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 (PVDs), our vision of a sandboxed environment where 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.

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. 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 workflow is not supported. In theory, provenance-aware databases could also help an analyst to track uncertainty and potential errors in her data, but would require the user to manually expose the uncertainty in data curation operations. In short, provenance can be a critical tool for enabling data exploration, but current systems for are lacking in several respects:

- **Regret-free exploration** - The user should be able to operate in a sandboxed environment where she is able to change past decisions and data derived based on these decisions should be automatically refreshed. Both base data and derived data is versioned. Conflicts created by automatic refresh are detected, but do not prevent creation of new versions.

- **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 automatically merged. For instance, a user may want to update an analysis based on recent changes to a database which requires merging these changes into the analysis pipeline.

- **Uncertainty as a first-class concept** - Whenever an operation introduce uncertainty, the system should keep track of this uncertainty, propagate it through operations over the uncertain data, and be able to explain whether an output is uncertain and how the uncertainty affects an analysis result.

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 [10]. However, the nonlinearity arising in data exploration tasks can not be easily handled. To the best of our knowledge, support for the versioning functionality that would be required to capture non-linear provenance does not exist in current systems. 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 provide the functionality mentioned above.

2. Running Example

Alice is an analyst at a hospital who wants to build an analysis workflow to determine the success rate of different treatments for
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 the Mimir system to understand how the uncertainty exposed by the lens affects her result and focus her curation 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 quite 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. Alice’s 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 introduce in the next section supports both requirements. Data curation and analysis over large data sets can be hindered by the cost of rerunning steps in a pipeline. The PVD approach we present can lazily materialize relation versions once they are needed (e.g., to refresh a visualization shown to the analyst).

3. 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-exist) 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 hypergraph 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., virtual version graphs instead of hyper-graphs.
3.1 Types of Transformations

We distinguish between two fundamental types of transformations: 1) Transformations such as view creation that create a new relation; 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 depending 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. 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 3 shows an encoding of Alice’s pipeline construction steps as a VVG (only one modification step to the lens is shown). Ignore node colors for now. Solid black edges are derivation edges and red 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,3 producing relation T which in turn is cleaned up using lens Lmissing to produce Tfixed. Finally, Q1 is run over this relation to produce relation SR storing the success rate of chancer treatments. Changing Q,4 (the query extracting relation T from JSON document J) into Q,4′ causes new versions of the relations derived from T to be created (these are relations T′, Tfixed and SR′). When Alice changes the parameters of the lens afterwards (L,missing), then new relations versions SR′ and Tfixed′ are created. Note how versions of the same relation are connected via version edges.

3.2 Automatic Updates 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 4 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: V1 created by query Q1, and V2 derived from V1 by running Q2. Once the update is applied, new versions V1′ and V2′ of these relations are created, are connected to the new version R′ and linked back to V1 and V2, respectively (shown on the right). In the figure, we have enclosed the subgraph derived from R in a red rectangle and its copy in a shaded rectangle.

3.3 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,2 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 5 shows a VVG for a variation of the previous example. There is a relation R plus two views defined based on R (V1 and V2). Two update operations U1 and U2 have been applied to R creating new versions of R and its dependent relations (the views). Assume the user determines that transformation U1 is buggy and should be modified. If the modified transformation U,1′ is applied to R, then all relations derived from the R′ (enclosed in a rectangle in the figure) the original output of U2 have to be updated too. Thus, a copy of this subgraph is created and connected to the new version of R. We use “...” 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 V1′ which was copied from V1 and is recorded to originate from V1, the origin of V1′.

3.4 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 V1 derived by a query Q1: SELECT name, age FROM R. If a user updates the relation R by deleting the age attribute, then the automatically created version V1′ of the view V1 derived by applying Q1 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 approach 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 viola-
tions 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 6 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.

3.5 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 and apply conflict detection and resolution strategies such as the ones mentioned in the previous section in addition to considering traditional types of conflicts such as conflicting updates to the same tuple.

4. 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 sandboxed environment in which the user can explore 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 compute from relations. They 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.

4.1 PVD Building Blocks

We argue that recent provenance-related techniques introduced by the authors - namely on-demand data curation through lenses [14] as well as provenance tracking and reenactment for updates [4, 5] - provides a solid foundation for implementing PVD. Lenses [14] 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 [4, 5] 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 and the ability to track provenance through such a composed transformations.

4.2 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 the workspaces to Big Data dimensions. 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. It will be critical to study the types of conflicts that can arise in VVGs given a class of allowed transformations, develop efficient techniques for detecting such conflicts, and study methods for conflict resolution. Last but not least, we would like to investigate feasible merge strategies for VVGs.

5. Related Work

The version graph model we have introduced is inspired by how version control systems such as Git (https://git-scm.com) treat versions. These systems typically allow 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 do not record the transformations that were used to derive a version $X$ from a previous version $Y$. Thus, they lack the virtualization capabilities (recompute a version instead of materializing it) that are an inherent feature in PVD. Furthermore, there is no notion of derived objects and automatic refresh of such objects. DataHub [6] is a system for dataset storage that is also inspired by version control systems and we share the vision underlying DataHub that the choice between materialization and re-computation can be turned into an optimization problem. In contrast to DataHub, we envision to apply declarative replay techniques and 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 and uncertainty. Since we expect many of the transformations in a VVG to be declarative, it will be important to study applicability of incremental view maintenance techniques [12, 8, 2] to speed up the materialization of new versions of relations when a relation they depend on is updated. Of particular interest are the techniques of DBToaster [13, 3] that enable large views to be refreshed efficiently, and techniques on Self Adjusting Computation [1]. Our approach to provenance management is informed by the large body of work on workflow [9, 10] and database provenance [7]. In contrast to typical approaches for workflow provenance, in our model there is no direct notion of a workflow - pipelines are build incrementally and data is automatically kept up to date. Our model shares with [10] 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 Perm [11] and GProM [4, 5].

6. 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.
A. Our Vision for a PVD User Interface

Figure 7 shows a mock GUI interface of a PVD system. The main interface is a notebook where the user can document her operations and insert new visualizations 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 conflict resolution. In this example, Alice has created a plot on top

B. Materialization of VVGs, Reenactment, and Compression

B.1 Materialization and Composition of Transformations

As mentioned before, a VVG keeps track of how relations (different versions of the same relation or separate relations) are de-
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 \( R \) resulting in \( M_G = \{ R, V_2' \} \). 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 4 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.