $v' \in V'$. In other words, we want to create a version tree that contains the set of focus versions but may differ in the actions used to construct those versions. In addition, our goal is to *reduce* the storage costs for these vertices in the new version tree (materialization costs should also decrease in most cases).

As discussed earlier, we restrict the set of actions involved to adds and deletes, allowing us to more easily construct inverses. In addition, we assume that each vertex represents an action that is atomic—it adds or deletes a single item. While it is often useful to create compound changes that encapsulate all operations that were triggered by a single user action, we assume atomic changes to simplify the problem. We note that any compound change can be expanded to a set of atomic changes.

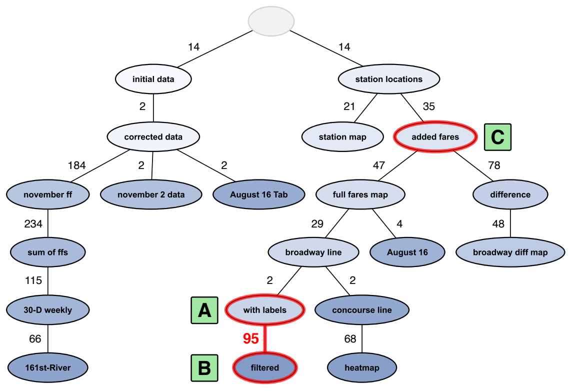
In what follows, we first discuss an optimal solution to the problem that creates a new version tree from scratch. Then, noting the infeasibility of such an approach as trees grow, we introduce two techniques that efficiently *transform* an existing version tree.

Optimal Solutions. An optimal solution to this problem is a version tree that has minimum storage cost for representing the vertices in V^*. Such a solution *minimizes redundant actions* and *maximizes the reuse of actions across different versions*. Thus, a sequence of changes to the same object, *e.g.,* add parameter a, delete parameter a, add parameter a, can be reduced, *e.g.,* add parameter a. In addition, if all versions in V^* use a particular object or have a certain property, the action that creates or sets it should appear once and all versions derive from it. Such a solution requires understanding not only how each version in V^* relates to each other but also each potential partial version.

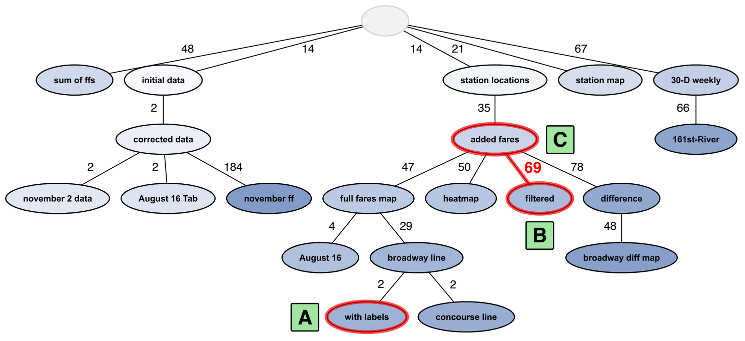
To make this more concrete, consider version trees that track the creation of sets. Any addition or deletion of an element from the set is independent; we do not have any constraints like adding A to a set S requires B to exist and C to be missing. The only requirement is that a delete action must delete an item that actually exists in the set. Now consider building a version tree for the sets $\emptyset$, $\{A, B, C\}$, and $\{A, B, D\}$. Since both non-empty sets share A and B, it would make sense to add those items first and branch when adding C and D. This creates a tree with storage cost 4 which is optimal.

In general, we can construct the optimal tree by solving the Steiner tree problem [10] over the complete power set graph. Of course, once we reach a few elements, enumerating that power set, let alone finding the minimum Steiner tree, which is NP-Hard, becomes computationally challenging. Furthermore, with sets, comparisons are straightforward. With workflows, which are graphs, we face the challenge of constructing all possible subgraphs and then solving the Steiner tree problem over that complete graph. Thus, heuristics will necessarily play a role in such computations.

Minimizing Version Trees. Minimizing a version tree consists of removing pairs of changes that are inverses of each other. In other words, if in the process of modifying a workflow, a user adds a module and later deletes it, those operations have no impact on the new workflow and can be removed. More formally, given a sequence of actions $a_1, \ldots, a_n$, if a_j is the *inverse* of a_i, $i < j$, the effect of $a_1, \ldots, a_n$ is equivalent to the effect of $a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{j-1}, a_{j+1}, \ldots, a_n$. Thus, we define *minimize-path* as a method that takes a sequence of actions and removes all complementary actions.



(a) Minimized Version Tree



(b) Refactored Version Tree

Figure 2: The version tree from Figure 1a has a storage cost of 2101. The minimized tree has the same shape but lower storage cost (1063) while the refactored tree moves vertices to take advantage of similarities, producing an even more compact tree with storage cost 785.

To minimize the entire tree, we minimize each path between any pair of nodes that should be preserved. These nodes are specified in the *skeleton* of the tree. Given a version tree $T(V, E)$ and a subset of focus vertices to be preserved, V^*, we define the skeleton $\mathit{Skel}(T, V^*)$ as the set of vertices V_S where

$$V_S = \{v \in V \mid v \in V^* \text{ or } \mathit{num\text{-}children}(v) > 1 \text{ or } v = \texttt{ROOT}\}$$

where $\mathit{num\text{-}children}(v)$ is the number of child vertices of v in T. Often, V^* includes all leaves of T but this is not required. The edges, E_S, are then compacted simple paths. Formally,

$$E_S = \{(v_1, v_2) \mid \exists\, \mathit{path}(v_1, v_2) \text{ and } \forall v \in \mathit{path}(v_1, v_2),$$
$$v = v_1, v = v_2, \text{ or } \mathit{num\text{-}children}(v) = 1\}$$

Then, minimization occurs for each edge of the skeleton. Specifically, we run $\mathit{minimize\text{-}path}(u, v)$ in T for each edge $(u, v) \in E_S$. In the example shown in Figure 1d, we can minimize the second group of actions to a single add (setting the value to 40.8). Figure 2a shows that such operations reduce the number of actions between workflows **A** and **B** from 115 to 95.

Refactoring Version Trees. One of the problems with minimization is that workflows that are similar but in different parts of the version tree cannot be moved closer, therefore, actions are repeated in different parts of the tree. To address these situations, we propose to compare *every pair* of workflows and reorganize them to create a more efficient version tree. However, because workflows are graphs and matching graphs is akin to subgraph isomorphism, a known NP-Complete problem, we leverage the existing version tree to make comparisons. Specifically, given the difference between two versions v_1 and v_2, computing the difference between v_1 and v_3 is easier if we know the actions used to transform v_2 to v_3. Instead of trying to match each pair from scratch, we instead use the shared history of changes to reduce the work.

As with version tree minimization, version tree refactoring takes an existing version tree T and set of selected vertices V^* and produces a new version tree T' with fewer actions. Here, however, we are not restricted by the original shape of the tree. Instead, we construct a complete graph G from the vertices V^* and compute a cost for each edge based on the differences between the workflows represented by each vertex. Then, we compute the minimum spanning tree of G and select the original root as the new root. The new tree is derived using the differences which are maintained as actions.

A key requirement is computing the difference between any two vertices in the version tree, and this is more involved when one vertex is not the parent of the other. We use a recursive strategy where we assume that the match between two vertices that are close (shorter path) in the tree can be extended to those further apart but using the same path. In determining the difference between two versions v_1 and v_2, we seek to equate actions that were taken independently but produced the same result. Note that in minimization, we determine cancelation by checking the unique identifiers of each object added or deleted, but in refactoring, we check the equivalence of the objects because they will *not share the same identifier*.

Our recursive algorithm $\mathit{diff}(v_1, v_2) = (d_1, d_2, d_3, d_4, d_5)$ returns a structure with five collections: adds specific to v_1, deletes specific to v_1, adds specific to v_2, deletes specific to v_2, and those adds which are equivalent. It begins with two base cases:

1. $\mathit{diff}(v, v) = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
2. $\mathit{diff}(\texttt{ROOT}, v) = (\emptyset, \emptyset, \mathit{minimize\text{-}path}(\texttt{ROOT}, v), \emptyset, \emptyset)$

Then, the general step is to find a parent difference, a difference located by moving up the tree, and integrating the actions between those nodes and the target differences. More formally, assume without loss of generality that v_1 is closer to the root than v_2. Then,

$$\mathit{diff}(v_1, v_2) = \mathit{update}(\mathit{diff}(v_1, \mathit{parent}(v_2)), \mathit{path}(\mathit{parent}(v_2), v_2))$$

where *parent* is defined on the skeleton of T, $\mathit{Skel}(T, V^*)$.

In the *update* function, we need to update the known parent diff based on the actions between the parent and child. Specifically, any *add* needs to be compared to the adds that already exist on the left side—if they are equivalent, they are moved to the equivalent collection and removed from the vertex-specific add collections. Similarly, if there is a new delete of an object that appears in the equivalent collection, we need to move the object to the left add collection. Other unrelated actions are added to the corresponding bins on the right side.

Once we have all of the differences computed, we can compute the *cost* of a difference as the number of actions that would be involved in transforming the workflow at v_1 to the one at v_2. Thus,

$$\mathit{cost}(v_1, v_2) = |d_1| + |d_2| + |d_3| + |d_4|$$

where $\mathit{diff}(v_1, v_2) = (d_1, d_2, d_3, d_4, d_5)$ as described earlier. Using the costs for each pair, we can build a minimum spanning tree on the complete graph and use that as the new version tree. Figure 2b shows how node **B** branches off of node **C** because $\mathit{cost}(\mathbf{C}, \mathbf{B}) = 69 < 95 = \mathit{cost}(\mathbf{A}, \mathbf{B})$.

However, we still have to transform the actions in the diff to viable actions. Specifically, going from one node to the other requires inverting the left-side actions and adding the right-side actions to them. Since we are dealing with add and delete operations, the inverses are straightforward. However, care must be taken to ensure that dependencies are preserved.

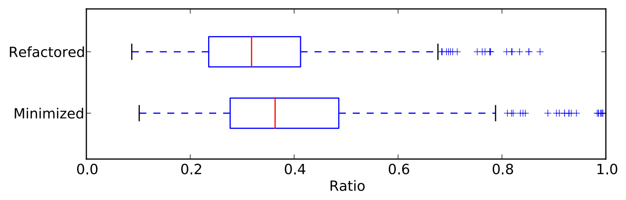


Figure 3: Boxplots showing the ratio of nodes in the new version trees to those in the original version trees for the two techniques.

4. Experiments

In order to evaluate the proposed techniques, we used workflow evolution provenance from VisTrails [8] traces and reordered the provenance in those examples using minimization and refactoring. Because VisTrails includes "change" actions which replace one value with another value in-place, we convert these to delete-add pairs of actions. This change increases the number of actions in the version tree but significantly simplifies the logic. Note that change actions could be reconstructed from delete-add pairs as a post-processing step. In addition, working with these traces requires checking dependencies when determining the equivalence of actions for refactoring. For example, knowing that two add parameter actions are equivalent requires checking that the actions affect the same module. Such dependencies mean we must check not only that the actions set the parameter to the same value but also that they are tied to the same module. Note that the equality of modules may have been determined through an earlier comparison.

We used 324 traces (vistrails) collected from students in a visualization course, and ran both the minimization and refactoring algorithms on those version trees. We selected tagged vertices (those annotated by the user) as focus vertices to minimize and refactor, and eliminated branches that contained no tagged vertices to make a fair comparison. The trees had between 95 and 14668 nodes (average 3345) and between 1 and 123 tags (average 18). We found that the techniques reduced the number of nodes in the trees to 0.41 and 0.35 of the original for the minimization and refactoring techniques, respectively. In addition, the refactoring technique performs better for provenance that is problematic for the minimization technique (see Figure 3) as expected because it can re-parent nodes. The techniques were implemented in Python 2.7 without optimization and were run on a 2.26 Intel Xeon processor. The minimization and refactoring techniques took, on average, 0.023 seconds and 5.67 seconds to compute new version trees. Note that because the refactoring technique must compare all focus vertices, we expect it to be significantly slower than the minimization technique which must only do a linear scan of all vertices.

5. Related Work

Provenance is important information, but if it takes too much space, users are unlikely to grant storage for it. To this end, there has been significant work to address provenance compression. Our work continues toward a similar goal for change-based provenance. Chapman et al. introduction factorization and inheritance techniques to store provenance more efficiently [4]. Anand et al. suggested techniques for exploiting redundancies in nested data collections using reductions [2], and Xie et al. applied web graph compression techniques to provenance [14]. Minimization has also been of interest for database provenance, and Amsterdamer et al. discuss *core provenance* and associated algorithms as a solution to reduce the provenance stored [1]. While we share with these approaches the goal to compact provenance information, we also aim to make the information easier to understand.

Change-based provenance is rooted in version control systems that maintain differences between files and directories. Such systems including svn and git maintain *differences* between files instead of creating complete new copies of each file [3]. Another version control system, darcs, establishes a patch theory that allows a more semantic meaning for each change to a file [5]. Change-based provenance is more akin to this type of system because it is based on user actions. Other related work in the area of version control includes the organization of versioned file hierarchies [6], and storing scientific array databases for efficient materialization [13].

6. Conclusion & Future Work

We have shown that reorganizing workflow evolution provenance produces more compact version trees, reducing storage costs and giving users more insight into which versions are similar. There are different applications that can benefit from provenance reorganization. For example, it can lead to more efficient workflow differences and analogies [12]. Another important application is understanding provenance from difference users. Suppose two users attack the same problem and we wish to understand how their approaches differ. By integrating the version trees using the reorganization techniques, we can better examine the differences and similarities. This may also be useful in summarizing workflow graphs [11].

There are two extensions that we plan to explore in future work. First, while creating version trees from scratch can be a challenging problem, we would like to explore techniques that build evolution provenance from collections of workflows using heuristics like those in phylogenetic tree construction [7]. Second, we plan to consider using *directed graphs* (instead of trees) to create smaller provenance footprints for workflow evolution. Directed graphs present potential merging issues as seen in darcs [5] but may also better allow integration of different explorations.

Acknowledgments. This work was supported in part by the National Science Foundation (CNS-1229185, IIS-1142013).

References

[1] Y. Amsterdamer, D. Deutch, T. Milo, and V. Tannen. On provenance minimization. In *Proc. PODS 2011*, pages 141–152. ACM, 2011.

[2] M. K. Anand, S. Bowers, T. McPhillips, and B. Ludäscher. Efficient provenance storage over nested data collections. In *Proc. EDBT 2009*, pages 958–969. ACM, 2009.

[3] S. Chacon. *Pro Git*. Apress, Berkely, CA, USA, 1st edition, 2009. `http://git-scm.com/book`.

[4] A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *Proc. SIGMOD 2008*, pages 993–1006. ACM, 2008.

[5] Darcs. `http://darcs.net/`.

[6] E. D. Demaine, S. Langerman, and E. Price. Confluently persistent tries for efficient version control. In *Algorithm Theory–SWAT 2008*, pages 160–172. Springer, 2008.

[7] J. Felsenstein. *Inferring phylogenies*. Sinauer Associates Sunderland, 2nd edition, 2004.

[8] J. Freire, D. Koop, E. Santos, C. Scheidegger, C. Silva, and H. T. Vo. *The Architecture of Open Source Applications*, chapter VisTrails. Lulu.com, 2011.

[9] J. Freire, C. Silva, S. Callahan, E. Santos, C. Scheidegger, and H. Vo. Managing rapidly-evolving scientific workflows. In *IPAW 2006*, LNCS 4145, pages 10–18. Springer Verlag, 2006.

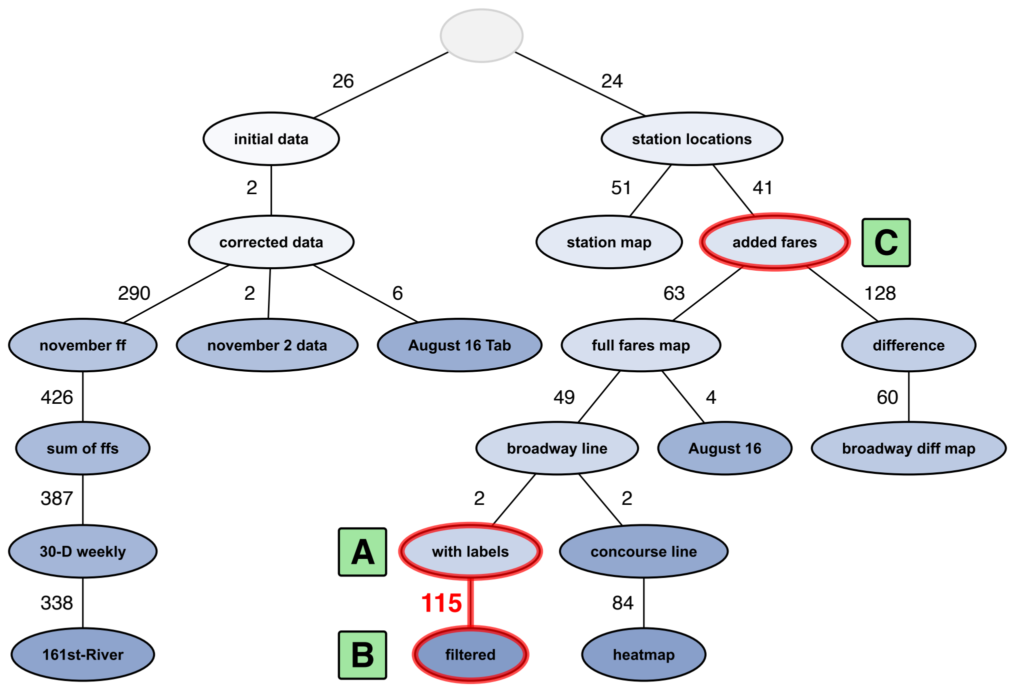
[10] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner tree problem*. Elsevier, 1992.

[11] D. Koop, J. Freire, and C. Silva. Visual summaries for graph collections. In *Proc. of IEEE Pacific Vis. Symp.*, pages 57–64, 2013.

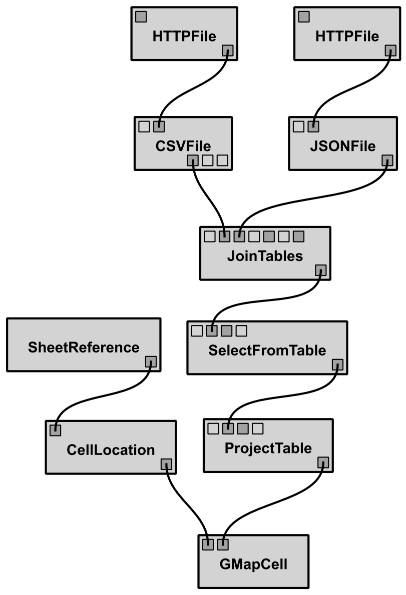
[12] C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and creating visualizations by analogy. *IEEE Trans. Vis. Comp. Graph.*, 13(6):1560–1567, 2007.

[13] A. Seering, P. Cudre-Mauroux, S. Madden, and M. Stonebraker. Efficient versioning for scientific array databases. In *Proc. ICDE 2012*, pages 1013–1024. IEEE Comp. Soc., 2012.

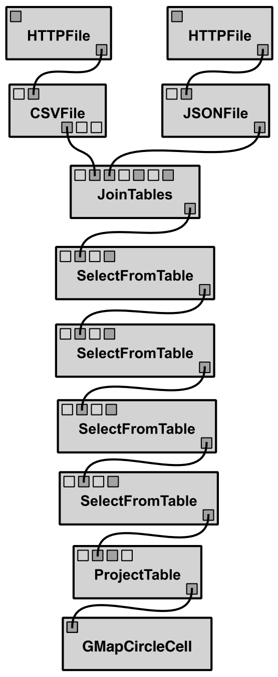
[14] Y. Xie, D. Feng, Z. Tan, L. Chen, K.-K. Muniswamy-Reddy, Y. Li, and D. D. Long. A hybrid approach for efficient provenance storage. In *Proc. CIKM*, pages 1752–1756. ACM, 2012.



(a) Version Tree



(b) Workflow **A**

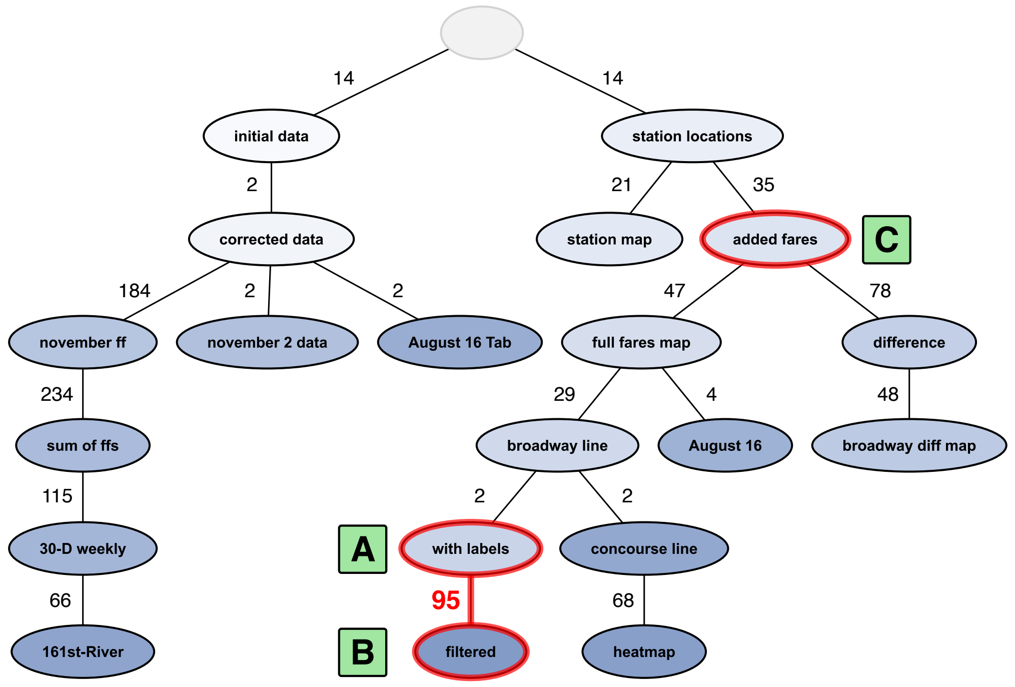


(c) Workflow **B**

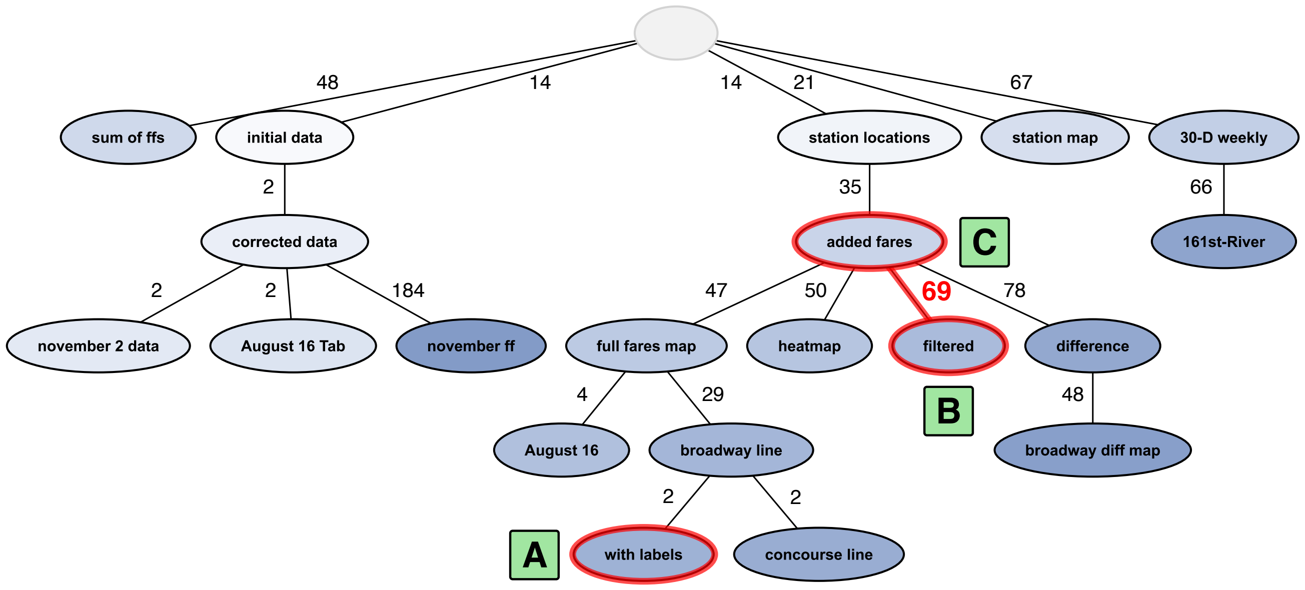
delete module “GMapCell”	
delete module “CellLocation”	
delete module “SheetReference”	①
delete module “ProjectTable”	
delete module “SelectFromTable”	
...	
add parameter “float_expr” to “SelectFromTable” with value “latitutde > 40.6”	
delete parameter “float_expr” from “SelectFromTable”	
add parameter “float_expr” to “SelectFromTable” with value “latitutde > 40.7”	②
delete parameter “float_expr” from “SelectFromTable”	
add parameter “float_expr” to “SelectFromTable” with value “latitutde > 40.8”	
...	

(d) **A** $\rightarrow$ **B** Actions

Figure 4: (Larger version of Figure 1.) (a) A version tree captured during an exploration of New York City subway fare data. The tree shows only the vertices tagged by users. The number of vertices between two nodes is shown on the edges. (b) & (c) Two workflows from the exploration. (d) A subset of the actions that transform (b) into (c).



(a) Minimized Version Tree



(b) Refactored Version Tree

Figure 5: (Larger version of Figure 2). The version tree from Figure 4a has a storage cost of 2101. The minimized tree has the same shape but lower storage cost (1063) while the refactored tree moves vertices to take advantage of similarities, producing an even more compact tree with storage cost 785.