B2: Bridging Code and Interactive Visualization in Computational Notebooks

Yifan Wu
UC Berkeley
yifanwu@berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

Arvind Satyanarayan
MIT CSAIL
arvindsatya@mit.edu

ABSTRACT

Data scientists have embraced computational notebooks to author analysis code and accompanying visualizations within a single document. Currently, although these media may be interleaved, they remain siloed: 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 bridge this divide, we present *B2*, a set of techniques grounded in treating data queries as a shared representation between the code and interactive visualizations. B2 instruments data frames to track the queries expressed in code and synthesize corresponding visualizations. These visualizations are displayed in a dashboard to facilitate interactive analysis. When an interaction occurs, B2 reifies it as a data query 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 an evaluative 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.

Author Keywords

Data science, computational notebooks, exploratory programming, interactive visualizations

CCS Concepts

•**Human-centered computing** → **Visualization systems and tools; Interactive systems and tools;**

INTRODUCTION

Computational notebooks (e.g., Jupyter and Observable) have become increasingly popular in data science because they enable *literate computing* [45]: 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 [12, 57]. Surveys and interviews with data scientists, however, highlight the impoverished use of visualization within computational notebooks. In contrast to visual analysis tools such as Tableau (née Polaris [49]) 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 [14, 57].

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 interleaved, these two types of artifacts remain siloed [14]. 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 [47] — 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 [28, 48, 60].

UIST '20, October 20–23, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7514-6/20/10 ...$15.00.
http://dx.doi.org/10.1145/3379337.3415851

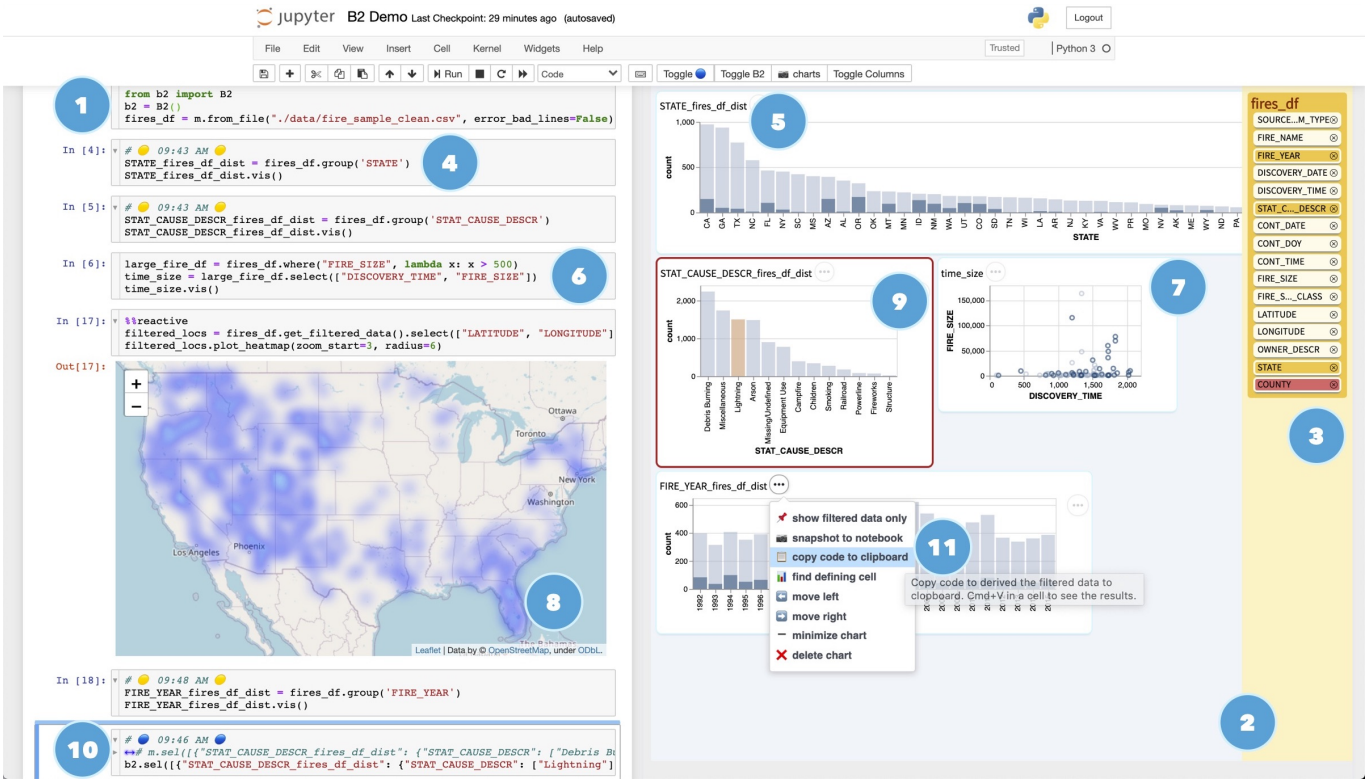

Figure 1. An analyst's workflow with B2. They start by (1) importing the library which creates (2) a resizable dashboard pane to the right of the traditional notebook. The analyst can (3) click on the columns, which creates (4) code that computes and (5) visualizes corresponding distributions. The analyst can also write (6) a custom data frame query to create (7) the scatter plot. (8) B2's *reactive cells* automatically recompute when new interactions 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.

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 Figure 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 [56] 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 document 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/ucbrise/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.

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. 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.

[1] `https://www.kaggle.com/rtatman/188-million-us-wildfires`

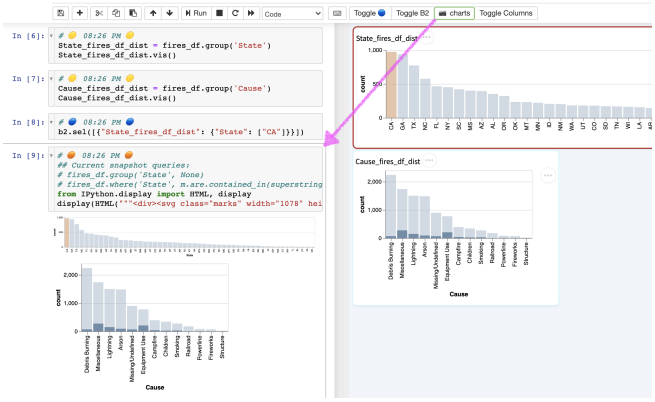

Figure 2. Snapshotting creates a cell in the notebook with an SVG of the visualization, persisting the transient interactive state.

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 dataframe 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. 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. 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

Figure 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.

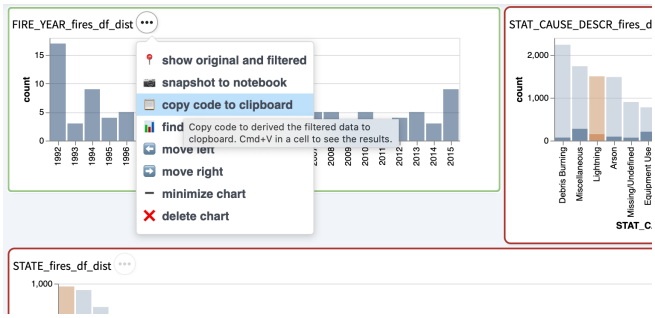

Figure 4. The full state of the chart, including interactive selections, is converted into code and made available through the `Copy Code to Clipboard` button.

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.

To model the rate of increase, Sam clicks on the menu item for the histogram, `Copy Code to Clipboard` (Fig. 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 dataframe that materializes the interactive state via the `get_filtered_data` API, which returns the rows of `df` filtered by the current selection (Fig. 5). Sam then **switches** back to the dashboard and clicks on the `Lightning`

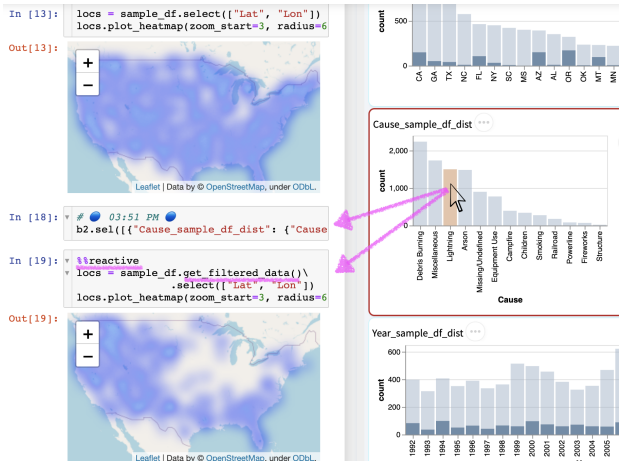

Figure 5. Contrast the top cell, which is static, to the reactive cell below. The reactive cell can be iterated on using interactions.

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.

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 [26] and work by Bret Victor [52, 53, 54] helped popularize instrumenting code editors with interactive controls [50]. 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* [33], R Markdown notebooks can be made interactive with *shiny* [21], and Streamlit [2] and Observable [4] 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 [3].

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 [1, 40, 41], widgets do not provide behaviors as rich as those found in interactive visualizations [62]. Second, widgets violate the literate computing goal of reproducibility [45] 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 [48]. 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 [23, 24], indicate hierarchical structure using colored blocks [61], or record and replay interactions with lists on web pages [13, 16]. 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 [18, 31, 32], 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 [25, 34], 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 [12, 17, 57]. 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 [11] and DEVise [42] 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 [20], 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.

Finally, B2 is contemporaneous with work by Kery et al. developing mechanisms to move "fluidly" between code and graphical interfaces within computational notebooks [38]. 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 integrating 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 [17, 56].

The Needs of Data Scientists

B2's goal of bridging code and interactive visualization is motivated by recent surveys and interviews of data scientists [12, 14, 17, 57]. 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 [57]. 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 [17]. For many data scientists, this overhead is sufficiently prohibitive that they eschew visual analysis tools altogether and restrict themselves to working only in code [57]. 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"* [14] 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 [12].

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 [17]. And, a naive integration of the two risks exacerbating existing concerns of notebooks being a "mess" [37], full of "ugly code" and "dirty hacks" [47]. 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 [36], on a per-cell basis [46], or for manually-defined snippets [35], or tools for gathering, cleaning, and comparing messy code [27]. 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 [27], 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* [58] while Mathisen et al. propose *literate analytics* [43]. 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.

THE GAPS BETWEEN CODE AND INTERACTIONS

Computational notebooks meld ideas from traditional scientific notebooks and literate programming as envisioned by Knuth [39]. Interactive visualization environments are inspired by vehicle dashboard design and the ideas of Exploratory Data Analysis as envisioned by Tukey [51]. 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 [12, 14, 17, 57], 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 [29], we consider those techniques that can be modeled as data queries. Here, we turn to Vega-Lite [48], 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 [62]. 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 [56], 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 anal-

ysis, 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 specification: 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.

SYSTEM DESIGN AND IMPLEMENTATION

B2 is implemented as a Jupyter notebook extension 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. 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 [48] 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 specification for a data frame visualization using established heuristics like column data types [9]. 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` [6] parameter chooses among bar charts, scatter plots, and line charts; `encoding` [5] configures which column is the x-axis, y-axis, how they should be interpreted (ordinal, quantitative, temporal), and sorted; `selection_type` and `selection_dimensions` [7] 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 [9] that only apply to static charts. Consider the scatter plot of the fire `Size` and `Time` (Listing 1, Line 2) 7, and the bar chart of the distribution of fires by their `Cause` (Line 3) 9. Because both dataframes 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 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 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

[2] `datascience` was designed for instructional purposes in large data science courses [19]. 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, 59]. To inter-operate, B2 data frames can be easily mapped to/from pandas via `df.to_pd` and `b.from_pd`.

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

visualization in another library/tool. The data in the *current* selection is accessed through the `get_filtered_data` API, which returns a standard dataframe 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.

```
# predicates
b2.current_selection
b2.all_selections
# data from the current selection
df.get_filtered_data()
# code from the current selection
df.get_code()
```

Listing 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.

Figure 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.

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 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. 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.

EVALUATION: FIRST-USE STUDY

To evaluate the usability of B2's bridges, we conducted a first-use study with 7 representative users[4], 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 dataframes, 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 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 [44], 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.

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

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 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 ($\mu = 3.7, \sigma = 0.9$), 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

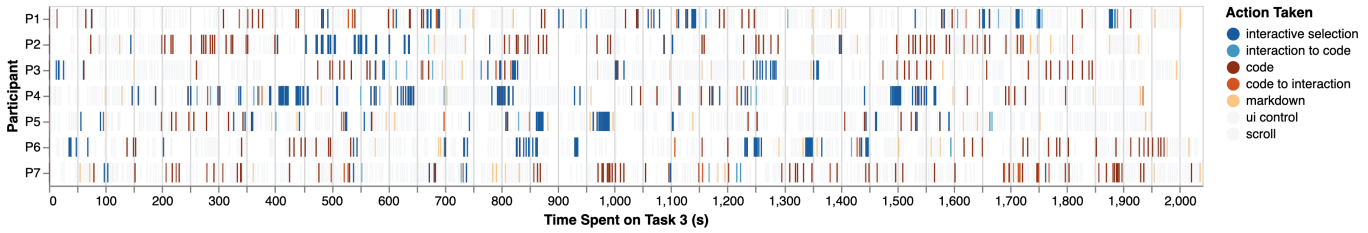

Figure 7. Participants' interaction traces while working on task 3, an open-ended task to model the dataset. Each participant's activity is represented by a horizontal strip plot, with participants arrayed down the visualization. Orange colors represents the work done in the code domain, and the blue colors represent the work done in the interactive visualization domain.

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. 8). 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. 9).

These types of patterns switching between code and interactive visualization, and vice-versa, were common across all participants. Fig. 7 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. 10). 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"*.

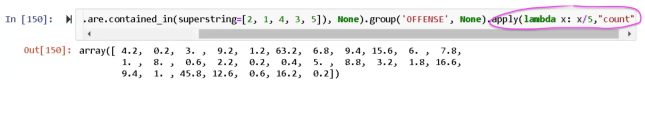

Figure 8. Modifying a generated query to further derive the average count, which is not possible through interactions.

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 participants 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 dataframe 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.

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.

Figure 9. Using code to create a custom plot with the average line in the chart, and computing percentages.

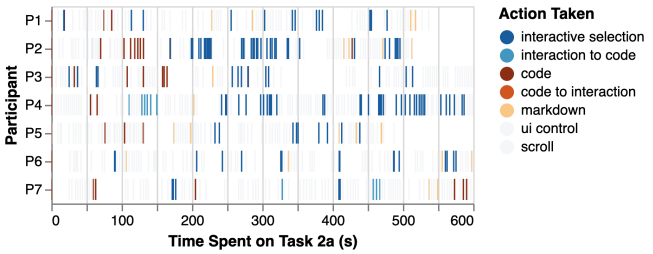

Figure 10. Participants' interaction traces while working on task 2. To plot the heatmap, all analysts initially interleaved between code and interactions. P3's gap between 300–400s is when they searched the web for information about the city. P7 used code, interaction, and the `get_filtered_data` API to inspect whether null values were present.

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 exploration [47] and we need to study further to understand the design and cognitive issues present.

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 [55]. 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 [30]. 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 [22] or Tensorflow [10] are often coupled with domain-specific charting packages like Tensorboard [8]. 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 [62] 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/ucbrise/b2`.

ACKNOWLEDGEMENTS

Thanks to Eugene Wu and Remco Chang for early discussions, Ryan Purpura for programming contributions, Brian Hempel and Andrew Head for helpful algorithmic and design advice, Anna Papitto for editing help, and the anonymous reviewers for their constructive comments. This work was supported by NSF No. 1564351, 1900991 and 1942659, and DOE No. DE-SC0016934.

REFERENCES

[1] 2019. bloomberg. (2019). `https://github.com/bloomberg/bqplot`

[2] 2019. The fastest way to build custom ML tools. (2019). `https://streamlit.io/`

[3] 2019. How to use interactive IPython widgets to enhance data exploration and analysis. (2019). https://towardsdatascience.com/interactive-controls-for-jupyter-notebooks-f5c94829aee6

[4] 2019. Observable. (2019). https://observablehq.com/

[5] 2019a. Vega-Lite Axis. (2019). https://vega.github.io/vega-lite/docs/axis.html

[6] 2019b. Vega-Lite Mark. (2019). https://vega.github.io/vega-lite/docs/mark.html

[7] 2019c. Vega-Lite Selection. (2019). https://vega.github.io/vega-lite/docs/selection.html

[8] 2020. TensorBoard: TensorFlow's visualization toolkit. (2020). https://www.tensorflow.org/tensorboard

[9] 2020. Use Show Me to Start a View. (2020). https://help.tableau.com/current/pro/desktop/en-us/buildauto_showme.htm

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

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

[12] Sara Alspaugh, Nava Zokaei, Andrea Liu, Cindy Jin, and Marti A Hearst. 2018. Futzing and moseying: Interviews with professional data analysts on exploration practices. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 22–31.

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

[14] Andrea Batch and Niklas Elmqvist. 2017. The interactive visualization gap in initial exploratory data analysis. *IEEE transactions on visualization and computer graphics* 24, 1 (2017), 278–287.

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

[16] Sarah Chasins, Shaon Barman, Rastislav Bodik, and Sumit Gulwani. 2015. 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*. 179–182.

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

[18] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and direct manipulation, together at last. *ACM SIGPLAN Notices* 51, 6 (2016), 341–354.

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

[20] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. 2020. 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.

[21] Garrett Grolemund. 2014. Introduction to interactive documents. (July 2014). https://shiny.rstudio.com/articles/interactive-docs.html

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

[23] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[24] Sumit Gulwani, William R Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105.

[25] Philip J Guo, Sean Kandel, Joseph M Hellerstein, and Jeffrey Heer. 2011. 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*. 65–74.

[26] Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R Klemmer. 2008. 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*. 91–100.

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

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

[29] Jeffrey Heer and Ben Shneiderman. 2012. Interactive dynamics for visual analysis. *Queue* 10, 2 (2012), 30–55.

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

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

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

[33] Jupyter. 2019. Using Interact. (2019). https://ipywidgets.readthedocs.io/en/stable/examples/Using%20Interact.html

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

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

[36] Mary Beth Kery, Bonnie E John, Patrick O'Flaherty, Amber Horvath, and Brad A Myers. 2019. 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, 92.

[37] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. 2018. 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, 174.

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

[39] Donald Ervin Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.

[40] Semi Koen. 2019. Bring your Jupyter Notebook to life with interactive widgets. (2019). https://towardsdatascience.com/bring-your-jupyter-notebook-to-life-with-interactive-widgets-bc12e03f0916

[41] Samuel Lau and Joshua Hug. 2018. *nbinteract: generate interactive web pages from Jupyter notebooks*. Ph.D. Dissertation. Master's thesis, EECS Department, University of California, Berkeley.

[42] Miron Livny, Raghu Ramakrishnan, Kevin Beyer, Guangshun Chen, Donko Donjerkovic, Shilpa Lawande, Jussi Myllymaki, and Kent Wenger. 1997. DEVise: integrated querying and visual exploration of large datasets. *ACM SIGMOD Record* 26, 2 (1997), 301–312.

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

[44] Berkeley PD. 2020. Calls for Service. (2020). https://data.cityofberkeley.info/Public-Safety/Berkeley-PD-Calls-for-Service/k2nh-s5h5

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

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

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

[48] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2016. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics* 23, 1 (2016), 341–350.

[49] Chris Stolte, Diane Tang, and Pat Hanrahan. 2002. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics* 8, 1 (2002), 52–65.

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

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

[52] Bret Victor. 2011. Explorable Explanations. (March 2011). http://worrydream.com/ExplorableExplanations/

[53] Bret Victor. 2012a. Inventing on Principle. (January 2012). http://worrydream.com/#!/InventingOnPrinciple

[54] Bret Victor. 2012b. Learnable Programming. (September 2012). http://worrydream.com/LearnableProgramming/

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

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

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

[58] Jo Wood, Alexander Kachkaev, and Jason Dykes. 2018. Design exposition with literate visualization. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 759–768.

[59] Yifan Wu. 2020. 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.

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

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

[62] Ji Soo Yi, Youn ah Kang, and John Stasko. 2007. Toward a deeper understanding of the role of interaction in information visualization. *IEEE transactions on visualization and computer graphics* 13, 6 (2007), 1224–1231.