Interaction History for Building Human-Data Interfaces

Yifan Wu

Electrical Engineering and Computer Sciences
University of California, Berkeley

May 13, 2021

Interaction History for Building Human-Data Interfaces

by

Yifan Wu

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph Hellerstein, Chair
Professor Marti Hearst
Associate Professor Björn Hartmann

Spring 2021

Interaction History for Building Human-Data Interfaces

Abstract

Interaction History for Building Human-Data Interfaces

by

Yifan Wu

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph Hellerstein, Chair

History provides context for the present. In the same way, past user interactions provide context for present explorations. This thesis investigates ways to reify user interaction history to address emerging challenges in the design and programming of human-data interfaces.

We leverage interaction history in three different but connected designs. The first is to enhance the design of interactions that suffer from delays, such as when working with remote databases. We use interaction history as a visual anchor to facilitate concurrent interactions, which ameliorate the cognitive burdens caused by delays. The second is to facilitate the programming of interactive visualizations involving asynchronous communication with remote databases. We capture event histories as a first-class programming construct, allowing the developer to declaratively specify what data to compute and how to update the state of the user interface. This way, developers avoid the low-level details of accessing remote data and coordinating events. The third and last design is to use interaction history to create a bridge between interaction design and programming. We capture and reify interaction history in both computational notebooks and interactive visualizations. Affordances on these reified histories help data scientists move fluidly between the two mediums.

To AGP.

Contents

List of Figures

List of Tables

Acknowledgments

Joe Hellerstein has been instrumental in my PhD journey. Your ability to think clearly, creatively, and rigorously helped me learn to conduct research independently. Your ample wisdom also helped me become more resilient to rejections. Eugene Wu at Columbia helped me publish my first paper when I was new to research. I am grateful for your patience. Over the years, your uncompromising scientific methods and writing standards has helped shape my research taste and skills. Remco Chang at Tufts University has been a cheer leader right from the start. My early years in grad school would have been bleak without your support and feedback. You helped me become a more confident researcher. Arvind Satyanarayan at MIT has been invaluable. Your brilliant observations, constructive questions, and spot-on writing advice were everything I had hoped for in a mentor and collaborator. Stratos Idreos, my undergraduate research advisor at Harvard University is the reason why I applied to graduate school at the first place. You went out of your way to share your enthusiasm for research and provide guidance.

I am also grateful to my committee members, Marti Hearst and Björn Hartmann for their insights and time spent on guiding my thesis at critical junctions. Aditya Parameswaran at UC Berkeley continues to provide great input. Steven Drucker, Matthai Philipose, Lenin Ravindranath provided helpful guidance during my summer at MSR. Jessica Hullman at Northwestern University offered valuable, albeit brief, mentorship.

I was lucky to have collaborated briefly with Larry Xu and Devin Petersohn at UC Berkeley, Dylan Cashman at Tufts University, and Alex Kale at University of Washington. A special thanks to Doris Xin for initiating the next leg of our journey and motivating a timely completion of this thesis. I remain in awe of your work ethic and creative energy. I look forward to more of our collaborations in the years to come.

Graduate researchers at the RISELab and BiD have been mentors, class project partners, and friends. In particular, Michael Whittaker was a great sounding board for technical challenges and a constant inspiration for giving better talks. Rolando Garcia always had thoughtful feedback with a philosophical flare. And to the rest of the Hellerstein crew, Chenggang Wu, Vikram Sreekanti, and Johann Schleier-Smith, it's been a good journey together since 2015. Andrew Head at BiD had been generous with his time. Lucie Choi and Ryan Purpura were great undergraduate research assistants who helped me become a better mentor. Outside of the two labs, Sanjay Krishnan, Fotis Psallidas, Jose Faleiro, Leilani Battle, Dominik Moritz, Kanit Wongsuphasawat, Katherine Ye, Elena Glassman, and Sarah Chasins offered valuable advice when our paths crossed.

Of course I owe thanks to the RISELab admins, Boban, Kattt, David, Shane, and Jon for keeping everything running. The EECS advising staff members, Shirley Salanio and Jean Nguyen, have also been kind and helpful. Outside of Berkeley, Elan has been an unfailing source of wisdom and a fantastic coach.

Chapter 1

Introduction

Data science is becoming relevant to all aspects of work, and with the increased use comes new challenges. Among the challenges that researchers have identified [73, 6, 23], scalable computations and better exploration are the two we will address in this thesis. The first challenge is to support interactions over larger datasets in human-data interfaces and the second challenge is to support easier interactive visualizations in a programming environment.

To tackle these challenges, we make use of *interaction history* in three ways. The first uses interaction history as a design component for a new interaction experience that is resilient to long data processing delays. The second uses interaction history as a programming construct to work with distributed data and asynchronous events. The third uses interaction history as a building block for bridges between programming and interaction design.

For Interaction Design

Interaction history has been an integral part of interactive visualization design: Shneiderman encouraged designers to "keep a history of actions to support undo, replay, and progressive refinement" [129]. Willett et al. augmented interactive widgets with a visualization of the history of people's interactions to provide cues for others to "forage" for information to navigate in new information space [106]. Callahan et al. called for the use of analytics provenance to track the "trails" of analysis to improve the scientific process [20]. Heer et al. analyzed ways to surface and model interaction histories in visualization tools like Tableau [59]. More recently, Feng et al. developed methods to directly represent interaction history in existing visualizations, creating markers in information saturated environments [39].

These prior works demonstrate the utility of interaction history as a design component: it allows users to develop their analyses over sequences of interactions without having to remember their past steps. We use this insight to tackle a challenge created by large datasets, where the system responds to user interactions after a long delay.

The reason why it is worthwhile to create a latency-resilient interface, rather than just

removing latency from the system side [90, 88, 97, 111, 139], is that upgrading an existing database to these systems can be expensive and prevents the creation of new applications. It would be valuable to develop a solution that only changes the frontend design.

Our observation is that existing interaction designs support only *one* interaction at a time[1]. If instead we can support *multiple concurrent interactions*, users may be able to amortize the cost of the delays. However, these concurrent interactions could be difficult to use because the results to the response may arrive out of order after unexpected delays [68]. To overcome the challenge, we hypothesized that interaction history can serve as a visual anchor to stabilize these otherwise unexpected visual updates.

We tested the hypothesis through a design technique—*Interaction Snapshots*. It captures previous interaction state in thumbnails and allows users to make multiple interactions and visit results at a later time. We found that the Snapshots design allows users to initiate multiple interactions without having to wait for each to load synchronously, thereby reducing the aggregate latency. Participants reported improved user experience dealing with the latencies. They also commented on better support to navigate the snapshots. The feedback points to the added cognitive complexity introduced by history-based designs and warns of potential limitations to adoption that need to be addressed by future work.

For Programming

Since interaction history has proven useful as design component for interactive visualizations, it would be ideal for a programming library to support designs that use interaction histories. For instance, the complexity of programming *Interaction Snapshots* would be reduced if an external library tracked the history of interaction and server response. To support the programming of interactive visualization applications, we hypothesize that interaction history, in the form of event logs, can be a useful abstraction.

Through our investigations of the hypothesis, we found that event logs offer additional programming benefits based on research in *distributed systems programming*, a field that uses event logs extensively [62, 7]. Distributed programming is a valuable lens to look at emerging interactive visualization use cases. When developers change from working with small, browser-based, data files to working with remote database and streaming data sources, they must grapple with distributed data and asynchronous events—the same challenges as in distributed systems.

Interaction history offers two key capabilities for programming distributed system. First, by capturing history as event logs, transient events are now persistent data. Rather than handling the events with imperative networking, developers can specify the state of the

---

[1] With the exception of *Pangloss* [97]: Pangloss is a progressive interactive visualization system that shows users a quick result based on sampled data, and users could request full results to be loaded in the background. We build on prior work but do not require a database migration to support online aggregation.

interface as a query over the event logs and underlying static data. They specify *what* data to get, rather than *how* to manipulate the data [28, 113]. This way, the system shoulders the burden of executing the queries instead of the developer. Second, the system augments the event logs with a logical timestep and ensures that the UI state updates in an atomic fashion. This way, developers can declaratively reason with the asynchrony and know that the interface will abide by the design invariant specified. They do not need to manually mutate state [62, 96] or keep track of the correspondences between events.

Based on these ideas, we created DIEL, an interactive visualization middle-ware. Through a series of evaluations on performance, expressibility, and notation usability, we found that DIEL provides a simple but functional declarative abstraction for interactive visualizations over distributed data and asynchronous events.

For Bridging Interactions and Programming

Since interaction history can support both the design and the programming of interactive visualizations, we hypothesize that it can serve as a bridge between visualization interactions and programs.

We tested the hypothesis in the medium of computational notebooks, e.g., Jupyter [79], a popular tool used by data scientists to both manipulate and visualize data [55, 23]. Notebooks supports both programming and interactivity through widgets [67] and HTML based cell outputs [122, 107]. However, the two mediums still remain siloed[2]: interactive visualizations must be manually specified as they are divorced from the analysis provenance expressed via dataframes, while code cells have no access to users' interactions with visualizations, and hence no way to operate on the results of interaction.

To break the silos and build a bridge between the two mediums, we look to their differences through the lens of interaction history. Notebooks persist history in "cells", which grows linearly. In contrast, interactions on visualizations are transient and hence the visualizations are displayed in a fixed parallel layout.

Besides helping us identify some gaps, history also helps us bridge gaps. To make the transient interactions persistent (and hence reproduceable), interactions on visualizations can be reified as programs executed in notebook cells, creating a persisted, executable log of the interactions. To relieve users the tedium of creating interactive visualizations, we can capture the data manipulation done in notebook cells to create relevant interactive visualizations automatically.

We instantiated these bridges in *B2*, a Jupyter notebook extension that brings seamless exploratory data analysis into the notebook.

---

[2] Bret Victor has also demonstrated novel programming interfaces that supports interactive controls [149], but these work are graphic editors and not data visualizations, which has a different set of abstractions.

1.1 Thesis Outline

This thesis contributes three novel techniques and artifacts that use interaction history to enhance the design and development of human-data interfaces. The rest of the thesis will present the main contributions in turn: *Interaction Snapshots*, DIEL, and *B2*. We summarize the three contributions to help the reader navigate the paper.

Interaction Snapshots is a design technique based on interaction history to improve the user experience in the face of high latency. It augments existing interactive visualizations with "snapshots" of past interactions and their respective results. As users interact, responses to requests from the interactions are asynchronously loaded into these snapshots. These snapshots serve as a visual anchor to help users work concurrently with multiple interactions. Our user study participants found the design helpful as they did not have to wait for each result to load and could easily navigate to prior snapshots.

DIEL is a programming toolkit based on interaction history to facilitate interactions with remote databases and streaming data. DIEL captures event history in event logs. These logs transform events into data, which developers can query directly to specify the state of the interface. As a result, developers avoid low-level details of networking data. Instead, DIEL compiles and optimizes a corresponding dataflow graph, and automatically generates necessary low-level distributed systems details. The event logs also provide logical timesteps that help developers reason about asynchronous events, such that their non-blocking interface avoids ending in inconsistent states and confusing the end user.

Lastly, B2 uses interaction history to bridge the divide between programming and interactive visualization in the computational notebook. B2 does so by leveraging history in both programming and interactions. On the programming side, it instruments dataframes to track the queries expressed in code and synthesize corresponding visualizations. These visualizations are displayed in a dashboard to facilitate interactive analysis. On the visualization side, B2 reifies interactions as data queries and generates a history log in a new code cell. Subsequent cells can use this log to further analyze interaction results and, when marked as *reactive*, to ensure that code is automatically recomputed when new interaction occurs. In a user study with data scientists, we find that B2 promotes a tighter feedback loop between coding and interacting with visualizations. All participants frequently moved from code to visualization and vice-versa, which facilitated their exploratory data analysis in the notebook.

1.2 Prior Publications and Authorship

Although I am the main author of the research in this thesis, the work is in collaboration with my advisor Joe Hellerstein and mentors Arvind Satyanarayan, Eugene Wu, and Remco Chang. I use the first person plural to reflect the collaborative effort.

Interaction Snapshots was published in InfoVis 2020 [163]. I arrived at the idea while working on *Devil* [166] with Joe Hellerstein and Eugene Wu. In that work, we explored different consistency models of the user interface by borrowing concepts from databases. In particular, the concept of *Multi-version Concurrency Control* in particular caught our attention. Remco's InfoVis and HCI expertise then helped guide the experiment designs, which was conducted by Larry Xu and I. Larry was a Master's student at UC Berkeley and we worked on early design of Snapshots, which we called *Chronicles* [167]. Three years after the initial work, I added the dashboard based designs that is the core of the Interaction Snapshots paper published in InfoVis 2020.

The work presented in DIEL is the result of a two prior publications [166, 161], though the main work is still in submission. DIEL was initially inspired by *Bloom* [7], work advised by Joe. By looking at interactive visualization programming through the lens of distributed systems, we were able to identify not only ways to improve the programming of interactive visualizations, but also expose potential UX issues caused by inconsistencies created by out of order events. UC Berkeley undergrads Lucie Choi and Ryan Purpura helped me with the programming of DIEL.

B2 was published at UIST 2020 [165]. I proposed the initial plan to Joe Hellerstein and later Arvind Satyanarayan, who then helped me iterate on the research questions, design of the artifact, experiment design and the final writing. Ryan Purpura also helped me with the programming of B2.

Chapter 2

Facilitating Exploration with *Interaction Snapshots* under High Latency

Most interactive visualizations are transient—each interaction replaces the previous state and updates the visualization with the current specification. *Interaction Snapshots* augment these visualizations design with interaction history in an unobtrusive manner. With the augmented history view, the interactions are no longer transient but rather persisted directly in the visualization.

We hypothesized that when using snapshots in visualizations with high interactive latency, users can, using snapshots as the stabilizing visual anchor, make multiple interactions concurrently. In our user study, participants performed analysis tasks more quickly and preferred the snapshots design. The results validate our hypothesis that interaction history can be used to improve user experience under high interactive latency.

Interaction Snapshots was published at InfoVis 2020.

2.1   Introduction

Current interactive data visualization systems rely on fast response times to provide a good user experience. This approach simplifies the design of the visualization UI and ensures direct manipulation interfaces that facilitate fluid user data exploration [89]. However, interactive data visualization is increasingly an integral part of big data analysis. The scale of the datasets and the required computational power has made it necessary to shift the data processing and storage to remote databases. In such a client-server architecture, client interactions are translated into server requests that incur both data processing and network latency. Ensuring ultra fast response times in the face of all these latencies is often challenging

if not impossible. The interface therefore should have a backup plan should the latency be high—the frontend needs to be resilient to high latencies.

Prior work, such as progressive visualization [63, 40, 171, 37, 112] and optimistic visualization [97], have also utilized interface design to deal with latency. However, these approaches still rely on considerable backend instrumentation, namely online aggregation [64, 63, 82] and approximate query processing [35, 5]. In many settings, designers do not have the opportunity, desire, or resources to make changes to the backend database systems.

Current UX-oriented solutions primarily address usability challenges stemming from a *single* user request. They focus on ways to shorten the time between the frontend sending the request to the backend and receiving the response. For instance, progressive visualization updates a single selected visualization with more accurate results over time. But what happens when the user wishes to make another interaction while the previous is still being processed? We now discuss the two predominant designs.

One such design is "blocking", where users are not allowed to perform a new interaction until the prior ones have rendered. This design makes the most sense when the latency is negligible, which is often the case when the data is small and fits in memory, such as the case for Vega [123] which runs on the browser, and Excel. The design is easy to implement and puts the least amount of load on the backend. As a result, even client-server systems like Tableau, which may not always guarantee negligible latency, adopt it.

Another common design is to allow new requests to be made and cancel previous requests. Allowing the user to interrupt existing requests makes the interface more *responsive* and ensures that "time-consuming operations that block other activity" can be aborted [69]. The interface renders the results of the most recent interaction only.

If the previous interaction is *not* cancelled, then more than one pending request will be processed concurrently, which has potential to reduce the overall latency and improve user experience. However, rendering interaction responses concurrently runs contrary to *direct manipulation*[131, 68], a commonly held user-interface design principle. Direct manipulation requires that "the object of interest is immediately visible", which in effect assumes a serial relationship between a user's action and the system's response in a one-to-one fashion. In contrast, allowing multiple responses to render concurrently behaves in an opposite manner— when a user interacts with a number of visual elements, the system might not respond to these interactions in the sequence the user's actions are performed or to replace the results too quickly. Both of which can be confusing to users by making it difficult to reason about the *correspondence* between interaction and response and to make sense of the responses.

To harness the benefit of concurrent interactions, we must address the design challenge it imposes. Our approach is to visualize the coordination between asynchronous request and responses explicitly. We do so by capturing the interaction results in a sequence of *snapshots*. This way, each new result of an interaction is appended to a history of results. Snapshots

provide a stable frame of reference that helps users make sense of uncertain latencies. Given this easy visual reference, the users could view the snapshots at a later time once the response is received. Snapshots mediate the asynchronous results through history and create a direct manipulation experience for multiple concurrent interactions.

Consider a cross-filter application, as shown in Fig. 2.1. After a user makes an interaction, a snapshot is created and appended beneath the visualization dashboard. The snapshot is a scaled-down display of the current visualization which continues to load while other snapshots are appended. The snapshot gives a visual indicator (e.g., spinner) of whether the data is still being processed. When the user sees the visual indication that the processing is complete, they can click on the corresponding snapshot, which loads the processed visualization into the main view for analysis. Users can also navigate through the snapshots quickly with left and right arrows, which "animates" through the selections.

We evaluated the efficacy of interaction snapshots on dashboards with 6 participants.
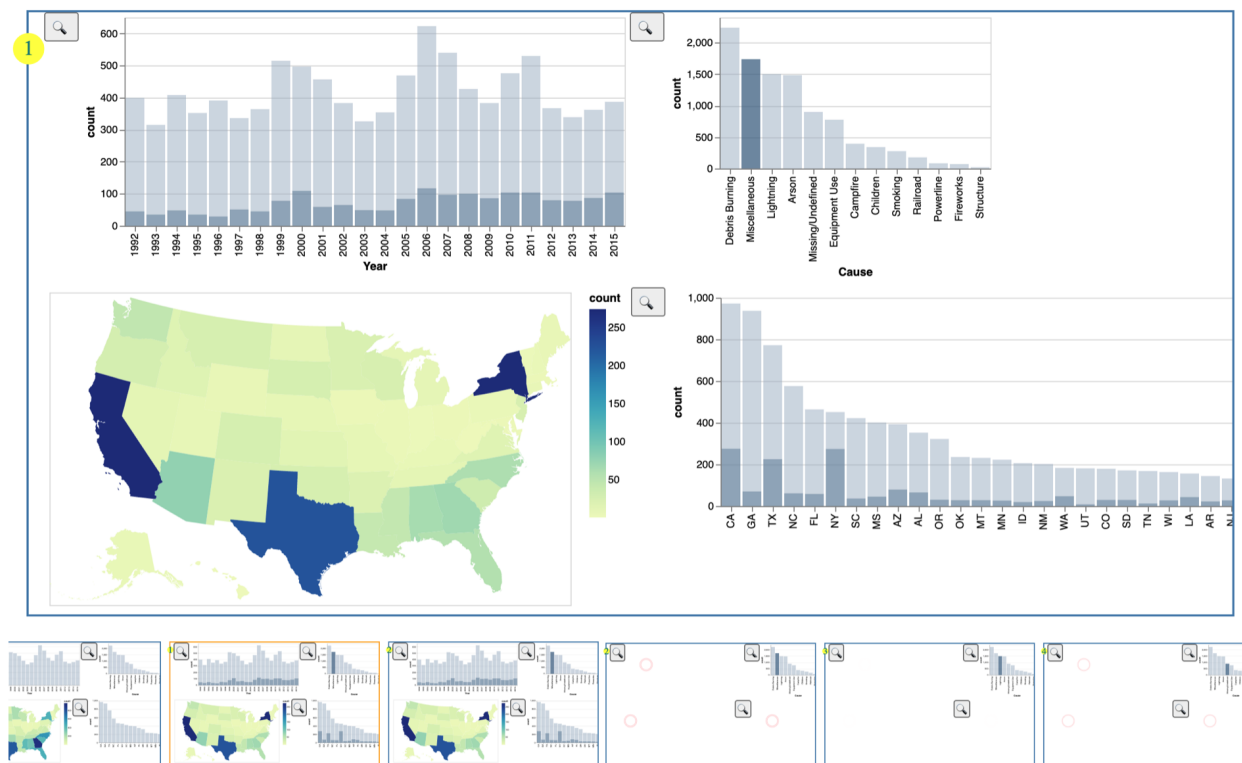
Figure 2.1: Applying interaction snapshots to a cross-filter visualization. Evaluation of US wildfire data. As users interact, snapshots are created. Users can perform *concurrent* interactions where they do not have to wait until the previous results arrive.

Traces of participant behavior demonstrate they can make effective use of concurrent inter-
actions and navigate to different snapshots. Qualitative feedback reveals that participants
find the interaction snapshot design helpful in the face of latency.

2.2   Related Work

Dealing with Interactive Latency: Prior work addressing the issue of latency in inter-
active visualizations can be divided along two axes: whether the solution is provided by the
backend or the frontend, and whether the query is evaluated on whole or samples of the
data.

Backend techniques to reduce latency for queries over all of the data include GPU-based
compute [90, 93, 45], custom indices [88, 139], and prefetching [98, 14]. Backend techniques
for a probabilistic query trade some amount of uncertainty for the reduction in processing
time. These include approximate query processing [5, 35] and online aggregation engines [64,
63] from which *progressive visualization* [64, 40, 171, 37, 112], which we discuss in the next
paragraph, is based on. While innovations in these backend techniques improve computation
capabilities, they do not eliminate the existence of latency in the real world. Users could be
dealing with legacy systems, large amounts of data, or, sometimes, slow network connection.

Compared to the pure backend efforts, our work is more closely related to progressive
visualization, which streams partial results (containing error bounds) to users. An augmen-
tation to the streaming design is Moritz et al.'s *optimistic visualization*, which allows the
user to first interact with an approximate query engine, and lets users mark an interaction
as "remembered" for a full, non-approximate, evaluation to check later [97].

We take inspiration from these works' approach of leveraging design to adapt to the real-
ities of "big data". In particular, optimistic visualization allows for users to optimize *across*
interactions, from which our design builds on. However, both progressive and optimistic
visualization rely on approximate query processing. Since on-the-fly sampling cannot cover
every small subset of data, many approximate query techniques also involve precomputing
samples, sketches, or other summary structures [31]. The preprocessing steps require time,
computation, and storage for each precomputed result. The additional effort may be worth-
while for developers who could afford backend changes, e.g., adapting advanced engines like
Sample+Seek. For those who would rather make changes just to the frontend, interaction
snapshots may be a more preferable trade-off—developers just need to add the new UX
technique and potentially limit their interaction designs to avoid ones that would trigger a
large amount of interactions, such as continuous brushing.

Interaction History: Much prior HCI work has used interaction histories to facilitate
user actions. Work in the CSCW community used trails of cursor positions to give temporal
context to the actions of remote participants [126]. In the visualization community, Heer et al.

model history as a sequence of movements through a graph of application states, presented in thumbnails in *Graphical histories* [59]. Feng et al. externalized interaction history by showing the "footprints" of interactions [38]. Optimistic visualization, as mentioned earlier, also makes use of history (the "remember" feature) to help users verify approximate results [97].

These prior work inform our design. Feng et al. observed that a direct encoding of interaction history supports visual recognition of previous interactions. The visual history does not require users to recall the past, which can be mentally taxing. The observation inspired us to visualize interaction history to reduce the cognitive challenges to asynchronous interactions. Graphical histories gave us inspirations for how to design the snapshot for dashboards, and optimistic visualization gives initial support that users do revisit interaction history when exploring data.

2.3 Design Iterations

To design with latency, we first analyze different ways interactions are handled in the presence of latency. The top diagram in Fig. 2.2 depicts a time-ordered model, where time increases from left to right. User inputs are depicted along the top line (interaction history), and the responses are rendered along the bottom line (render history). A dashed arrow between the interaction and render history corresponds to the time to respond to the request.

Fig. 2.2 (1) shows the ideal case where requests respond instantaneously. However, when there is latency, a number of possible scenarios can occur: (2) shows the blocking case where the user is not allowed to submit a new request until the prior one completes; (3) shows the non-blocking case where users freely interact with the visualization and new requests supersede and cancel previous requests; (4) shows the concurrent case where neither the input nor the output is blocked. The benefit of this approach is that the total time is shorter, but the downside is that the interface could be difficult to interpret, especially when the amount of latency varies (5).

To address the dis-coordination between interaction request and response, as seen in Fig. 2.2 (4,5), we hypothesize that displaying past interactions in snapshots could serve as a stabilizing visual anchor. A simple mechanism is to encode the step by which the interaction was made using a visual encoding channel. The right of Fig. 2.3 shows an example where selections are rendered in snapshots.

We conducted a pilot study to verify the hypothesis. In the pilot, participants used a simple visualization shown in Fig. 2.3: a bar chart that displays sales data for a company across years, split into facets of the months. There are three conditions: baseline, treatment 1 and treatment 2. The first two uses the interfaces on the left of Fig. 2.3, and the last one the right of Fig. 2.3. The baseline is the blocking interaction, as illustrated in Fig. 2.2 (2). Treatment 1 has the same UI as baseline, but asynchronously renders the results, as in Fig. 2.2 (4,5). Treatment condition 2 shows interaction snapshots.
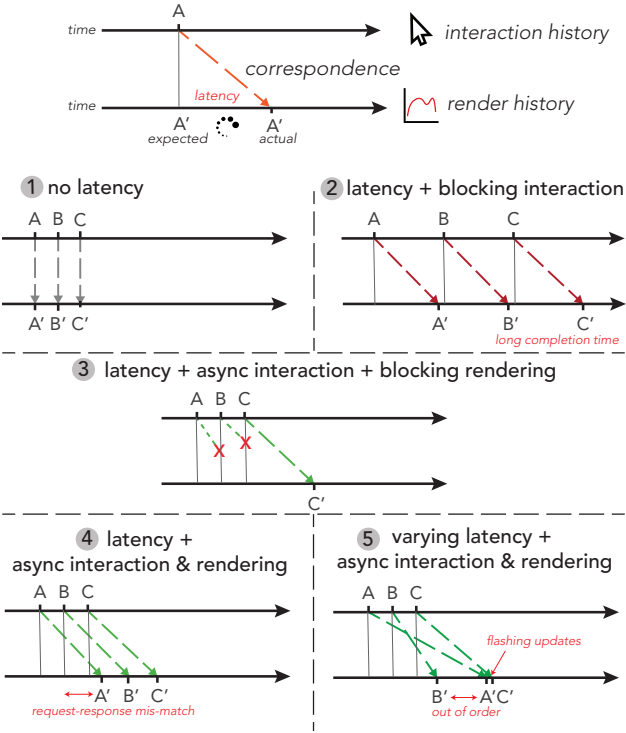
Figure 2.2: A sequence of interaction requests and responses under different conditions visualized on a horizontal time axis. Colored arrows represent request/response pairs over time. Light vertical lines highlight request times. Case (1) is the ideal no-latency scenario commonly assumed by visualization designers—everything works as expected. Case (2) is a latency scenario where the user is forced to wait for each response to load before they are allowed to interact again. Case (3) is another latency scenario but unlike the previous, the user does not need to wait. Previous interaction responses that are in-flight are not rendered. Case (4) differs from the previous in that all responses are rendered. Lastly, case (5) shows an example of (4) where the responses are shown in a different order than the order requests were issued.

Since asynchronous rendering causes reordering of the results (per Fig. 2.2 (5)), we chose a visual task that is not order dependent, to encourage asynchronous interactions. For a bar chart that displays sales data for a company across months and years, we asked users to identify if any of the months crossed the sales threshold of 80 units sold. We generated data to ensure the task was not perceptually difficult. There were no data points that were close to the threshold of 80 units.

We measured the accuracy of the response and the total time to complete a task in seconds (the time between when the participant is allowed to start interacting and when the they submit an answer). We also logged all events on the UI, such as interactions, responses
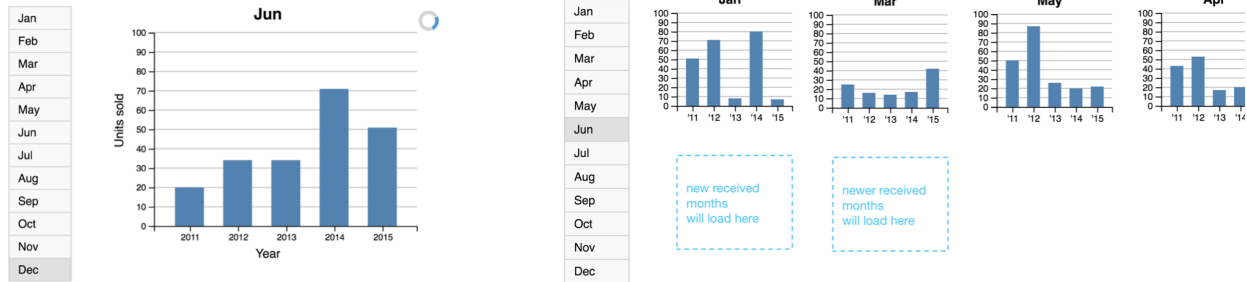
Figure 2.3: Pilot experiment: on the left is the basic design where interaction results update in place, on the right is a design that displays snapshots of interaction results as small multiples.
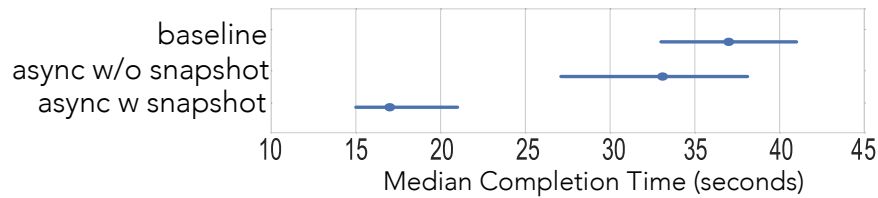
Figure 2.4: Comparison of median users task completion times, with the *interaction snap-shots* condition being much faster than the others.

received, and response rendered.

We recruited participants online through Amazon Mechanical Turk (17 participants for baseline, and 30 for the two treatments, 58% with bachelors degree or higher, and 46% female, ranging from 23 to 67 years of age). Participants were compensated $0.30 per task, with a $3-5 completion bonus, compliant with Californian minimum wage. Participants were randomly sorted into either the baseline or treatment group. They were shown instructions about the task and trained to complete two sample tasks beforehand, followed by 10 actual tasks.

The differences were in task completion time, shown in Fig. 2.4. We report the unsigned Wilcoxon Rank-Sum test: baseline median=37 sec (N = 31), treatment (condition 1) without snapshots median=33 sec (N=52), Z=0.63, $p < 0.5$, and treatment (condition 2) with snapshots median=17 sec (N=54), Z=3.22, $p < 0.002$ where N denotes the count of the group. There were no significant differences in accuracy between the three conditions. We can see that participants were able to complete the tasks much faster with the snapshots design. We did not observe a learning effect—there were no significant differences between the metrics on early tasks and later tasks. We speculate that this is because the participants
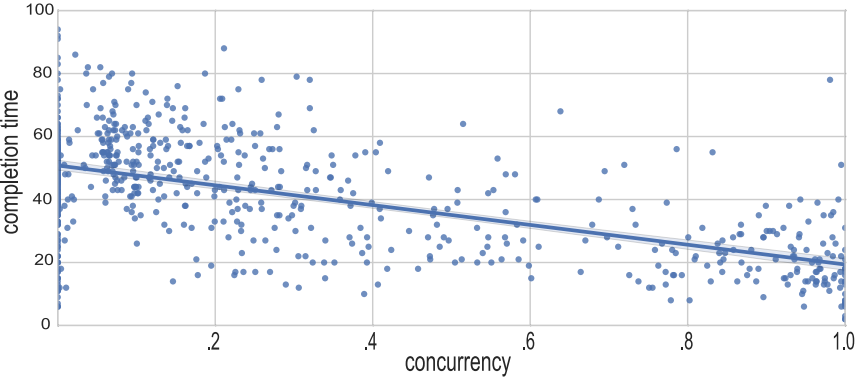
Figure 2.5: Completion time correlated with level of concurrency. A negative correlation suggests that concurrent interactions may help alleviate the effect of latency.
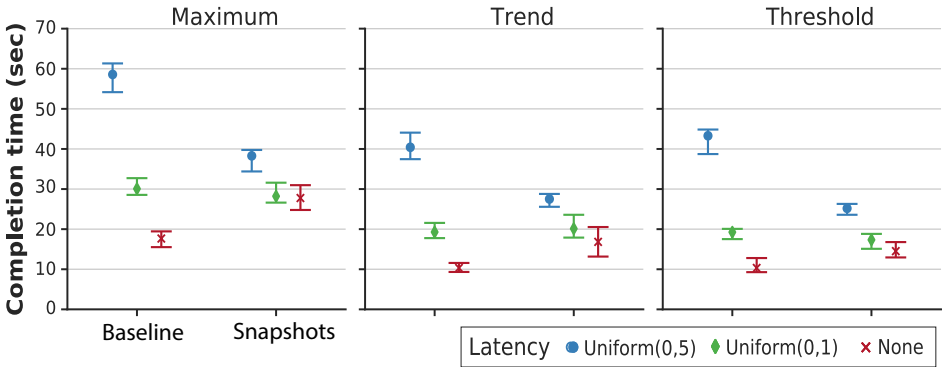
Figure 2.6: Each chart of the plot visualizes median task completion time with 95% CI (y-axis).

had already familiarized with the interface in the training session prior to tasks.

To understand why this was the case, we visualized the concurrency—the percentage of task completion time where there was more than one concurrent request. Fig. 2.5 show that participants could complete tasks faster with higher concurrency tend to complete tasks faster, which is made possible because of asynchrony and encouraged by the snapshot design.

To ensure that the result also generalizes to other tasks, we conducted a second pilot study to include two more tasks: identifying the month with a maximum value, and the month with a certain trend. These two tasks represent the "find maximum/extremum" and "characterize distribution" tasks in Amar et al.'s analytic activity taxonomy [8]. Both of these tasks are known to be more challenging than the threshold task in the first pilot study,

which falls under "retrieve value" in the taxonomy.

Each participant completed the tasks with either no snapshots (baseline) or with snapshots designs (treatment). We also have two latency conditions: one is uniformly sampled from 0 to 1 second (short), and the other is uniformly sampled between 0 to 5 seconds (long). Each participant completes three conditions (none, short, and long). For each group, we recruited 50 Mechanical Turk participants.

Again, we found that there were no significant differences in task accuracy and observed no learning effect. Task completion time, however, was very different, as shown in Fig. 2.6. We see that participants complete all tasks significantly faster in the with-snapshots condition when there was long latency.

More qualitatively, participants commented that completing the tasks with latency with no snapshots is "painful", "frustrating", "tedious", and "awful". Some explained that responses were hard to remember—*"I had a hard time remembering what I'd just seen a second ago."*. In contrast, participants commented on the ease of use when snapshots are present—*"The ability to load several months at once definitely offsets any loading latency – difficulty was roughly the same as one month with no latency. One month with latency was a bit painful."*. Interestingly, the perceived speed of loading seemed to have changed as well—*"Some of the tasks loaded really slow, single month got irritating waiting. Most of the multiple tasks loaded fairly quickly."* While the perception of latency is not the focus of this study, the feedback suggests the benefits of the use of interaction snapshots beyond improving task-completion times [53].

The interaction snapshots design was not free from fault. One participant commented that "It took a few tries to get used to how it worked". We also see evidence of this in Fig. 2.6—under the no-latency condition, participants took on average longer to complete tasks when using the new design compared to baseline.

2.4   Dashboard Snapshots: Design and Evaluation

The pilots evaluated interaction snapshots' effectiveness on a single visualization for fixed tasks. We now evaluate the technique for a more complex setting—dashboard—with open ended exploration.

One key design challenge with snapshots for dashboard is *space*. Replicating the interaction results as small multiples is not feasible for dashboards. We address this constraint by creating a separate representation of the snapshot—a smaller "thumbnail" view, much like Graphical Histories [59] and Pangloss [97], which can be clicked on and expanded. Fig. 2.1 is an example application of the technique on a cross-filter visualization of wildfires in the US [71]. The interaction details and code are included in supplementary materials.
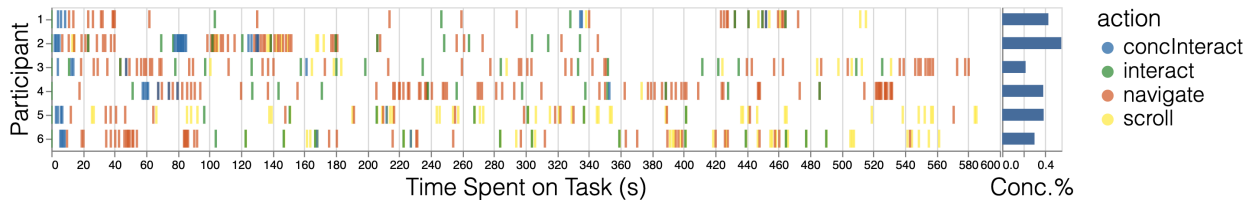
Figure 2.7: On the left is a visualization participants' interaction traces while exploring US wildfire data. The traces visualized are interactions—which contains both concurrent interactions (`concInteract`), and non-concurrent interactions (`interact`). On the right is the percent of concurrent interactions of all interactions made.

Methods

Rather than assigning participants specific tasks (as in previous pilots), we plan to observe how participants explore data and report qualitative metrics. First is whether and how much the participants make use of concurrent multiple interactions. A higher usage would suggest that the snapshots design is able to facilitate concurrent interactions, and that it is useful to the participants. Second is how often snapshots are revisited—the more participants engage in older snapshots, the more they are leveraging interaction snapshots' unique capabilities. Third is direct feedback.

We conducted a first-use study with 6 users, all college students who have taken data science courses, with a self-reported "somewhat experienced" with visual data analysis on a 5-point Likert scale ($\mu = 3.0, \sigma = 0.89$). Due to the COVID-19 "shelter-in-place" order, we conducted the studies over video. We began each study with a 5-minute tutorial of the interaction snapshot enabled dashboard on mass mobilization protest data [27], and then asked participants to analyze US wildfires using a similar dashboard (Fig. 2.1). Participants were prompted with a specific question ("identify the states with the most wildfires in the years 2000 to 2004") followed by free-form exploration for about 8 minutes. Then, participants verbally summarized their findings. All interaction latency was set to between 5 to 7 seconds. At the conclusion of the study, we administered an exit survey to measure the effectiveness of the interface and to debrief participants about their experiences. The session took 20-25 min and participants were compensated $15 in Amazon gift cards.

Quantitative Results

We instrumented the interface to log all user interactions, including interactive selections of elements in the charts, navigating to prior interaction states, and scrolling through the snapshots. To analyze this data, we visualized the traces in Fig. 2.7. For the interactions that happened *before* the completion of the previous interaction result (since there is a 5 to 7 second delay), we mark them as *concurrent interactions* (`concInteract`, as labeled

in Fig. 2.7), and the rest of the interactions as `interact`. We also provide a distribution of the percent of concurrent interactions out of all interactions in the bar chart to the right, with ($\mu = 0.38, \sigma = 0.10$). Participants interacted with the interface on average 20 times ($\mu = 20.33, \sigma = 5.99$) during the session, and navigated twice that on average ($\mu = 43.17, \sigma = 17.49$).

On 5-point Likert scales, participants positively rated the interface overall ($\mu = 4.33, \sigma = 0.47$), as well as the history feature ($\mu = 4.17, \sigma = 0.69$). In terms of how frustrating the delay was, participants rated it as only a little ($\mu = 2.0, \sigma = 0.58$).

Qualitative Results

We observed participants quickly grasped how to make use of concurrent interactions through the snapshots. One common pattern was to make multiple interactions concurrently when the participants had a question in mind. P3 mentioned that it was *"nice to have [the interaction result] pre-loaded"*, and that reminded them of *"opening search results in multiple tabs [in the background] when browsing web pages."* P6 also indirectly commented that *"Tabs [i.e., snapshots] made the wait less painful/annoying."*

However concurrent interactions were not always used. When participants had a specific question or targets in mind, such as "I want to compare how different causes of fires differ geographically", they knew exactly what interactions they would need and opted for concurrent interactions. When the participants did not have such a question, they relied on the results to their immediate selections to generate ideas for further interactions. Hence they waited for the response to load instead of making other interactions while waiting.

Interestingly, the snapshots still proved useful while they *waited*. P4 mentioned that when they were waiting for the result to load, they would *"look at the history to remember what I was doing earlier to keep track of what I'm looking for"*. P1 also used the delay to positive effect, saying that *"the delay gives my brain a time to reflect on what I'm expecting and what to do. I think it's actually better [than instantaneous response]."* None of the participants found the delay to be worse than "a little frustrating".

All of the participants browsed through history when summarizing their findings (the final task), recalling relevant insights they made prior. The evidence is shown in Fig. 2.7, a visualization of participant activities. The visualization records "`navigate`" events, which is created when a user navigates to a different snapshot. In the figure, we can see dense patches of `navigate` ticks towards the end of the sessions. Qualitative feedback also corroborates this pattern: P6 mentioned that *"History tool allowed me to go back to my previous thoughts easily, and made it easy to reference observations."*

Participants also desired more features. P4 mentioned that the snapshots quickly became cluttered, which made it difficult to navigate and detracted from the positive aspects of history. P3 suggested ways to introduce more guides for the snapshots, such as adding text
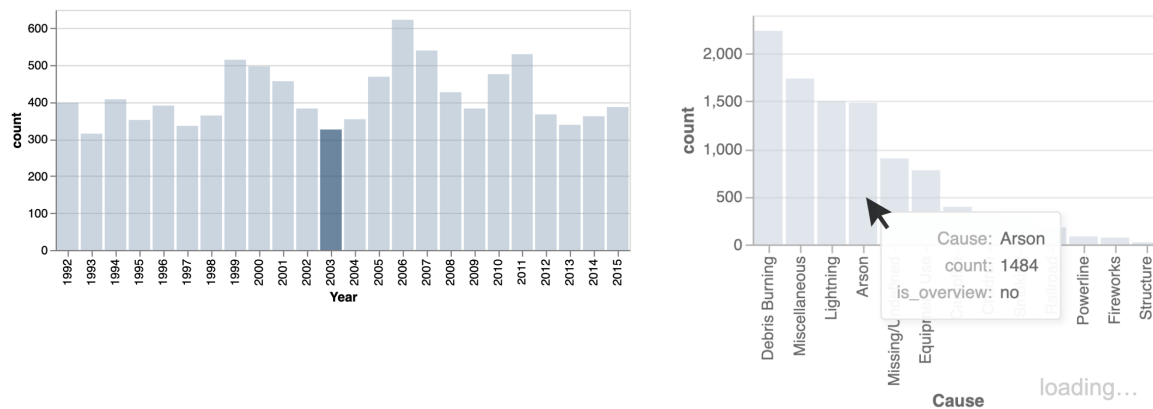
Figure 2.8: Fig. 2.1 showed an example where the charts pending updates are completely replaced by a spinner (see the last three snapshots) to indicate the loading status. This figure shows an alternative design. The loading status of the chart is indicated by the transparency of the chart plus the "loading..." text. More importantly, the "base" chart (light blue portion of the `Cause` chart to the right) is kept despite not having received a response. This way, the user can directly make another interaction without having to navigate to another snapshot where the chart has loaded.

to describe the interaction, or a way to either arrange or color encode the snapshots by the chart interacted with. P3 and P4 asked about the ability to remove snapshots that they found irrelevant.

2.5 The Interaction Snapshot Design Process

Having presented examples of interaction snapshots, we now conclude with a generalized design process. Interaction snapshots require three elements: (1) the user's past interactions, (2) the effect of each interaction, and (3) the temporal ordering of the interactions. Together, they show the correspondence between the user's interaction requests and the response of the system over time. There are different ways to satisfy these requirements. For the bar chart, we used the position encoding channel, which can be applied to other single visualizations. For the dashboard, we used snapshots, which can be applied to other multiple coordinated visualizations.

Interaction snapshots prescribes some design choices and leaves others intentionally under-specified, such as how the visualizations update before the response is received, including the design of relevant spinners and removal of previous results. These designs are not part of Snapshots because the interface likely already has a preexisting design that composes with the addition of snapshots. While these designs compose, there are still some design lessons
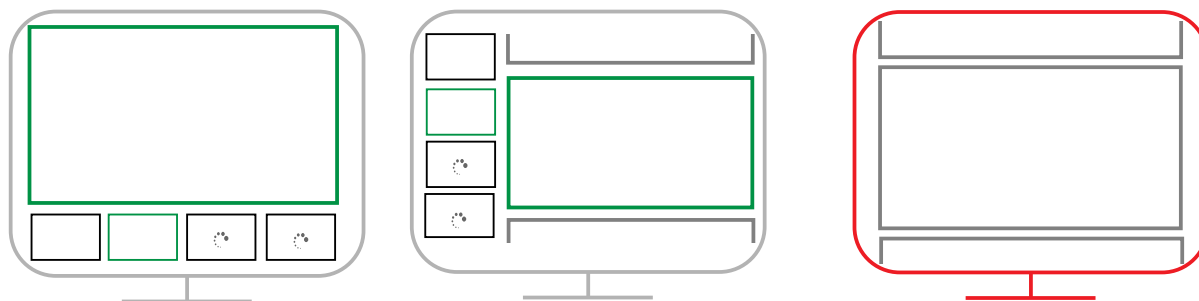
Figure 2.9: Illustration of different ways the snapshots could be created for the interactive. The design on the right does not have visible snapshots and we found it less conducive to concurrent interactions and does not improve the UX under high latency.

that we learnt from our own implementations. We now share these lessons with readers interested in applying Snapshots.

Easier Concurrent Interactions The running example implements the UX where charts with pending interaction results are immediately replaced with spinners. This means that the user cannot directly continue to make a new interaction on the chart. Rather, the user would have to go to a previous snapshot to interact.

To remove this additional hurdle , the designer could make a simple modification. As illustrate in Fig. 2.8, the the base interactive visualization is preserved despite the loading of the linked chart. Since the underlying chart is stable across interactions, the user could continue to make another interaction directly. There are other plausible designs for indicating that the system is processing, and we defer the reader to existing literature [127, 126].

Loading Indicators. When a dashboard is complex, it may become difficult to see if the result has loaded in a snapshot. Being able to see the snapshot results at a glance helps users navigate multiple snapshots quickly. To make the results more visible, we can visualize the loading status into an encoding channel, such as the color of the snapshot bounding box, or a colored dot or spinner.

Ensuring Snapshots Visibility. Snapshots of visualization dashboards easily run into interface real-estate limitations and we found it important to ensure that snapshots are immediately visible.

For example, for the initial interface used in our user study, we appended full-sized snapshots to the interface (right of Fig. 2.9). Users had to scroll to up to see prior interactions results. Furthermore, they do not have an immediate sense what snapshots has finished loading. Both of these factors discouraged participants from performing concurrent interactions.

Another example of ensuring snapshot visibility is automatically scrolling the newest snapshot into view. This makes it easier for the user to locate their current interactions.

In general, these findings and design reflect the core idea that snapshots only are as effective as their ability to help users navigate multiple concurrent interactions.

Implementation Method. Asynchronous interactions are notoriously difficult to implement. However, the snapshot design obviates the complexity of coordinating asynchronous events by allowing the developer to append new results to the current view. When the response is missing, simply render a spinner (or any other progress indicator) in its place. By recording the request/response pairs into a *log*, the entire interface becomes a visualization of the event log data. Chapter 3 will present a programming library that provides direct support for this programming pattern based on event logs.

2.6 Conclusion

Through many pilots and design iterations, we explored aspects of the design space for multiple concurrent interactions. We found a positive answer to our initial question of whether frontend design alone could offer some alleviation to the pain of latency.

Limitations and Future Work

While our findings inform us of the utility of the snapshots design, there still remain important questions unanswered. Since we have only evaluated the latency of around 5 seconds, we do not know exactly how much latency the snapshots design could permit.

Our hypothesis is that the latency that the snapshots design "buys" the system is congruous to the time it takes to comprehend the visualizations and formulate a question. Of course, the exact duration would depend on the visualization, the goal of the exploration, and the participant's data analysis experience. Future work can aim to identify the common range from a distribution of data collected. For instance, we anticipate that a latency of an hour will certainly prove the interface unusable for interactive explorations even with snapshots.

Future evaluations could explore eliciting subjective user experience and measuring quantitative metrics on a spectrum of latencies. Following prior work [89], we can systematically study how snapshots change user behavior in terms of their rates of making observations, drawing generalization, and forming hypothesis. We can also compare interaction snapshots and progressive/optimistic visualizations [97].

Another concern with snapshots is balancing the trade-off of cognitive ease versus complexity. On the one hand, we see that snapshots facilitates concurrent interactions, which makes latency less taxing. On the other hand, we see that users have to put in additional

effort to make sense of snapshots. Given that prior work on using history to facilitate web searches and data analysis have seen mixed results—ranging from limited adoption [10, 59, 56, 118] to widespread success in computational notebooks [79]—we need a longitudinal study to more accurately evaluate the efficacy and trade-offs of the snapshots design.

Besides studying the long-term utility of Snapshots, we can also explore ways to make it more usable. Following user study feedback, we can further develop controls around the snapshots such as editing and organizing. Another option is to augment the linear history with when users "branched" off into a different interaction to capture richer context— the snapshots could double as an interaction provenance graph. More broadly, *interaction snapshots* present opportunities to bring features common in literate computing to interactive visualizations.

Chapter 3

DIEL: Interactive Visualization Beyond the Here and Now

We continue to investigate the use of interaction history to facilitate interactive visualizations over remote datasets. In this work, rather than the *interactive* experience, we look to the *programming* experience.

When working with large datasets, the design challenge is latency and the programming challenge is complexity. Scaling from a browser-based application to one that leverages compute on remote databases adds complexity because the system is now *distributed*: data is on *both* the client browser and a remote database. As a result, handling interactions is now much more complex. Developers need to communicate asynchronously between the client and the remotes, sometimes translating between two different languages (JavaScript and SQL).

To address this challenge, we found that interaction history allows us two critical capabilities. First, capturing the interaction in an event log unifies events as data. This makes it possible for the programmer to define the application state more declaratively. Second, the logs provide the temporal context of events, which allows the programmer to reason with more user friendly, non-blocking, designs of the interface.

We make use of these two capabilities in the DIEL model and prototype. Through a series of performance, notation, and example based evaluations, we validate that history is not only a useful construct for design, but also for programming.

3.1   Introduction

Recent advances have made authoring browser-based interactive visualizations quite simple, via novel abstractions for specifying encodings [157, 19], layout, data transformations, and interactions [125, 122]. Critically, these abstractions enable *declarative* specification: devel-
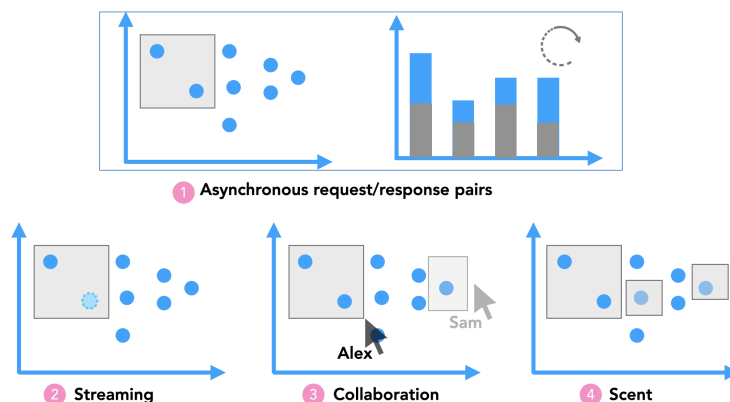
Figure 3.1: Example interactive visualization use cases "beyond here and now". (1) An interactive visualization over a databases, where the data is remote and the events are evaluated asynchronously. (2) A streaming interactive visualization where a new data point arrives within the existing brushed selection (colored green with dotted line), creating potential ambiguity for the brush selection behavior. (3) A collaborative interactive visualization where multiple people are interacting with a visualization and their activities are shared asynchronously. (4) "Scented" visualization where prior interactions are shown to provide users with analysis context.

opers can compose these visualization elements and defer execution concerns to the toolkit runtime. Declarative abstractions thus enable developers to build and iterate on designs quickly and expressively, while reducing the errors that would arise from imperative implementation [58, 125]. However, these abstractions do not gracefully support use cases where data may be distributed across multiple locations, or where events cannot be synchronously handled to completion. To illustrate, we discuss two interactive visualization settings—one quite traditional and another more complex—and reflect on the difficulty of addressing these issues with current practices.

Consider the *client-server architecture*, a classical design for scalable interactive visualizations that remains common even in recent research [98, 97]. A concrete example is shown in Fig. 3.1(1), where the brush selection over the scatterplot is computed by a remote database. To handle the brush selection, a developer can no longer rely on existing declarative abstractions (e.g., Vega-Lite's selections [122]) alone. Instead, they must now translate interaction logic from a visualization specification to the query language of the database, network with the remote database, and manually bookkeep the provenance of interactions and database responses to ensure that the interface is displaying the correct results, in the correct order, in the face of non-deterministic processing latency.

A more specialized example is *streaming data*, increasingly investigated in research [168,

83, 154] and industry [173]. An example is shown in Fig. 3.1(2), where the points can be streamed in the scatterplot and selected by a brush interaction. The two processes — the user's brushing interaction, and the tweet stream — introduce ambiguity about what should happen when both attempt to update the visualization concurrently. For instance, consider the case when a new point streams in, falling within a brushed region. Should this new point be part of the existing selection, or should it be isolated in a new separate selection? Should the new point be blocked until the user removes the brush selection, or should the point be added and the brush selection removed? These decisions can be further complicated if the brush selection is processed by a remote server, which introduces another asynchronous event that multiplies the complexity.

For a designer to explore decision space fluidly [46], we want to to minimize the friction of experimenting with these choices. But distributed and streaming visualizations still have a high programming barrier, despite progress on declarative libraries for browser-based data visualizations. The effect is that designers get "cornered" into whatever is easiest: for example opting for a high-latency blocking design, or disallowing users from interacting with streaming data, purely for ease of implementation rather than superior usability. To improve design fluidity in these use cases, we identify two key abstractions that are missing from existing interactive visualization frameworks:

Distributed Data. Data is increasingly distributed at different compute nodes (e.g., across a browser and a remote database). As such, developers currently need to write code to access the data, which they often need to further optimize. Our goal is to develop abstractions that (1) free developers from sending data manually between local and remote processes, (2) allow developers to easily change their data storage and compute sources (e.g., between the main browser thread, a WebWorker [99], a local database, or cloud databases) without rewriting their application code, and (3) unobtrusively perform low-level optimization (e.g., caching).

Asynchronous Events. When data is local on one client, user interactions are the only events that a developer has to work with. Thus, computation caused by interactions, such as filtering data based on a selection, can be handled synchronously. However, when data is remote, asynchronous events are the norm: responses from a server, streaming data from real-world events, or events generated by other users. These asynchronous events are outside of the application's direct control and, thus, have to be handled as they arrive. Our goal is to develop abstractions that allow developers to (1) create consistent user experiences in the face of concurrency and out-of-order execution, and (2) easily experiment with different designs for handling asynchronous events.

To effectively work in this new paradigm of distributed data and asynchronous events, we adapt two patterns from the field of distributed systems programming:

Logical Constraints. Rather than imperatively manipulating data, developers express logical constraints (i.e., queries) over data, and the system determines the methods used to execute the queries [28, 113]. By decoupling the *how* from the *what*, logical constraints

allow the system to plan and optimize the execution of the query (possibly spanning multiple databases) [113], relieving the developer of this burden. This approach also has practical implications. In particular, SQL and data frame libraries express logical constraints over data, and have been widely adopted by developers and database engines [133, 108, 9] alike.

Immutable Events. Past events, along with their time steps, are stored as an immutable log by the system. Developers declare the current state of the application as a logical constraint over the immutable log rather than transiently handling events and mutating state via side effects [62, 7]. As a result, developers can coordinate events declaratively without worrying about how to update the application state through callbacks over transient events.

In this work, we bring these ideas from distributed systems programming to the context of interactive visualization. DIEL[1] models interactions and asynchronous data computation events as timestamped records in event tables, and the state of the interface as a query over the event tables and data tables. The DIEL runtime maintains an event loop to reevaluate queries as event tables change, ensuring that the interface is responsive to interactions. With DIEL, we are able to achieve the requirements outlined earlier — developers no longer need to implement low-level networking with remote databases or perform optimizations, and have a way to easily coordinate asynchronous events that maintain a consistent interface.

To evaluate DIEL's expressiveness, we construct a diverse set of interactive visualizations over distributed data and asynchronous time. Examples include ways to handle request-response asynchrony [166, 163], interactions on streaming data, composing two related interactions, and interaction scents [158, 38]. These examples show that DIEL's abstractions allow developers to rapidly and concisely explore different interface designs. We also present a heuristic analysis of the usability of these abstractions using the *Cognitive Dimensions of Notation* framework [17], highlighting both gains to fluidity, and compromises to premature commitment. Finally, we verify the viability of DIEL through performance measurements of a prototype, and show that DIEL adds low overhead and scales as data increases in size.

3.2   Related Work

DIEL builds on prior work in databases, visualization systems, and distributed systems programming.

Database and Visualization Systems. Research at the intersection of visualization and database systems has traditionally focused on enhancing performance — for instance, by offloading data aggregation and filtering to a remote database [134, 97] or by embedding a high-performance database-like query engine into the browser [122]. In contrast, our work is *not* motivated primarily by performance but instead by the programming experience of visualization developers. In particular, DIEL should be thought of as a programming abstraction

---

[1] **D**eclarative **I**nteraction **E**vent **L**og

between a database and a client. Although DIEL embeds a SQL query engine in the browser, it does so for its benefits as a *programming construct*, and its ability to interoperate with a remote database. Critically, DIEL is agnostic to the specific database system, which could range from a SQLite instance in the browser's main thread [92], to commonly used databases on a server, such as SQLite, Postgres [108], or new research systems [90, 111].

FORWARD is a general web application programming framework that directly binds database records and query results to DOM elements [41]. DIEL is similar in that it manages how elements in the visualization are updated based on changes in the underlying database that represents user interactions. However, DIEL uniquely addresses the challenges of asynchronous events, which are shown to be especially challenging for interactive visualizations [166]. DIEL does so through a novel declarative programming API over event histories.

Visual Programming over Tables. There is a close connection between visual querying and textual queries — visual querying systems combine textual query specification with direct manipulation and data visualization. For instance, VIQING maps visual selections, joins, and reordering to SQL operators [102]. Polaris endowed the pivot operator with powerful interactive capabilities [134]. Wrangler brings together interactions and a query based DSL for data transformations [72]. B2 leverages query lineage to instrument interactive cross-filtering visualizations in computational notebooks [165]. DIEL builds on these close connections using relational queries to define the data transformation logic of interactions. However, DIEL differs from these systems in that it is a programming abstraction and a middleware layer for general purpose interactive visualization programming.

Interactive Visualization Libraries. While DIEL makes use of interactive visualization libraries, such as D3 [19] and Vega [122], it is not one itself. DIEL relies on the front-end visualization libraries to map data to visual encodings, perform visualization-specific transformations (such as `voronï`, `treemap`, or `wordcloud` [144]), create interactors, and capture selection values. Existing abstractions in these libraries support interactions in a *synchronous* setting where the computation is expected to finish immediately. As a result, the events are also transient by default and assumed to be unnecessary for subsequent events. In contrast, DIEL supplements existing frontend visualization libraries by filling in the gap of working with distributed data and asynchronous events. Of particular note is how DIEL differs from Vega, which also models interaction events as streaming data. Like other libraries, events in Vega are transient by default and, although a rich set of data transformations are offered, they only operate over client-side data. DIEL, on the other hand, records all events into a persistent log and offers a simple set of relational abstractions over distributed data. Future research could consider how to further extend Vega's dataflow abstractions [125, 124] to better integrate with, or adopt DIEL's abstractions.

Frameworks for Provenance and Asynchrony. With current programming paradigms, developers need additional instrumentation to support features like logging or undo/redo.

For example, *Trrack* is a purpose-built library that augments existing interactive visualizations with a provenance graph of state history [32]. DIEL, in contrast, offers a more general-purpose set of abstractions which provide provenance tracking via first-class event histories.

Similarly, to be able to share and synchronize multiple users' changes over the network, programmers of collaborative groupware applications rely on special-purpose frameworks. Toolkits such as *Janus* help developers resolve concurrent and possibly conflicting events to ensure that each participant views a consistent artifact [126]. DIEL takes inspiration from these systems, e.g., the time-stamped events in Janus [126], but also adds specialization for interactive visualizations, such as providing support for working with distributed data and tracking request-response provenance. DIEL was directly inspired by distributed programming language research, notably the Bloom [7] language and CRDT [110] data structures. Like Bloom, DIEL focuses on the semantics of atomic timesteps immediately after the user interacts, helping developers reason about out-of-order events. DIEL differs from the prior work in its focus on reifying a log of history as a core aspect of its data model.

3.3   Programming Interactions Across Space & Time

To address the challenges of asynchronous events and distributed data, we look to distributed systems programming for inspiration and adapt the ideas to interactive visualization programming.

Asynchronous Interactions: Immutable Events

Asynchronous events, unlike their synchronous counterparts, need to be coordinated by developers to ensure a good user experience [166]. This coordination can be challenging as developers need to maintain relevant information of past events and selectively trigger event handlers. And, since different parts of the event handling and bookkeeping logic are scattered across functions and variables, these imperative bookkeeping and callbacks can become tedious to maintain and prone to errors. This situation is exacerbated when new types of events need to be supported (e.g., a new interaction). To address these challenges, we look to key methods in distributed systems programming.

Events as Data. An important technique in distributed systems programming is making events immutable and first class [62]. Events are stored as a log, and the state after each event is defined as a function (or query) of the log [3]. This design abstracts away the details of maintaining state (no more mutations with callbacks) and makes it easier to reason about the consistency of the application [7]. We can apply this framing directly to interactive visualizations: user interactions are events, and queries over these events (and other tables) can specify the state of the interface.

Atomic Timesteps. Compared to synchronous events, asynchronous events can, by defini-
tion, arrive out of order. This behavior can lead to a variety of "inconsistent" visualization
states, as documented by Wu et al. [166]. One simple example is naively rendering a response
when it is received, even if it is for an "old" interaction made prior to the most recent one.
To deal with unpredictable sequences of events, developers need to reason about what prior
events occurred, and when they did so.

Given this challenge of inconsistency, one obvious solution is to synchronize the events.
However, this solution violates a core HCI principle: *responsiveness* [68, 69]. Making events
synchronous means that the user is *blocked* from performing new interactions until their
previous interaction result has been executed completely (including being sent to the server,
computed on the server database, received by the client, and rendered to the UI). As this
process can take a significant amount of time to complete, making the interface unresponsive
yields a frustrating user experience.

To maintain a smooth user experience, it is important to work *with* asynchronous events.
Here, one idea from distributed programming is particularly helpful: *atomic timesteps* [7],
whereby the system guarantees an atomic unit of evaluation by computing the state of the
application in full before admitting another input event. With atomic timesteps, developers
reason independently, "frame by frame", about what computation needs to be computed
synchronously within a given timestep. Moreover, developers can reference the explicit
timesteps stamped on events, with the guarantee that an event with a smaller timestep
precedes an event with a larger timestep. Finally, developers can be sure that the interface
satisfies the constraints specified at every timestep, which could support robust UX in the
face of asynchrony.

Distributed Data: Logical Constraints

To be able to scale an interactive visualization application, a developer must be able to
flexibly change where the data is stored and computed upon. They may initially build
a prototype over small datasets that are stored and computed in the main thread of the
browser. To use a larger dataset, they may utilize WebWorkers [99], which allow the data-
rich tasks to run asynchronously in the background and not block the interface. Then, as the
developer deploys the application to real-world datasets, they may move the computation to
a database on their local machine or a cloud database.

To achieve this flexibility, the system should abstract data access details from developers.
The field of databases has tackled this problem using a concept that is not unfamiliar to
interactive visualization developers: *relations* (also called *tables*). Relations are sets of data
tuples with a fixed schema, and computation over relations is governed by an algebra of
select, project, join, and group by operators [28]. Relational query languages bring with
them two important properties for our purposes:

Physical Data Independence. Currently, to change from storing and computing data in the main browser thread to any other options requires custom code: developers may need to map interaction logic from JavaScript to SQL, handle the database connection, and perform networking. This work is needed for two reasons. First, developers often specify *how* to access the data. If the data location changes, the program changes. Second, the computation abstractions on the client and other locations may be different. Relational languages can address both of the factors. They allow developers to specify *what* data to access, making the program independent from the physical details of the data. Furthermore, if a standard relational language is used everywhere, be it the client, a WebWorker thread, or a remote database, the developer can work with one abstraction and not have to translate between specifications.

Rich Optimization Currently, developers may implement optimizations manually, such as caching. This is not ideal for two reasons: one is a higher programming barrier, and another is that the developer may not have enough time for more involved optimizations beyond a simple cache. Relational languages relieve the optimization burden from the developer and allow the system to compile the logical specification into a physical execution plan, using a wealth of optimization techniques [113].

3.4 The DIEL Model

To address the challenges in Section 1, DIEL manages user interactions, remote databases, and the communication between local and remote. On the browser frontend, the developer specifies the data that the user can interact with and select (e.g., using Vega-Lite [122]) and translates them into events that are sent to DIEL via its JavaScript API. DIEL stores events in the local in-browser local database and manages query processing on the remote databases, which are connected to DIEL through a set of remote-side APIs. Through standard database connection libraries such as `node-postgres` and `sqlite3`, DIEL allows the developer to connect to remote databases ranging from SQLite, to PostgreSQL, to cloud databases like Amazon Redshift.

Note that remote databases and their asynchronous complexities can arise in surprising settings. In addition to databases on remote servers (such as Redshift), remote databases also arise when data is managed by a different process on the user's computer. For instance, browser WebSockets communicate with the web page's main thread via asynchronous messaging. For this reason, DIEL is designed to work for any distributed database setting, irrespective of whether the databases are on the cloud.

The developer provides a specification that DIEL uses to coordinate query processing between the local and remote databases. The specification defines the application state that DIEL will manage, and is written in a SQL-like language. We chose SQL as the basis because it is the most widely used data processing languages today [103], and is familiar to
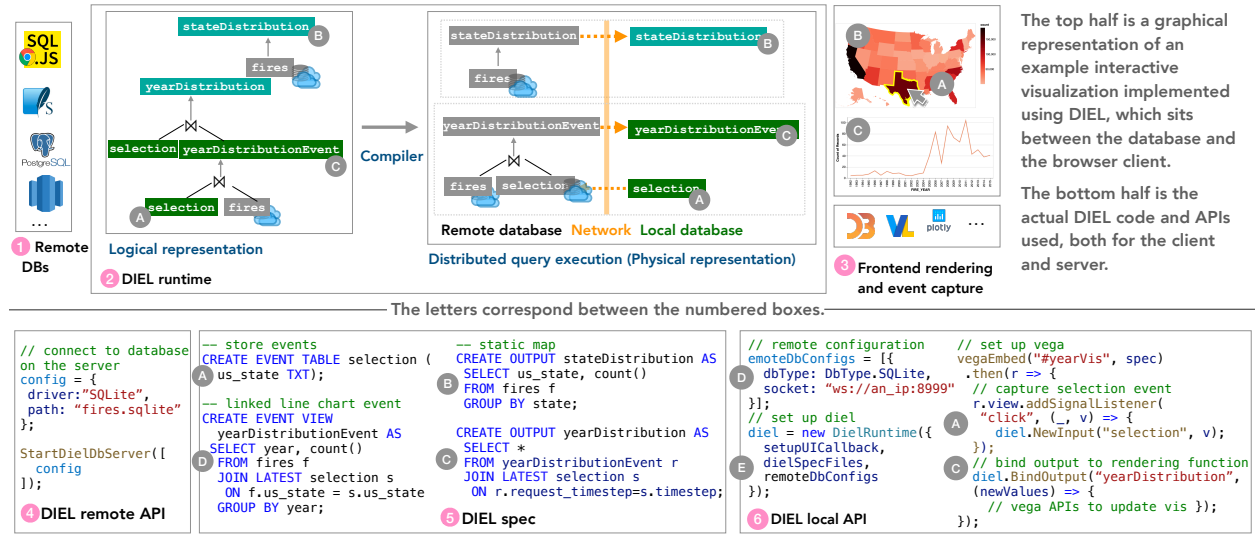
Figure 3.2: Example code using DIEL to power the interactive visualization of wild fires in the US. The user can click on a US state to filter the distribution of fires over the years (3). To create this visualization, developers use DIEL in conjunction with data visualization libraries and remote databases (1). Developers query over timestamped event logs (`selection`) and base dataset (`fire`) to specify the state of the application (a logical spec (5)), and DIEL orchestrates the computation between the local and remote databases (an execution plan (2)).

developers working with large datasets. However, DIEL is not tied to SQL, and alternative relational languages. It is straightforward to layer a data frame-like API atop DIEL's abstractions. Given the specification and runtime API calls, DIEL coordinates the local and remote database to exchange data and query results. The query results then update visualization state back to the application via a JavaScript API so that it can render results and update the visualizations.

We next discuss how developers specify different types of application state, and how it is managed during run time. Fig. 3.2 depicts the running example, and Tables 3.1 and 3.2 summarize the syntax.

Data Model

Static Visualizations Static visualization is a well-accepted domain, where tabular data is fed to APIs that implement a grammar of graphics [122, 157]. For static visualization, DIEL is responsible for preparing the data to be rendered, and invoking the visualization API when the data is ready. The tables used to create visualizations could either be stored tables or the results of queries. For instance, in Fig. 3.2, the map visualization maps the

keyword	syntax CREATE
`event`	`EVENT TABLE <table_name>(<column>)...`
	`EVENT VIEW <view_name> AS SELECT...`
`output`	`OUTPUT <output_name> AS SELECT...`

Table 3.1: DIEL syntax for creating tables. An `EVENT TABLE` stores the history of an event. Developers can access the table with two additional columns maintained by DIEL: *timestep* (logic time of the event) and *timestamp* (wall-clock time of the event). An `EVENT VIEW` is a named SQL query (view) that spans the local and server databases. Developers access these views the same way they access event tables, with an additional column *request_timestep*, which indicates when the view was requested. An `OUTPUT` is a view whose results, after each timestep, are evaluated and passed to the rendering function bound via the `BindOutput` API.

fields (`us_state, count`) from the query result in (5B) to polygons and fill color in the visualization. When the application is first loaded, DIEL evaluates the initial queries to render the static visualizations. Using the `BindOutput` API (6D), the developer registers a callback *rendering function* that is called when the query results are updated. The developer annotates the queries that the rendering function will access using the `OUTPUT` keyword. The system will use the information to inform the dataflow as well as instrument the `BindOutput` API.

Events DIEL stores events as timestamped tables. When the developer defines visualization interactions, such as selections, the data records that are selected are ingested into DIEL through the `NewEvent` API and stored in tables. For instance, the selection on the map (3A) is mapped via a Vega listener (6A) and stored in the `selection` table (5A), with a single column, `us_state`, representing the state selected, e.g., "CA". An event may be one or more rows. At runtime, each new event is augmented with a *logical timestep*, which we discuss in Section 3.4 , and a *physical timestamp*, recording the wall-clock time of the event. Other external events, such as those generated from an automated process in the browser on behalf of the user (e.g., timers), follow the same model.

Developers mark event tables with the `EVENT` keyword (5A). DIEL generates the corresponding JavaScript handler for the API `NewEvent`. Developers are free to use their favorite frontend libraries (e.g., Vega-Lite, D3) to generate events; they simply invoke the DIEL API to store new events in the event table.

Interactive Visualizations User interactions add events to DIEL event tables that ultimately update the visualizations. This happens naturally because the developer can query the `EVENT` tables in the same way as any normal data table. Any such queries marked with `OUTPUT` will automatically generate the corresponding `BindOutput` API to register a rendering function callback (used in e.g., 5D).

API	syntax
input	`diel.NewEvent('<event>', <object>)`
output	`diel.BindOutput('<output>',`
	`<rendering_func>)`

Table 3.2: DIEL JavaScript APIs to interface with the frontend. `NewEvent` takes in events from the developer, e.g., user's selection. `BindOutput` binds the outputs to a rendering function specified by the developer that takes a table and renders a visualization.

When a developer is working with a remote database, they need to query *across* both the local and the remote databases. For instance, the `fires` table is on a remote database, whereas `selection` is in the local database. DIEL automatically manages the asynchronous communication between the two databases. Due to this asynchrony, we treat such queries as events as well, and ask developers to explicitly mark these queries with `EVENT VIEW` so they are aware of the table being an event log (despite the fact that DIEL can automatically detect them). For each `EVENT VIEW`, DIEL maintains the corresponding timesteps of the event that caused its reevaluation, and stores the timestep values as a column named `request_timestep` in the `EVENT VIEW`. Developers then use the data in `EVENT VIEW`s to specify `OUTPUT`s, with the help of the timesteps to specify how asynchronous events should be handled in the face of out-of-order events.

For example, the line chart in Fig. 3.2 (3C) visualizes the distribution of fires by year. The query in (5D) fetches the distribution of fires based on the `us_state` selected: it combines the `fires` and `selection` event tables using the `JOIN` operator, and then filters the rows in `fires` by applying the filter on the selected `us_state` selected using the `WHERE` operator. (5C) is a query that selects the year distribution results of the most recent interaction is selected, based on the `timestep` data. This ensures that the interface displays the correct data despite out of order responses.

DIEL Execution

DIEL orchestrates the local database and one or more remote databases in response to new events. Each event is evaluated *synchronously* in the local thread, corresponding to one *logical timestep*. As a result, the application-level effects of events with more recent logical timestamps are guaranteed to occur after events with earlier logical timestamps. From the perspective of the local database, user interactions and responses from remote databases are both events, however they are handled slightly differently, as described in Fig. 3.3.

Interaction Event: The application creates new events from user interactions (1a), and inserts them into the corresponding `EVENT TABLE` using the `NewEvent` API call (1b). DIEL recomputes the local `OUTPUT` tables, and triggers the corresponding render function callbacks
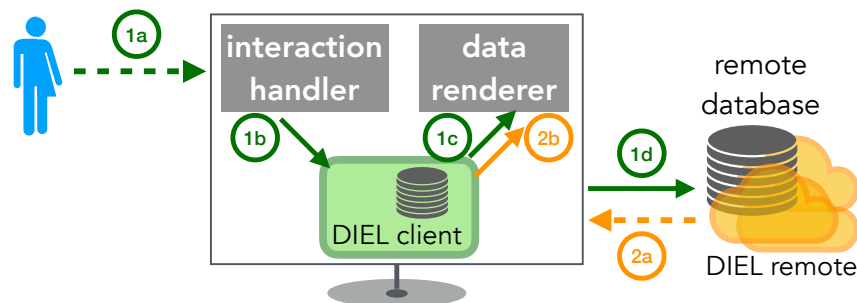
Figure 3.3: Illustration of the DIEL runtime. The dashed arrows indicate asynchronous events that advance the system logical timestep forward. There are two in this diagram, one from user interaction (green) one from a database response (orange). The solid arrows indicate synchronous evaluation after each new event.

(1c). Finally, DIEL handles any `EVENT VIEW`s that depend on the new event. If it accesses remote data, DIEL first sends the appropriate query to the remote database and handles the response as described below (1d). Otherwise, DIEL executes the local `EVENT VIEW` query.

Data Response Event: Once the local DIEL runtime receives a response from a remote database (2a), which is tagged with the originating event's timestep, it calls `NewEvent` to insert it into the `EVENT VIEW`. Finally, the `OUTPUT`s are reevaluated, and the results are sent to the application rendering functions.

3.5   A Prototype Implementation of DIEL

To verify the feasibility of the DIEL model, we implemented a prototype. It ingests a DIEL specification and produces a distributed dataflow that is executed in conjunction with the frontend JavaScript libraries and backend databases. There are two main challenges in this process. First, queries over distributed data cannot be executed directly since the computation requires data to be co-located. Second, the DIEL model poses performance challenges.

An implementation should both create a dataflow such that necessary data is exchanged between the local and remote databases, and overcome these performance hurdles to keep the interface responsive. As such, DIEL builds the distributed dataflow in four phases. The first three phases address the challenge of distribution, and the last phase address the challenge of performance.

Two libraries manage these steps: (1) a local (browser-based) library updates the JavaScript state, and (2) a remote library configures access to the remote database. The developer provides these respective libraries with the connection configurations (e.g., Fig 3.2 (4) and (6)).

In the first step, DIEL parses and compiles the specification into an abstract syntax tree that captures the relational operations between tables. The tree diagrams in Fig. 3.2(2) (left) illustrates that of the running example. With the tree of operators, DIEL can now reason about the data exchange needed to co-locate tables for evaluation.

To reason about the exchange, DIEL next identifies the tables and their schema in the remote databases, i.e. a table catalog. Then, with the catalog, DIEL identifies queries that involve tables on both the local and remote databases. DIEL determines, for such queries, what data need to be exchanged between these databases. This process transforms the high-level logical specification into a distributed dataflow that can now be executed.

An example of such a transformation done by the current prototype is shown in Fig. 3.2(2) (right). DIEL decides to perform the evaluation of `yearDistributionEvent` on the remote database, thus requiring the `selection` table to be sent over the network to the remote database from the local database, and the result `yearDistributionEvent` to be sent over the network from the remote to the local database. The output `yearDistribution` is then evaluated within the local database upon each step. Note that this is not the only distributed dataflow possible. Alternative implementations of DIEL could instrument more advanced algorithms that can further optimize for performance by changing what data needs to be shared over the network . This can be done by partitioning the execution of queries into more granular subqueries.

In the final phase, DIEL optimizes the physical execution plan from the previous step. Four mechanisms are used.

1. Selective output invocation. Interactive dashboards often contain multiple visualizations, and sometimes ones with many visual elements that are expensive to render. A naive plan will reevaluate all the queries and re-render all the visual marks, which could block the main browser thread from responding to user events. To address this issue, the DIEL prototype analyzes the execution plan and builds a dependency graph of the DIEL queries. Using the dependency graph, DIEL selectively evaluates and invokes rendering functions for only the output views dependent on the current event.

This issue can be addressed via static analysis of DIEL queries at compile time. From the queries, we can extract dependencies across queries, views and tables syntactically. For example, in Fig. 3.2, there are two outputs—`stateDistribution` and `yearDistribution`. Looking closely at 3.2 (2), note that `selection` "flows into" `yearDistribution`, but does *not* flow into `stateDistribution`. This reflects the syntax of those outputs in 3.2 (5), where `selection` appears only in the definition of `yearDistribution`. Using this information, DIEL knows to only selectively update `yearDistribution` when there is a new user interaction (`selection`).

2. Materialized intermediate views. One interaction can require updating multiple visualizations. Data transformations are sometimes shared when computing these updates. For

```
CREATE VIEW filtered AS
 FROM fires f
  JOIN LATEST selection s ON f.state = s.us_state;
```



```
CREATE EVENT VIEW
 causeDistributionEvent AS
 SELECT cause, count()
   FROM filtered GROUP BY cause;


CREATE EVENT VIEW
 yearDistributionEvent AS
 SELECT year, count()
   FROM filtered GROUP BY year;
```
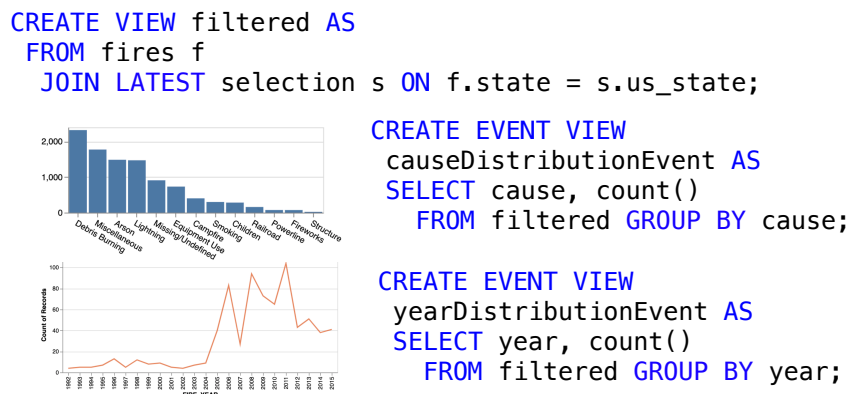
Figure 3.4: A example of reuse and materialization. The filtering logic based on the selection in Fig. 3.2, `filtered`, is shared by two visualizations. DIEL analyzes the query plan and materializes `filtered` to avoid evaluating it twice.

instance, the `filtered` view in Fig. 3.4 is shared by two outputs. Since views are just named queries in SQL, this can lead to redundant work: whenever the database evaluates any query that references a view, the view is reevaluated as part of the query. This wasteful reevaluation can be prevented if the view results are materialized into a table for use by downstream queries [25]. For materialized views to work correctly, they must be updated appropriately when the data they query changes. We tackle this problem pragmatically in our prototype: we implement update mechanisms for views in the our local database code; for remote databases, we count on their native support for materialized view maintenance.

3. Automatic caching. Developers often build a client-side cache by hand to amortize the compute and network time to a server. For any given specification, DIEL can automatically instrument this functionality, thus saving the developers' time. The server response data are already stored in the event log. To make use of the past values, DIEL analyzes the execution plan at this stage and instruments a layer of indirection to the evaluation of the EVENT VIEWs when a new event arrives: DIEL hashes the parameters of the relevant rows of the event tables and looks up the hash in an *event cache table* instrumented by DIEL. If the hash is present, DIEL returns that value, and if not, DIEL dispatches the dataflow. DIEL's automatic caching process saves storage by not having multiple copies of the responses, which is especially limited on the client. In addition to helping save the developers' time, DIEL's automatic caching process also alleviates the additional memory usage caused by DIEL's log of events.

4. Automatic Index Selection. Good database performance is typically tied to building indexes that suit your query workload. For a given DIEL program, this workload is known in advance as part of the spec, hence DIEL analyzes the query structure to statically determine indexes that will result in good performance.
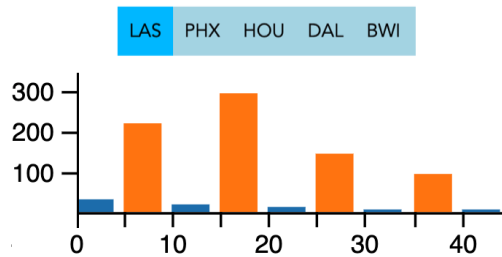
Figure 3.5: Flight distribution, where the blue is flight count filtered by LAX, and the orange is the overall flight delay.

Besides optimizing the dataflow, we also optimized the execution speed of the local database. Unlike the remote databases, the implementation of the local database is controlled by DIEL. The local database runs in the browser's main-thread and any delay would directly block user interactions and HTML updates. Therefore, it is critical that the local database executes quickly. A recent advance in browser technology, WebAssembly [155], allows for fast execution of SQLite, which we use as the local database, through optimized transpilation.

Additional Language Support

Through using and observing others use DIEL, we have identified and implemented three extensions to DIEL that improve usability.

Syntax Sugar with Templates and Defaults

Programmers can reuse their DIEL specifications through SQL VIEWs, but VIEWs do not support all the cases for code reuse. For instance, Fig. 3.5 shows a visualization where the developer need to provide the distribution of the flight delays by the origin airport. Suppose that they have already written the logic to compute the distribution over all the airport in a query `distributionAll`. To define the filtered query `distributionFiltered`, there is no way to reuse `distributionAll`, because the origin airport column is now filtered out. The query would have to be written from scratch.

To facilitate reuse, we implemented a simple **templating engine**. The syntax is shown in Listing 3.2 line 2 and 4. We illustrate the template's use with an example. To now add the additional filter of the origin airport, instead of copying the entire query `flightDelayDist` from Listing 3.5 and adding the filter, the developer can use the DIEL *template* feature, shown in Fig 3.1. Here, the `distributionTemplate` template allows the developer to define a table as a variable called `variable_table` in line 3. In the case of the flight without the airport filter, the `variable_table` parameter is `flight`, and in the case of the flight filtered by the airport, `filteredFlights`.

```
1  CREATE TEMPLATE distributionTemplate(variable_table) AS
2    SELECT ...
3    FROM {variable_table}
4     JOIN LATEST zoomDelayItx z
5    ...;
6
7  CREATE EVENT TABLE originSelectionInteraction (origin TEXT);
8
9  CREATE VIEW filteredFlights AS
10   SELECT *
11   FROM flights
12    JOIN LATEST originSelectionInteraction
13     ON origin;
14
15 CREATE OUTPUT distributionAll AS
16   USE TEMPLATE distributionTemplate(variable_table='flight');
17
18 CREATE OUTPUT distributionFiltered AS
19   USE TEMPLATE distributionTemplate(variable_table='filteredFlights');
```

Listing 3.1: Example templating feature in DIEL that promotes code reuse in cases of filtering. The omitted select clause on line 2 to 4 is the same as `flightDelayDist` from Listing 3.5 line 2 to 9, with the `flights` table replaced by the template variable `variable_table`.

```
1  -- define template
2  CREATE TEMPLATE <template_name>(<var_name>) AS SELECT...;
3
4  -- use template
5  USE TEMPLATE(<variable_name>=<value>)
6
7  -- copy schema
8  CREATE TABLE <new_table_name> AS <existing_relation>;
9
10 -- LATEST expanded
11 SELECT... FROM <relation> WHERE timestep = (
12   SELECT MAX(timestep) FROM <relation>);
13
14 -- state programs (side effects)
15 CREATE PROGRAM AFTER(<event1>, <event2>, ...)
16 BEGIN
17   INSERT INTO <history_table> ...;
18   SELECT <udf>(), ...;
19 END;
```

Listing 3.2: Different DIEL extensions to improve programmer experience: templates, schema copy, LATEST, and state programs.

Side Effects with History Programs

In DIEL, all tables are append-only logs through the invocation of the JavaScript API `NewEvent`. However, we found that in more advanced cases, such as undo, it can be use-

```sql
1 │ CREATE EVENT TABLE clickInteraction(id INT);
2 │ CREATE EVENT TABLE undoInteraction();
3 │ CREATE TABLE allSelections(id INT);
4 │ -- record the items at every timestep into state
5 │ CREATE PROGRAM AFTER (clickInteraction, undoInteraction)
6 │   BEGIN INSERT INTO allSelections SELECT * FROM currentSelection; END;
7 │ -- derive current selection base on both the click and the undo
8 │ CREATE OUTPUT currentSelection AS
9 │   SELECT COALESCE(s.id, e.id) AS id
10│   FROM LATEST clickInteraction e
11│     LEFT OUTER JOIN curUndoSel s ON 1;
12│ -- using basic math to derive the selection base on undo
13│ CREATE VIEW curUndoSel AS
14│   SELECT * FROM allSelections AS s WHERE rowid = ((
15│     SELECT MAX(rowid) FROM allSelections
16│   ) - (
17│     SELECT COUNT() * 2 - 1
18│     FROM undoInteraction u JOIN LATEST clickInteraction c
19│       ON u.timestep > c.timestep
20│   ));
```

Listing 3.3: Linear undo in DIEL for click selections.

ful to be able to have some "side-effects", where developers can modify state more flexibly.

We implemented this functionality, called *state programs* (after [3]), and the tables they insert to are *history tables*—DIEL augments the history tables with the timestep column, but does not invoke the event loop.

We illustrate its use through a simple example of undo, which is a common interaction [170]. For simplicity, we discuss an example linear undo implementation (used in applications such as Emacs [109]). In this example, the user clicks on a visual mark to select, and the undo re-selects the previous selection. Say the user clicks on A, B, and C, and then presses undo twice. The sequence of actions is (A, B, C, Undo, Undo), and the user will see the sequence of selections: (A, B, C, B̲, C̲), where B̲ means that B is selected due to an undo action.

Listing 3.3 provides an implementation. Since an undo is another interaction event, it is defined as an EVENT TABLE; allSelections will track the sequence of selections as described above, where the rank is simply the sequential id. The *state program* populates allSelections with the actual selection for each logical timestep (i.e., each click and undo event). To compute the current selection, it first checks whether the most recent interactions were undo events or normal clickInteraction. If there are undo events, then it subtracts their count from the maximum rank in allSelections to identify the selection that the undos represent (e.g., B̲ after the first undo above).

```
1 CREATE VIEW filtered_view AS
2   SELECT a1, a2
3   FROM t1 WHERE a1 > 10
4   CONSTRAIN a1 NOT NULL;
```

Listing 3.4: Example Constraints

Debugging with Constraints

Logic programs can be difficult to debug when the error manifests downstream from the code containing the bug. For instance, if the developer accidentally filtered on the wrong value in a view *V1*, and output *O* references *V1* and *V2*. If the developer is debugging missing results from *O*, they need to inspect *both* of the views. The situation can be improved with constraints. We have implemented this feature in DIEL. Listing 3.4 shows one such constraint syntax.

3.6 Evaluating DIEL's Expressivity

To assess the benefits of DIEL, we show examples of interactive visualizations that deal with the variety of challenges that arise when managing asynchrony and distribution, including coordinating responses from remote servers, streaming data, and composing interactions. Our discussion focuses on the aspects addressed by DIEL and omits implementation details of the frontend. We also show how DIEL specifications compose in a modular manner, by building each new interaction on the running example of Fig. 3.2. Since the challenges are independent of the particular choice of visual encodings, we do not focus on varying the visual designs.

Coordinating Requests and Responses. Latency from remote databases' responses can cause inconsistencies in the interface, if not handled properly [166]. For instance, Fig. 3.6 shows a timeline of user interactions (`selection`) and server responses (`yearDistributionEvent`). The user clicks `TX` and then `CA`, but the remote database responds with the result for `CA` first. Rendering the most recent result (`TX`) would surprise a user who is expecting `CA`. To avoid inconsistent interfaces, developers can use DIEL to specify a range of designs in a few lines of code. We walk through three possible designs shown in Fig. 3.6.

Option Ⓐ always displays the result of the most recent interaction. The DIEL spec first identifies the interaction by selecting the rows with the highest timestep (`LATEST selection`), then selects the rows in `yearDistributionEvent` with matching request-response timesteps (`d.request_timestep = e.timestep`). If no match is available yet, nothing is returned. The front-end visualization logic could indicate as such, e.g., using a spinner as shown.

Option Ⓑ always displays the most recent response and its corresponding selection,

```
CREATE OUTPUT yearDistribution AS
  SELECT *
  FROM LATEST selection e
  JOIN yearDistributionEvent d
    ON d.request_timestep=e.timestep;
```

```
CREATE OUTPUT yearDistribution AS
  SELECT d.*, e.us_state
  FROM LATEST yearDistributionEvent d
  JOIN selection e
    ON d.request_timestep=e.timestep;
CREATE OUTPUT pending AS
  SELECT state FROM selection WHERE timestep NOT IN
    (SELECT request_timestep
      FROM yearDistributionResult);
```

```
CREATE OUTPUT snapshots AS
  SELECT *
  FROM selection s
  JOIN yearDistributionEvent y
    ON y.request_timestep=s.timestep;
```

Figure 3.6: Designs to coordinate asynchronous requests and responses when querying over distributed data: Ⓐ renders the most recent interaction requested; Ⓑ renders the most recent response received as well as any *pending interactions*; Ⓒ renders snapshots of all interactions and their corresponding results [163].

as well as pending selections. The DIEL spec first selects the most recent response (LATEST yearDistributionEvent), then joins it with the selection table to retrieve the corresponding value of the selection (e.us_state) by matching their timesteps. The second output pending represents pending selections and is computed by finding the request(s) that do not have a corresponding response.

Option Ⓒ displays "snapshots" of all interactions [163], where past interactions and their results are scaled down into a scrolling pane of small multiples at the bottom. The DIEL spec simply selects all the interactions from the event table selection, joined with the corresponding responses by their timesteps. The snapshots allow a user to interact with the visualization and navigate to prior states, concurrent with the loading of new responses.

Without DIEL, implementing these designs would require the developer to manually keep track of events — store the points of interest selected, their respective responses, and the global ordering of all the events — and coordinate multiple event handlers. While each step is simple in isolation, put together the complexity of this low-level data-recording and event handling compounds substantially. Developers may have trouble reasoning about the *overall* design.

DIEL, on the other hand, encourages a consistent experience by asking the developer to specify which of the events should be in the output at any given time. There is no accidental design resulting from interrupt-driven event handling, such as immediately rendering whatever response arrives. Furthermore, DIEL takes care of recording event history and provenance. The developer can query these data directly — for example identifying the responses from remote databases with timestep data that DIEL automatically maintains. As a

```
CREATE EVENT          CREATE OUTPUT fireMap AS
TABLE incident(       SELECT f.lat, f.lon
 id TXT,              FROM incident f
 type TXT,            WHERE type='new'
 lat FLOAT,            AND f.id NOT IN (
 lon FLOAT,            SELECT f.id
 time INT);            FROM incident
                       WHERE type='contained');
```
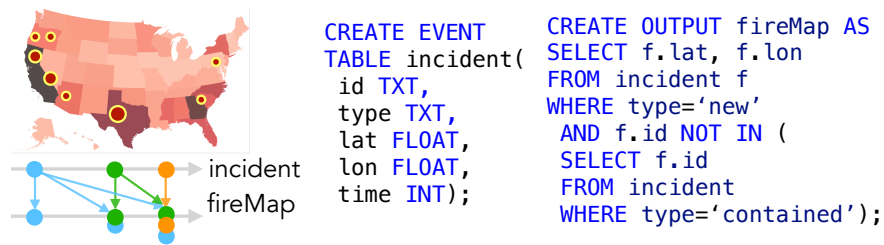
Figure 3.7: Example DIEL spec for the symbol overlay of active fires, determined by selecting `incidents` that do not yet have a row with the column `type` of `contained`.


```
CREATE EVENT TABLE brush(    CREATE VIEW brushedIncidents
  min_lat FLOAT,             AS SELECT m.id
  max_lat FLOAT,                FROM fireMap m
  min_lon FLOAT,                JOIN LATEST brush b
  max_lon FLOAT);                 ON contained(m.*, b.*)
```
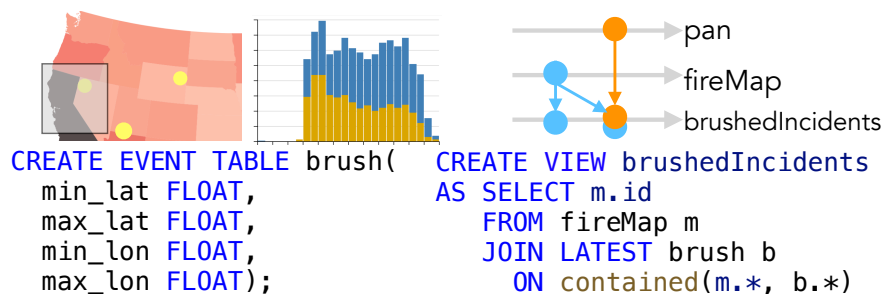
Figure 3.8: Example DIEL spec for brushing interaction: `brushedIncidents` selects fires in the symbol map `fireMap` that falls into the brushed region `pan` to update the bar chart (query omitted).

result, the developer can iterate on alternate designs without any instrumentation overhead.

Streaming Data. Given the real-time nature of fires, the developer may want to incorporate streaming data into their visualization. Fig. 3.7 overlays the choropleth in Fig. 3.2 with a symbol map of active current fires. The event `incident` contains the location of the fire and whether it is new or contained. When a new `incident` event arrives, fires are either added to or removed from the symbol map overlay. To implement this design with DIEL, the event `incident` is captured as an event table and the data for the overlay is captured by the output table `fireMap`. Each tuple from the latter is used to query the `incident` table to identify fires that have not yet been controlled, via the `NOT IN` subquery. Note how the developer can rely on a few lines of DIEL code, instead of programming custom JavaScript functions to store and manipulate streaming events[2].

Composing Events: Interaction and Streaming. Cross-linking is a common interaction technique [130]. Fig. 3.8 shows a new brush selection added to the map visualization, linked

---

[2] As with materialized views in Section 3.5, we take a pragmatic approach in our prototype: we handle streams in the client database, but do not support streaming in the remote databases.

```
CREATE EVENT TABLE pan AS brush;
CREATE VIEW effectiveBrush AS
  SELECT b.*
    FROM LATEST brush b, LATEST pan p
A WHERE b.timestep > p.timestep
B WHERE contained(b.*, p.*);
```

brush

pan

effectiveBrush
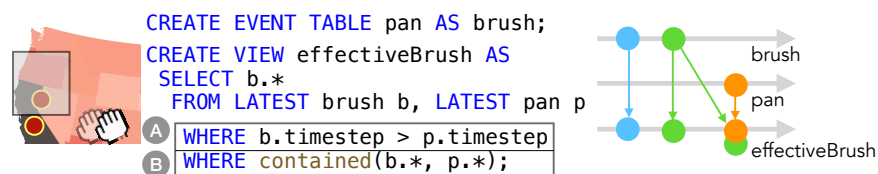
Figure 3.9: Example DIEL specification of composing two interactions: panning and brushing (from Fig. 3.8). There are two ways to coordinate: Ⓐ removes the brush selection whenever there is a more recent panning event, and Ⓑ removes the brush only when the brush is panned out of view.

.

to the bar chart shown to the right (the query not shown for brevity). `contained` checks if the `lat, lon` bounds in `fireMap` are within the `min` and `max` of the brush. It is a utility function defined by DIEL, using the *user defined function* construct in SQL [113]. The new interaction composes with the streaming `firemap` view from before — if there is a new event received that falls into the brushed area, the incidents selected in `brushedIncidents` will also be updated, and any dependent output views will be updated as well. This subtle instrumentation, automatically performed by DIEL, may be difficult for a developer to catch in a traditional implementation where the logic may be dispersed into different event handlers.

Composing Events: Brushing and Panning. Different interactions serve different purposes and more than one interaction could be employed for the same visualization, which the developer may need to coordinate. Following the running example, suppose now the developer wishes to introduce a pan-zoom interaction, shown in Fig. 3.9. The `brush` table is defined in geographic coordinates, so the user can pan the map to an area where the brush is no longer visible, and the value of the selection is in question. We present two possible designs to address this ambiguity. Both derives a new brush, `effectiveBrush` for use in place of the raw `brush`.

Option Ⓐ invalidates the brush when the user initiates a new panning interaction. The DIEL spec selects the most recent brush (`LATEST brush`) only if it occurred *after* the most recent pan (`LATEST pan`). If we replace option Ⓐ with option Ⓑ, we instead invalidate the brush only when a new panning event moves the brush *out of view*. In either case, developers do not manually modify the callbacks to the panning interaction handler to check and remove the previous brush selection. The "removal" is specified declaratively and enforced implicitly by DIEL as the logical timesteps progress.

Interaction Scent. Research has shown the benefit of visualizing interaction history: *Hind-Sight* visualizes the user's prior interactions in the visualization [38]; *Scented Widgets* visualize interactions by other users [158]; *Interaction Snapshots* visualize historical interactions

```
Ⓐ  CREATE OUTPUT brushScent AS
     SELECT UNIQUE(b.*) FROM brush b;
```

brush/selection
brushScent/selectionScent

```
   CREATE OUTPUT selectionScent AS
    SELECT us_state, count()
     FROM selection_all
     GROUP BY us_state;
```
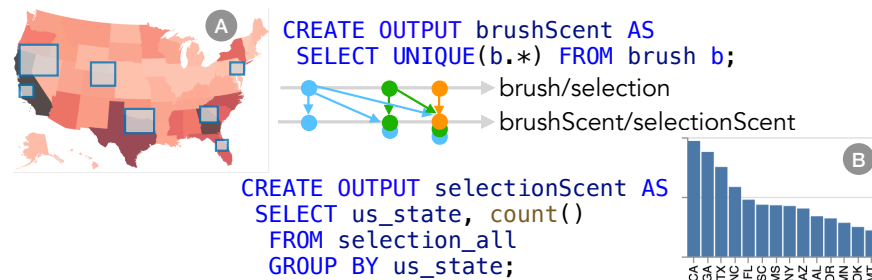
Figure 3.10: Example DIEL spec for Ⓐ Hindsight: select all unique brushes in the `brush` event table, and Ⓑ Scented Widget: a bar chart that shows the frequency of selections across users.

and their results [163]. We have shown an example of *Interaction Snapshots* in Fig. 3.6; we now discuss the other two designs in Fig. 3.10.

Ⓐ instantiates a *HindSight* design, where all prior brushes are shown [38]. The DIEL spec for `brushScent` selects unique brushes through the `UNIQUE` operator. Ⓑ shows an example design of *scented widgets*, where a histogram of all prior `us_state` selections are aggregated and counted [158]. The `selection_all` is a table in the database that stores all the event tables from each session, which is saved by the developer from event tables to the remote databases (and not handled automatically by the current DIEL prototype), e.g., `INSERT INTO selection_all SELECT * FROM brush`.

With DIEL, the data backing these visualizations is already in event tables ready to be queried, and the visualization scents are automatically updated. Without DIEL, the developer would have to manually store the events and re-render the visualization scents after each interaction, or use libraries like *Trrack* [32].

More Advanced Usage of DIEL

Since DIEL uses SQL as its host language, DIEL inherents SQL's advanced features as well. We illustrate with a few examples.

Dynamic Histogram Zoom.. When zooming into a histogram, the user may want to see a more fine-grained view of the distribution. Listing 3.5 provides an example flight delay distribution. `zoomInteraction` captures the selection, and the bin size dynamically adjusts to the size of the selection.

Drop Down Menus. Drop down menus are a common design in interactive visualization dashboards. Listing 3.6 shows an example where the user could toggle between the origin or destination column when sorting the table. The toggling is implemented with the `CASE...WHEN` clause supported by SQL. Another SQL feature we leverage is `CHECK`, a constraint
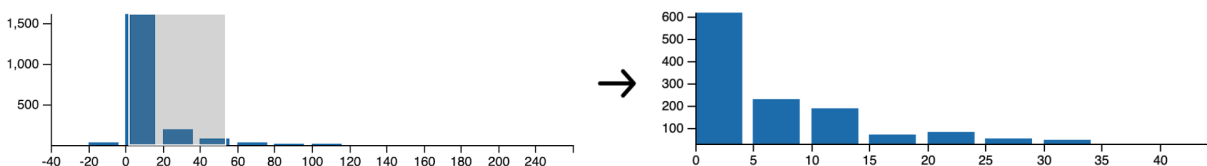
Figure 3.11: Example zoom interaction. Where as the user zooms in more detail, the bin size gets more granular.

```
1  CREATE EVENT TABLE zoomInteraction(minD INT, maxD INT);
2
3  CREATE VIEW flightDelayDisttribution AS
4    SELECT
5      ROUND(delay/((z.maxD-z.minD)/10))
6               * ((z.maxD-z.minD)/10) delayBin,
7      COUNT() count
8    FROM flights JOIN LATEST zoomInteraction z
9    GROUP BY delayBin
10     HAVING delayBin < z.maxD AND delayBin > z.minD;
```

Listing 3.5: Example zoom interaction (Fig 3.11) where the more the user zooms in, the smaller the bin size.

checking mechanism that makes sure that the `column` entry is well defined (Line 2).

Dynamic Sample Size. Listing 3.7 shows an example where the user can sample from the table, and adjust the sample size. Line 2 ensures that all invalid user entries are ignored, and line 5 randomly orders the data to avoid skew.

3.7   Evaluating DIEL's Performance

We evaluate the feasibility of the DIEL model using our prototype implementation. We focus on the overhead that the DIEL middleware introduces during initial setup and user interactions, as well as its ability to scale to large datasets as compared to leading visualization libraries.

We used Kaggle's wildfire dataset [1], creating the visualization shown in Fig. 3.2, and conducted evaluations on a MacBook Pro with 2.7 GHz Quad-Core Intel Core i7 and 16 GB of memory. In the experiments, the "remote" database server is a SQLite process running on a web application deployed on Heroku ("Free Dynos" tier with 512 MB of RAM). The network bandwidth was 18.5 Mbps. Changing the dataset size or database will not affect the

```
1  CREATE EVENT TABLE columnSelectionInteraction (
2     column TEXT
3     CHECK
4       column='origin'
5       OR column='destination'
6       OR column='delay'
7    );
8
9  CREATE OUTPUT flightTable AS
10    SELECT f.*
11    FROM flights f JOIN LATEST columnSelectionInteraction i
12    ORDER BY CASE i.col
13      WHEN 'origin' THEN origin
14      WHEN 'delay' THEN delay ELSE destination
15     END;
```

Listing 3.6: Example DIEL code that reconfigures which columns to sort by. The `CASE` clause is a SQL syntax shortcut that helps developer express the logic dynamically change the selection based on some predicates.

```
1  CREATE EVENT TABLE sampleSizeInteraction (
2     sizeSelection INT CHECK size > 0
3  );
4
5  CREATE OUTPUT flightSample AS
6     SELECT * FROM flights ORDER BY RANDOM()
7     LIMIT (
8       SELECT sizeSelection
9       FROM LATEST sampleSizeInteraction);
```

Listing 3.7: Example DIEL code that reconfigures the number of rows to be seen. The developer can dynamically control how much data is shown based on user input, achieved by limiting the number of row by a sub-query (line 6).

overhead that the DIEL middle-ware incurs. Thus we chose to focus on these three evaluation configurations.

Initialization. Fig. 3.12 shows the time taken to setup DIEL. The *local* setup only involves a local database in the main browser thread. DIEL (1) sets up the synchronous database in the main thread; (2) compiles the DIEL spec into logical representations, including optimization steps such as caching; and, (3) sets up the views in the local database. The *remote* setup accesses the remote database server. DIEL shares step (1) and (2) of the local setup, then it builds a *catalog* of tables in remote databases and sets up a distributed dataflow based on the catalog. The *remote+network* results are the same as *remote*, but with the addition
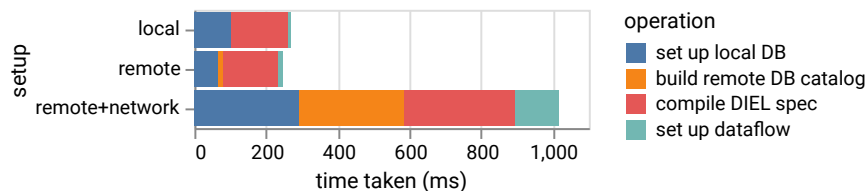
Figure 3.12: The x-axis represents the respective time taken for DIEL's initialization steps, in milliseconds, and the y-axis represents the type of setup being measured.

of nontrivial network communication overheads. These introduce fixed costs—such as the 200ms overhead to fetch the SQL.js library in order to set up the local database—as well as variable costs to send results between the client and server. These costs depend on the network bandwidth. For instance, decreasing the network bandwidth to 1.4 Mbps caused the set up local database to be 8687ms.

In all three cases, we see that the initialization time does not pose a usability challenge, especially given that the initiation is a one time cost at the beginning of loading the web page containing the interactive visualizations.

Event Handling. Fig. 3.13 shows the time taken to handle events. One type of event is user interaction, which took less than 5 ms to handle. During that time, DIEL (1) saves the input event to the local database (serializing from JavaScript to SQLite); (2) outputs the view data (deserializing from SQLite to JavaScript); and, (3) networks with the remote database. Another type of event is a remote database response, which takes about 20 ms to handle. During that time, DIEL (1) processes the remote message; (2) saves the input event to the local database; and, (3) outputs the view data. Processing the data from the remote database and serializing that data into the local database takes the bulk of the time. We see that the overhead in either case is well under the limit of 100 msec prescribed by Card et al. to sustain perceptual causality [21].

Different Data Sizes. To demonstrate a DIEL program's ability to scale, we created samples from 10 thousand rows to 4 million rows and benchmarked two implementations of the visualization shown in Fig. 3.2: one with DIEL (the SQLite server and static Vega visualizations) and one with Reactive Vega running in the browser. We measured the time taken between the handler receiving the interaction and outputting the computed result. This excludes the rendering logic, which is common across the experimental runs. The SQLite and DIEL results also exclude the network latency since it is shared in both cases.

Fig. 3.14 shows that DIEL is able to handle the increasingly large data by leveraging resources beyond the client, whereas a client-only Reactive Vega application freezes the browser at 4 million rows of data. We also measured the query evaluation time directly
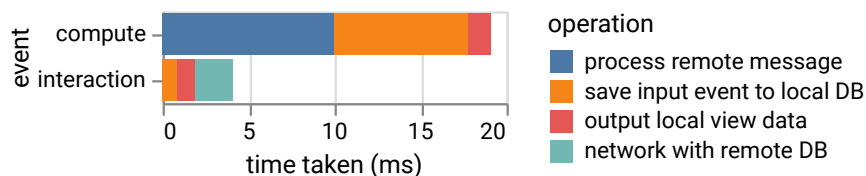
Figure 3.13: The x-axis represents the time taken for the event handling steps, and the y-axis the type of event being measured. These results are evaluated against both the *remote+network* and the *remote* only setup. The numbers follow the same distribution because the tick performance is decoupled from the database query processing performance. The steps are all *local* evaluations after the asynchronous events have been received. The actual latencies for each individual response are depicted in Fig. 3.14.
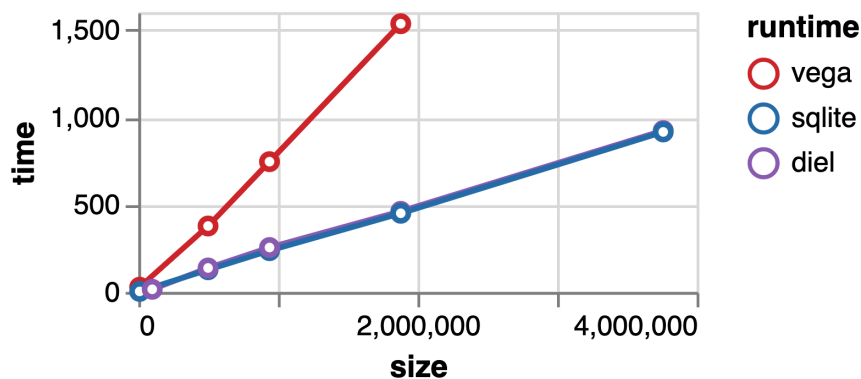
Figure 3.14: A visualization of how long the query processing takes as the data grows in size. One program is implemented with Vega, another with DIEL used in conjunction with a *remote* SQLite database, whose raw query time is also plotted. DIEL is able to make use of the backend database and process data beyond the capacity of the browser, with less than 1s of additional overhead for (de)serialization when working with the remote database.

against the remote SQLite database — DIEL's processing time follows closely with negligible serialization/serialization overhead. Additionally, the DIEL implementation benefits from a *non-blocking* design because it does not take up costly resources on the main thread.

While this study does not exhaustively analyze DIEL's performance, we believe it addresses DIEL's core motivations of usability, expressivity and scalability via remote data servers.

3.8   Evaluating DIEL's Usability

We evaluate the programming experience of DIEL using the *Cognitive Dimensions of Notation* [17], a set of considerations to evaluate the effectiveness of notational systems. Cognitive dimensions have been used by prior visualization systems to evaluate their usability [125, 119, 172, 120]. We pick a relevant subset and contrast the effectiveness of DIEL against that of current common practices.

Viscosity *(resistance to change).* DIEL facilitates change in a few ways. First, its declarative query specification abstracts *what* to compute from *how* to compute. As a result, switching from a client-only application to a client-server one (or to one using WebWorkers) requires only a few lines of change to configuration details. Fig. 3.2(4E) provides an example of the configuration, `remoteDbConfigs`, a JavaScript object containing high-level specification such as the type of the database used and what socket to connect to the DIEL database wrapper. Query changes are moreover isolated from table changes thanks to the classical data independence property of the relational model [113]. For instance, if the `incident` schema changes because the data provider now also includes the reporting station, none of the downstream queries, `fireMap` or `pannedIncidents`, would need to change; more complex schema changes can be hidden behind relational views. Finally, views also provide a way to reuse logic within an application. Fig. 3.4 shows an example where a change to the internal specification of the query `filtered` would be abstracted away from the dependent views as long as its `SELECT` clause is unchanged.

Closeness of Mapping *(closeness of representation to domain).* Since SQL is widely used in modern databases and for general data manipulation tasks [103], DIEL closely represents the data processing domain. However, DIEL does not fully represent the complex domain of data visualizations. For example, the Vega authors identify that visualizations involving small multiples often require hierarchical structures with second-order quantification [124].

We agree that the expressiveness of SQL is more limited than interactive visualization frameworks. Relational languages like DIEL expresses only first-order logic. In particular, DIEL can define and quantify relationships between entities, but cannot quantify over a data-dependent set of table names or column names. Yet this limitation is key for distributed execution and optimization. Neither physical data independence nor the rich optimization methods discussed in Section 3.3 would be easy to implement without the relational abstraction. Moreover, developers could transform the data into other forms by manipulating the output tables provided by DIEL in any JavaScript functions. For instance, they can use Vega to transform the tabular data into nested groups to render small multiples. These data transformations happen at the end of DIEL's dataflow and does not diminish the effectiveness of DIEL's ability to orchestrate distributed query evaluations.

Consistency *(similar semantics are expressed in similar syntactic forms).* In DIEL, the only data structure is a table. One of the key contributions of DIEL's model is to unify both

"live" events and "stored" data in a single frame of reference—tables—which store both data and history. Once events are reified as data in an event table, DIEL presents a unified data-centric language.

Premature Commitment *(constraints on the order of doing things).* For DIEL to be effective, it imposes a premature commitment. Developers must represent the state of visualizations using tables (rather than arbitrary data structures) upfront. This premature commitment can hamper a rapid prototyping process, but we believe the advantages of the table format outweigh these concerns. As discussed in Section 3.3, the table format facilitates working with distributed data, and makes explicit possibly concurrent processes.

Role-expressiveness *(the purpose of an entity is readily inferred).* DIEL reuses an existing, well-established data model of relational tables and queries, and introduces only two additional constructs: `EVENT`s and `OUTPUT`s. This approach has proven sufficiently expressive, as demonstrated by the examples in Section 3.6. DIEL operates as a middleware layer between the frontend and backends, and lifts the logic of data exchange between client and remotes. The relatively small surface area of DIEL's abstractions stands in contrast to the existing imperative code written to support such use cases, which is often dispersed across custom functions with a commensurate burden of role-expressiveness.

Hidden Dependencies *(important links between entities are not visible).* DIEL makes dependencies quite explicit. The only type of dependency DIEL introduces is between tables, which are syntactically evident in queries: the table a query creates is dependent on the tables it references. In contrast, current imperative practice distributes dependencies across different functions, each with custom logic and bookkeeping formats that require additional effort to navigate and make sense of.

Hard Mental Operations *(high demand on cognitive resources).* There are two potentially challenging programming tasks in DIEL. One challenge is debugging in SQL [42]. Consider the case where the developer is debugging a view *O* which involves both *V1* and *V2* views; they need to inspect *both* of the views to locate the error. To address this challenge, we built *view level constraints* (as discussed in Section 3.5), similar to SQL table constraints [113], so that developers could make assertions on intermediate queries. Another challenge is not being able to mutate state. It could be challenging to define the state of the visualization with only raw events, especially when the logic is more complex (e.g., *undo-redo*). To help, we took a page from the construct of *state programs* in *relational transducers* [3], which allow developers to maintain derived state by inserting values into tables after events.

Diffuseness *(verbosity of language).* The current DIEL syntax hews close to SQL, and as such does not have syntactic conveniences one might like for visualization (e.g., *binning* [82]). Some aspects of DIEL's current verbosity can be alleviated by introducing syntactic sugar for common operations. Through our own experience working with DIEL and analyzing code snippets, we identified the most common programming patterns and implemented a handful of syntactic sugars. For instance, `LATEST` selects the most recent event (discussed

in Section 3.5). Similarly, the *default* asynchrony policy for output views over distributed data creates an event table for the developer and selects the response for the most recent interaction. We provide additional details in the supplement.

In sum, DIEL introduces some premature commitment and hard mental operations. However, we believe these are outweighed by the decrease in viscosity, and more explicit dependencies, role-expressiveness, and consistency.

3.9 Conclusion

By adapting two key ideas from distributed systems programming — immutable events and logical constraints — DIEL contributes a substantive step towards declarative programming over distributed data and asynchronous events for interactive visualization. Through examples, we demonstrate that developers can use DIEL to declaratively specify a variety of emerging interactive visualization use cases, ranging from working with remote data to visualizing interaction history. And, to assess the challenge that DIEL's use of a relational language poses to developers, we conducted a heuristic evaluating using the Cognitive Dimensions of Notation [17]. We find that although DIEL introduces premature commitment and possible hard mental operations, these disadvantages are outweighed by a decrease in viscosity for working with data of various sizes and changing the designs appropriately, and the increase in consistency between specifying operations over distributed data. Moreover, as our performance benchmarks suggest, this declarative model allows DIEL to reason about the specification and optimize the execution plan.

DIEL is an open source system available at https://github.com/yifanwu/diel. By designing a unified abstraction over distributed data and asynchronous events, we hope to help developers prototype and explore alternative designs for emerging interactive visualizations faster.

Limitations and Future Work

Expressibility and Usability. DIEL uses SQL as a host language, and this limits expressibility, which we analyzed in Section 3.8. Since DIEL expresses only first-order logic, it limits the the expressibility. In fact, the very existence of the additional language features we implemented (Section 3.5) points to this limitation. For instance, Listing 3.6 shows SQL variables can only range over data values, not over schema values such as table and column name. Hence, we cannot write a DIEL expression that will handle an arbitrary column name as a variable—the columns have to be enumerated literally in the query.

As explained in Section 3.8, these limitations were to facilitate scalability and they can be bi-passed by using more expressive libraries like Vega [122] in conjunction with DIEL. While

this separation and additional composition is a significant improvement from the chore of setting up the client-server architecture, it is still far from ideal.

Future iterations of DIEL may benefit from using a data frame library based syntax instead, which could be more flexibly integrated into JavaScript [162]. We could also explore integration with Vega and Vega-Lite to directly support scalability.

Optimizations. As data grows in size and computation grows in complexity, optimizing the performance of interactive visualization application is a hot topic. DIEL's unique middleware architecture that spans the local and remotes allows for a number of research opportunities. To start, operator-level materialized-view maintenance techniques [25] can make the frontend database even faster. *Federated* databases that optimize globally across multiple databases [34] can help us optimize data exchange between the local database and remote database. Another possibility is to automatically parallelize query evaluation [113] across multiple threads of computation, e.g. multiple WebWorkers in a browser. Finally, we can enhance the performance of each timestep with "garbage collection" by removing rows that are no longer in use from logs. This pattern is common in many areas, such as in replicated database systems [117], multi-version concurrency control [16] and distributed systems programming [29].

Collaboration. DIEL does not yet support an important distributed use case, *collaborative interactive visualizations* (Fig 3.1(3)). Coordinating communication between multiple users is a classic challenge in distributed systems and CSCW [137]. A global order of events across multiple editors cannot be guaranteed without explicit coordination that decreases the interface's responsiveness. Instead, various coordination-free proposals have emerged that use more involved metadata than simple local timesteps to provide distributed consistency guarantees, e.g., [156, 30]. It would be interesting to extend DIEL with ideas from this work.

Chapter 4

B2: Bridging Code and Interactive Visualization in Computational Notebooks

The two previous chapters demonstrate that interaction history is an abstraction useful for interactions and programming. The fact that it is useful for both suggests that it may serve as a shared abstraction between the two mediums. This led us to investigate ways to leverage history to bring the two mediums closer. With a more unified medium, the analyst can have the best of both worlds—the quick feedback loops of interaction and also the expressiveness and reproducibility of programming.

We hypothesize that interaction can serve as a bridge to unlock the best of both worlds in computational notebooks. This hypothesis gave birth to B2, an extension in Jupyter. B2 bridges three gaps between programming and the interactive visualizations in the notebook, using capabilities provided by interaction history. Participants in our user study engaged with B2 extensively and switched frequently between interacting and programming. They confirmed that B2 helped them with faster data exploration in the notebook. B2 was published at UIST 2020.

4.1 Introduction

Computational notebooks (e.g., Jupyter and Observable) have become increasingly popular in data science because they enable *literate computing* [105]: a single document captures analysis code, textual observations, and visualizations of results. These computational notebooks remain useful far beyond the initial act of authoring: e.g., for auditing, reproducing, or sharing data insights. Moreover, the structure of these notebooks — a series of executable *cells* — facilitates a more iterative and interactive coding process well-suited for data science workflows [6, 159]. Surveys and interviews with data scientists, however, highlight the
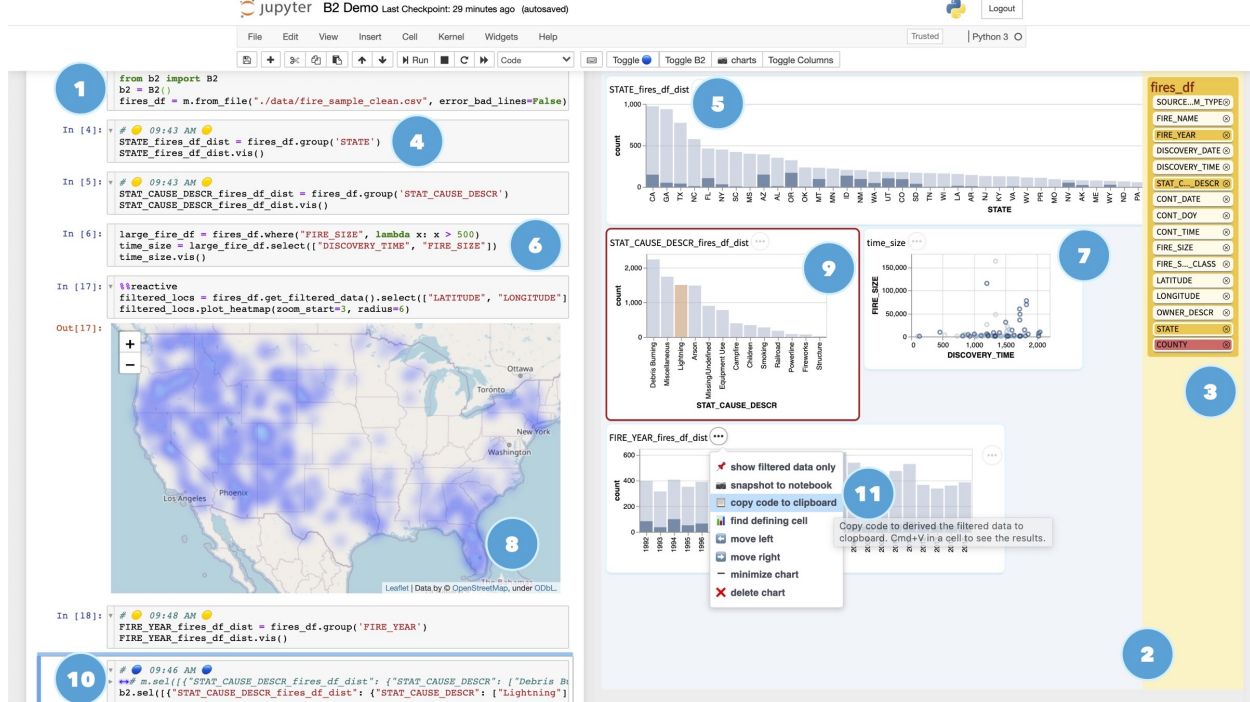
Figure 4.1: An analyst's workflow with B2. They start by (1) importing the library which
creates (2) a resizable dashboard pane to the right of the traditional notebook. The ana-
lyst can (3) click on the columns, which creates (4) code that computes and (5) visualizes
corresponding distributions. The analyst can also write (6) a custom data frame query to
create (7) the scatter plot. (8) B2's *reactive cells* automatically recompute when new inter-
actions occur on visualizations. Interactions involve (9) *selections* of marks, which link or
cross-filter the other visualizations in the dashboard, and are reified in code cells as either
(10) an interaction history or by (11) copying their composed predicate definitions.

impoverished use of visualization within computational notebooks. In contrast to visual
analysis tools such as Tableau (née Polaris [134]) or Microsoft PowerBI, which offer rapid or
automated specification of visualizations and direct manipulation interactions to coordinate
multiple linked views, visualizations in notebooks are largely manually specified single, static
views [13, 159].

Recent work suggests that although notebook users could benefit from richer support
for interactive visualization, the friction of switching between the two paradigms remains
too high. For example, although notebooks allow code cells and visualizations to be inter-
leaved, these two types of artifacts remain siloed [13]. Code cells cannot access the results
of interactive operations performed on visualizations—for instance, after brushing a region
on a scatter plot, data scientists cannot extract the selected points for subsequent analysis.
This gulf is exacerbated by the fact that interactive results are *transient*, a property that

violates literate computing. Unless an analyst explicitly documents them — a rare practice due to the friction it introduces [116] — these results are lost when a notebook session ends. A similar disconnect also exists in the other direction. Code cells express a rich analysis provenance, which often has a natural correspondence to interactive visualizations. Analysts, however, are unable to leverage this provenance and are, instead, forced to manually specify interactive visualizations from scratch. Finally, interleaving code and visualization cells may itself be an impediment. As there may be several cells between successive visualizations, it is unlikely to have more than one visualization visible on screen; thus, interaction techniques become confined to operating over a single visualization at a time, which provides only limited utility.

In response, we present B2, a library of techniques to bridge the divide between code and interactive visualizations in computational notebooks. To map between these two sides, we need a shared representation of the work occurring on either side. The fundamental task of data analysis involves iterative data transformation, and both code and interactive visualizations can capture this task as a *data query*. In code, queries are typically constructed through data frame manipulations or as SQL statements while, in visualization, queries are often expressed as interactive selections [57, 121, 164].

To allow analysts to more seamlessly move from code to interactive visualizations, B2 wraps a data frame library and records the abstract syntax tree of queries that occur as a result of data frame transformations. Based on this data lineage, B2 offers an additional `vis` API method on data frames which, when invoked, automatically synthesizes an appropriate *interactive* visualization. For instance, consider the example shown in Fig. 4.1. An analyst imports a dataset about wildfires in the United States[1] as a B2 data frame. In a subsequent series of cells, they first group the data by `State`, and then by `Cause`, producing a new data frame each time. B2 tracks these steps and using the data lineage, creates two histogram visualizations that can be interactively cross-filtered. By design, these visualizations do not appear in the normal flow of notebook cells. Rather, they appear within a secondary dashboard panel to facilitate richer multi-view coordination [153] regardless of where in an analysis process they are created.

To bridge the gulf in the other direction, B2 instruments its visualizations to track the interactive selections that occur. An API method materializes the selected state as a data frame, thereby allowing analysts to conduct follow-up analysis of interactive results in code. When such cells are marked as *reactive*, they are automatically reevaluated as new interactions occur. B2 also creates a new code cell to maintain a log of interaction history — old entries are commented out and new entries appended, with selections represented by their underlying predicate definitions. In doing so, B2 *reifies* interactivity and persists it in the flow of the literate computing notebook. For instance, analysts can (un)comment entries in the log to replay their interactions or compare states, can use code comments to doc-

---

[1] https://www.kaggle.com/rtatman/188-million-us-wildfires
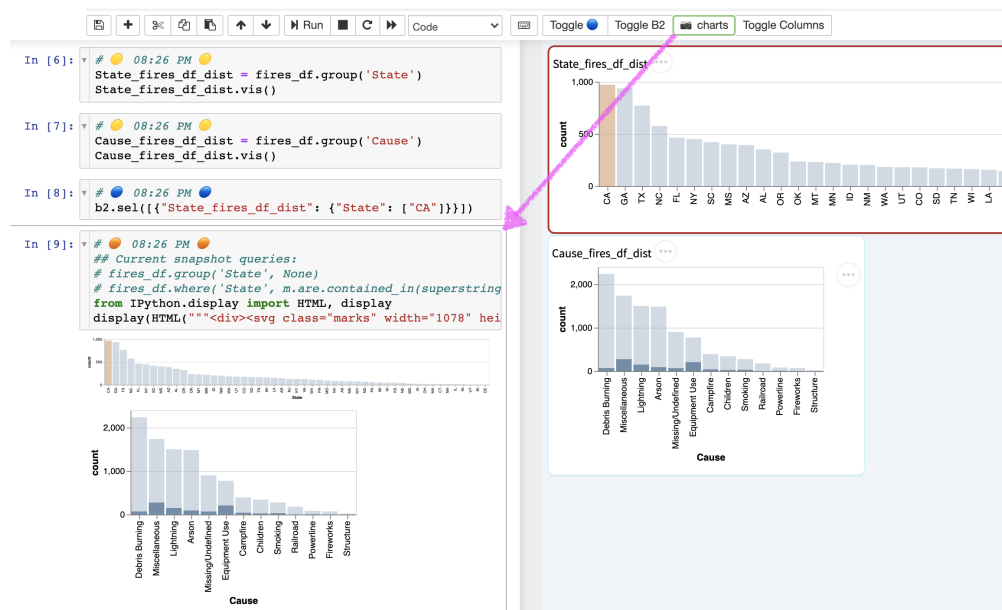
Figure 4.2: Snapshotting creates a cell in the notebook with an SVG of the visualization, persisting the transient interactive state.

ument meaningful interactive discoveries, and can copy and paste selection predicates for downstream analysis.

We implement B2 as an open source extension for Jupyter notebooks available at https://github.com/yifanwu/b2, and evaluate its efficacy through a first-use study with 7 participants. Traces of participant behavior demonstrate they make use of B2's "bridges" to frequently switch between code and interactive visualization, and qualitative comments indicate that B2 helps facilitate the exploratory data analysis process.

4.2 A Demo of B2

To place the design and goals of B2 in context, we present a full demo following the wildfire example in Fig. 4.1. We identify the times when the analyst, Sam, **switches** from code to visualizations and **switches** from visualizations to code. We also include a supplementary video demo of this section.

Sam first initiates B2 with code `b2 = B2()`, which creates a dashboard to the right. She then loads in the fires dataset from a CSV file using `b2.from_file`, which creates a list of columns in the pane to the right. To start exploring, Sam **switches** to the dashboard. She sees a `State` column and wonders how the count of fires varies across states. She clicks on this column. B2 then adds and executes a code cell that derives the

distribution `state_dist=df.group('State')` and creates a visualization in the dashboard `state_dist.vis()`. Sam then **switches** to a markdown cell to record the insight that *"CA has the highest number of wildfires."*

To speed up code execution, Sam takes a sample of the data frame (`sample_df=df.sample(1000)`). She then **switches** to the dashboard to investigate the `Cause` for `CA` fires. She clicks on the column `Cause` (which again generates a code cell deriving a new data frame and visualization) and then clicks on the resulting `State` histogram to select the bar representing `CA`. This interaction cross-filters the `Cause` histogram, with the filtered `CA` fires shown in darker blue. To document this result, Sam clicks *"Snapshot Charts"* which copies the visualizations to a notebook cell (Fig. 4.2).

Sam now wonders if there are fewer fires in the early morning since it is cooler. After clicking on the `Time` column, she wishes to sort by time but notices some null values. So she **switches** back to the code generated and filters out the null values, formats the time, and specifies the visualization to sort by the x-axis (Fig. 4.3). She verifies in the visualization that her original hypothesis was true.

Besides distribution visualizations, B2 also supports *custom* visualizations. She hypothesizes that there may also be a correlation between fire size and time. She **switches** to code, writes `sample_df.select(['Time', 'Size']).vis()`, and **switches** to the dashboard to inspect the resulting chart (7) and interacts with it to explore. Sam notices that the cause of large fires in the afternoons are mostly Lightning and wonders if the fires caused by lightning in `CA` increases year over year, so she clicks the `Year` column in the yellow column pane and then on `Lightning` bar in the `Cause` histogram.
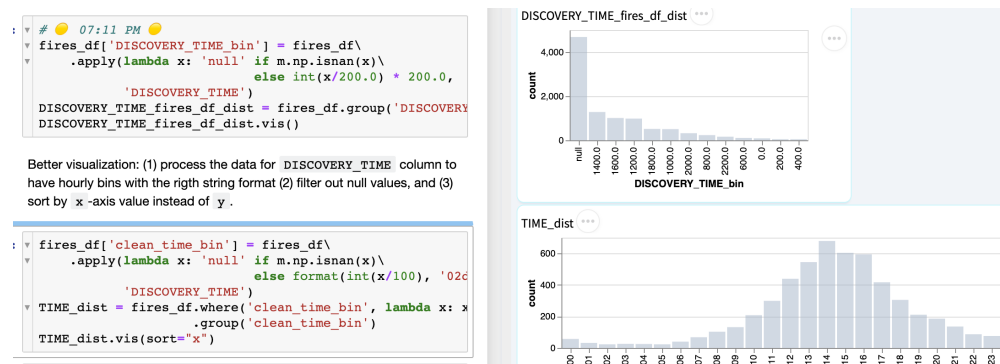
Figure 4.3: The top code cell is generated by B2 to visualize the distribution of `Time`, after a selection on the column pane by the analyst. The bottom code cell is edited by the analyst from the code above, using functions such as `format`, and `where`, to further refine the visualization.
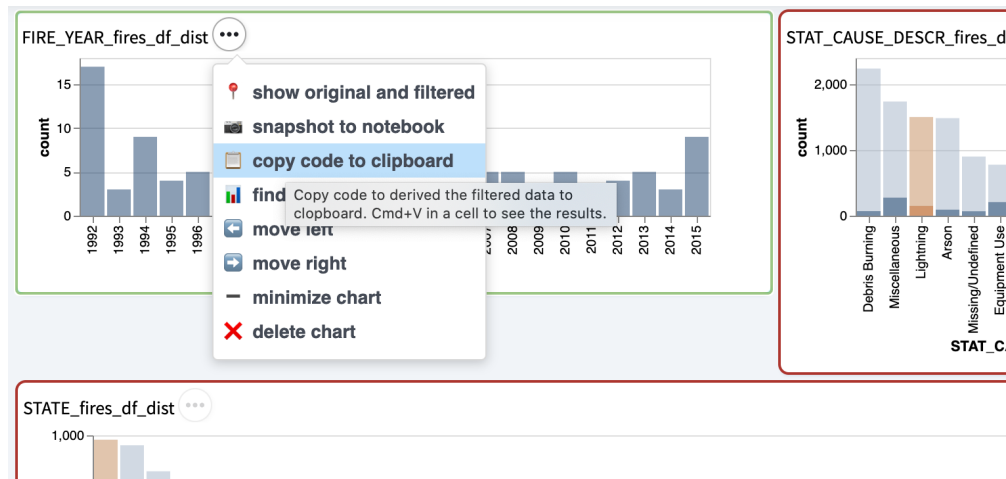
Figure 4.4: The full state of the chart, including interactive selections, is converted into code
and made available through the *"Copy Code to Clipboard"* button.

To model the rate of increase, Sam clicks on the menu item for the histogram, *"Copy
Code" "to Clipboard"* (Fig. 4.4), and **switches** to paste the code into a new cell. The pasted
code expresses the interactive selections as composed query predicates—`sample_df.where`
`('State', 'CA') .where('Cause','Lightning')`. Sam replaces `sample_df` with `df`, and
writes a simple linear model to fit the full dataset. She verifies that there is indeed a trend
upwards and notes the finding in a new markdown cell.

Having explored the "low hanging fruit", Sam decides to dive deeper into fire locations.
She **switches** back to code, and uses a Python library to draw a heatmap using the (`Lat`,
`Lon`) coordinates. To enable interactive analysis, she marks this cell as `%%reactive` and uses
a data frame that materializes the interactive state via the `get_filtered_data` API, which
returns the rows of `df` filtered by the current selection (Fig. 4.5). Sam then **switches** back
to the dashboard and clicks on the `Lightning` bar in the `Cause` histogram. The reactive cell
updates, showing that Lightning is skewed towards the north-central states.

Finally, Sam sends this notebook to her collaborator Alex. Alex starts by wanting a high-
level overview of Sam's process. He clicks the *"Toggle ⬤"* button at the top of the notebook
to hide the interaction history cells, to more easily view Sam's code and markdown notes,
as well as charts she explicitly chose to persist via *"Snapshot"*. To validate Sam's insights
for himself, Alex un-toggles the interaction histories, and *replays* interactions by unfolding
and (un)commenting relevant lines in the history, and re-executing the cells. The charts in
the dashboard, as well as the reactive heatmap update in response.
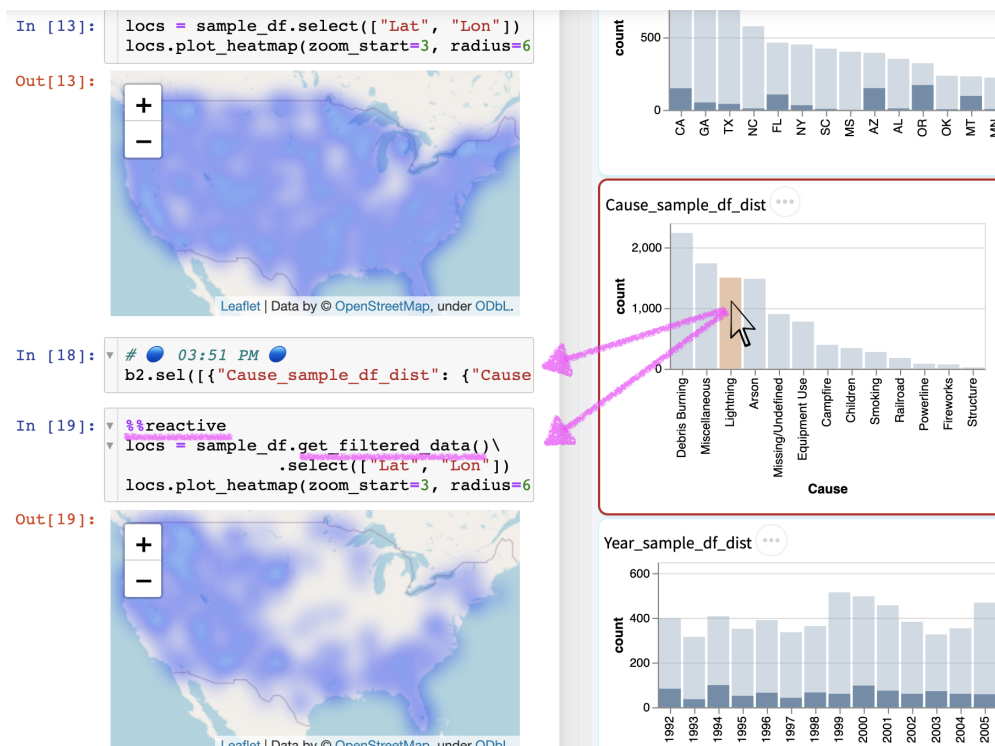
Figure 4.5: Contrast the top cell, which is static, to the reactive cell below. The reactive
cell can be iterated on using interactions.

4.3 Related Work

Prior work has primarily investigated two mechanisms for integrating code and direct manipulation interactions: using interactions to parameterize code or generate code. Here, we review these approaches and draw contrasts to B2, and motivate its design through prior surveys of data scientists.

Interactions Parameterizing Code

Early systems like Juxtapose [54] and work by Bret Victor [148, 149, 150] helped popularize instrumenting code editors with interactive controls [135]. Computational notebook platforms offer analysts ways of instrumenting code with HTML widgets (e.g., range sliders, radio buttons, checkboxes, and drop-down menus). For instance, Jupyter offers *Jupyter widgets* [70], R Markdown notebooks can be made interactive with *shiny* [48], and Streamlit [141] and Observable [101] provide a standard library of options. Widgets can be manually instantiated by analysts, or can be automatically inferred using code semantics (e.g., the `@interact` function decorator found with Jupyter widgets). Across these platforms, widgets

primarily serve to *parameterize* code — i.e., each widget maps to a single variable in the code, and manipulating the widget re-executes the corresponding code. In doing so, widgets help tighten the feedback loop by allowing analysts to rapidly explore alternate input parameters instead of rewriting and rerunning whole cells [67].

Widgets, however, are only an initial step towards endowing code with interactive semantics. First, widgets have limited expressivity — although they can be composed together into interactive dashboards [18, 81, 86], widgets do not provide behaviors as rich as those found in interactive visualizations [170]. Second, widgets violate the literate computing goal of reproducibility [105] as interactions with them are transient — the results associated with a particular widget state are lost on subsequent interactions. B2 builds on the benefits that widgets bring to code, and extends them to interactive visualizations. Akin to the automatic synthesis found in features like Jupyter's `@interact` decorator, B2 tracks the lineage of data frame derivations to generate visualizations and automatically instruments them with interactivity. And, rather than simply parameterizing individual variables, interactive operations populate intensional and extensional predicates called selections [121]. Finally, B2 persists these operations by maintaining logs of interaction histories in new code cells.

Interactions Generating Code

The programming-by-example (PBE) community has a long history of studying how user input can be used to synthesize programs. For example, a user can provide concise input-output pairs [50, 51], indicate hierarchical structure using colored blocks [169], or record and replay interactions with lists on web pages [12, 22]. These systems receive user interactions as input and synthesize as output a program in a general-purpose programming language.

While B2 takes inspiration from this line of work, our approach most directly follows systems that establish a *bidirectional* relationship between direct manipulation interaction and textual specification of code. One example is Sketch-N-Sketch [26, 65, 66], which allows users to write a program to generate SVG output, and then directly manipulate the SVG canvas to modify the original code. Another example in the data domain is Wrangler [52, 72], a data transformation interface that provides a direct-manipulation tabular interface reminiscent of a spreadsheet, and maps user interactions into editable textual histories that can be compiled into standalone code.

These approaches are motivated by recognizing that neither direct manipulation nor coding is best suited for all tasks, but combining them yields an accumulation of benefits — users can rapidly and intuitively specify designs via direct manipulation, but then switch to code to construct reusable abstractions. This goal resonates with the results of recent surveys and interviews of data scientists which find that visual interfaces are most useful if their output can be captured in code [6, 23, 159]. Thus, akin to Sketch-N-Sketch and Wrangler, B2 provides bidirectional bridges between code and interactive visualizations: B2 synthesizes appropriate visualizations by tracing the data lineage expressed in code,

and interactions performed on the visualizations are logged to code cells to enable further analysis. Critically, B2 differs in the domain it addresses (cf. Sketch-N-Sketch) and in its support for richer interactive visualizations integrated with general-purpose programming languages (cf. Wrangler's domain-specific language).

Systems like GUESS [4] and DEVise [91] are more closely aligned with B2's goal of bridging code and interactive visualization. In particular, GUESS offers an environment where interactions with graph visualizations can be captured in Python-based REPL (read-evaluate-print loop), and textual commands manipulate the visual output. DEVise identifies that interactive visualizations can be modeled as SQL expressions, and that multiple views can be coordinated by analyzing their schemas — an approach analogous to B2's automatic synthesis of interactions based on data frame lineage. However, B2 differs in two key ways. While GUESS and DEVise are standalone systems, B2 is embedded within the existing data science ecosystem — namely in computational notebooks and by leveraging data frame APIs. In doing so, B2 must bridge an additional set concerns that these prior systems did not grapple with: how best to combine the highly iterative nature and two-dimensional layout of interactive visualizations with the persistence and linear layout of computational notebooks.

Meeting data scientists where they work is a motivation that B2 shares with Wrex [36], a recent system that embeds a visual data wrangling interface within the Jupyter notebook. Building on the previous theme, a key insight of Wrex is that it is not enough to simply embed PBE systems in context; rather, to respect literate computing principles, the code these systems synthesize must be human-readable. B2 follows this insight in two ways. First, the interaction history that B2 produces is expressed as a series of human-understandable API calls, rather than low-level event logs. And, second, to preserve the linear flow of literate computing, B2 records these interaction histories in new code cells placed directly after the most recently executed cell.

B2 is contemporaneous with work by Kery et al. developing mechanisms to move "fluidly" between code and graphical interfaces within computational notebooks [78]. In particular, Kery et al. introduce `%mage`, a Jupyter extension that provides APIs for graphical interfaces to affect notebook state. They demonstrate how `%mage` can be used to provide a spreadsheet interface to interactively manipulate data frames, and extract or materialize interactive selections performed on visualizations. Although `%mage` and B2 share a common set of goals, the two systems differ in their scope: B2 targets integrating code with interactive visualizations specifically, whereas `%mage` looks to graphical interfaces more broadly. This difference in scope yields salient differences and tradeoffs in how the two systems achieve their desired outcomes. For example, `%mage` uses string templates and pattern matching to translate interactions to code — an approach that many different types of graphical interfaces can target, but that can also be brittle when trying to map code changes back to the interfaces. In contrast, B2 records interaction histories as *predicates*, a representation that is tailored to interactive visualization but is also more robust to bidirectional changes. Moreover, by taking a more focused scope, B2 identifies and addresses an additional challenge with inte-

grating code and visualizations that may not apply to graphical interfaces more generally: restricting interactive visualizations to a linear flow of interleaved cell outputs limits the creation of richer multi-view coordination [23, 153].

Since B2's publication, there has been newer works published to help data exploration in a notebook programming setting. *Lux* by Lee et al., is a Python library that facilitate data exploration through automation [87]. Unlike B2, which captures user intention through data frame query operators, Lux captures user intentions explicitly through the `.intent` API and provides visualizations for potential next-steps. B2 and Lux share the common goal of facilitating data exploration in notebooks. However they are complimentary: Lux could incorporate in the history of data frame queries to enhance their recommendation algorithm, and B2 could incorporate recommendation heuristics and algorithms to enhance the exploration process.

The Needs of Data Scientists

B2's goal of bridging code and interactive visualization is motivated by recent surveys and interviews of data scientists [6, 13, 23, 159]. In particular, Wongsuphasawat et al. find that data scientists often switch between several tools including textual environments (e.g., MATLAB or Jupyter) and graphical interfaces (e.g., Tableau or Microsoft PowerBI) during their analysis sessions [159]. As Chattopadhyay et al. report, this switching behavior forces analysts to repeat themselves by manually translating work they conducted in code to visual interfaces, or vice-versa [23]. For many data scientists, this overhead is sufficiently prohibitive that they eschew visual analysis tools altogether and restrict themselves to working only in code [159]. Indeed, Batch and Elmqvist identify that visualizations *"should be first-class members of the analytical process so that actions and transformations interactively performed in the component can be exported and passed on to the next component in the sequence"* [13] and Alspaugh et al. call for new systems that combine the expressiveness of programming and scripting languages, with the efficiency and ease-of-use of visual analysis tools [6].

These studies also indicate the challenges of integrating code and interactive visualizations within notebook environments. For instance, analysts report frustrations with how the cell-based structure of notebooks limits the usefulness of visualizations [23].

And, a naive integration of the two risks exacerbating existing concerns of notebooks being a "mess" [76], full of "ugly code" and "dirty hacks" [116]. This problem is exacerbated by the structure of code cells, which can be executed out of written order, and the inclusion of interactive widgets, which control input parameters in a transient fashion. A naive integration of code and interactive visualization could significantly compound this problem. Recent work has explored a spectrum of strategies to ameliorate this latter issue including version control that occurs automatically for all artifacts in a notebook [75], on a per-cell basis [115], or for manually-defined snippets [74], or tools for gathering, cleaning, and comparing messy code [55]. Inspired by these solutions, and in particular by their lightweight

and in situ nature, B2 records a history of interactions in new code cells. Critically, to not further contribute to the spatial dimension of mess [55], B2 merges contiguous selections into a single cell, with old interactions commented out and folded.

Finally, recent work has also explored how to extend the literate computing paradigm to visual analysis. For instance, Wood et al. introduce *literate visualization* [160] while Mathisen et al. propose *literate analytics* [94]. While both approaches share our goal of bridging literate computing and exploratory visual analysis, their focus is on the narrative aspects of the process. In particular, literate visualization introduces a schema validator to prompt users to document their design decisions, while Mathisen et al. implement InsideInsights, a system for structured and hierarchical annotation of insights that are a result of visual analysis. B2, by contrast, is concerned with enabling analysts to move between the code-driven work of literate computing and the interactive visualizations of exploratory visual analysis. Rather than focusing on promoting documentation of insights, B2 synthesizes visualizations from code semantics, and allows code to operate on the results of interactions performed on visualizations.

4.4   The Gaps Between Code and Interactions

Computational notebooks meld ideas from traditional scientific notebooks and literate programming as envisioned by Knuth [80]. Interactive visualization environments are inspired by vehicle dashboard design and the ideas of Exploratory Data Analysis as envisioned by Tukey [142]. However, there are significant gaps between the metaphors of notebooks and dashboards, and the goals of programming and data exploration.

A rich integration of interactive visualization into notebooks should strive for a *composition* of the benefits offered by both paradigms. However, we identify three gaps that currently hinder such integration: a *semantic* gap that prevents each side from understanding the work that is happening in the other; a *temporal* gap that allows only code to persist, and only interactions on visualizations to be transient; and a *layout* gap between the notebook's linear structure and rich coordinated multi-view visualizations. In this section, we describe these gaps and their impact on an analyst's workflow. We identified these gaps by studying the pitfalls of computational notebooks as reported by data scientists [6, 13, 23, 159], and by examining the design choices and tradeoffs manifest in prototypes we developed through our iterative design process.

The Semantic Gap

At base, we assume that raw data to be analyzed is accessible to both code and visualizations — via data frames for code, and encoded as visualizations via the grammar of graphics. However, neither side is able to capture *the work* that occurs in the other. The code side has no access to the work involved in interactions that are performed on a visualization.

Thus, visualizations become a "dead end" from code, unable to drive subsequent analysis unless an analyst chooses to manually code up insights they identified via visual interaction. Conversely, visualizations do not understand the work expressed in code: specifically it is blind to the lineage of transformations and derivations on a data frame. As a result, an analyst must manually construct appropriate interactive visualizations from scratch even if the code that specifies the data frame captures semantics that can automate visualization design. For instance, when data results from a `group` operator, it is typical to favor a *bar* marks to produce a histogram. Similarly, visualizations of two derived data frames that share a common ancestor can often be usefully linked or cross-filtered. In both directions, this semantic gap introduces friction into an analyst's process and prevents them from being able to "round-trip" their data and their work without repeated specification of intent.

To bridge the semantic gap, it helps to have a *shared abstraction* to represent the work occurring on either side. The fundamental task of data analysis involves the iterative transformation of data, and both code and interactive visualizations capture this task as *data queries*. In code, queries are expressed as data frame manipulations — following our demo example, `df[df['FIRE_SIZE'] > 500]` returns a data frame of large fires while `df.group('STATE')` groups tuples by the `STATE` field. For visualization, although a wide variety of interactive techniques are available [60], we consider those techniques that can be modeled as data queries. Here, we turn to Vega-Lite [121], which identifies a *selection* as a fundamental building block for interaction design that is suitably expressive to cover an established taxonomy of interaction techniques in data visualization [170]. In particular, every Vega-Lite selection includes a definition for a *predicate*, or a data query that determines which tuples lie within the selection. These predicates are defined in one of two ways: an *intensional* predicate specifies a set of data points based on logical conditions that must be satisfied, while an *extensional* predicates explicitly enumerates a set of selected data points. In essence, an intensional predicate is an expression (a piece of code), and an extensional predicate is a fixed set of data.

With this shared representation in place, we can begin to translate the data queries occurring in code to interactive visualization, and vice-versa. For instance, by tracking the queries expressed via data frame manipulations, we can automatically synthesize appropriate visualizations and instrument them with linked or cross-filtering interactions. Similarly, interactive selections can be captured in code by their predicate definitions, or by materializing them as data frames. We detail how to operationalize these bridges in the subsequent section.

The Temporal Gap

Iteration is a critical process in data science. Both code and interactive visualizations support iterative workflows, but they occur at different time scales. The key mechanism for iterating in code is cell execution: data scientists author their analysis as a series of discrete steps (or "cells") which can be executed individually, sequentially, or out of the order they were

originally written in. Although cells may be edited and re-executed any number of times, analysts often prefer to copy and paste their code to a new cell to be able to compare different iterations of an idea. Critically, however, all these cells and their output are *persistent* — a property that facilitates literate computing goals of sharing and reproducing analyses, but also contributes to the burdensome mess that data scientists report. In visualizations, iteration occurs through repeatedly performing interaction techniques that manipulate the view (e.g., through highlighting or filtering points of interest). However, in contrast to the persistent nature of iteration in code, iteration through interacting on a visualization is entirely transient. This transience violates a key tenet of literate computing: it hinders an analyst's ability to refine or share insights they arrived at interactively. However, it also lowers the threshold for engaging in iteration by shifting the process from one of authoring and editing code to one of browsing and exploration.

Thus, the temporal gap introduced by persistence on one side and transience on the other either limits how much an analyst might iterate, or the degree to which they capture and share insights that result from iteration. To bridge this gap, we need to enable users to make their exploratory iterations in visualization persist when appropriate, and make their code iteration more transient when appropriate. Doing so will allow interactions performed on visualizations to serve, alongside the code, as a reproducible log of work—this also reduces the code versioning burden of a large trail of slightly different cells that analysts currently generate. In the next section, we detail the mechanisms B2 provides for crossing this gap including capturing a snapshot of visualizations, reifying interactions as a history log or a materialized data frame, and by introducing reactive cells that automatically re-execute when new interaction occurs.

The Layout Gap

Given their narrative roots in scientific notebooks and literate programming, notebooks naturally have a linear layout. When visualizations are incorporated into notebooks, they are interleaved between code cells like figures in a research paper. While this format facilitates sharing and communicating the logic of analytic workflows, it often puts a physical distance between related charts, which limits an analyst's ability to exploit visual signals that arise from multiple charts—especially signals resulting from chart interaction.

By contrast, data dashboards offer a compact co-location of charts on a 2-dimensional canvas. This layout enables the dynamic, multi-view coordination across charts we see in exploratory visual analysis tools [153], where charts can be linked so that interactions on one chart can meaningfully change the view in others. While this is useful for short-timescale analysis, it lacks the notebook's ability to structure and narrate a multi-step investigation.

For the best of both worlds, we can integrate a dashboard layout into the notebooks. Doing so, however, brings with it several design challenges. First, a dashboard layout breaks the formerly tight coupling between a visualization and the code cell containing its specifi-

cation: although a visualization may be visible in the dashboard, its specification cell may have scrolled out of view. Analysts may wish to locate these cells in order to understand how a visualization was constructed, or modify its design. Second, a dashboard layout has implications for the bridges we introduced to address the temporal gap. In particular, as interactions on visualizations now occur in the dashboard rather than as part of the linear notebook layout, should code cells containing interaction histories be automatically created or manually requested? In either case, where should they be placed in the linear structure of the notebook to fulfill the readability goals of literate computing, without further contributing to the mess data scientists currently grapple with? In the next section, we describe how B2 resolves these tensions when bridging the layout gap.

4.5   System Design and Implementation

B2 is implemented as a Jupyter extension. It is composed of two components: (1) on the code side, a Python back-end component that provides an instrumented data frame library, an event-loop reacting to interactions, and an API to access the state of the interactive visualizations current and past (2) on the visualization side, a JavaScript front-end component that renders visualizations, captures user selections, and generates synthesized notebook code cells corresponding to visual interaction. In this section, we describe how the notebook and dashboard work together to bridge the three gaps previously described. Throughout, we refer back to Fig. 4.1 for illustration.

Bridging the Semantic Gap

To bridge the semantic gap between code and visualization, we need to translate the work happening on either side to the other. As discussed above, this work is represented in a shared abstraction of data frame queries, which can be generated from each side and translated across.

Code to Interactive Visualizations

B2 provides a simple Python API to generate interactive visualizations from data frame code. This API is backed by novel techniques for auto-generating interaction logic in Vega-Lite [121] based on the Python lineage of a data frame.

   B2 delivers this API in the context of a Python data frame library called `datascience`[2], by adding a single method, `.vis`. To minimize the activation energy of using B2 visualizations, the `.vis` method does not require any parameters to work—B2 can infer the

---

[2] `datascience` was designed for instructional purposes in large data science courses [33]. Pandas is the most popular Python data frame library, but it is notoriously complex, with an API that permits many different ways to express the same logic, making operator tracking difficult [15, 162]. To inter-operate, B2 data frames can be easily mapped to/from pandas via `df.to_pd` and `b.from_pd`.

specification for a data frame visualization using established heuristics like column data types [143]. The `.vis` method can be controlled directly via a set of optional parameters based on Vega-Lite configuration specifications, which are augmented with instrumentation from B2 for cross-filtering. The `mark` [146] parameter chooses among bar charts, scatter plots, and line charts; `encoding` [145] configures which column is the x-axis, y-axis, how they should be interpreted (ordinal, quantitative, temporal), and sorted; `selection_type` and `selection_dimensions` [147] configure how the selection should happen and whether the selection is the x-axis, y-axis, or both. Any parameters that the analyst chooses to specify are locked to their specification, the rest are inferred.

Critically, we do not ask the user to specify any interaction logic—we *infer* that logic through the data lineage of the queries. This goes beyond traditional visualization inference techniques like ShowMe [143] that only apply to static charts. Consider the scatter plot of the fire `Size` and `Time` (Listing 4.1, Line 2) (7), and the bar chart of the distribution of fires by their `Cause` (Line 3) (9). Because both data frames derive from the same parent, `fires_df`, B2 infers that they should be linked. As a result, a brush selection on the scatter plot cross-filters the bar chart by overlaying a second bar chart on the first. This overlay is defined by a new, automatically-generated query that first filters the rows of fires whose `Time` and `Size` is in the selected region (Line 5), then re-applies the grouping on `Causes` (Line 6) to the filtered base data frame. This query is derived by tracing the operators used to derive each data frame, and then applying the selections to the base data frames of the "source" data frame, then replacing the filtered data frame with the base of the "target" data frame.[3]

```
1 # given
2 size_time_df = fires_df.select['size', 'time']
3 cause_df = fires_df.group('cause', count)
4 # derived
5 filtered_df = fires_df.where(lambda r: r.size < max_size and r.time
      < max_time)
6 cause_df_filtered = filtered_df.group('cause', count)
```

Listing 4.1: Example queries and automatically synthesized interactions.

Visual Interactions to Code

B2's chart interactions are expressed as *selections* of data. We want to enable users to use chart interactions for easy, familiar tasks, and fluidly reify selections into code so they can bring them back into the customizable logic of the notebook pane. To illustrate, we work through a scenario in which an analyst identifies an interesting area in the scatter plot of the fire size and time of the fire in (7), and brushes to highlight the area. The brush specifies

---

[3] Technical details that describe the scope of the inference, methods and algorithms are discussed in Section 4.6.

a selection bounding the size and time of the fires. This selection can be accessed in three different ways that exercise different features of B2.

Data. After brushing, the analyst sees a filtered chart containing an interesting distribution of fire causes. They wish to directly access the data of the filtered chart to evaluate custom functions using the data — for example, to join with data of state population and compute the correlation or customize the visualization in another library/tool. The data in the *current* selection is accessed through the `get_filtered_data` API, which returns a standard data frame object.

Code. The analyst also wishes to access the code that derives the data of the state histogram to (1) run the code on a different dataset with the same schema, (2) share or record the code so the result can be reproduced directly, or compared with other analysis. This code can be accessed through the `get_code` API, as well as the *"Copy Code to Clipboard"* dashboard button (11).

Predicates. The analyst realizes that they also want to access selections that occurred previously. The `all_selections` API returns the full history of selections as a list of B2 objects, and a corresponding API returns the `current_selection`. These B2 objects can be reused using additional API calls that give access to either a data frame or code representation.

```
1 # predicates
2 b2.current_selection
3 b2.all_selections
4 # data from the current selection
5 df.get_filtered_data()
6 # code from the current selection
7 df.get_code()
```

Listing 4.2: B2 APIs, from interactive visualization to code

Bridging the Layout Gap

The goal of the B2 dashboard pane is to allow visualizations that correspond to many, possibly distant notebook cells on the left to be co-located spatially on the right. Hence the dashboard pane is a 2D canvas that scrolls independently of the notebook pane. Users are given affordances to control the position of charts within the dashboard.

To bridge the layout gap between notebook and dashboard metaphors, we have to consider both how notebook features map into the dashboard, and how dashboard interactions are reified back into the sequential layout of the notebook.

From Notebook to Dashboard

Invoking the Python `vis` API of B2 causes a visualization to be generated, which needs to be placed in the 2D canvas of the dashboard. Given the layout flexibility of the dashboard and the goal of allowing users to colocate charts, we simply append new visualizations to the bottom right of the current dashboard pane, and scroll the pane to the new chart. Users can then reposition the visualization via a menu on the chart.

From Dashboard to Notebook

To make an interaction persistent, we need to place its reified code into a notebook cell, which requires choosing a location in the sequential layout of the notebook. One option is to paste the code into whatever cell currently contains the Juypter notebook cursor. We ruled out this choice after pilot tests showed that analysts are often not aware of where the cursor is. We also considered always placing selections at the end of the notebook, or even all in one dedicated cell, but in that design interactions are separated the flow of work, conflicting with our goal of "closing the loop" and integrate coding with visual exploring in a single flow of work.

We ended on a design that seemed intuitive to pilot users: placing generated code cell after the *most recently executed* cell. To cue the user as to where new cells will be placed, we maintain a horizontal blue bar in the notebook under the most recently executed cell.

Beyond the initial code placement, users need long-term affordances to investigate the connection between charts on the right and cells on the left. To see the connection between a visualization and the cell where it was specified, analysts can click on the button, *"find defining cell"*, in a drop-down menu (11), and the notebook will navigate to the cell where the visualization call `vis` is invoked.

Bridging the Temporal Gap

The temporal gap between notebooks and interactive visualizations require introducing persistence to interaction, and allowing code cells to respond interactively to user input. We describe how we support these in turn.

Bringing Persistence to Interaction

The ability to reify interactions into code allows users to persist their interactions. This can substantively change the user experience of interactive visualization, integrating it into the longer-term, narrative metaphor of a computational notebook.

However, not every real-time data exploration gesture merits being solemnized in the narrative of a notebook. If we record every transient visual state, it would bloat the notebook significantly. Instead, we record the minimum amount of code to specify interactive

Naively generated cells.

```
▼ # ● 08:14 PM ●
  m.sel([{"TIME_dist": {"clean_time_bin": ["15", "14"]}}])
```

```
▼ # ● 08:15 PM ●
  m.sel([{"STAT_CAUSE_DESCR_fires_df_dist": {"STAT_CAUSE_DESCR": ["Fireworks"]}}])
```

```
▼ # ● 08:20 PM ●
  m.sel([{"STAT_CAUSE_DESCR_fires_df_dist": {"STAT_CAUSE_DESCR": ["Lightning"]}}])
```

```
▼ # ● 08:22 PM ●
  m.sel([{"STAT_CAUSE_DESCR_fires_df_dist": {"STAT_CAUSE_DESCR": ["Lightning"]}}, {"TIME_dist": {"clean_time_bin": ["14","15"]}}])
```

History compressed into comments in the same cell.

```
▼ # ● 08:22 PM ●
  # m.sel([{"TIME_dist": {"clean_time_bin": ["15", "14"]}}])
  # m.sel([{"STAT_CAUSE_DESCR_fires_df_dist": {"STAT_CAUSE_DESCR": ["Fireworks"]}}])
  # m.sel([{"STAT_CAUSE_DESCR_fires_df_dist": {"STAT_CAUSE_DESCR": ["Lightning"]}}])
  m.sel([{"STAT_CAUSE_DESCR_fires_df_dist": {"STAT_CAUSE_DESCR": ["Lightning"]}}, {"TIME_dist": {"clean_time_bin": ["14","15"]}}])
```

Comments folded away in the cell.

```
▶ # ● 08:22 PM ●↔
  m.sel([{"STAT_CAUSE_DESCR_fires_df_dist": {"STAT_CAUSE_DESCR": ["Lightning"]}}, {"TIME_dist": {"clean_time_bin": ["14","15"]}}])
```

Figure 4.6: A demonstration of three designs of selection cells. The top four cells represent the result of creating new cells every time there is an interaction. The second last cell represents all four selections, with three commented out. The last cell represents the previous cell folded, and is the design we finalized on for B2.

visualization states. For interactions with the data pane, we create a *column distribution cell* that contains the logic of grouping and visualizing the values (e.g., (4)). For selections on the visualizations, we create a *selection cell* that contains the predicates of the current selections (e.g., (10)). The automatically injected cells allow analysts to reference and replay previous interactions directly in the notebook.

The rest of the visualization state can be "pulled" explicitly into the notebook narrative at relevant times by clicking on *"snapshot"*, which captures an SVG representation of the state of all visible visualizations in a notebook cell and the code to derive the data for the respective visualizations in comments.

However, even just with the selection specifications in the reified cell, their accumulation could still produce clutter. To further reduce clutter, if multiple selections are made without the analyst switching to the notebook pane, all their reifications are merged into a single cell, with the code for all but the last selection commented out and folded (with a `code_mirror` API), as shown in Fig. 4.6. This way, the default space devoted to each interaction "session" in the notebook is small and constant, but the history is easily accessed by code unfolding for replay, copy-paste, and other purposes. If analysts still find the injected cells disruptive to their notebook flow, they can toggle the hiding of all generated cells with the *"Toggle ●"* (11).

Bringing Interactivity to Code Cells

To introduce interactivity into the notebook, B2 implements *reactive cells*. These are in some sense the mirror image of reified cells—instead of persisting interactive events, they make the (persistent) code cells interactive. With reactive cells, analysts can create their own custom interactions using a visualization package of their choosing, expanding the expressiveness of the default interactive visualizations. A cell is made reactive by simply prefixing its code with a Jupyter "magic" command, `%%reactive`. B2 ensures that the notebook's JS component executes the cell after every visual selection. If the magic command has a `-df <variable>` flag, it executes after the visualization that is named after the variable in the flag (e.g., `"STATE_fires_df_dist"` in (4) and (5)). In every other way, reactive cells are regular notebook cells—for example, they can be moved and deleted.

4.6 Technical Highlights

For readers interested in implementing B2-like techniques, we describe the algorithm used to synthesize interactions and the system architecture.

Synthesizing Interactions from Queries

Section 4.5 describes the high level process we use to synthesize the interactions from the queries. Here we explain in more detail the technical detail of the algorithm used. The algorithm is developed based on data lineage research in database research. Data lineage allows us to map a row of a query's output to the set of rows of the input tables that contributed to its content [24]. In the context of interactive visualizations, data lineage allows us to connect a visual selection (rows from the table which the visualization is based on) to other subsets of data, i.e., a filter.

B2 is not the first visualization system to make use of this connection. *Smoke* was an important reference prior work that used the concept of data lineage to implement cross-filtering [111]. In Smoke, all operators are instrumented such that developers of an interactive visualization express linked visualizations as "lineage queries". Smoke deploys *physical instrumentation* of data lineage because it allows for more efficient query execution. However, B2 requires a logical approach to allow the user to access the results in a readable query format, in addition to the result of the lineage query.

As a result, we looked to prior work in logical lineage capture in B2. *Perm* is a system that derives the lineage query via transforming the original query with "provenance rewrite rules" that capture a wide (though not complete) range of queries [43]. B2's algorithm is developed based on *Smoke* and *Perm*. We first walk through an example shown in Fig. 4.7 to outline the method and we then explain the algorithm shown in Listing 1.
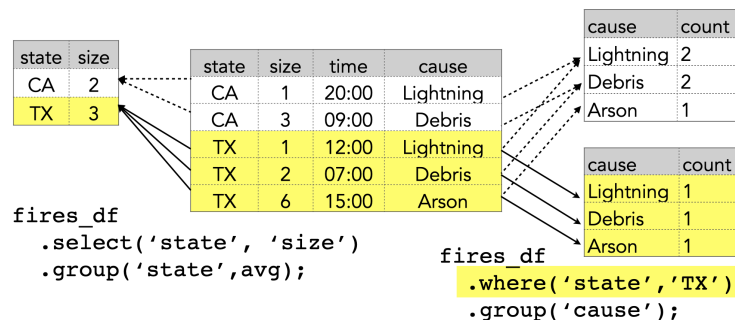
Figure 4.7: An illustration of how B2 synthesizes interaction queries: first identify rows in `fires_df` responsible for producing the selected row in `state_df`, and then apply the derivation for `cause_df` based on the rows responsible.

Fig. 4.7 shows the backing tables of two visualizations: average size of fire by US state, and count of fire by cause. Given a selection of the US state `TX`, we can identify the rows in the original data frame[4], `fires_df` (highlighted in yellow). Then we can apply the same logic of computing the distribution of the causes of fires by applying the logic on the rows in `fires_df` as part of the selection's lineage.

Listing 1 provides the pseudocode of the algorithm. The main function, SYNTHESIZEIN-TERACTION, applies the current selections to the target data frame and returns a data frame filtered by the selections. Line 2 obtains the original data frame of the query through the function GETORIGINALDF. The details of its implementation are omitted, but the gist is that it recursively walks through the operators traced until an original data frame is reached. Since we only support queries without joins, there will be only one original data frame. The tracing is achieved by creating a wrapper `datascience` library to capture the specification of each data frame as an abstract syntax tree (AST), recorded in a hidden field of the data frame, `_ops`.

Now that we have the original data frame, we iteratively apply each selection to the data frame using APPLYSELECTION (lines 4 to 5). The filtered data is then used to replace the original data frame (line 7), yielding the final result (line 8). Function APPLYSELECTION applies the selection to the data frame by deriving the rows of the original data frames linked to the selected column values. In Line 11, we first get the original data frame of the column that was selected via function GETSELORIGINALDF. We then check if this original data frame matches that of the target data frame. If it does not, we attempt to join the two original data frames (lines 13 to 15). If the join is not possible, the function returns the original data frame without any filtering logic applied (line 17). Lastly, we apply the

---

[4] "Original" data frames are defined as data frames loaded from external sources and not derived from any other data frames.

selection as a predicate to the data frame `sub_ops`) and return the result (lines 22 and 23).

Note that B2 only supports lineage for selections over columns from the original data frame and do not yet support queries involving joins. However this is not a critical limitation because users could always join the data into one table (denormalization) before they use B2. This is a common pratice in visual analytic tools [138]. That being said, future work could continue to apply methods from database lineage literature to further enhance queries supported.

System Architecture

Components. A B2 instance is composed of: (1) a Python runtime that tracks data frame operations and handles UI requests, (2) a JavaScript runtime that injects the dashboard component and manages the event-loop within the browser, and (3) a communication layer between the Python and JavaScript runtimes, `Comm` [132].

Algorithm 1 Interaction Synthesis Algorithm

```
 1: function SYNTHESIZEINTERACTION(sels, df)
 2:     original ← GETORIGINALDF(df)
 3:     filtered ← COPY(original)
 4:     for sel ∈ sels do
 5:         filtered ← APPLYSELECTION(sel, filtered)
 6:     end for
 7:     new_df ← REPLACEBASE(filtered, df)
 8:     return new_df
 9: end function
10: function APPLYSELECTION(sel, original)
11:     sel_original ← GETSELORIGINALDF(df)
12:     if sel_original! = original then
13:         joins ← GETJOINABLEDFS(sel_original)
14:         if original ∈ joins then
15:             sub_ops ← JOIN(original, sel_original.ops)
16:         else
17:             return original
18:         end if
19:     else
20:         sub_ops ← COPY(original.ops)
21:     end if
22:     pred ←CREATEPREDICATE(sel)
23:     return WHERE(sub_ops, pred)
24: end function
```
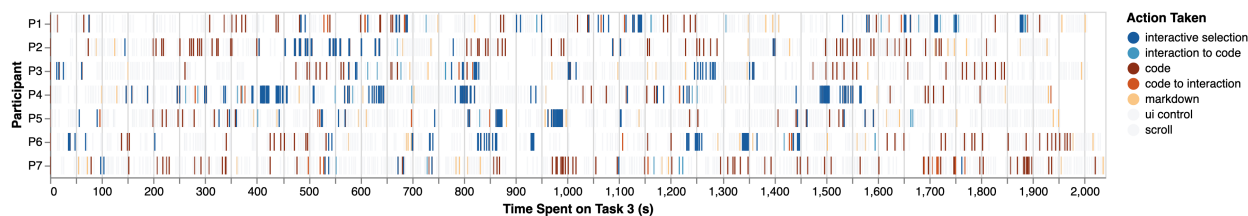
Figure 4.8: Participants' interaction traces while working on task 3, an open-ended task to model the dataset. Each participant's activity is represented by a horizontal strip plot, with participants arrayed down the visualization. Orange colors represents the work done in the code domain, and the blue colors represent the work done in the interactive visualization domain.

Event Flow. When a user selects a visual mark, the visualization implemented in Vega captures the selection event and invokes a B2 function that sends the selection values to the Python runtime. The Python runtime then synthesizes the selection code based on existing active selections. The synthesized code is sent to the JavaScript runtime, which then executes the code in a ● selection cell. The function invoked then loops through all the currently visualized data frames and apply the selection to each and sends the serialized values to the JavaScript runtime. The JavaScript runtime renders the filtered values in the visualizations and checks if the new selection matches the current selection. If not, the new selection is drawn to synchronize state between the Python and JavaScript runtimes. Note that this double round-trip design is necessary to support the bi-directional execution of interactions from not just the UI but also the code.

4.7 Evaluation: First-Use Study

To evaluate the usability of B2's bridges, we conducted a first-use study with 7 representative users[5], including 6 college students who have taken an upper-division data science course, and 1 data scientist from industry. All participants had experience using Jupyter notebooks and data frames, and their average self-reported data science expertise was 3.7 on a 5-point Likert scale ($\sigma = 0.6$). Participants also reported regularly using static visualizations for their day-to-day analysis ($\mu = 4.2, \sigma = 0.7$), and only sometimes use interactive visualizations ($\mu = 2.9, \sigma = 1.0$).

Methods

Due to the COVID-19 "shelter-in-place" order, we conducted the studies over video conference using a hosted Jupyter Notebook. We began each study with a 30-minute tutorial of

---

[5] COVID-19 affected our ability to broadly recruit participants.

B2's features, and then asked participants to complete three data analysis tasks. The tasks used an open dataset of logged calls to the local police department [104], which we chose to be interesting to participant. Tasks began specifically-focused and then transitioned to being more open-ended: (1a) identify the top two types of offenses on the weekend; (1b) verify that the result from the previous task holds on another dataset; (2a) identify how the locations of calls skew based on different factors; (2b) note factors you have not looked at; (3) explore the data further and share observations and recommendations for the police department. We asked the participants to record their findings in markdown cells and to think aloud.

Participants took 45–60 minutes to complete the three tasks. At the conclusion of the study, we administered an exit survey to measure the usefulness of B2 features, and to debrief participants about their experiences. Participants were compensated with $30 Amazon gift cards.

Quantitative Results

We instrumented the Jupyter notebook to log all user interactions with elements of the notebook and dashboard, including B2 API invocations, interactive selections on the visualizations, clicks on the column pane, and clicks on the drop-down menus of individual visualizations. To analyze this data, we computed the count of logged entry by type, and report the average and standard deviation across participants.
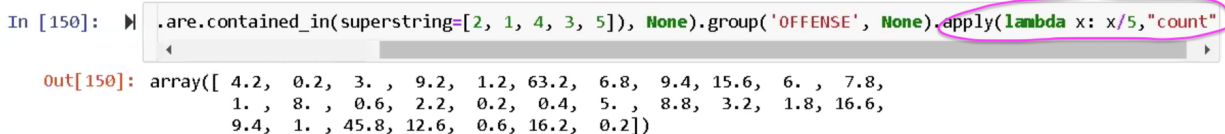
Crossing the Semantic Gap: On average, participants clicked columns from the dashboard listing 17 times ($\sigma = 4.4$), and selected marks in the visualizations 63 times ($\sigma = 41.9$). Participants manually invoked the B2 APIs (Listing 4.2) in code an average of 26.7 times ($\sigma = 21.9$); this number rises to an average of 112 times ($\sigma = 84.8$) when we include automatic invocations as a result of reactive cells.

Crossing the Layout Gap: Participants controlled the dashboard (e.g., adjusting its size, hiding or (re)moving visualizations, etc.) an average of 12.6 times ($\sigma = 5.3$), and scrolled to navigate the notebook's linear flow an average of 536 times ($\sigma = 140.7$). To navigate from the dashboard to the notebook, participants clicked the *"Find Defining Cell"* button an average of 2.2 times ($\sigma = 1.9$) and none of the participants used B2's functionalities to navigate from the notebook to the dashboard.

Crossing the Temporal Gap: On average, participants recorded transient interactive visualizations to the notebook using snapshots 3.7 times ($\sigma = 2.9$), and made cells interactive with the `%%reactive` magic command 4.43 times ($\sigma = 2.94$).

Post-study survey results

On 5-point Likert scales, participants positively rated B2 overall ($\mu = 3.7, \sigma = 0.6$), with similar ratings for interactive visualization ($\mu = 4.2, \sigma = 0.75$), static visualization ($\mu =$

```
In [150]: ▶  .are.contained_in(superstring=[2, 1, 4, 3, 5]), None).group('OFFENSE', None).apply(lambda x: x/5,"count"
          ◀                                                                                                        ▶

Out[150]: array([ 4.2,  0.2,  3. ,  9.2,  1.2, 63.2,  6.8,  9.4, 15.6,  6. ,  7.8,
                  1. ,  8. ,  0.6,  2.2,  0.2,  0.4,  5. ,  8.8,  3.2,  1.8, 16.6,
                  9.4,  1. , 45.8, 12.6,  0.6, 16.2,  0.2])
```

Figure 4.9: Modifying a generated query to further derive the average count, which is not
possible through interactions.

$3.7, \sigma = 0.9$), B2's programmatic API ($\mu = 3.6, \sigma = 0.7$), and interaction histories ($\mu = 3.5, \sigma = 0.92$). In terms of ease-of-use, participants rated B2 a 3.1 ($\sigma = 0.3$), but responded that they were likely to use B2 in the future ($\mu = 3.6, \sigma = 0.8$).

Qualitative Results

We observed participants quickly grasped how to use code and interactions together in a complementary fashion. One common pattern was using code to first process data before visualizing it. For instance, P2 first attempted to visualize the time column by clicking on the pane, but B2 flagged that there were too many unique values and it would not be able to synthesize the code to visualize the data. In response, P2 switched to the notebook and inspected the values in the `Time` column with code. They then extract out the `Hour` from the `Time` column with a regex function and visualized the distribution by `Hour` using the `.vis` API. Having built the hour distribution visualization, P2 then interacted with it by brushing time ranges to further filter other visualizations. Another common pattern was using code to compute statistics using the interactively filtered visualizations. For instance, using *"Copy Code to Clipboard"*, P3 copied the code used to derive an interactive visualization and added their own functions to compute the average values (Fig. 4.9). Similarly, P7 was inspired by B2's visualization, and wrote their own visualization in `matplotlib` using the data frame B2 reified in the previous code cell, before then computing statistics in code (Fig. 4.10).

These types of patterns switching between code and interactive visualization, and vice-versa, were common across all participants. Fig. 4.8 visualizes the interaction traces of participants in task 3, and we can see frequent interleavings between coding and interacting with the visualizations for all participants and throughout the duration of this open-ended task. Indeed, in the post-study debrief, participants shared enthusiastic comments about B2's features. For instance, P5 wanted them to be able to *"create custom visualizations for their day to day work"* while P6 wanted them in order to *"share the raw underlying data of a chart [with coworkers]"*.

When working on task 2, which prompted exploration of call location, all but one participant chose to use interactive visualizations. In particular, these participants chose to create an initial heatmap, make it reactive using B2's `%%reactive` magic command, and then

select different values in the dashboard histograms. In contrast, P7 primarily used code to
re-derive the `Lat Lon` information. P7's manual iteration was slower and resulted in fewer
insights in the time given (Fig. 4.11). In either case, participants frequently needed to refer
to documentation, even when using popular libraries such as `numpy` and `matplotlib`. This
need to consult resources outside of the notebook appeared to impose a high cost, and several
participants interacted with the dataset using B2's visualizations before committing to using
extensive coding to answer the task. For instance, when thinking out loud, P5 shared that
"I need to dig into this [with code] but maybe later".

The fact that the interactive visualizations were automatically synthesized proved to be
important. P1 commented that the features offered by B2 are *"hard to do [for them] with
plotting libraries"*. P5 said that B2 *"spares the user from tedious commands"* and *"facilitates
the EDA process"*. This ease of use also prompted comments about targeting B2 at those
less familiar with code. P7 said B2 *"engages people who would like to conduct research and
data analysis but don't have much programming experience"*, and P2 said *"I think this would
be awesome to show students in my data science class"*.

There was also evidence that incorporating interactions into the programming process
may require a mental shift. We observed this shift in thinking most saliently in how partici-
pants chose to complete Task 1b. To verify if the answer to task 1a holds true for a different
dataset with the same schema, five analysts exported their interactive work to code using the
"Copy Code to Clipboard". With this code, they replaced the data frame from task 1a with
the new data frame loaded in 1b, executed the query and evaluated the results in code. The
other 2 analysts manually re-applied the interactions from the Task 1a on the new dataset
and were not aware of using B2's capability to translate interaction results to code.
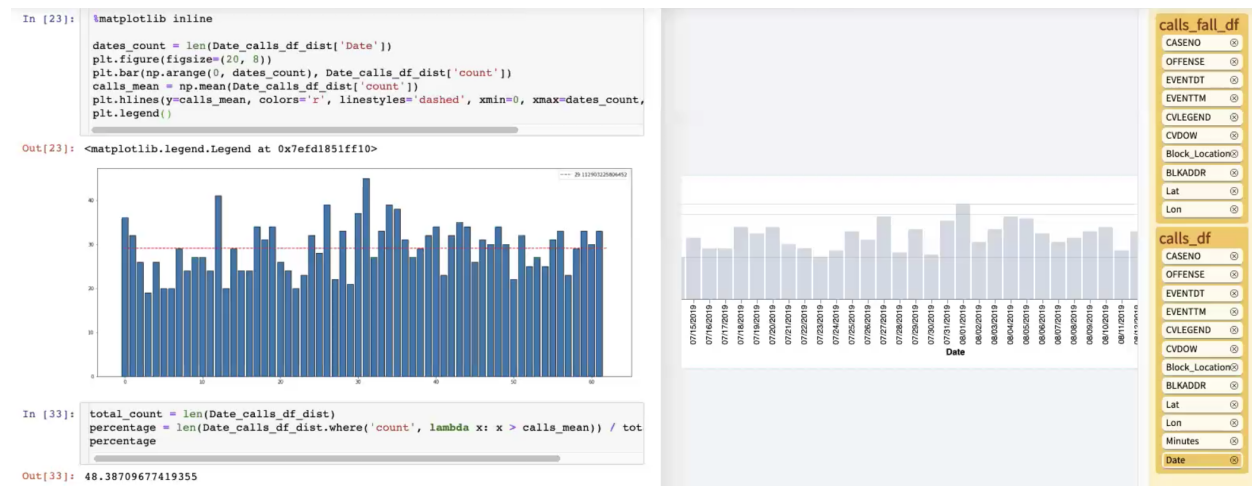
Figure 4.10: Using code to create a custom plot with the average line in the chart, and
computing percentages.
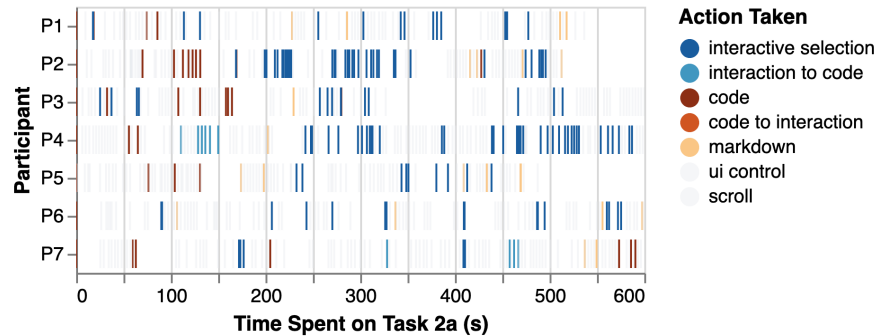
Figure 4.11: Participants' interaction traces while working on task 2. To plot the heatmap, all analysts initially interleaved between code and interactions. P3's gap between 300–400s is when they searched the web for information about the city. P7 used code, interaction, and the `get_filtered_data` API to inspect whether null values were present.

When asked in the exit interview about their approach, they commented that the alternative approach of mapping interactions to code hadn't crossed their minds. In other words, the first approach requires that the analyst understands how B2 let them work across the two mediums, and it seems from the results that it's not always apparent. P4 said *"It might be because I am not very familiar with the idea of interactive visualizations, I find it not so easy to adapt to the tools given my background"*. Further study is required to determine whether this mental shift will be ameliorated with increased and longer-term exposure to interactive visualizations and exploratory visual analysis, or whether it represents a more fundamental overhead of switching between these two paradigms that future versions of B2 can address.

Participants had more mixed opinions about the auto-injected reified selection cells. For instance, P2 mentioned that they could *"make the notebook very crowded"* and suggested that we *"have it appear in some separate tab"*; P3 mentioned that it was *"annoying that the left side gets populated by lines of code each time I click on the visualization on the right"*. However, the most experienced data analyst (with years of industry experience), appreciated the feature, saying *"being able to remember the current selection and history"* is important *"because I often forget what I've done if I go get coffee or lunch"*, and that the feature may help with reproducing the analysis (*"I often have one team reach out to get a version of an analysis that I did for another team"*).

Interestingly, although B2 does provide a feature to combat this issue (the *"Toggle ●"* button), neither P2 or P3 made use of it. They explained that they forgot the toggle control feature. Perhaps the issue could be addressed by more use with the tool. It could also be that, fundamentally, some analysts do not like their notebook to be modified. Regardless, this initial finding speaks to well-known explanation-exploration conflict in notebook ex-

ploration [116] and we need to study further to understand the design and cognitive issues present.

Lastly, although participants were able to easily adjust B2 to have more screen real-estate for code (or visualizations), none of the participants did. This could be due to a lack of familiarity to the many controls B2 provide. Though it may be more likely that participants did find both code and visualizations helpful at all times.

4.8   Conclusion

We contribute B2, a library of techniques to bridge the gaps between code and interactive visualizations in computational notebooks. We identified these gaps by studying prior work that surveyed and interviewed data scientists, and then evolved our understanding based on the tradeoffs manifest in prototype implementations we piloted with representative users. These gaps — and the way they arise from the metaphors, layouts, and timescales of the two styles of work — are a significant influence on the specific bridges we ultimately chose to instantiate in B2. Our first-use study validates that these bridges indeed help users iterate between code and interactive visualization more seamlessly.

Having addressed this first set of questions, we have uncovered a number of additional challenges that appear to be rich topics for further investigation.

Non-Linear Workflows & Asynchronous Collaborations: Our user studies yielded relatively short sessions, but notebook explorations could often become *non-linear*, where analysts may want to jump between sections of analysis. These jumps cause changes in *context*, both in terms of the program state and analysts' mental models. The challenge of managing segments of analysis state is also faced in collaboration settings, where analysts sometimes jump through cells and need to understand cell dependencies [151]. Supporting analysts in navigating between segments of analysis in space and time poses additional challenges for the layout and temporal gaps. Future work could explore bringing in techniques such as bookmarking, linking, and annotations from asynchronous collaboration literature [61]. For instance, we could consider bookmarks containing groups of visualizations that match sections of the notebook that may correspond to a set of visualizations. We could also link the states of charts and data frames—when a data frame is modified, the charts reactively update in accord.

DSLs Beyond the Data Frame: A data frame API is an attractive semantic foundation for bridging visualization and code, because (a) it is broadly useful across many classes of data (tables, matrices) and analysis steps (data preparation, database-style queries, linear algebra), and (b) it maps well to the core visual aspects of EDA. However there are other data science microcosms where it might be interesting to bridge code and data. One popular example today is deep learning, where APIs like Keras [49] or Tensorflow [2] are often coupled with domain-specific charting packages like Tensorboard [140]. How might a different set of

data frame operations and visualizations impact the gaps we identify in this paper, and might they suggest new gaps and bridges?

Links Beyond Cross-Filter: In this paper, we focus on synthesizing cross-filters as our main interactive mechanism, but there are many other possible techniques to consider. For instance, Yi, et al. present one taxonomy of this space, with seven different categories of interaction techniques [170] covering dozens of ideas in prior work. For many of these, it would be interesting to consider how data properties and operation lineage could aid in automatic synthesis of useful interactions. As a more general question, if we expand B2 to accommodate many different interaction models, how might we prioritize automatic selection of the most appropriate model, or design semi-automatic interfaces for interaction model selection?

B2 is available as open-source software at https://github.com/yifanwu/b2.

Chapter 5

Summary and Future Work

In this thesis, we contributed three novel artifacts that collectively support more scalable and fluid human-data interfaces, and verified the utility of interaction history. In this chapter, we summarize the key insights and highlight interesting future research directions.

New Interaction History Affordances

In Interaction Snapshots and B2, study participants found access to past interactions so helpful that they suggested ways to edit and iterate on the reified interaction histories. One suggestion is to make reified interaction history editable. For Snapshots, users wanted to be able to organize snapshots into groups, remove ones no longer needed, and add annotations. For B2, users wanted to define multiples dashboard view and connect the dashboard views with segments of the notebook cells. We hypothesize that the design of "scrollytelling", popularized in recent journal articles [128], can help. As the user scrolls through to different part of the notebook, the dashboard can update accordingly.

Another direction builds on top of a related research area, *visual analytics provenance*. Researchers in the field find that the generation of visualizations are often limited to the final visual artifact, and the "trails" of analysis that not captured. *VisTrails* tackles this problem by capturing the visualization pipeline, or dataflow; users can then query and edit the dataflow, and to apply the dataflow to other inputs [20]. Gotz et al. tackles a similar problem, insight provenance, using a novel *action* abstraction as semantic building blocks [44].

Computational notebooks are well suited to leverage these research ideas, and we hypothesize that the use of history and the inter-operation provided by B2 can help. The code cells in notebooks already provides natural units of provenance, and the visualization interactions are reified into code cells through B2. Users can manually change the dataset and replay the code cells that captures the interactions and related operations. However, because notebooks permit out-of-order executions, reproducibility is still a challenge. Future work may leverage program slicing techniques [55] or lean into a new execution model like

those in *Observable* [101] and *Streamlit* [141].

Human-Data Interfaces as Distributed Systems

The lens of distributed system on interactive visualizations has not only been fruitful for our research. It also highlights open questions and opportunities.

One direction is to support the programming of *collaborative* human-data interfaces, which is an emerging use case for interactive visualizations [118, 100]. Programming for collaboration has been widely studied in the collaborative groupware community [47, 137, 136], but less so in the interactive visualization space, which supports different user interactions than document edits.

Programming for collaboration tools is fundamentally different from that for a single user. The DIEL model only supports one source of change—those observed on the client through its API. However in a collaborative setting, there are multiple sources of changes: user A and user B could both be interacting in separate interfaces concurrently. While we can certainly try to enforce a single global ordering of events via a linearizable coordination service using an algorithm like MultiPaxos [84]. However such a solution often leads to latencies unacceptable to modern UI design principle of responsiveness [69].

To support non-blocking interfaces in the context of collaboration, we need to support a distributed notion of time. This is where the lens of distributed systems could help us. Distributed system programmers have long recognized that distributed time supports only partial ordering (as opposed to a global ordering). One key concept to to help distributed systems developers reason about concurrent events is *vector clocks* [85]. While vector clocks have been used for collaborative editors [156], programming with it had been challenging. Recent advances in languages for distributed programming, such as like *Bloom* [7], offer simple mechanisms for strong coordination using monotonic programming constructs. Bloom allows for multiple nodes to generate events simultaneously, and provides language support to determine when the concurrency can result in inconsistent experiences or outcomes across users [7]. Conway explored augmenting Bloom for collaborative editing [30], but future work is still needed to specialize the abstractions for interactive visualizations.

Beyond programming, the distributed systems lens is also useful for design. By modeling the user as an agent in a distributed system along with the UI and remote databases, we can capture and reason with UX anomalies. Like systems, users also take time to process signals and perform actions. We know that users take more than 100ms to observe a change on the screen and perform an action (like clicking) [21]. We also know that some interactions, such as drawing a brush, or making multiple selections by clicking on each item, take time to perform. Given these latencies in human processes, it is possible for different events to go out of order, causing the UI to end up in an unexpected state. We have explored these inconsistencies and reconciliation methods in a prior work [166], but much remains to be done.

A New Medium of Programming and Interactions

B2 is only one step in the direction of tools that inter-operate between programming and interactions. One obvious future work direction is to support interactions beyond cross-filtering. For instance, we can inter-operate code with a Tableau-like shelf interactions through pivot operators [134]. Another future work direction is to expand beyond relational operators and support composable abstractions in other domains. For instance, machine learning is a good candidate given its highly structured workflows.

B2 could also serve as an inspiration to help complete "unfinished" bridges in prior work, where the interactions can generate code, but not the other way around. Completing the unfinished bridges from the code to interaction may help non-technical users make sense of the code in a process that they understand. It may also serve as an education tool to help users learn coding. One example is data wrangling: *Wrangler* leverages a DSL based on Potter's Wheel [114] to synthesize executable programs from interaction sequences [72]. Another example is SQL query composition: *Sieuferd* is an interface that allows users to expresses all of SQL through direct manipulation [11]. Future work could explore ways to synthesize interactive controls from programming history.

Of course, B2 is not the only approach to building this bridge. Interaction history is just one way to share specific types of state between the two mediums. One alternative is templating. `mage` allows tools to represent themselves as both code and GUI as needed through templates [77]. Ivy is another recent project using templates to help users synthesize Vega-Lite specifications [95]. Another option is programming by example: *Falx* allows users to interactively define programs via thesis [152], and *Wrex* allows users a Excel flash fill experience but with Pandas [36]. Yet another line of work is output-directed programming [65]. Users can either make direct manipulations or write code to edit the final artifact. This way, edits not possible through direct manipulation can instead be performed via code edits. It would be interesting work to see how interaction history could be used to enhance these tools to push the envelope for the future of programming.

Bibliography

[1] "188 million US wildfires". In: URL: https://www.kaggle.com/rtatman/188-million-us-wildfires.

[2] Martın Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "Tensor-flow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 265–283.

[3] Serge Abiteboul, Victor Vianu, Brad Fordham, and Yelena Yesha. "Relational transducers for electronic commerce". In: *Journal of Computer and System Sciences* 61.2 (2000), pp. 236–269.

[4] Eytan Adar. "GUESS: a language and interface for graph exploration". In: *Proceedings of the SIGCHI conference on Human Factors in computing systems*. 2006, pp. 791–800.

[5] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. "BlinkDB: queries with bounded errors and bounded response times on very large data". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 29–42.

[6] Sara Alspaugh, Nava Zokaei, Andrea Liu, Cindy Jin, and Marti A Hearst. "Futzing and moseying: Interviews with professional data analysts on exploration practices". In: *IEEE transactions on visualization and computer graphics* 25.1 (2018), pp. 22–31.

[7] Peter Alvaro, Neil Conway, Joseph M Hellerstein, and William R Marczak. "Consistency Analysis in Bloom: a CALM and Collected Approach." In: *CIDR*. 2011, pp. 249–260.

[8] Robert Amar, James Eagan, and John Stasko. "Low-level components of analytic activity in information visualization". In: *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005*. IEEE. 2005, pp. 111–117.

[9] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. "Spark sql: Relational data processing in spark". In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. ACM. 2015, pp. 1383–1394.

[10] Eric Z Ayers and John T Stasko. *Using graphic history in browsing the World Wide Web*. Tech. rep. Georgia Institute of Technology, 1995.

[11] Eirik Bakke and David R Karger. "Expressive query construction through direct manipulation of nested relational results". In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 1377–1392.

[12] Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. "Ringer: web automation by demonstration". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2016, pp. 748–764.

[13] Andrea Batch and Niklas Elmqvist. "The interactive visualization gap in initial exploratory data analysis". In: *IEEE transactions on visualization and computer graphics* 24.1 (2017), pp. 278–287.

[14] Leilani Battle, Remco Chang, and Michael Stonebraker. "Dynamic prefetching of data tiles for interactive visualization". In: *SIGMOD*. 2016.

[15] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. "AutoPandas: neural-backed generators for program synthesis". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–27.

[16] Philip A Bernstein and Nathan Goodman. "Concurrency control in distributed database systems". In: *ACM Computing Surveys (CSUR)* 13.2 (1981), pp. 185–221.

[17] Alan F Blackwell, Carol Britton, A Cox, Thomas RG Green, Corin Gurr, Gada Kadoda, MS Kutar, Martin Loomes, Chrystopher L Nehaniv, Marian Petre, et al. "Cognitive dimensions of notations: Design tools for cognitive technology". In: *Cognitive technology: instruments of mind*. Springer, 2001, pp. 325–341.

[18] *bloomberg*. 2019. URL: https://github.com/bloomberg/bqplot.

[19] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. "D^3 data-driven documents". In: *IEEE transactions on visualization and computer graphics* 17.12 (2011), pp. 2301–2309.

[20] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos E Scheidegger, Cláudio T Silva, and Huy T Vo. "VisTrails: visualization meets data management". In: *Proceed-

ings of the 2006 ACM SIGMOD international conference on Management of data.
2006, pp. 745–747.

[21] Stuartk Card, THOMASP MORAN, and Allen Newell. "The model human processor-An engineering model of human performance". In: *Handbook of perception and human performance.* 2 (1986), pp. 45–1.

[22] Sarah Chasins, Shaon Barman, Rastislav Bodik, and Sumit Gulwani. "Browser record and replay as a building block for end-user web automation tools". In: *Proceedings of the 24th International Conference on World Wide Web.* 2015, pp. 179–182.

[23] Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. "What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities". In: (2020).

[24] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. *Provenance in databases: Why, how, and where.* Now Publishers Inc, 2009.

[25] Rada Chirkova, Jun Yang, et al. "Materialized views". In: *Foundations and Trends®️ in Databases* 4.4 (2012), pp. 295–405.

[26] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. "Programmatic and direct manipulation, together at last". In: *ACM SIGPLAN Notices* 51.6 (2016), pp. 341–354.

[27] David Clark and Patrick Regan. *Mass Mobilization Protest Data.* Version V4. 2016. DOI: 10.7910/DVN/HTTWYL. URL: https://doi.org/10.7910/DVN/HTTWYL.

[28] Edgar F Codd. "A relational model of data for large shared data banks". In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[29] Neil Conway, Peter Alvaro, Emily Andrews, and Joseph M Hellerstein. "Edelweiss: Automatic storage reclamation for distributed programming". In: *Proceedings of the VLDB Endowment* 7.6 (2014), pp. 481–492.

[30] Neil Robert George Conway. "Language Support for Loosely Consistent Distributed Programming". PhD thesis. UC Berkeley, 2014.

[31] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. "Synopses for massive data: Samples, histograms, wavelets, sketches". In: *Foundations and Trends in Databases* 4.1–3 (2012), pp. 1–294.

[32] Zachary T Cutler, Kiran Gadhave, and Alexander Lex. "Trrack: A Library for Provenance Tracking in Web-Based Visualizations". In: (2020).

[33] John DeNero, David Culler, Sam Lau, and Alvin Wan. *A Berkeley library for introductory data science.* 2015. URL: https://github.com/data-8/datascience/.

[34] Amol Deshpande and Joseph M Hellerstein. "Decoupled query optimization for federated database systems". In: *Proceedings 18th International Conference on Data Engineering.* IEEE. 2002, pp. 716–727.

[35] Bolin Ding, Silu Huang, Surajit Chaudhuri, Kaushik Chakrabarti, and Chi Wang. "Sample+ seek: Approximating aggregates with distribution precision guarantee". In: *Proceedings of the 2016 International Conference on Management of Data.* 2016, pp. 679–694.

[36] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. "Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems.* ACM. 2020.

[37] Jean-Daniel Fekete and Romain Primet. "Progressive analytics: A computation paradigm for exploratory data analysis". In: *arXiv preprint arXiv:1607.05162* (2016).

[38] Mi Feng, Cheng Deng, Evan M Peck, and Lane Harrison. "HindSight: Encouraging Exploration through Direct Encoding of Personal Interaction History". In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), pp. 351–360.

[39] Mi Feng, Cheng Deng, Evan M Peck, and Lane Harrison. "Hindsight: Encouraging exploration through direct encoding of personal interaction history". In: *IEEE transactions on visualization and computer graphics* 23.1 (2016), pp. 351–360.

[40] Danyel Fisher, Igor Popov, Steven Drucker, et al. "Trust me, I'm partially right: incremental visualization lets analysts explore large datasets faster". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM. 2012, pp. 1673–1682.

[41] Yupeng Fu, Kian Win Ong, Yannis Papakonstantinou, and Erick Zamora. "Forward: Data-centric UIs using declarative templates that efficiently wrap third-party JavaScript components". In: *Proceedings of the VLDB Endowment* 7.13 (2014), pp. 1649–1652.

[42] Sneha Gathani, Peter Lim, and Leilani Battle. "Debugging database queries: A survey of tools, techniques, and users". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems.* 2020, pp. 1–16.

[43] Boris Glavic and Gustavo Alonso. "Perm: Processing provenance and data on the same data model through query rewriting". In: *2009 IEEE 25th International Conference on Data Engineering*. IEEE. 2009, pp. 174–185.

[44] David Gotz and Michelle X Zhou. "Characterizing users' visual analytic activity for insight provenance". In: *Information Visualization* 8.1 (2009), pp. 42–55.

[45] Graphistry. *Supercharge Your Investigations*. 2017. URL: https://www.graphistry.com/ (visited on 09/12/2017).

[46] Thomas RG Green. "Instructions and descriptions: some cognitive aspects of programming and similar activities". In: *Proceedings of the working conference on Advanced visual interfaces*. 2000, pp. 21–28.

[47] Saul Greenberg and David Marwood. "Real time groupware as a distributed system: concurrency control and its effect on the interface". In: *Proceedings of the 1994 ACM conference on Computer supported cooperative work*. ACM. 1994, pp. 207–217.

[48] Garrett Grolemund. *Introduction to interactive documents*. July 2014. URL: https://shiny.rstudio.com/articles/interactive-docs.html.

[49] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[50] Sumit Gulwani. "Automating string processing in spreadsheets using input-output examples". In: *ACM Sigplan Notices* 46.1 (2011), pp. 317–330.

[51] Sumit Gulwani, William R Harris, and Rishabh Singh. "Spreadsheet data manipulation using examples". In: *Communications of the ACM* 55.8 (2012), pp. 97–105.

[52] Philip J Guo, Sean Kandel, Joseph M Hellerstein, and Jeffrey Heer. "Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts". In: *Proceedings of the 24th annual ACM symposium on User interface software and technology*. 2011, pp. 65–74.

[53] Chris Harrison, Brian Amento, Stacey Kuznetsov, and Robert Bell. "Rethinking the progress bar". In: *Proceedings of the 20th annual ACM symposium on User interface software and technology*. 2007, pp. 115–118.

[54] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R Klemmer. "Design as exploration: creating interface alternatives through parallel authoring and runtime tuning". In: *Proceedings of the 21st annual ACM symposium on User interface software and technology*. 2008, pp. 91–100.

[55] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. "Managing Messes in Computational Notebooks". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM. 2019, p. 270.

[56] Marti Hearst. *Search user interfaces*. Cambridge university press, 2009.

[57] Jeffrey Heer, Maneesh Agrawala, and Wesley Willett. "Generalized selection via interactive query relaxation". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2008, pp. 959–968.

[58] Jeffrey Heer and Michael Bostock. "Declarative language design for interactive visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 16.6 (2010), pp. 1149–1156.

[59] Jeffrey Heer, Jock Mackinlay, Chris Stolte, and Maneesh Agrawala. "Graphical histories for visualization: Supporting analysis, communication, and evaluation". In: *IEEE transactions on visualization and computer graphics* 14.6 (2008).

[60] Jeffrey Heer and Ben Shneiderman. "Interactive dynamics for visual analysis". In: *Queue* 10.2 (2012), p. 30.

[61] Jeffrey Heer, Fernanda B Viégas, and Martin Wattenberg. "Voyagers and voyeurs: supporting asynchronous collaborative information visualization". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. 2007, pp. 1029–1038.

[62] Pat Helland. "Immutability changes everything". In: *Queue* 13.9 (2015), p. 40.

[63] Joseph M Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, Tali Roth, and Peter J Haas. "Interactive data analysis: The control project". In: *Computer* 32.8 (1999), pp. 51–59.

[64] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. "Online aggregation". In: *ACM SIGMOD Record*. Vol. 26. 2. ACM. 1997, pp. 171–182.

[65] Brian Hempel and Ravi Chugh. "Semi-automated svg programming via direct manipulation". In: *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. 2016, pp. 379–390.

[66] Brian Hempel, Justin Lubin, and Ravi Chugh. "Sketch-n-Sketch: Output-Directed Programming for SVG". In: *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. ACM. 2019, pp. 281–292.

[67] *How to use interactive IPython widgets to enhance data exploration and analysis*. 2019. URL: https://towardsdatascience.com/interactive-controls-for-jupyter-notebooks-f5c94829aee6 (visited on 04/25/2020).

[68]   Edwin L Hutchins, James D Hollan, and Donald A Norman. "Direct manipulation interfaces". In: *Human–Computer Interaction* 1.4 (1985), pp. 311–338.

[69]   Jeff Johnson. *GUI bloopers 2.0: common user interface design don'ts and dos*. Morgan Kaufmann, 2007.

[70]   Jupyter. *Using Interact*. 2019. URL: https://ipywidgets.readthedocs.io/en/stable/examples/Using%5C%20Interact.html (visited on 06/11/2019).

[71]   kaggle. *1.88 Million US Wildfires, 24 years of geo-referenced wildfire records*. URL: footnote%7B%5Curl%7Bhttps://www.kaggle.com/rtatman/188-million-us-wildfires%7D%7D..

[72]   Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. "Wrangler: Interactive visual specification of data transformation scripts". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2011, pp. 3363–3372.

[73]   Sean Kandel, Andreas Paepcke, Joseph M Hellerstein, and Jeffrey Heer. "Enterprise data analysis and visualization: An interview study". In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (2012), pp. 2917–2926.

[74]   Mary Beth Kery, Amber Horvath, and Brad A Myers. "Variolite: Supporting Exploratory Programming by Data Scientists." In: *CHI*. 2017, pp. 1265–1276.

[75]   Mary Beth Kery, Bonnie E John, Patrick O'Flaherty, Amber Horvath, and Brad A Myers. "Towards Effective Foraging by Data Scientists to Find Past Analysis Choices". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM. 2019, p. 92.

[76]   Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. "The story in the notebook: Exploratory data science using a literate programming tool". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM. 2018, p. 174.

[77]   Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. "mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks". In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 2020, pp. 140–151.

[78]   Mary Beth Kery, Kanit Wongsuphasawat, Kayur Patel, Donghao Ren, and Fred Hohman. "The Future of Notebook Programming Is Fluid". In: *CHI*. 2020.

[79]   Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain

Corlay, et al. "Jupyter Notebooks-a publishing format for reproducible computational workflows." In: *ELPUB*. 2016, pp. 87–90.

[80] Donald Ervin Knuth. "Literate programming". In: *The Computer Journal* 27.2 (1984), pp. 97–111.

[81] Semi Koen. *Bring your Jupyter Notebook to life with interactive widgets*. 2019. URL: https://towardsdatascience.com/bring-your-jupyter-notebook-to-life-with-interactive%20-widgets-bc12e03f0916.

[82] Tim Kraska. "Northstar: An interactive data science system". In: *Proceedings of the VLDB Endowment* 11.12 (2018), pp. 2150–2164.

[83] Milos Krstajic and Daniel A Keim. "Visualization of streaming data: Observing change and context in information visualization techniques". In: *Big Data, 2013 IEEE International Conference on*. IEEE. 2013, pp. 41–47.

[84] Leslie Lamport et al. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[85] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 179–196.

[86] Samuel Lau and Joshua Hug. "nbinteract: generate interactive web pages from Jupyter notebooks". PhD thesis. Master's thesis, EECS Department, University of California, Berkeley, 2018.

[87] Doris Jung-Lin Lee, Dixin Tang, Kunal Agarwal, Thyne Boonmark, Caitlyn Chen, Jake Kang, Ujjaini Mukhopadhyay, Jerry Song, Micah Yong, Marti A. Hearst, and Aditya G. Parameswaran. *Lux: Always-on Visualization Recommendations for Exploratory Data Science*. 2021. eprint: arXiv:2105.00121.

[88] Lauro Lins, James T Klosowski, and Carlos Scheidegger. "Nanocubes for Real-Time Exploration of Spatiotemporal Datasets". In: *TVCG* (2013).

[89] Zhicheng Liu and Jeffrey Heer. "The effects of interactive latency on exploratory visual analysis". In: *IEEE transactions on visualization and computer graphics* 20.12 (2014), pp. 2122–2131.

[90] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. "imMens: Real-time Visual Querying of Big Data". In: *Computer Graphics Forum*. Vol. 32. 3pt4. Wiley Online Library. 2013, pp. 421–430.

[91] Miron Livny, Raghu Ramakrishnan, Kevin Beyer, Guangshun Chen, Donko Donjerkovic, Shilpa Lawande, Jussi Myllymaki, and Kent Wenger. "DEVise: integrated

querying and visual exploration of large datasets". In: *ACM SIGMOD Record* 26.2 (1997), pp. 301–312.

[92]   Ophir Lojkine. *SQLite compiled to JavaScript through Emscripten.* https://github.com/kripken/sql.j 2017. (Visited on 03/17/2018).

[93]   MapD. *Platform for Lightning-Fast SQL, Visualization and Machine Learning.* 2017. URL: https://www.mapd.com/ (visited on 09/12/2017).

[94]   Andreas Mathisen, Tom Horak, Clemens Nylandsted Klokmose, Kaj Grønbæk, and Niklas Elmqvist. "InsideInsights: Integrating Data-Driven Reporting in Collaborative Visual Analytics". In: *Computer Graphics Forum.* 2019.

[95]   Andrew McNutt and Ravi Chugh. "Integrated Visualization Editing via Parameterized Declarative Templates". In: *arXiv preprint arXiv:2101.07902* (2021).

[96]   Erik Meijer. "Your mouse is a database". In: *Queue* 10.3 (2012), p. 20.

[97]   Dominik Moritz, Danyel Fisher, Bolin Ding, and Chi Wang. "Trust, but verify: Optimistic visualizations of approximate queries for exploring big data". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems.* ACM. 2017, pp. 2904–2915.

[98]   Dominik Moritz, Bill Howe, and Jeffrey Heer. "Falcon: Balancing interactive latency and resolution sensitivity for scalable linked visualizations". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems.* 2019, pp. 1–11.

[99]   Mozilla. *Using Web Workers.* 2018. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers (visited on 02/23/2018).

[100] Rupayan Neogy, Jonathan Zong, and Arvind Satyanarayan. "Representing Real-Time Multi-User Collaboration in Visualizations". In: *arXiv preprint arXiv:2009.02587* (2020).

[101] *Observable.* 2019. URL: https://observablehq.com/.

[102] C Olston, Michael Stonebraker, Alexander Aiken, and Joseph M Hellerstein. "VIQING: Visual interactive querying". In: *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on.* IEEE. 1998, pp. 162–169.

[103] Stack Overflow. *Developer Survey Results, Most Popular Programming Languages.* https://insights.stackoverflow.com/survey/2019. 2020. (Visited on 06/03/2020).

[104] Berkeley PD. *Calls for Service.* 2020. URL: https://data.cityofberkeley.info/Public-Safety/Berkeley-PD-Calls-for-Service/k2nh-s5h5.

[105]   Fernando Pérez. *"Literate computing" and computational reproducibility: IPython in the age of data-driven journalism.* 2013. URL: http://blog.fperez.org/2013/04/literate-computing-and-computational.html.

[106]   Peter Pirolli and Stuart Card. "Information foraging." In: *Psychological review* 106.4 (1999), p. 643.

[107]   Plotly. *Plotly.* 2020. URL: https://plotly.com/.

[108]   PosgreSQL. *PostgreSQL: The World's Most Advanced Open Source Relational Database.* https://www.postgresql.org/. 2019. (Visited on 02/10/2019).

[109]   Atul Prakash and Michael J Knister. "A framework for undoing actions in collaborative systems". In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 1.4 (1994), pp. 295–330.

[110]   Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. "A commutative replicated data type for cooperative editing". In: *2009 29th IEEE International Conference on Distributed Computing Systems.* IEEE. 2009, pp. 395–403.

[111]   Fotis Psallidas and Eugene Wu. "Smoke: Fine-grained lineage at interactive speed". In: *arXiv preprint arXiv:1801.07237* (2018).

[112]   Sajjadur Rahman, Maryam Aliakbarpour, Ha Kyung Kong, Eric Blais, Karrie Karahalios, Aditya Parameswaran, and Ronitt Rubinfield. "I've seen" enough" incrementally improving visualizations to support rapid decision making". In: *Proceedings of the VLDB Endowment* 10.11 (2017), pp. 1262–1273.

[113]   Raghu Ramakrishnan and Johannes Gehrke. *Database management systems.* McGraw Hill, 2000.

[114]   Vijayshankar Raman and J Hellerstein. "Potters wheel: an interactive framework for data cleaning and transformation". In: *Working draft* (2001).

[115]   Adam Rule, Ian Drosos, Aurélien Tabard, and James D Hollan. "Aiding collaborative reuse of computational notebooks with annotated cell folding". In: *Proceedings of the ACM on Human-Computer Interaction* 2.CSCW (2018), pp. 1–12.

[116]   Adam Rule, Aurélien Tabard, and James D Hollan. "Exploration and explanation in computational notebooks". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems.* ACM. 2018, p. 32.

[117]   Sunil K. Sarin and Nancy A. Lynch. "Discarding obsolete information in a replicated database system". In: *IEEE Transactions on Software Engineering* 1 (1987), pp. 39–47.

[118] Ali Sarvghad and Melanie Tory. "Exploiting analysis history to support collaborative data analysis". In: *Proceedings of the 41st Graphics Interface Conference*. 2015, pp. 123–130.

[119] Arvind Satyanarayan and Jeffrey Heer. "Lyra: An interactive visualization design environment". In: *Computer Graphics Forum*. Vol. 33. 3. Wiley Online Library. 2014, pp. 351–360.

[120] Arvind Satyanarayan, Bongshin Lee, Donghao Ren, Jeffrey Heer, John Stasko, John Thompson, Matthew Brehmer, and Zhicheng Liu. "Critical reflections on visualization authoring systems". In: *IEEE transactions on visualization and computer graphics* 26.1 (2019), pp. 461–471.

[121] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. "Vega-lite: A grammar of interactive graphics". In: *IEEE transactions on visualization and computer graphics* 23.1 (2016), pp. 341–350.

[122] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. "Vega-lite: A grammar of interactive graphics". In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), pp. 341–350.

[123] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. "Reactive vega: A streaming dataflow architecture for declarative interactive visualization". In: *IEEE transactions on visualization and computer graphics* 22.1 (2015), pp. 659–668.

[124] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. "Reactive vega: A streaming dataflow architecture for declarative interactive visualization". In: *IEEE transactions on visualization and computer graphics* 22.1 (2016), pp. 659–668.

[125] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. "Declarative interaction design for data visualization". In: *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM. 2014, pp. 669–678.

[126] Cheryl Savery and TC Graham. "It's about time: confronting latency in the development of groupware systems". In: *Proceedings of the ACM 2011 conference on Computer supported cooperative work*. ACM. 2011, pp. 177–186.

[127] Steven C Seow. *Designing and engineering time: The psychology of time perception in software*. Addison-Wesley Professional, 2008.

[128] Doris Seyser and Michael Zeiller. "Scrollytelling–an analysis of visual storytelling in online journalism". In: *2018 22nd International Conference Information Visualisation (IV)*. IEEE. 2018, pp. 401–406.

[129] Ben Shneiderman. "The eyes have it: A task by data type taxonomy for information visualizations". In: *Proceedings 1996 IEEE symposium on visual languages*. IEEE. 1996, pp. 336–343.

[130] Ben Shneiderman. "The eyes have it: A task by data type taxonomy for information visualizations". In: *The craft of information visualization*. Elsevier, 2003, pp. 364–371.

[131] Ben Shneiderman. "The future of interactive systems and the emergence of direct manipulation". In: *Behaviour & Information Technology* 1.3 (1982), pp. 237–256.

[132] *Simple Widget Introduction*. 2019. URL: https://jupyter-notebook.readthedocs.io/en/stable/comms.html (visited on 01/03/2020).

[133] SQLite. *Small, fast, self-contained, high-reliability, full-featured, SQL database engine*. https://www.sqlite.org. 2017. (Visited on 03/17/2018).

[134] Chris Stolte, Diane Tang, and Pat Hanrahan. "Polaris: A system for query, analysis, and visualization of multidimensional relational databases". In: *IEEE Transactions on Visualization and Computer Graphics* 8.1 (2002), pp. 52–65.

[135] Matúš Sulír, Michaela Bačíková, Sergej Chodarev, and Jaroslav Porubän. "Visual augmentation of source code editors: A systematic mapping study". In: *Journal of Visual Languages & Computing* 49 (2018), pp. 46–59.

[136] Chengzheng Sun and David Chen. "A multi-version approach to conflict resolution in distributed groupware systems". In: *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*. IEEE. 2000, pp. 316–325.

[137] Chengzheng Sun and Clarence Ellis. "Operational transformation in real-time group editors: issues, algorithms, and achievements". In: *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. 1998, pp. 59–68.

[138] Tableau. *Tableau*. 2018. URL: https://www.tableau.com/ (visited on 09/03/2018).

[139] Wenbo Tao, Xiaoyu Liu, Çagatay Demiralp, Remco Chang, and Michael Stonebraker. "Kyrix: Interactive visual data exploration at scale". In: CIDR. 2019.

[140] *TensorBoard: TensorFlow's visualization toolkit*. 2020. URL: https://www.tensorflow.org/tensorboard.

[141] *The fastest way to build custom ML tools*. 2019. URL: https://streamlit.io/.

[142] John W Tukey. *Exploratory data analysis*. Vol. 2. Reading, Mass., 1977.

[143] *Use Show Me to Start a View.* 2020. URL: https://help.tableau.com/current/pro/desktop/en-us/buildauto_showme.htm.

[144] *Vega Transforms.* URL: https://vega.github.io/vega/docs/transforms/.

[145] *Vega-Lite Axis.* 2019. URL: https://vega.github.io/vega-lite/docs/axis.html.

[146] *Vega-Lite Mark.* 2019. URL: https://vega.github.io/vega-lite/docs/mark.html.

[147] *Vega-Lite Selection.* 2019. URL: https://vega.github.io/vega-lite/docs/selection.html.

[148] Bret Victor. *Explorable Explanations.* Mar. 2011. URL: http://worrydream.com/ExplorableExplanations/.

[149] Bret Victor. *Inventing on Principle.* Jan. 2012. URL: http://worrydream.com/%5C#!/InventingOnPrinciple.

[150] Bret Victor. *Learnable Programming.* Sept. 2012. URL: http://worrydream.com/LearnableProgramming/.

[151] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. "How data scientists use computational notebooks for real-time collaboration". In: *Proceedings of the ACM on Human-Computer Interaction* 3.CSCW (2019), pp. 1–30.

[152] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. "Falx: Synthesis-Powered Visualization Authoring". In: *arXiv preprint arXiv:2102.01024* (2021).

[153] Michelle Q Wang Baldonado, Allison Woodruff, and Allan Kuchinsky. "Guidelines for using multiple views in information visualization". In: *Proceedings of the working conference on Advanced visual interfaces.* 2000, pp. 110–119.

[154] Franz Wanner, Andreas Stoffel, Dominik Jäckle, Bum Chul Kwon, Andreas Weiler, Daniel A Keim, Katherine E Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, et al. "State-of-the-art report of visual analysis for event detection in text data streams". In: *Computer graphics forum.* Vol. 33. Citeseer. 2014, pp. 1–15.

[155] *WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine.* URL: https://webassembly.org/.

[156] Stéphane Weiss, Pascal Urso, and Pascal Molli. "Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks". In: *2009 29th IEEE*

International Conference on Distributed Computing Systems. IEEE. 2009, pp. 404–
412.

[157]   Leland Wilkinson. *The grammar of graphics*. Springer Science & Business Media,
2006.

[158]   Wesley Willett, Jeffrey Heer, and Maneesh Agrawala. "Scented widgets: Improving
navigation cues with embedded visualizations". In: *IEEE Transactions on Visualiza-
tion and Computer Graphics* 13.6 (2007), pp. 1129–1136.

[159]   Kanit Wongsuphasawat, Yang Liu, and Jeffrey Heer. "Goals, Process, and Challenges
of Exploratory Data Analysis: An Interview Study". In: *arXiv preprint arXiv:1911.00568*
(2019).

[160]   Jo Wood, Alexander Kachkaev, and Jason Dykes. "Design exposition with literate
visualization". In: *IEEE transactions on visualization and computer graphics* 25.1
(2018), pp. 759–768.

[161]   Eugene Wu, Fotis Psallidas, Zhengjie Miao, Haoci Zhang, Laura Rettig, Yifan Wu,
and Thibault Sellam. "Combining Design and Performance in a Data Visualization
Management System." In: *CIDR*. 2017.

[162]   Yifan Wu. "Is a Dataframe Just a Table?" In: *10th Workshop on Evaluation and
Usability of Programming Languages and Tools (PLATEAU 2019)*. Schloss Dagstuhl-
Leibniz-Zentrum für Informatik. 2020.

[163]   Yifan Wu, Remco Chang, Joseph Hellerstein, and Eugene Wu. "Facilitating Explo-
ration with Interaction Snapshots under High Latency". In: *2020 IEEE Visualization
Conference (VIS)*. IEEE. 2020.

[164]   Yifan Wu, Remco Chang, Eugene Wu, and Joseph M Hellerstein. "DIEL: Transparent
Scaling for Interactive Visualization". In: *arXiv preprint arXiv:1907.00062* (2019).

[165]   Yifan Wu, Joseph M Hellerstein, and Arvind Satyanarayan. "B2: Bridging Code and
Interactive Visualization in Computational Notebooks". In: *Proceedings of the 33rd
Annual ACM Symposium on User Interface Software and Technology*. 2020, pp. 152–
165.

[166]   Yifan Wu, Joseph M Hellerstein, and Eugene Wu. "A DeVIL-ish approach to incon-
sistency in interactive visualizations." In: *HILDA@ SIGMOD*. 2016, p. 15.

[167]   Yifan Wu, Larry Xu, Remco Chang, Joseph M Hellerstein, and Eugene Wu. "Making
Sense of Asynchrony in Interactive Data Visualizations". In: *arXiv preprint arXiv:1806.01499*
(2018).

[168] Zaixian Xie. "Towards exploratory visualization of multivariate streaming data". In: *IEEE Vis/InfoVis/VAST Doctoral Colloquium.* 2007.

[169] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. "A colorful approach to text processing by example". In: *Proceedings of the 26th annual ACM symposium on User interface software and technology.* 2013, pp. 495–504.

[170] Ji Soo Yi, Youn ah Kang, and John Stasko. "Toward a deeper understanding of the role of interaction in information visualization". In: *IEEE transactions on visualization and computer graphics* 13.6 (2007), pp. 1224–1231.

[171] Emanuel Zgraggen, Alex Galakatos, Andrew Crotty, Jean-Daniel Fekete, and Tim Kraska. "How Progressive Visualizations Affect Exploratory Analysis". In: *IEEE Transactions on Visualization and Computer Graphics* (2016).

[172] Jonathan Zong, Dhiraj Barnwal, Rupayan Neogy, and Arvind Satyanarayan. "Lyra 2: Designing Interactive Visualizations by Demonstration". In: *IEEE Transactions on Visualization and Computer Graphics* (2020).

[173] Zoomdata. *Analyze the Freshest Data.* https://www.zoomdata.com/resource/streaming-data-why-it-makes-sense-and-how-work-it. 2018. (Visited on 03/31/2018).