Enhancing Web-based Analytics Applications through Provenance

Akhilesh Camisetty, Chaitanya Chandurkar, Maoyuan Sun, and David Koop

Abstract—Visual analytics systems continue to integrate new technologies and leverage modern environments for exploration and collaboration, making tools and techniques available to a wide audience through web browsers. Many of these systems have been developed with rich interactions, offering users the opportunity to examine details and explore hypotheses that have not been directly encoded by a designer. Understanding is enhanced when users can replay and revisit the steps in the sensemaking process, and in collaborative settings, it is especially important to be able to review not only the current state but also what decisions were made along the way. Unfortunately, many web-based systems lack the ability to capture such reasoning, and the path to a result is transparent, forgotten when a user moves to a new view. This paper explores the requirements to augment existing client-side web applications with support for capturing, reviewing, sharing, and reusing steps in the reasoning process. Furthermore, it considers situations where decisions are made with streaming data, and the insights gained from revisiting those choices when more data is available. It presents a proof of concept, the Shareable Interactive Manipulation Provenance framework (SIMProv.js), that addresses these requirements in a modern, client-side JavaScript library, and describes how it can be integrated with existing frameworks.

Index Terms—Collaboration, provenance, streaming data, history, web.

1 Introduction

With the emergence of web libraries for interactive visual exploration and analysis of data [5, 12, 14, 16, 48, 55], users are able to analyze and explore a wide variety of data from their web browsers. Support for animated transitions and event handling gives users the ability to interactively filter data, change views, or dynamically steer computational processes. As these frameworks and tools continue to evolve, there has been a tradeoff between the fluidity offered by client-side frameworks and the complexity of the applications. Specifically, where server-side applications capture information as pages are navigated and offer mechanisms to return to previous steps or settings, client-side applications offer little infrastructure to recall or persist past actions. This either limits the complexity of the visualization or leads to user frustration. However, server-side solutions require servers to provide storage and interfaces and users to trust servers with their information. In addition, it is usually more difficult to customize the content and granularity of information captured, and the data tends to be less connected to the actions a user makes. We have explored the space of solutions for adding provenance to support reflection and collaboration, and developed a prototype JavaScript framework (SIMProv.js) to show how client-controlled provenance can augment web applications.

Provenance—data describing how a result was achieved—has been shown to be useful for a variety of tasks [20, 30, 49, 52]. It is important in documenting work accomplished, engendering trust in results, and supporting reproducibility of past results. However, most work on capturing computational provenance, which is focused on the history of the design and execution of tasks, has concerned static computations. In practice, data exploration and analysis involves dynamic interaction with the computed results, and those actions may trigger computational updates usually presented to the user immediately. In addition, the design of the computation is an important step to document, not only for accuracy, but also to see what changes were developed as the computation evolved [21]. Finally, the goals and decisions of a user, as they digest and interact with visual displays are an important piece in understanding past and current analyses [46], often enhanced by user-provided annotations.

With the dominance of the web, it is becoming more common for users to use and collaborate on web analytics applications. PolyChrome [3] is a solution that seeks to facilitate this using general web-standard techniques and focuses on fine-grained events; we seek coarser actions to enable users to navigate and reflect on their past work. Provenance, generated as a user performs analysis steps, can be useful in helping a user understand another’s work when users are working asynchronously and a handoff [60] occurs. While the promise of provenance is well-recognized, user privacy has also become a part of any conversation about web-based tools and applications. Sites that store such information on servers must make users aware of such details, and any project that involves sensitive information often cannot function in a hosted environment unless those servers are controlled by the investigator’s organization or compliant with specific regulations and security protocols. Thus, we wish to ensure that all provenance data is maintained on a user’s machine leveraging browser storage like IndexedDB. Ideally, applications may be delivered and executed over the web via the browser, but all history is stored locally—no provenance travels back to the server without user permission.

Another emerging area where provenance can help is streaming data. As visual analytics deals with updating data, web applications are well-positioned to link with data streams, but we need to consider how the provenance and the streaming data match (or don’t). It is useful to both recall or revisit the exact state of the application and data when reviewing provenance, but it can also be enlightening to see the application in the same state but the data updated with more or more recent data. Thus, it makes sense to support both of these choices when possible.

In this paper, we describe a framework that enables developers to add provenance features to their interactive web applications and users to improve their analyses through these features. Key contributions include:

- Desiderata and strategies for capturing, using, and sharing provenance in web applications.
- A framework that supports both private, user-controlled provenance and asynchronous and synchronous collaboration.
- Strategies for capturing and revisiting provenance when web analysis applications use streaming data.
- A reflection on decisions made in evolving the framework to better embrace modern web technologies and support real-time analytics.
- A developer-focused perspective that considers the barriers to the adoption of provenance in web applications.

Please see the supplemental video for a better understanding of how these aspects have been integrated into a prototype system.

• Akhilesh Camisetty, Chaitanya Chandurkar, Maoyuan Sun, and David Koop are with UMass Dartmouth. E-mail: {acamisetty, cchandurkar, smaoyuan, dkoop}@umassd.edu.
2 Related Work

Ragan et al.'s survey contains an organizational framework for provenance in visualization and data analysis [49]. In that framework, our work can be characterized as focusing on the interaction and visualization types with the direct purposes of replication, action recovery, and collaboration. Note that the types of provenance may also be categorized more generally. Computational provenance focuses on the lineage of a specification or result regardless of who or what impacted each step [20]. Analytic provenance focuses on a user's actions and purpose [46]. One may try to infer from computational provenance what a user was trying to do, and from a high-level understanding, what data was used or computations run. This intersection is most salient where a user action leads directly to a computational action (or vice versa). Gotz and Zhou organize provenance based on semantic richness with high-level tasks (e.g., identifying insights) at one end and low-level events (e.g., mouse clicks) at the other [24]. Our work focuses on the action tier, capturing exploration actions and allowing users to perform insight actions like annotation and meta actions like undo and redo.

Provenance Structure & Display A challenge with provenance is effectively supporting visual and interactive exploration [30]. GRASPARD introduced the history tree as a way to organize snapshots of the steps scientists take in an investigation [7], and Derthick and Ross showed how such a tree could be used to efficiently visit past states [17]. VisTrails introduced change-based provenance as a way to encode states of the history (or version) tree [21]. Unlike version control software like Git [22], change-based provenance relies on prescriptive changes, designated when an action occurs rather than when modifications are committed. Network visualizations like Image Graphs [40] and ExPlates [56] help organize related states and can also display information about the mutation of data of visual representation. Thumbnails can be a key component in displaying history [4, 29], and studies have shown they can be effective even at low resolutions [35, 54]. Furthermore, thumbnails may be cropped or zoomed to focus on particular changes that help users identify types of actions [37].

Collaboration in Visual Analytics The visualization community has acknowledged the importance of collaboration, and there has been significant work to classify the different contexts and outline associated challenges [28, 33, 51]. Modes of collaboration are divided according to both space and time. Co-located users will collaborate with different strategies than those who are distributed in different locations. Synchronous collaboration has users working at the same time while asynchronous collaboration allows users to work more independently [33]. An important challenge in asynchronous collaborative sensemaking is the handoff problem where one user’s analysis is picked up by another user who must understand the work that has already been done and where to go next, a type of knowledge transfer [60]. In synchronous collaboration, the problem of concurrent updates is often addressed via operational transformation which can utilize timestamps and buffers to keep clients in the same state [18, 45].

Collaboration Infrastructure and Interfaces Collaborative analysis requires users be aware of others’ work as well as their own; interfaces need to facilitate user access and awareness. Using server infrastructure, VisPortal persisted visualizations and notes across sessions and allowed users to build off each other’s work [34]. Hajizadeh et al. investigated views where a collaborator’s selections could be overlaid over a user’s visualization [27]. The CLIP tool showed not only that analysis benefit from externalizations of notes and discoveries, but also introduced partial and full merging strategies for integrating collaborators’ work into a user’s workspace [41]. A network or cluster view that organizes insights juxtaposed with a timeline has been shown to aid users in synchronous [11] and asynchronous [10] collaboration scenarios. McGrath et al. investigated the branching and merging in their protocol for co-located, synchronous collaboration, providing methods for users to switch between coupled and decoupled exploration as an analysis proceeds [42]. PolyChrome is a native web framework for synchronous collaboration, and pushes DOM events between browsers [3]; our work is similar but targets the action tier to provide a coarser provenance.

Web Provenance Provenance on the web has been examined in the context of linked data or the lineage of information displayed on a web page [25, 26]. It also includes analytic provenance tasks, tracking actions across different sites for sensemaking [44]. Because of the inter-connected nature of the web, various data sources may be combined in an interactive system. Thus, an insight gained during interaction depends on the data sources, the integration and transformation of that data, and the actions of a given user. Our focus is on web applications where a user interacts with a single page that updates interactively rather than examining the browsing history of a user. However, we expect that provenance schemes that work across pages could be coupled with our focused provenance.

Using Provenance The ability to use the captured provenance is important, and visualizations of history help users navigate and explore past work while algorithms might also suggest directions for further analysis. Work in this space includes explicitly showing dimension coverage during collaborative analysis [50] and displaying personal interaction histories via direct encoding [19]. Similarly, scented widgets help guide users to explore certain parts of the visualization [57]. Dubek and Callan show how building a grammar-based model using interaction history might help suggest or correct directions during analysis [15]. REACT uses captured history to generate personalized action recommendations for query changes [43], and EVLIN extends the idea to a richer provenance model and also generates visualization recommendations [38].

Web App Debugging There are a variety of tools to help replay or debug web applications. Tools like Unravel seek to help developers better understand or reverse engineer websites [31]. Timelapse provides an interpreter that can be embedded with modern web browsers to capture detailed debugging information [8]. Visualization is also useful for understanding asynchronous interactions [1]. One of our goals is to keep users in the same environment they are used to, augmenting it instead of requiring users to use special software or tools when browsing the web. Other tools, like Vega’s visual debugger [32], provide targeted debugging for specific libraries.

Logging Mechanisms Other solutions exist for logging actions as users interact with a web application and run in a standard browser. Google Analytics is a popular framework that allows developers to track user page views and developer-identified events through a JavaScript library [23]. In such frameworks, data is automatically sent to and stored on servers where it can be analyzed by users in a dashboard interface. In addition, it is not data a user can use to help recall their exploration. Other solutions, like mimic.js, provide fine-grained data about interaction events [6]. Again, while these solutions allow for post-hoc analyses based on different provenance data, we seek solutions to help users as they work by providing external memory for past states and analyses.

Libraries Methods to store and recall past state range from fragment encoding to libraries that seek to provide undo and redo capabilities equivalent to desktop applications on the web. Web developers may encode state in URLs with fragment identifiers; the fragment identifier defines parameters that impact the current state of the page. Client-side web storage (IndexedDB, WebGL) provides larger storage for such data. Libraries that provide undo and redo capabilities (e.g., [2]) align with one of the core goals that users are able to use the recorded provenance immediately. However, unlike logging or provenance solutions, undo stacks are often not serializable, and they do not support branching or jumping between states.

3 Design Desiderata

Interactive web applications have become an important medium for visual analytics work. Many tools are being developed for web browsers because they can support users in many different settings. At the same time, rich toolkits and more libraries have made it possible to develop more complex applications. As this trend continues, it is important that we support analysts who are using these tools. One important aspect of this support is to augment the reasoning process by freeing users from
having to remember every step of their analysis. For example, Fig. 1 shows three thumbnails of the many states an analyst might encounter when comparing the performance of two motorcycle racers. These thumbnails provide markers in a user’s sensemaking process.

While provenance has been shown to have value in a variety of settings and for a variety of purposes [49], there has been less work on designing web frameworks to support the analysis process. Clearly, many of the principles from desktop applications or workflows can be leveraged on the web, but the web also presents its own challenges and opportunities. Among these, collaboration is an important benefit that the web offers. Instead of users working alone, networked environments open many opportunities for analysts to discuss insights and iterate with others on new ideas. However, the web also exacerbates issues that exist in desktop environments. Specifically, security and privacy are of paramount importance especially in a sensitive analysis, and as applications receive and send data, it can be difficult to know what information is being exposed. With the frantic pace of web development, the longevity of a web application can suffer due to libraries going unsupported or rapid changes in languages. Our goal has been to develop a framework that supports provenance for visual analysis in the web environment, and we have identified desiderata both by examining provenance in other contexts as well as current web applications and frameworks.

**Reflection** We wish to support *self-reflection at different time scales*. An important feature for reflection is the ability to revisit past states, including states that were just created, as part of an analysis. The web offers many features, but undo and redo are not standard features outside of text entry fields. When these features are offered, it is often as checkpoints that are stored by the server. A second scale is revisiting work from hours or perhaps days ago. Here, the goal may be to pick up where one left off or to return to a known base for new exploration. Finally, returning to archived provenance can aid a user who months or years later needs to recall how a particular result was achieved. Here, it may be more instructive to step through multiple states to examine the process used rather than simply locating the final result.

**Collaboration** We wish to support *collaboration at different levels*. While self-reflection is important, collaboration has become a focus in today’s interconnected world. The web allows work to be posted for anyone to examine, but in the context of analysis, it may be more likely for users to share with a smaller group or even just one other person. For smaller, more controlled environments, a user should be able to control the data, storing it locally rather than posting it on a server. However, collaboration at scale brings about many interesting opportunities. If many users are analyzing a specific problem, all within the same application, we can start to build collective intelligence in a more automated fashion, identifying common paths of inquiry as well as unique results. Instead of users all beginning at the same place and performing similar first steps, it is possible for them to jump in at existing waypoints and extend those. Another aspect of collaboration is whether it is shared or independent. In other words, should one user’s work impact all others, or should users simply be notified of each others’ progress? Shared collaboration produces a single result and works well when different aspects can be analyzed at the same time, but if users are constantly getting in each other’s way, this can be problematic. Collaboration with independent views is usually enabled by notifications of changes that a user may choose to examine and perhaps integrate, but that user may also continue to work without the same type of interruption that active shared collaboration may introduce.

**Streaming Data** Much analysis is still carried out on static datasets, but with tasks that demand immediate examination even if the data has not all loaded, or on datasets that are too large to open at once, or when current data is more important that past data, we need to support analysis on changing data. Streaming data includes data being generated in real-time, but data that is loaded incrementally in batches is currently a more likely use case. Here, revisiting a past state is fraught with issues. The data may not be available any longer, or an analysis on past data may not make sense when examining more current data. We seek a provenance framework to allow past states to be recalled in a manner that is either consistent with past data or can be updated for current data.

**Interface Features** Users need interfaces where navigation is featured to explore and make use of provenance. These interfaces should support common tasks like undo and redo but also allow jumps back to known states. To permit such jumps, it is important that one can identify states, and this often requires some description or depiction of that state. At the same time, understanding the relationship of one state to another is important. Were two results derived from a similar waypoint; was another developed at the same time?

**Recording Insights** Users need to be able to provide their own insights. Clearly, if a machine was able to perform all of the tasks involved in sensemaking, there would be no need for humans in the loop, but this is not the current situation. Instead, we need to make sure that users can relate their own insights to the structures and data automatically captured. If a particular state is important, allow the user to tag and comment on it.

**Application Integration** We want any provenance framework to be general enough to integrate with different styles of applications. Of course, the time involved in adding collaborative provenance capabilities can be prohibitive, but we believe that the core requirements are identifying the actions or states that should be captured and being able to serialize and materialize them. In many cases, the visual analytics application does not generate significant state with respect to the underlying data or analysis. In other words, because a user drives the changes, the provenance is purely user-generated which means that this data need not grow as significantly as the provenance of a distributed computation that is capturing each function call and parameter setting.

**Storage & Running Time** Another important consideration is the minimization of time and space constraints needed for provenance. While fully client-side web applications have not always been used for involved analyses, this may be partially due to the need for tools...
that help users keep track of their progress and process. As these applications become more complex, they may generate more complex provenance data that consumes more space and takes more time to materialize. Here, we can examine techniques that use storage to maintain checkpoints that are faster to load than replaying changes.

4 SOLUTIONS

Now, we seek to address these desiderata and focus on some ideas that extend current practices. For streaming data, we propose two modes for revisiting and replaying provenance information. We also explore using the same provenance information to support both asynchronous and synchronous collaboration. We examine strategies to improve the efficiency of application integration as well as to improve the efficiency for users via a storage-running time tradeoff.

4.1 Interface

An important aspect of the interface is that it supports not only capture but allows users to explore the data as it is captured. This becomes more important when streaming data and collaboration are involved as more variables complicate a user’s mental map of the analysis. Furthermore, the interface should serve as an additional overlay or integration with existing web applications, not a replacement or container for them. In this sense, the interface should be minimal but expand to provide helpful overviews of the provenance including tables of past actions and trees or relationships between states.

In order to support reflection, we see thumbnails as an important ingredient to aid users looking to discover past work [35, 54]. However, a thumbnail that shows the entire view may be of limited utility when changes are relegated to a particular region of the page. Thus, we might allow developer configuration of how thumbnails are captured from a particular application. Specifically, there may be three types of thumbnails: overview, action-focused, and result-focused. An overview thumbnail shows the entire visualization in its current state. Action-focused thumbnails, then, focus on what the user changed (e.g. a change in selection) while result-focused thumbnails show the region of the visualization that was updated as the result of the action.

4.2 Streaming Data

A streaming data application is as an application where the data is not loaded in a single pass [13]. Importantly, this means that not all of the data has been loaded when the interface is first shown, and the interface continues to update as more data is ingested. Note that streaming data applications may be able to access all of the data that eventually comes in, or they may choose to purge the raw data as filtering or aggregation is completed. In some cases, the data is too big to all be stored, and in other cases, the data becomes stale and thus less useful for online analytics.

Crouser et al. identify some challenges that visual analytics faces with streaming data, including the importance of recognizing and understanding changes in insights and decisions as the data changes [13]. Because the data keeps changing as the user analyzes it, that user can quickly face cognitive overload as they try to keep track of and compare new features with views in the past. Because the user is engaged in analysis, they are updating views while the streaming data is also enacting change on the same views. To this end, support for recalling past views as they were originally seen is important. To support this, we can associate an order and identifier to the chunks of data as they come in and bind the most recent chunk identifier in each provenance action as it is captured.

If an application has the entire dataset available and can recall past states of the analyses, analysts also have an interesting opportunity to compare that view from the past with an updated version that reflects the current reservoir of data. The past view is the persistent, set-in-stone snapshot of the analysis when decisions were made, but an updating mode changes as new data is made available, as if the user had stopped analysis and watched as the application’s views updated as new data came in (see Fig. 2 for an example). However, this non-persistent mode should allow the user to revisit any previous state and apply the latest data to it. This enables reflection on whether the additional, perhaps more recent, data changes the analyst’s perspective on any past insights.

4.3 Collaboration

Our goal is to facilitate both asynchronous and synchronous collaboration through the same mechanism—provenance. Synchronous collaboration can take advantage of a wide swath of conferencing tools and screen-sharing software, while asynchronous collaboration can be supported through various serializations, but we believe that standardizing the mechanism will allow users to transition between modes more easily. Shared views can be helpful in initial analyses, but independent work that can later be examined could allow a greater diversity of ideas. Because provenance is stored in the same way for both modes, it is easy to switch between the modes. Fig. 3 shows how two users might

<table>
<thead>
<tr>
<th>Name</th>
<th>Collaboration</th>
<th>Sharing/Privacy</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsyncUser</td>
<td>Asynchronous</td>
<td>User-Controlled</td>
<td>File</td>
</tr>
<tr>
<td>AsyncPub</td>
<td>Asynchronous</td>
<td>Public</td>
<td>Web File</td>
</tr>
<tr>
<td>SyncPriv</td>
<td>Synchronous</td>
<td>Private</td>
<td>Network Ad-hoc</td>
</tr>
<tr>
<td>SyncOrg</td>
<td>Synchronous</td>
<td>Organization</td>
<td>Network Hosted</td>
</tr>
</tbody>
</table>

Table 1: Different collaboration strategies that can be supported in a web application.
especially with the proliferation of personal information stored and transferred about the web, we must be wary about how provenance is shared and allow users to maintain privacy when desired [59]. To support this, we seek to capture all information locally; provenance should not travel outside a user’s own machine without their consent. This is possible through the use client-side browser storage like IndexedDB; the application and dependent libraries can be loaded across the web, but the captured provenance stays in a local database. To support collaboration, we allow this data to be exported in JSON format to a file or directly as a web-hosted file that can be immediately shared. Table 1 shows different strategies and technologies to enable collaboration and what the privacy characteristics are.

A final consideration is the join between streaming data and collaboration. As streaming data presents challenges beyond standard visual analytics, collaboration may be useful in this context. Specifically, if one user has done analysis on a subset of the data, and another user revisits that analysis in the non-persistent mode, the second user may, like the first user, be able to gain different insights. More interestingly, if the data streaming is not uniform, different users may see views of the data, allowing a collaboration to more quickly identify insights. In the synchronous case, this becomes problematic because the provenance needs to store the state of the stream at each action, but it could be possible to collaborate in a non-persistent mode where users see different views of the data but adhere to the same action history.

4.4 Efficient Storage and Access

The storage of provenance in a version tree builds on the concepts introduced in VisTrails including change-based provenance [21]. We extend change-based actions to normalize the data involved in a change as it may be used in multiple changes (e.g. a forward and inverse change). In addition, we distinguish changes that update the visualization from those that update state. Finally, the materialization algorithm is extended to consider the costs of each change when evaluating potential paths.

4.4.1 Definitions

A version tree is tree where each node is a version—a state—of the entity being tracked by the tree, and each edge signifies that a child node was derived from its parent node. In contrast to undo stacks, version trees persist all versions, adding branches where multiple versions have been derived from the same starting state. An edge contains information about how a version is transformed. Each change is a function that updates the state, while a version is a transformation of the state. Ideally, the change data specifies only the information that changed from one version to the other, but it could, for the simplicity of implementation but at the cost of efficiency, encode the entire state. Inverse changes support efficient undo capabilities, encoding a transformation up the tree. The root node of the version tree must have an associated reset method that resets an application to its original state; using this and the sequence of changes along a path from the root, we can derive any other version.

A change is a specific type of transformation that mutates one version into another. The focus is on the transformation—how the state is updated—rather than exactly how a user enacted it. Formally, given a version \( v \) and a change \( c(v) \) should produce a new valid version \( v' \). Where a change specifies the type of transformation, the change data specifies the data and parameters that a change uses to construct a derivative version. Ideally, the change data and changes allow efficient encoding of the transformation between states, but we aim to support the range of possibilities because inefficient provenance can often still be a good starting point. With (forward) changes, going backward to previous states requires resetting the visualization to its starting state and replaying actions until the desired state is reached. Inverse changes are changes that go from a child version to a parent version—the “undo” direction. Note that in some cases, we can share change data across changes, even more so with inverse actions since their data often overlaps with a corresponding forward action.

While encoding the entire state of an application as the change data for every change may be inefficient, having such information would allow us to skip replaying entire change sequences in order to get to a particular version. Instead, we simply update the application’s state according to the data stored with that version. To take advantage of this shortcut more generally, we use a checkpoint for a version that encodes the full state for that version. More concretely, given a checkpoint function \( C \) and checkpoint data \( d_c \), for a version \( v \), \( C(d_c) \), no matter what the current version is, will update the visualization to the version \( v \). We also allow developer-defined checkpoint rules that specify the conditions for creating a checkpoint. Each rule is passed the version tree and the current node, and returns a boolean indicating whether a checkpoint should be persisted.

Because checkpoints essentially shadow a web application’s state, we can also consider changes that act on that shadow state instead

<table>
<thead>
<tr>
<th>Change</th>
<th>A transformation from one version to another</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateChange</td>
<td>A Change that specifies the transformation of the serialized application state of one version to the state of another</td>
</tr>
<tr>
<td>ChangeData</td>
<td>Data referenced by one or more Changes</td>
</tr>
<tr>
<td>Checkpoint</td>
<td>Serialization of application state for a version</td>
</tr>
<tr>
<td>Action</td>
<td>An encapsulation of Changes (change, inverse, stateChange, stateInverse), corresponding ChangeData object(s), an optional Checkpoint, and metadata (timestamp, thumbnail, etc.)</td>
</tr>
<tr>
<td>Trail</td>
<td>An encapsulation of all Actions and the corresponding version tree</td>
</tr>
</tbody>
</table>

Table 2: Terms related to the provenance representation
we happen to be in the version tree. Because we may have both forward changes and inverses, checkpoints and forward state changes, or checkpoints and inverse state changes, are available. Generally, implementing more features leads to more efficient switches between versions.

of updating the application directly. Specifically, we define a state change, \( c_s \), as a change that takes a checkpoint corresponding to a particular version \( v \) and returns a new checkpoint corresponding to another version \( v' \). Note that SIMProv.js allows users to define both a state change and an inverse state change in addition to a non-state change and inverse. Often, updating shadow state can be more efficient because there are no interface updates that need to be processed until the final state is loaded. In addition, state changes may be chunked [29], and these chunks compressed.

Finally, an action encapsulates all data associated with a node and the edge to its parent in version tree. It includes at least one change that specifies how the state represented by the parent node is transformed to the state represented by the given node. However, it may also include any of an inverse change, a state change, and an inverse state change. It may store an optional checkpoint that represents the state corresponding to the node. Importantly, changes may share state data. If the inverse of adding a particular filter is removing it, the specification of that value may serve as the change data for both the change and the inverse change. It may also serve as the change data for state and inverse state changes.

4.4.2 Materializing Versions

A key operation on version trees is the materialization of a specific version, transforming the application to that version's state from wherever we happen to be in the version tree. Because we may have both forward and inverse changes as well as the state-changes of both types, there are a number of potential paths to reach the desired state (see Fig. 4). At a minimum, we need a reset callback and forward changes to return to any version. Specifically, we can reset and then run all changes along the path from the root to the desired node (Path 1 in Fig. 4). With inverses, we add the ability to follow the inverses back to a common parent and then down to the desired node (Path 2 in Fig. 4). With only checkpoints and forward state changes (no inverses), we have the ability to transform the shadow state from the checkpoint to the desired node (Path 3 in Fig. 4). Finally, if we add inverse state changes, we can utilize nearby checkpoints even if they are not ancestors of the target node (Path 4 in Fig. 4). Note that even with state changes, inverses, and checkpoints, we still may find that returning to the root node may be an efficient solution.

The cost of each strategy depends on a number of factors and may also be impacted by the tradeoff between efficient materialization and storage costs. In order to determine the most efficient method to obtain a target version \( v' \) from the current version \( v \) via provenance, we must first consider various starting nodes and then the paths from those nodes to \( v' \). Besides the current node, the root node and checkpoints are also valid starting points. We use a breadth-first search from the target node to identify the nearest checkpoints. Due to inverses, a checkpoint may be descendant of \( v' \) in addition to an ancestor. Each checkpoint has an associated cost of materialization, and resetting the application to its initial state has a cost as well. Next, we calculate the path from each potential starting node to \( v' \) and check it for both "normal" change and state change sequences. First, we check that the necessary inverses exist. Recall that if there are no inverse changes defined, we may still materialize from the root node. Given a valid sequence of changes, we calculate the associated cost. Each change has an associated cost, and thus, each path has a total cost based on the sums of the change costs. The minimal cost path is selected from all possible starting nodes.

5 Prototype Frameworks

We have completed two prototypes of the proposed framework. The first prototype focused on showing that provenance could be captured and navigated in web applications via a plug-in style system. It had a thumbnail strip, gallery view, and annotation panel. It also supported undo and redo, and integrated a change-based model to track and replay provenance. The second prototype focused on support for modern web applications, featuring asynchronous calls and integration with IndexedDB storage so that provenance could be robustly saved and maintained even when a user navigated away from an application. Behind the scenes, we rewrote code to better delineate between the core features and the user interface. In addition, this prototype focused on the challenges of streaming data and real-time collaboration.

One goal of the SIMProv.js framework is to allow developers flexibility in capturing provenance information and users flexibility in persisting, sharing, and using it. Because a general approach for integrating provenance in interactive web applications cannot cover every potential situation, we provide developers with tools to facilitate provenance capture and let them designate both where and how provenance is captured. SIMProv.js provides features for persisting provenance, capturing thumbnails of the current state, and storing annotations. In addition, it provides capabilities for importing, reviewing, and replaying past provenance information. Since JavaScript is the language of the web, SIMProv.js is written in JavaScript, building on the growing stack of existing technologies and libraries.

5.1 Interface

While the ability to capture and persist provenance is clearly important, our goal for the framework is to make the provenance usable. This requires interfaces that permit navigation, browsing, and annotation. We provide a set of panels and widgets inline in the library so users can interact with them (along with proper style rules) into existing pages. The toolbar interface is a widget that can be relocated about the page. Fig. 5 shows the toolbar and the associated panels. The toolbar provides undo and redo buttons, an annotation panel, a scrollable thumbnail view, and a menu to access methods to store and import provenance data. It also provides access to a full-screen gallery that shows a table with detailed information about the versions and thumbnails (see Fig. 7) and a tree showing the relationships between different versions. In addition, the interface provides two submenus to facilitate streaming data and synchronous collaboration.

**Thumbnail Strip** Because thumbnails allow quick decisions about which versions to return to, we provide a thumbnail panel that has all of the versions along the current path (just like with undo and redo). A user can select a thumbnail to materialize the version associated with it. When hovering over a particular thumbnail, a tooltip displays amount of time that has passed since that version was created in a human-readable form (e.g., “a few seconds ago”) along with a string describing the change. For visualizations built entirely in canvas or a single svg, capturing these thumbnails is fairly straightforward. However, many applications use multiple graphical views and also encode data in HTML elements. Thus, we also need to be able to capture entire regions of the page. In addition, with transitions, we must often wait to capture the thumbnail until the transition has ended; sometimes this results in empty thumbnails if a user moves quickly. We use rasterizeHTML.js [9] as a helper library in these cases because it works to respect style rules at all levels. Indeed, even when trying to capture a simple svg, style rules that reference parent elements do not work correctly, leading to misrendered thumbnails.
5.2 Designing Capture

A key concern in creating useful provenance is having a design that captures meaningful changes at a reasonable granularity. Thus, a developer must identify the transformations that must be captured and the listeners or callbacks that can be utilized to do so. If using checkpoints and state changes, the developer needs to identify a schema for that state. Often, consulting the application’s own representations, if they exist, is useful here. Then, representations for the change data and changes must be defined. Identifying overlaps between changes and their inverses or state-based counterparts lead to more efficient representations. Note that the change data must be serializable regardless of whether it is being used for a state change.

5.3 Persistence and Collaboration

As a client-side framework, SIMProv.js stores its provenance in the browser’s local storage, and does not send information back to a server. This was a key change in our revised framework, as the persisting provenance used to require that the user export it from memory, but now provenance is written to browser storage after each action. Not only does this reduce the burden on the server, eliminating the need to store everyone’s provenance, but it also puts the user in control of data derived from their interactions. If a user wishes to remove provenance information, SIMProv.js provides a delete database method to clear out all provenance information. Note that we will need to alert the user if space in browser storage is running low. If a user wishes to keep the information private, SIMProv.js can archive it to a local disk as a JSON file for future review. However, users may also choose to share that file or directly upload their provenance to a server where their collaborators or perhaps the public can view it. Collaborators can also work synchronously using a serverless architecture where provenance is shared between clients as it is generated. For streaming data used in an application, it may be infeasible to store the data with the provenance. However, we can tag each provenance action with the timestamp of when they are connected again. During synchronous collaboration, this allows a user to dynamically configure whether they should revisit a state from the previous one and methods to take this data and enact the change to move between states. To facilitate faster speed at the cost of storage, a developer can also specify a checkpoint method that dumps all of the state along with a corresponding updateState method to rematerialize that state. Fig. 6 shows how a new action to support tracking changes in the tabs of a multi-tab analytics application was developed.

5.4 Navigating and Organizing Provenance

As described, the framework provides a thumbnail strip as well as a gallery that includes both a list view and tree view that with shared highlighting. A concern here is having too much information, and strategies to address this include undo-as-delete [29], chunking [37], and pruning [36]. In addition, the organization of the provenance can aid in identifying states [40, 56]. In the first prototype, we highlighted nodes in the thumbnail strip where branches existed, but the navigation proved difficult so it was removed. We do think there is a way to allow navigation from the strip in the future. The list view provides the
configuration = { ... 
  actions: { Tab: actionTab, ... } }

... 

function checkpoint() {
  ...
  state['tab'] = $('#uc-tab .active')[0].id.
  slice(4); }

function updateFromState() {
  ...
  if (state['tab']) {
    $('#btn' + state['tab']).click(); }

function createTabAction(tab, prevTab) {
  actionData = {
    forwardData: tab,
    inverseData: prevTab,
    type: 'Tab',
    inverseAction: 'Tab',
    information: 'Changed tab to ' + tab;
  } simprov.acquire(actionData);
}

function actionTab(state, action, isState) {
  data = action.actionData;
  if (isState) {
    state['tab'] = data.forwardData;
    return state;
  } else {
    $('#btn' + data.inverseData).click(); }
}

// use click events to create action
$('#btnExpenditures').click(function () {
  createTabAction('Expenditures', lastTab);
  lastTab = 'Expenditures';
});

Fig. 6: To add a new action to be captured, we need to edit methods that capture and update state, add new methods to create and enact the action, and register the action with simprov.

opportunity to sort states by various criteria including visual thumbnail similarity or state differences. The version tree can also be reorganized to facilitate exploration by similarity [36], but this impacts the ability to see derivations. Because of our goal in maintaining the relationship between views, we do not currently employ these strategies but plan to incorporate them in the future as we investigate the scalability of the framework.

6 CASE STUDIES

In order to evaluate the prototypes, we have focused on applications created using the dc.js framework which builds visualizations on top of the crossfilter library [16]. However, with the first version, we also explored how to use Vega to support provenance in the Polestar visualization design tool [58]. An important factor in these examples was that we tried to not only instrument a single application but also examine how multiple applications could be enabled by building a bridge from those libraries to SIMProv.js.

6.1 Crossfilter Explorations

When users can interact with a visualization, it allows them to explore their own ideas and scenarios in addition to the default settings or prescribed configurations. We examined three different applications created in dc.js that facilitate these types of explorations. The NASDAQ stocks example (shown in Fig. 2) allows a user to explore the performance of the stocks on that exchange over time. If that data is streamed in, examining only the data until 2000 could lead to very different conclusions than those obtained by examining the data through 2012. Another application examined characteristics like speed, acceleration, and banking angle for two different motorcycle racers, one a professional and the other a journalist (shown in Fig. 1). These attributes could be filtered by range which filtered the data shown on a map.

6.2 Collaborative Visualization Design

Visualization design is an iterative process, often requiring exploration and refinement. When working with a client or colleague, sharing the state of the visualization is useful. It is also useful to share multiple designs and show their relationships. Finally, a static view is not nearly as useful as the ability to share working, interactive visualizations, where users may change and tweak shared views. Keeping track of this design history may be important not only for reflection and learning, but also to spur future ideas. Consider a design that did not work well for a particular dataset but would be great for another. Being able to review the aspects of the old design, though it was never published would be very useful.

The Polestar application provides an interface for creating a visualization from multi-attribute datasets, similar to applications like...
Tableau [58]. By mapping attributes to different visualization channels, a user can explore different visualizations of a dataset. Polestar provides its own undo and redo capabilities by tracking specific variables when an event occurs as well as bookmarks where visualization specifications can be preserved in the browser’s local storage. However, it currently lacks ability to show the steps involved in creating a book-marked visualization—the iterative design process. While SIMProv.js provides its own interfaces, we were able to integrate the functionality into Polestar without adding a new layer to the interface. Undo and redo are then provenance-backed, and the gallery view and import/export capabilities can be included in Polestar’s toolbar.

As an example, when exploring the standard cars dataset, a user might be interested in a relationship between fuel economy and model year. That user might try adding encodings of other attributes and then decide to post the provenance on the web; SIMProv.js allows users to automatically upload the provenance as a gist on Github. If another user downloads the provenance and examines some of the past visualizations, she may choose different techniques like small multiples or different attributes to emphasize like country of origin. As other users post their updates, the original user can see how they have built on his ideas and iterate on them. Again, the derivation of the design or those designs that were discarded may be relevant to others, and SIMProv.js seeks to preserve this provenance. See Fig. 7 for some of the provenance produced by this exploration.

7 Discussion

SIMProv.js provides an initial solution for capturing and using provenance from web-based visual analyses. There are a number of potential issues that should examined both when including provenance and improving the support for and utility of the provenance. For provenance to be of use years or even days in the future, the data and the specification of the interactive application must be available in the form they existed when the original analysis was performed. That use of provenance may allow not only the replay of past work but also analysis of user decisions.

Updates A clear concern when storing provenance is the evolution of the interactive application or the data used. If a web developer fixes a bug in a visualization or changes a variable name, past provenance may not load or may produce a different result than when it was generated. Schemes that were used for versioning web services may be useful here [39]. Another option is to archive the page and its resources (the data and dependent libraries) as part of a provenance bundle. If the data comes from a curated store, it alleviates the need for maintaining another copy, and for this reason, the choice on what to include can be left to the user. Similarly, some library versions may not need to be explicitly included if they exist in a centralized archive. While content delivery networks (CDNs) help centralize standard libraries, they can also serve to archive past versions. A final option is for the application developer to provide upgrade schemes that translate past provenance from a previous version of the application to the latest format. For renaming or reorganization, helper routines can mitigate the required effort here.

Understanding Users Existing work [41, 50] has shown that providing users with more information about what they have explored and the insights already gained improves their ability to analyze data. Our concern in evaluating our framework is trying to understand provenance-specific concerns. Specifically, we are curious how the various presentations of provenance (thumbnails, list view, tree view) aid in recall and navigation. Additionally, we plan to evaluate strategies that users employ to revisit past states, known and unknown, after a few minutes versus a few days. For streaming data, we are interested in understanding when users toggle between persistent and updating modes, and how this affects analyses.

Developer Cost A brief survey of prototypes and demos from VIS 2017 projects that were made publicly available [47] showed few allowed users to persist states, and we believe one reason for this is that the cost to enable provenance from scratch is high. The reason we examined examples from dc.js and vega is that they have fairly standard interfaces for responding to specific changes. For dc.js, we were able to define a reusable base of changes in a provenance bridge. The bridge was used for the case study to enable three different dc.js visualizations. Configuring the differences meant identifying the charts and their types, and providing label information. While vega provided a uniform method for listing for signals, understanding exactly which data to capture, how to inject state, and the dependencies between signals required a deeper understanding of the underlying visualization design. We believe that if developers build visualizations with provenance in mind, greater support in the libraries will help simplify the links between the visualizations and provenance mechanisms.

Referenced Data Another potential issue developers may face in extending an existing application to support provenance is a reliance on in-memory semantics. For example, suppose there is a check on whether a specific object exists in a particular container; if that object was passed from the container by reference and eventually ended up in the data, we expect that such a query would return true. However, if that object was deserialized from provenance change data and the container was deserialized from a provenance checkpoint, the normal comparison of those objects will show them as dissimilar. A related issue is that referenced data in an action or checkpoint may be modified by later actions. In order to guard against this, we clone the object so that it will not be mutated before the provenance is persisted. Again, this may introduce the issue where two clones of the same object are viewed as unequal instead of the same, but we have not yet encountered this. We recommend not relying on any operators that require equality checks based on memory location for provenance-enabled applications.

8 Conclusion and Future Work

We have examined a number of aspects that important for users to capture, recall, store, and share provenance information from web-based applications. As more work is done in client-side web environments, the ability to capture and preserve the explorations is increasingly important. We believe that large-scale client-side web visualizations and analytic environments are rare because web provenance frameworks do not exist. We see bridges to existing frameworks as a path to increase adoption. A benefit of client-side web environments is that they protect privacy in a medium where that is usually not expected. If the initial specifications come from a server, a user can take the framework, plug in their own data, and capture provenance without it being reported back to a server. At the same time, users may choose to share provenance with a colleague by sending the provenance file or share it with the world by posting it to a common site or potentially the server itself. They can also use provenance to facilitate synchronous collaboration and deal with the cognitive burden that streaming data brings.

While SIMProv.js provides an initial prototype, we believe that the initial work with streaming data may be interesting to apply to progressive computations [53], where a user’s steering might interrupt computations before they finish. In those cases, what should the provenance record? Clearly, we can record the point at which a computation was interrupted or a new computation begun, but how do we capture the state of the data or visualization at that point?

More opportunities come from the wealth of data that can be produced by capturing provenance. Especially for focused applications where many users may interact with a visualization (e.g. interactive visualizations published by the media), techniques to aggregate or analyze the data will certainly be useful in understanding trends or patterns. Another opportunity from the data is to try to extract some of the higher-level insight or task provenance which might be aided by a large amount of data [15].

References


