Provenance-aware Versioned Dataworkspaces

Xing Niu[1], Bahareh Sadat Arab[1], Dieter Gawlick[2], Zhen Hua Liu[2], Vasudha Krishnaswamy[2], Oliver Kennedy[3], Boris Glavic[1]

[1]Illinois Institute of Technology
{xniu7,barab}@hawk.iit.edu, bglavic@iit.edu

[2]Oracle Corporation
{dieter.gawlick,zhen.liu, vasudha.krishnaswamy}@oracle.com

[3]SUNY Buffalo
okennedy@buffalo.edu

Abstract

Data preparation, curation, and analysis tasks are often exploratory in nature, with analysts incrementally designing workflows that transform, validate, and visualize their input sources. This requires frequent adjustments to data and workflows. Unfortunately, in current data management systems, even small changes can require time- and resource-heavy operations like materialization, manual version management, and re-execution. This added overhead discourages exploration. We present *Provenance-aware Versioned Dataworkspaces* (*PVD*s), our vision of a sandboxed environment in which users can apply — and more importantly, easily undo — changes to their data and workflows. A PVD keeps a log of the user's operations in a light-weight *version graph* structure. We describe a model for PVDs that admits efficient automatic refresh, merging of histories, reenactment, and automated conflict resolution. We also highlight the conceptual and technical challenges that need to be overcome to create a practical PVD.

```
{ "Treatment": [
  {
    "Patient": "John",
    "Disease": "Lung Cancer",
    "Doctor": "Xing",
    "Treatment": "Chemotherapy",
    "Suc": true,
    "Finish": true
  },
  {
    "Patient": "Bob",
    "Disease": "Stomach Cancer",
    "Doctor": "Bahareh",
    "Treatment": "Radiation",
    "Suc": false
  },
  ...
]}
```

(a) JSON document J

Treatment	Disease	Success
Chemotherapy	Lung Cancer	TRUE
Chemotherapy	Stomach Cancer	NULL
Surgery	Lung Cancer	FALSE
Radiation	Stomach Cancer	FALSE
Surgery	Lung Cancer	NULL

(b) T (Before lens application)

Treatment	Disease	Success
Chemotherapy	Lung Cancer	TRUE
Chemotherapy	Stomach Cancer	FALSE
Surgery	Lung Cancer	FALSE
Radiation	Stomach Cancer	FALSE
Surgery	Lung Cancer	TRUE

(c) T_{Fixed} (After lens application)

Figure 1: JSON document J storing treatment data (a), extracted relational treatment data T (b), and cleaned version T_{Fixed} (c).

1. Introduction

Data exploration and curation require analysts to develop and test hypothesis, typically by applying transformations (e.g., cleaning, re-structuring, or analytic querying) before interpreting the result. Ideally, such an analysis would be linear, with the analyst iteratively resolving successive problems in the data, tightening his/her analysis until the desired outcome is reached. However, in the real world, it is often unclear what data is needed for an analysis, how to obtain that data, how to curate and clean it to increase its quality, and how to express an analysis query. Thus, an analyst's day-to-day exploration often demands lots of interactive backtracking. For example, the analyst might recognize a past mistake, correct this mistake, and re-start the analysis from this step. Many data curation operations introduce uncertainty, e.g., when resolving constraint violations there often exist alternative ways of cleaning the data. Uncertainty leads to additional backtracking when a user explores different options for an uncertain choice - specifically if there are no tools available to understand the uncertainty and its effect on analysis results. Before introducing our vision of a version model that aids exploratory analysis, we present an exemplary curation process to further illustrate the challenges faced by analysts.

1.1 Running Example

Alice is an analyst at a hospital who wants to build a workflow to determine the success rate of different treatments for lung cancer. Treatment information is available as a JSON document J (an example is shown in Figure 1 (a)) that contains an array of treatments storing the patient, the type of disease, the responsible doctor, the treatment method, whether the treatment is finished, and whether it is successful. The information about success (`Suc`) and termination (`Finish`) of treatments is not available for all records. As a first step, Alice uses a query Q_{JT} to map the data from the JSON document into a relational schema to create relation T shown in Figure 1 (b). Assume that Alice misunderstood the data representation and retrieved the values of attribute `Success` from the field `Finish` in the JSON document. Realizing that this attribute is null in some of the tuples, Alice wants to apply a data curation step to fix these missing values. Using the Mimir system [7], she could create a missing value imputation lens. Lenses are data curation operators that apply a data cleaning operation like replacing `NULL` values with "best-guess" values selected heuristically by a classifier. Under the hood, a lens uses probabilistic database techniques to keep track of uncertainty introduced by the heuristic curation operation. This uncertainty is propagated to the results of transformations that are applied to a lens output and the Mimir system provides an API to expose the uncertainty. For example, the missing value imputation lens trains a model to predict the value for a missing attribute. In this case the uncertainty is based on the fact that the trained model predicts values with a certain probability. In our example, Alice chooses to not explore the uncertainty introduced by the lens immediately, but rather to continue building her analysis pipeline on top of the lens output (relation T_{Fixed} shown in Figure 1 (c)). She runs the SQL query Q_1 shown below to compute the success rate of treatment methods for lung cancer.

```
SELECT SUM(CASE WHEN Success = TRUE
                THEN 1 ELSE 0 END) / count(*)
         AS SuccessRate,
         Treatment
FROM   T_fixed
WHERE Disease = "Lung Cancer"
GROUP BY Treatment
```

When interpreting the results of this query Alice realizes that she made a mistake early on, i.e., when casting the JSON document into a relational form. Alices corrects her query Q_{JT} to extract `Suc` instead of `Finish` from the JSON document resulting in a query ${Q_{JT}}'$ and a new version T' of the treatment relation. What is frustrating for Alice is that she now has to repeat the lens creation and analysis query steps of her pipeline. Typically, the model trained for replacing missing values provides a good estimate, but should be tweaked to improve data quality if the attribute with missing values is critical for the analysis - as is the case in the running example. Alice can use Mimir to understand how the uncertainty exposed by the lens affects her result and focus her efforts to fix parts of the data and pipeline that are relevant for her analysis. This may require repeatedly modifying the parameters of the lens until a satisfactory result is achieved. This part of the process is labor-intensive, as most databases lack support for efficient modification of materialized view queries, forcing Alice to rerun the lens and analysis query after each modification. Her work would be greatly simplified using a system that automatically refreshes the extracted relation T, lens, and analysis query output for any change to Q_{JT} as well as the lens output and analysis query result if the lens is modified. Furthermore, during her exploration she should be able to keep track of past versions of her pipeline and how they relate to each other. The model we propose, supports both requirements.

1.2 Challenges and Requirements

Exploratory data curation systems should support the analyst in tracking, understanding, and eventually resolving uncertainty in a way that keeps her focused on her data and analysis. Current data management platforms support neither this exploratory mode of operation, nor do they expose uncertainty introduced by curation operations. Provenance-aware workflow systems do help the analyst to keep track of her operations, but only once a workflow has been developed and applied to inputs. Pay-as-you-go construction and modification of workflows is not supported. Provenance-aware databases can track uncertainty and potential errors in data, but would require the user to manually expose the uncertainty in curation operations. In short, provenance is a critical tool for enabling exploration, but current systems are lacking in several respects:

- **Regret-free exploration** - The user should be able to operate in a *sandboxed* environment where she can change past decisions and data derived based on these decisions should be automatically refreshed. Both base data and derived data are versioned.
- **Full accountability through provenance tracking** - The system should maintain both a record of the transformations executed by the user and their dependencies as well as be able to provide provenance at the data-level.
- **Automatic conflict detection and resolution** - The system should automatically detect conflicts that exist in the data as well as conflicts that are based on automatic refresh of derived data. Furthermore, when detecting a conflict, the system should propose potential resolution strategies.
- **Merging of transformation pipelines** - The system should enable analysis pipelines to be merged. For instance, a user may want to update an analysis based on recent changes to a database which requires merging the changes into the pipeline.
- **Uncertainty as a first-class concept** - Whenever an operation introduces uncertainty, the system should track and propagate this uncertainty through further operations, and be able to explain whether an output is uncertain and how the uncertainty affects an analysis result. This requires fine-grained provenance.

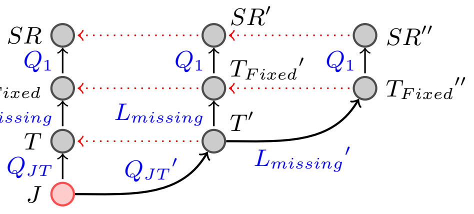

Figure 2: Virtual version graph for the running example.

Note that existing provenance-aware workflow systems can track transformations applied by a user to data in a workflow and may even keep track of changes to the workflow [6]. However, automatic refresh of derived data based on changes to previous steps in a workflow (e.g., Alice's change to Q_{JT}) is not supported. In this work we present the **Virtual Version Graph** model (VVG), a simple, yet powerful model for representing version histories that supports derived objects which are updated automatically as well as provides a clean semantics for changing past decisions (e.g., by modifying a transformation). Furthermore, we discuss Provenance-aware Versioned Dataworkspaces (PVD), our vision for a new type of sandboxed curation and analysis environment based on the VVG model. Data curation and analysis over large data sets can be hindered by the cost of rerunning steps in a pipeline. The PVD approach can lazily materialize relation versions once they are needed (e.g., to refresh a visualization shown to the analyst).

2. Virtual Version Graph Model

The functionality outlined in the introduction and motivated by the running example requires an expressive model for tracking objects, versions of objects, transformations, and dependencies among objects. To this end we introduce a simple, yet powerful, model that we call **Virtual Version Graph Model**. This model is a form of version control mechanism (multiple parallel histories can co-exit) with explicit tracking of transformations (version control systems typically do not track what transformation created a version), automatic updating of dependent objects, a principled and non-invasive way of changing past transformations, and a lightweight way to represent objects and versions that enables objects to be materialized on-demand. Although the model can be generalized, in this work we limit our discussion to objects that are relations.

A version graph G in our model is a directed acyclic hyper-graph where each node in the graph represents a version of a relation. Any transformation (e.g., query or update) creates a new relation that will be connected to the previous version via an edge labelled with the transformation. Furthermore, edges in the graph are classified into one of two categories: derivation or version (hyper-)edges. *Derivation hyper-edges* are used for operations that create a relation from one or more input relations (e.g., a relational join operator). *Version edges* are used to connect different versions of the same relation. To simplify the exposition we only consider transformations with a single input relation, i.e., no hyper-edges.

2.1 Types of Transformations

We distinguish between two fundamental types of transformations: 1) Transformations such as view creation that create a new rela-

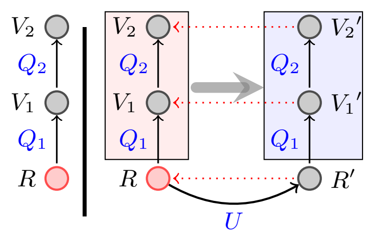

Figure 3: Recording an update U to a relation R in the VVG

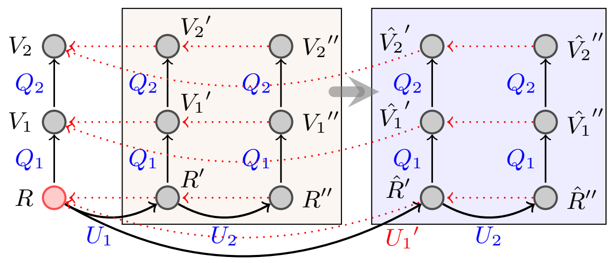

Figure 4: Modifying transformation U_1 to U_1'

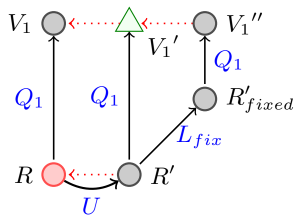

Figure 5: VVG with an ill-defined relation version

tion; 2) Transformations such as updates that create a new version of an existing relation. Both types of operations are represented as derivation edges in the version graph, but they are treated differently in terms of refreshing depended relations. For a derivation operation α (first type) applied to input relation R we create a relation node with a new label S that does not occur in the graph and connect R to S via an derivation edge labelled with α. For instance, the relation T in the running example is created by such an operation. When a versioning operation β (the second type of transformation) is applied to a relation R, then we create a new version of R, say R'. This version is connected to R via an edge labelled β. We also add a version edge from R' to R denoting that R' is a new version of R. A versioning operation also creates new versions for the full subgraph of dependencies rooted at R. These new versions of dependent relations will be connected to R'.

EXAMPLE 1. *Figure 2 shows Alice's pipeline construction as a VVG (only one lens modification step is shown). Ignore node colors for now. Solid edges are derivation edges and dotted edges are version edges. The original version of her pipeline is shown on the left: JSON Document J is used as an input to Q_{JT} producing relation T which in turn is cleaned up using lens $L_{missing}$ to produce T_{fixed}. Finally, Q_1 is run over this relation to produce relation SR storing the success rate of chancer treatments. Changing Q_{JT} (the query extracting relation T from JSON document J) into Q_{JT}' causes new versions of the relations derived from T to be created (these are relations T', T_{fixed}' and SR'). When Alice changes the parameters of the lens afterwards ($L_{missing}'$), then new relations versions SR'' and T_{Fixed}'' are created. Note how versions of the same relation are connected via version edges.*

2.2 Automatic Update of Derived Relations

For a modification of a relation R (e.g., an update) we create new versions of all relations that are derived from R, i.e., all relations that are reachable from R in the VVG. Thus, we create a copy of the subtree rooted at R, the input of the edge corresponding to the modification, and connect it to the end point of the edge corresponding to the modification (the new version R' of the updated relation). We also add a backlink from each copy S' of a node to its previous version S to indicate that it is a new version of S.

EXAMPLE 2. *Figure 3 shows how applications of an update operation affects a VVG. In the graph before the update (shown on the left), there are two relations that depend on R: V_1 created by query Q_1 and V_2 derived from V_1 by running Q_2. Once the update U is applied, new versions V_1' and V_2' of these relations are created, are connected to the new version R' and linked back to V_1 and V_2, respectively (shown on the right). In the figure, the subgraph derived from R is enclosed in a red box and its copy in a blue box.*

2.3 Modifying Transformations

In addition to creating new versions of data, we would like to support updates to transformations as well, e.g., such as Alice's modification to the JSON extraction query Q_{JT} in the running example. By modeling the transformation's arguments (e.g., the extraction query) as data to be versioned as well, transformation updates are naturally supported in the VVG model as well. When a transformation's arguments are modified, a new version of the argument's node is created, and the effects are propagated throughout all of the dependency edges as well. Note that for simplicity we do not show nodes for transformation arguments in the example graphs.

EXAMPLE 3. *Figure 4 shows a VVG for a variation of the previous example. There is a relation R plus two views defined based on R (V_1 and V_2). Two update operations U_1 and U_2 have been applied to R creating new versions of R and its dependent relations (the views). Assume the user determines that transformation U_1 is buggy and should be modified. If the modified transformation U_1' is applied to R, then all relations derived from R' (enclosed in a rectangle in the figure), the original output of U_1, have to be updated too. Thus, a copy of this subgraph is created and connected to the new version of R. We use "ˆ" to denote versions in this new subgraph. Note that for modifications to transformations the new versions of derived relations that are created based on this transformation retain the same versioning history as the versions they are copied from. This effectively creates a branch in the history of such relations. For instance, consider $\hat{V_1}'$ which was copied from V_1' and is recorded to originate from V_1, the origin of V_1'.*

2.4 Detecting and Dealing with Conflicts

Our approach of automatically creating new versions of derived relations when a new version of a relation is created may lead to the creation of relation versions that are ill-defined. For instance, consider a view V_1 derived by a query Q_1: `SELECT name, age FROM R`. If a user updates the R relation by deleting the `age` attribute, then the automatically created version V_1' of the view V_1 derived by applying Q_1 to R' is invalid. This is the case, because the `age` attribute no longer exists. One option would be to reject every operation that would lead to invalid derived relations. Given the pay-as-you-go nature of our technique we prefer to use a different approach. When new versions of derived relation are created we apply basic sanity checks and if a transformation that created a relation is no longer well-defined, then this relation is marked as invalid in the graph. We envision to make a wide variety of semi-automated and automatic conflict detection and resolution strategies available to the user. For example, if we allow integrity constraints to be defined for each relation, then we can automatically detect violations to these constraints and propose constraint-based data cleaning methods (e.g., lenses) to the user to resolve conflicts. For instance, assume that the user has defined a primary key ($name$) for relation R. An insert of a new person into R with the same name as an existing person would violate this constraint. Conflict resolution operations are represented as transformations in our model.

EXAMPLE 4. *Figure 5 shows an extension of the example described above. Assume that an ill-defined relation version V_1'*

(green triangle) was caused by the automatic refresh of view V_1 based on an update U dropping the `age` *attribute. The conflict caused by the missing attribute* `age` *has been resolved by applying a lens L_{fix} to compute values for the missing attribute* `age`*.*

2.5 Merging VVGs

An important functionality we would like to provide is merging of VVGs, e.g., to combine analysis pipelines or to repeat an analysis with new data. When merging VVGs we face similar challenges as do version control systems since there is no linear history and, thus, merging strategies, conflict detection, and conflict resolution are required. In contrast to version control systems we could take transformation semantics into account when merging. Furthermore, we will consider conflict detection and resolution strategies such as the ones mentioned in the previous section in addition to traditional types of conflicts such as conflicting updates to the same tuple.

3. PVD

We envision PVD to operate based on the version model introduced in the previous section. To start an analysis a user would create a new PVD as a virtual copy of the current state of a database (or another PVD). This creates a sandbox environment for data curation and analysis tasks where every operation is reflected in a VVG initialized during PVD creation. A PVD would be exposed through an interface similar to iPython (`http://ipython.org/`). The user can execute operations on current versions of relations and define a set of visualizations based on relations. These visualizations are automatically updated whenever a new version of an underlying relation is created. Visualizations are represented as distinguished nodes in the VVG. Only nodes that correspond to visualizations have to be materialized. Thus, we can choose materializations in a way to optimize a particular objective, e.g., the amount of storage required or the cost of deriving new materializations.

3.1 PVD Building Blocks

We argue that recent provenance-related techniques introduced by the authors - namely on-demand data curation through lenses [7] as well as provenance tracking and reenactment for updates [2] - provide a solid foundation for implementing PVD. Lenses [7] would provide PVD with powerful uncertainty aware data curation operations. The build-in support for fine-grained data provenance in lenses that records how uncertainty affects a query result could be complemented with our approach for provenance for updates [2] to be able to also track uncertainty through update operations. Furthermore, reenactment, the declarative replay technique that we have developed to retroactively compute provenance for updates can be used to virtualize update operations. By translating an update into an equivalent query we can compose it with other updates and queries. This results in additional options for how to materialize a relation version (see Appendix B.2) and the ability to track provenance through such a composed transformations.

3.2 Implementation Challenges

While the aforementioned technologies can be the foundation for PVD, many challenges remain. Developing effective strategies for determining which relation versions to materialize, when to materialize, and how to materialize will be essential in scaling workspaces. Similarly, methods for compressing VVGs, e.g., sharing nodes among copies of a subgraph, will help keep the size of such graphs manageable. Incremental view maintenance techniques can provide a PVD system with additional, potentially more efficient, options for materializing relations. We will study the types of conflicts that can arise in VVGs given a class of allowed transformations, develop efficient techniques for detecting and resolving such conflicts, and study methods for merging VVGs.

4. Related Work

The version graph model we have introduced is inspired by how version control systems such as Git (`https://git-scm.com`) treat version histories with parallel branches which is a natural way to model revisions of past decisions in data curation and exploration. However, version control systems lack the virtualization capabilities (recompute a version instead of materializing it) and automatic refresh of derived objects that are an inherent to PVD. DataHub [3] is a system for dataset storage inspired by version control systems. We share DataHub's vision that the choice between materialization and re-computation can be turned into an optimization problem. In contrast to DataHub, we target data exploration pipelines which informs our choice to propose automatic refresh of derived relations and provide explanations for outcomes based on fine-grained provenance. Many transformations in a VVG will be declarative. Hence, incremental view maintenance, e.g., the techniques provided by DBToaster [1], should be used to speed up refresh of derived relations. Our approach to provenance management is informed by the large body of work on workflow [5] and database provenance [4]. In contrast to typical approaches for workflow provenance, in our model there is no direct notion of a workflow - pipelines are built incrementally and data is automatically kept up to date. Our model shares with [6] the aspect that version histories do not have to be linear. In contrast to most database provenance approaches we keep track of the transformations and how they create versions of relations. The virtual nature of relation versions in our model aligns well with on-demand provenance techniques such as the ones of GProM [2].

5. Conclusions

We have introduced our vision for *Provenance-aware Versioned Dataworkspaces*, a novel approach for providing data analysts with a sandboxed environment with several unique features that ease their day-to-day work. A PVD enables users to keep full account of their operations and supports exploratory application of data curation and analysis operators, because 1) derived objects including visualizations are automatically refreshed if the object they depend on is updated, 2) any past transformation can be modified, and 3) conflicts caused by refresh are detected automatically. Versions of relations in PVD do not have to be materialized giving a system implementing PVD a wide range of options for optimization.

References

[1] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5 (10):968–979, 2012.

[2] B. Arab, D. Gawlick, V. Radhakrishnan, H. Guo, and B. Glavic. A generic provenance middleware for database queries, updates, and transactions. In *TaPP*, 2014.

[3] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. Parameswaran. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *PVLDB*, 8(12):1346–1357, 2015.

[4] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[5] S. B. Davidson, S. Cohen-Boulakia, A. Eyal, B. Ludäscher, T. McPhillips, S. Bowers, and J. Freire. Provenance in Scientific Workflow Systems. *IEEE Data Engineering Bulletin*, 32(4):44–50, 2007.

[6] J. Freire, C. T. Silva, S. P. Callahan, E. Santos, C. E. Scheidegger, and H. T. Vo. Managing rapidly-evolving scientific workflows. In *Provenance and Annotation of Data*, pages 10–18. Springer, 2006.

[7] Y. Yang, N. Meneghetti, R. Fehling, Z. H. Liu, and O. Kennedy. Lenses: an on-demand approach to etl. *PVLDB*, 8(12):1578–1589, 2015.

```
CREATE VISUALIZATION SUCCESSPLOT
STYLE BARCHART
PLOT FROM SR
WITH X = TREATMENT AND Y = SUCCESSRATE;
```

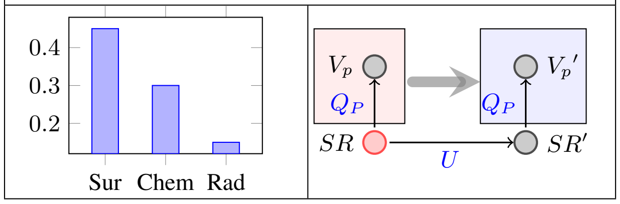

Figure 6: Mock user interface for a PVD system. The user can execute transformations and create visualizations that are automatically refreshed whenever the user modifies a relation. Additionally, the user can explore the VVG graph for her workspace and execute operations such as replacing a past transformation (as decribed in Section 2).

A. Our Vision for a PVD User Interface

Figure 6 shows a mock GUI for a PVD system. The main interface is a notebook where the user can document her operations and insert new visualizations and data curation operations by writing code. The system would allow the user to browse the VVG for her workspace and this would also be the starting point for modifying past transformations and conflict resolution. In this example, Alice has created a plot on top of her analysis result (relation SR) that shows a bar chart for the success rates of treatments. In the VVG, this plot is represented as a derived node V_p that is refreshed whenever the input relation SR is updated. For instance, if Alice updates SR using an update operation U, a new version V_p' of the visualization is created based on the updated relation SR' and the plot in the GUI is modified accordingly. Note that in such a user interface not all object versions described by the VVG are visible to a user. For instance, in the example shown in Figure 6 only V_p' is visible to Alice whereas the content of SR' is currently not exposed. As explained further in the next section, a PVD system can exploit this observation by delaying materialization of a relation version in a VVG until the content of this relation is exposed through the user interface.

B. Materialization, Reenactment, and Compression

B.1 Materialization and Composition of Transformations

As mentioned before, a VVG G keeps track of how relations (different versions of the same relation or separate relations) are derived from each other through the application of transformations and only some of these relation versions may actually be materialized at a certain point in time. This can be modelled as a set M_G of nodes in the VVG G that are materialized. Based on such a set we can determine whether the relation version corresponding to a non-materialized node R can be materialized based on the currently materialized relations in M_G. Any relation version S can be materialized as long as there exists a path in the graph connecting a node $R \in M_G$ to relation S. To materialize S we would apply a composition of the transformations on that path to R. Supporting lazy materialization of relation versions is important for our approach, because it enables a system implementing Provenance-aware Versioned Dataworkspaces to 1) avoid materializing relation versions that are not needed and 2) to choose an optimal way of materializing a relation version, (e.g., optimal could mean choosing the composed transformation with the minimal expected runtime). In particular, eager materialization of automatically refreshed derived relations may be expensive if relations they depend upon are updated frequently. Thus, lazy schemes are essential for efficiently implementing PVDs.

EXAMPLE 5. *Reconsider Figure 3 and assume that $M_G = \{R\}$ (nodes in M_G are highlighted in red and have a bold outline). Relation $V_2{}'$ from Figure 3 can be computed from R by composing the operations on the path from R to $V_2{}'$ and applying them to the start point of the path. In this case the composed operation would be $Q_2 \circ Q_1 \circ U(R)$.*

B.2 Nondestructive and Composable Updates Using Reenactment

Note that in Section B.1 we did assume that relation versions are immutable and that transformations can be composed. That is, an application of a transformation creates a new relation instead of destructively modifying the input relation. For instance, for relational updates this can be achieved by applying the declarative replay technique (*reenactment*) we have developed in [2] where sequences of updates are expressed as queries that return the updated state of a relation. Nonetheless, destructive modification, e.g., standard relational update semantics, can be also be incorporated in our model. These types of modifications create a new version of a relation by destructively modifying the previous materialized version. In the VVG this would be reflected by marking the input as non-materialized and marking the output of the transformation as materialized.

EXAMPLE 6. *Continuing with Example 5 we would like to execute the composed transformation $Q_2 \circ Q_1 \circ U(R)$ to materialize $V_2{}'$. Using a regular relational database we would have to execute this transformation in two steps: running the update U and then executing a query $Q_2 \circ Q_1$ over the updated relation. This would destructively modify R. After this execution, relation version R is no longer available. In fact, it has been replaced by R'. In addition $V_2{}'$ is now materialized. That is, $M_G = \{R', V_2{}'\}$. Using reenactment an update U can be transformed into a so-called reenactment query $\mathbb{R}(U)$ which returns the updated version of R. Using this approach we could execute $Q_2 \circ Q_1 \circ \mathbb{R}(U)(R)$ to materialize $V_2{}'$ without modifying R. In the resulting graph, the set of materialized relation versions is $M_G = \{R, V_2{}'\}$. We are not claiming that either of these options is superior to each other, but would like to note that they represent different trade-offs. Even for this simple example there exists additional plans for materializing $V_2{}'$, e.g., by materializing $V_1{}'$ and then executing Q_2 over $V_1{}'$.*

B.3 Compressing Graphs Through Composition of Transformations

Our model also allows for compression of the graph structure at the cost of loosing details about transformation sequences. For instance, a path $R_1 \xrightarrow{T_1} R_2 \xrightarrow{T_2} R_3$ can be replaced with $R_1 \xrightarrow{T_2 \circ T_1} R_3$. Applications of this rule enable parts of the graph to be compressed at the cost of loosing details about transformations. It would be interesting to study such compression techniques in more depth. Another interesting research direction is to incorporate incremental view maintenance techniques. For example, for the VVG in Figure 3 we could apply such techniques to determine how the U propagates to V_1 which would give us a direct transformation from V_1 to $V_1{}'$. Thus, incremental view maintenance can provide us with additional options for materializing relations.