


# Towards Reusable and Reactive Widgets for Information Visualization Research and Dissemination

John A. Guerra-Gomez   
 Northeastern University, Oakland Campus

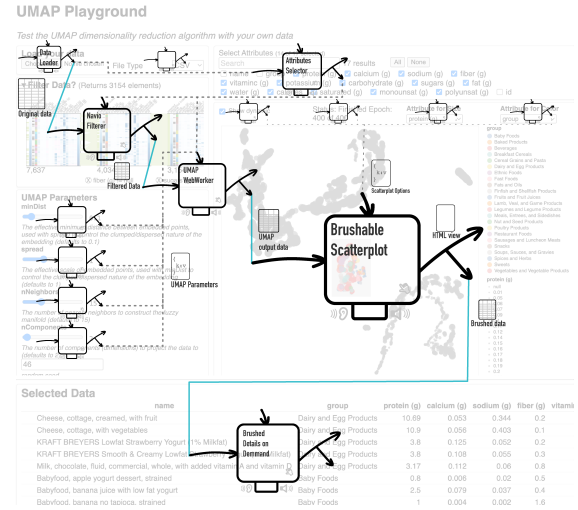
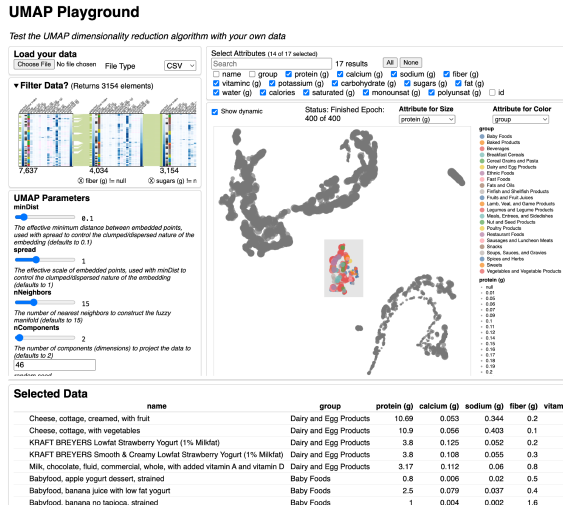


Figure 1: Example Information visualization application built using reusable and reactive widgets. It allows users to load their data, zoom, and filter using navio [8], process it with the UMAP algorithm [15, 7], and then brush on the generated Scatterplot to obtain details on demand, all reactively on user interactions. The left side of the figure shows the application interface, while the right side shows an overlay of some of the reactive widgets used to build it. Demo <https://johnguerra.co/viz/umapPlayground/>.

## ABSTRACT

The information visualization research community commonly produces supporting software to demonstrate technical contributions to the field. However, developing this software tends to be an overwhelming task. The final product tends to be a research prototype without much thought for modularization and re-usability, which makes it harder to replicate and adopt. This paper presents a design pattern for facilitating the creation, dissemination, and re-utilization of visualization techniques using reactive widgets. The design pattern features basic concepts that leverage modern front-end development best practices and standards, which facilitate development and replication. The paper presents several usage examples of the pattern, templates for implementation, and even a wrapper for facilitating the conversion of any Vega [27, 28] specification into a reactive widget.

**Index Terms:** Information Visualization, Software Components, Reactive Components, Notebook Programming, Direct Manipulation, Brush and Linking

## 1 INTRODUCTION

The information visualization community produces outstanding research contributions that could greatly benefit society as a whole. However, despite how effort-intensive it is to produce research software prototypes as supporting materials for papers, these tend not to get widely disseminated because they tend to be hard to reuse, replicate, and integrate with other systems.

\* e-mail: [jguerra@northeastern.edu](mailto:jguerra@northeastern.edu)

This paper presents a software design pattern that the research community could use to facilitate the dissemination of their research contributions. The pattern leverages the concept of reactive and modular widgets that follow simple specifications and can be easily put together to build larger applications such as the ones created in typical IEEE VIS systems or application papers. Figure 1 presents an example of such an application and an overlay of the modular reactive widgets that were used to build it. As can be seen in the source notebook <https://observablehq.com/@john-guerra/umap-playground>, the UMAP playground is a basic application that lets users process their own data using the UMAP algorithm [15, 7]. Moreover, the application is completely reactive, allowing users to change the hyperparameters of the algorithm, as well as to change the attributes to be processed, filter the data, and even brush in the scatterplot representation of the UMAP result to obtain details on demand in a table [31]. All of these functionalities are implemented by reusing reactive widgets highlighted in the figure, that are interconnected to produce the final result. Using this architecture was not only simpler to build this interface, but also allows the developer to exchange one of the widgets for another one that supports the same functionality.

## 2 THE NEED FOR A NEW VISUALIZATION COMPONENT PATTERN

Creating research prototypes is a fundamental part of conducting research in information visualization. These prototypes are fundamental for conducting usability studies or controlled experiments, or for releasing the prototype as supporting evidence of the contributions two mention two examples. Upon publishing these prototypes, researchers usually want to encourage the reusability of their contributions and facilitate the dissemination and reproducibility of their results. However, building such prototypes takes signif-

ificant effort and time, and depending on how the prototype is released, these benefits are commonly not obtained. Take, for example, SpaceTree [24] a renowned visualization technique released in 2002 that allows users to explore large trees through rich interactions. Even though the original project was released as a desktop Java application, using the technique as part of another software would require a skilled software developer to reimplement the core concepts of the technique, and therefore not many SpaceTree implementations are used out there. Moreover, even researchers who want to evaluate how their proposed technique compares to SpaceTree would not be able to test them, as there is no simple way of integrating it into a web application used for testing.

On the other hand, D3 [4], Vega and Vega-lite [28, 27, 10], are some of the most widely used tools for information visualization development, in part because of its usage of web standards, modularity and easy adaptation to other software. A great example of this is the success of the Altair project [34], an adaptation of the Vega-lite grammar for the Python programming language that is widely used. As researchers, our goal should be to release software prototypes that are modular, reusable, and can be easily embedded into other software. For this, multiple researchers [11, 36, 32] have attempted to release visualization plugins and libraries in hopes that they can be reused. However, these have experienced limited adoption, probably because they don't leverage modern standards and some of them rely on outdated programming technologies.

This paper proposes a model for releasing information visualization software using reusable and reactive widgets, adhering to modern web development standards and best practices. Instead of creating large monolithic software applications for testing our contributions, this paper proposes to build and release modular widgets that could be easily integrated into new applications. These could even be easily included in the testing framework of the next researcher who wants to test and compare new ideas with the state of the art. If designed correctly, these widgets can be put together to create the larger visualization solutions such as the ones that are commonly published in our field, while facilitating distribution and reproducibility. Moreover, following these reactive widget models could simplify prototype development by facilitating reusability even inside our research teams. If, on top of that, we add developing these applications in Notebook-based platforms, the potential for speeding up development can be increased even more. As an example, see this video that demonstrates how to go from nothing to an interactive and configurable visualization application in less than 20 minutes using Observable Notebooks <https://www.youtube.com/watch?v=mxEEktWy15o&t=26s>.

One of the most interesting features of information visualization is direct manipulation and the ability to coordinate multiple views via brush and linking. This proposed widget design facilitates this type of development by implementing coordinated views through interconnected reactive widgets. Each widget will have a clear function in the larger application, input and output data, and will emit and listen for standard input events to coordinate the rest of the application. Each widget will output an HTML element and some selected data, which would facilitate interconnection even with applications written in other programming languages such as Python via libraries such as Anywidget [14].

### 3 RELATED WORK

The visualization community is well acquainted with the concepts of reusability and modularization. D3 [4] and the Vega grammar family [28, 27] are great examples of successful research contributions that have been widely disseminated and follow a very modular approach. The widget pattern presented here follows a similar approach, advocating the importance of building research contributions using reusable widgets. Moreover, this work contributes a technique to facilitate the interconnection of these widgets by

means of reactivity based on Web standard events. The design presented by this work also takes great inspiration from the concept of collaborative notebook-based programming introduced by Mike Bostock et al. with Observable [2, 23], particularly with the idea of Custom Inputs [22] and Synchronized inputs [21], and previous ideas towards reusable charts [17]. This paper takes this idea even further, prioritizing core interactive visualization concepts such as direct manipulation [29, 30] and brush and linking [32, 5, 1, 26, 37], to facilitate interconnection of visualization components and the construction of larger applications. The reactive visualization widgets design proposed here is designed for JavaScript, following the de facto standard for front-end development, but it can also be easily expanded to other programming languages such as Python by using libraries such as Anywidget [14]. This could facilitate the dissemination of other proposed ideas such as [36, 13, 11].

The approach of this work is different from others as Wongsuphasawat's grammar for components [36] in the fact that we focus on JavaScript components using Web Standards rather than creating a new grammar, which could facilitate the adoption of the pattern.

### 4 DEFINING THE REACTIVE VISUALIZATION COMPONENT DESIGN

As shown in Figure 2, a reactive and reusable widget should include the following characteristics:

- **Input data** It could receive some input data (usually an array of objects in tidy format [35], but of course it could be in any other format) and configuration options, which are commonly passed as an object. Listing 4 lines 2-6. It is common to include an initial value as part of the options object.

- **HTML Element** It must return an *HTML Element* [19] that represents the view of the widget in the application. Listing 4 lines 12 and 41, variable *target*.

- **Reactive Value** It could hold some internal value that will convey the user's interactions with the widget. e.g. selecting a subset of the data by selecting a mark in the visualization. This value should be returned as the value attribute of the returned object. Listing 4 lines 8, 29 to 35, variable *internalValue*. By following this rule the widget would work nicely with Observable's *viewof* operator [22, 3], while still can be easily used in vanilla JavaScript, e.g. <https://observablehq.com/@john-guerra/search-checkbox#useVanillaJS>.

- **input events** Emit standard input events [18] when the reactive data changes, e.g. the user selected different elements. It also should listen to *input* events which should trigger a redrawing of the reactive value, e.g. move the brush position.

- **Exposed extras**, e.g. a color palette created for the visualization widget, or the different scales.

This widget definition leverages many of the best practices of Observable's Inputs, but it isn't limited to the platform. Since all of it is based on open modern Web standards, it can work seamlessly with vanilla JavaScript applications, as well as with frameworks such as React [16], or Svelte [25].

#### 4.1 Patterns for connecting widgets

The key advantage of using reactive widgets is to be able to combine them to build larger applications. One could use a reactive widget that features direct manipulation, and then easily connect it with other coordinated reactive widget to show selected items using brush and linking. Connecting reactive widgets is straightforward in Observable as described in the following examples, but might as well be applied in other modern Web environments by listening and reacting to standard *input* events.

- **Parent-children.** You have one widget whose reactive value output is the input of another one. The children's widget should be immutable and will be recreated every time the parent changes. A common example of this is a filtering widget <https://>

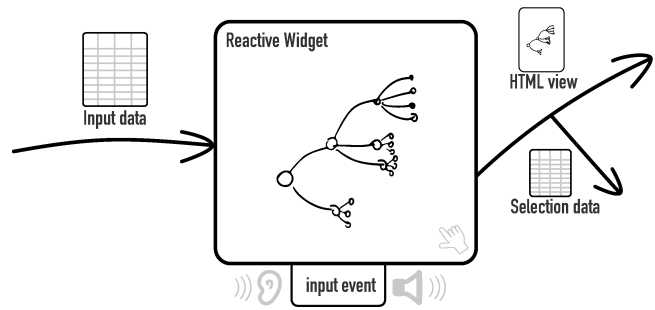


Figure 2: Reusable reactive widget should receive some input data, and return an HTML element (view) and an output value (selected data). Reactive widgets will listen and re-render when receiving standard HTML *input* events (ear) and will emit an *input* event when the user selection changes.

[observablehq.com/@john-guerra/faceted-search](https://observablehq.com/@john-guerra/faceted-search), which filters data before passing them on to another widget to show details on demand. In this case, the children will not be sending changes to the parent.

```
viewof selected = FacetedSearch(data);
Histogram(selected);
```

- **Synchronized inputs via `Inputs.bind`.** When having two widgets that represent the same reactive data in different ways, they can be connected using a convenience function `Inputs.bind` [20] as demonstrated in <https://johnguerra.co/reactiveWidgets/#count>. `Inputs.bind` listens for input events from one of the inputs and updates the reactive values of the other widget. However, it selects one widget as primary and dispatches only input events on that one to avoid infinite loops.

```
viewof selection = BrushableHistogram(cars, {
  x: (d) => d[xAttr],
  value: [12, 32] // initial position
})
Inputs.bind(
  BrushableHistogram(cars, { x: (d) => d[xAttr]
    => }),
  viewof selection
)
```

- **Shared mutable value**  
One final alternative to synchronizing two widgets when they are equally important is to create a shared mutable store and synchronize both widgets to it.

```
viewof shared = Inputs.input(initialValue);
Inputs.bind(WidgetOne(data), viewof shared);
Inputs.bind(WidgetTwo(data), viewof shared);
```

## 4.2 Types of components

The main type of widget proposed in this paper is a *direct manipulation*, reactive and reusable visualization widget that lets users use the visualization as an input or selector. However, as in any common component design, widgets can specialize for common tasks needed for building visual analytics applications. Other common widgets and components that would fit well would be:

*Input Wrapper.* A module that can wrap a widget to make it, for example, persistent on local storage on the browser as demonstrated on <https://observablehq.com/@john-guerra/persist-input>. This module would accept a

```
1 function ReactiveWidgetTemplate(
2   data,
3   {
4     value = 0 // following observable inputs, the
      ↳ value option contains the initial value
5   } = {}
6 ) {
7   // • The interval value selected by the user
      ↳ interaction
8   let intervalValue = value;
9
10  // • The HTML element that we will return
11  // ♣ Add here the visual representation of your
      ↳ widget
12  let target =
13    ↳ htl.html`<output>${intervalValue}</output>
      ↳ <button onClick=${() =>
      ↳   setValue((intervalValue +=
      ↳   1)})>></button>`;
14
15  // • Usually you have a function to reflect in
      ↳ the UI the current value
16  function showValue() {
17    // ♣ Add here the code that updates the
      ↳ current interaction
18    target.querySelector("output").value =
      ↳ intervalValue;
19  }
20
21  // • And a function to update the current
      ↳ internal value. This triggers the input
      ↳ event
22  function setValue(newValue) {
23    intervalValue = newValue;
24    showValue();
25    target.dispatchEvent(new CustomEvent("input",
      ↳ { bubbles: true }));
26  }
27
28  // • The value attribute represents the current
      ↳ interaction
29  Object.defineProperty(target, "value", {
30    get() { return intervalValue; },
31    set(newValue) {
32      intervalValue = newValue;
33      showValue();
34    }
35  });
36
37  // • Listen to the input event to show the
      ↳ current interaction
38  target.addEventListener("input", showValue);
39
40  // • Finally return the html element
41  return target;
42 }
```

Listing 1: Reactive Widget Template. Please, see <https://johnguerra.co/reactiveWidgets/> for an interactive code example, with helper functions, and D3 and Vega Lite examples.

```

// ----- With Observable -----
viewof selection = MyWidget(data, {opt1: val1, ...});
// viewof selection returns the html element
Inputs.table(selection);
// selection contains the widget's user selection

// ----- Vanilla Javascript -----
const target = MyWidget(data, { opt1: val1, ...});
document.querySelector("#target")
  .appendChild(target);
target.addEventListener("input",
  () => showOutput(target.value));
// target.value contains the user's selection

```

Listing 2: Example code for using reactive widgets.

widget as an input, and output another one augmented, which facilitates the development of more complex behaviors.

*Data Wrangling widgets.* This is a specialization of widgets whose main focus is to take some data and wrangle it into different formats; e.g. group tabular data by attributes to create hierarchies <https://observablehq.com/@john-guerra/multi-auto-select> or a widget that filters data <https://observablehq.com/@john-guerra/faceted-search>

*Custom Input widgets.* Widgets that don't rely on visualizations but rather on augmented input controls collect users' input; e.g. Selecting many attributes at once with search capabilities <https://observablehq.com/@john-guerra/search-checkbox>.

## 5 APPLICATION EXAMPLES

This section presents a series of examples that serve as case studies to validate and demonstrate the flexibility and generalizability of the proposed reactive widget design for information visualization. The examples range from more examples of widgets that followed the pattern and are ready to use by the community, to larger applications that implement them. Moreover, we present a wrapper widget that can be used to transform any Vega [28] or Vega-lite [27] spec into a reactive widget.

*SpaceTree* <https://observablehq.com/@john-guerra/spacetree> is a re-implementation of many of the original paper's features [24] while exposing a reusable and reactive widget that can be easily used to allow users to explore and select elements in a large hierarchy. The reactive value of this widget is the currently selected path.

*UMAP Playground* <https://observablehq.com/@john-guerra/umap-playground> shown on Figure 1 an application that demonstrates the UMAP [15] implementation UMAP-js [7] library using the USDA's nutrients dataset [33]. It features several reusable components, including *data-input* <https://observablehq.com/@john-guerra/data-input> which allows users to load data files from different formats and parse them; *multi-auto-select* <https://observablehq.com/@john-guerra/multi-auto-select> for selecting the attributes to be considered in the UMAP; *navio* [8] for visualizing and filtering the tabular data; and many other inputs from Observable's standard Input library [22].

### 5.1 IEEEVIS 2023 Papers Explorer

This example <https://observablehq.com/@john-guerra/ieeevis-2023-papers> features the IEEEVIS 2023 Conference papers organized by their text (title, type, and abstract) similarity using sentence embeddings computed with HuggingFace's transformers.js library [12]. It applies dimensionality reduction algorithms using Cytura's Druid.js [6]. All of this is completely com-

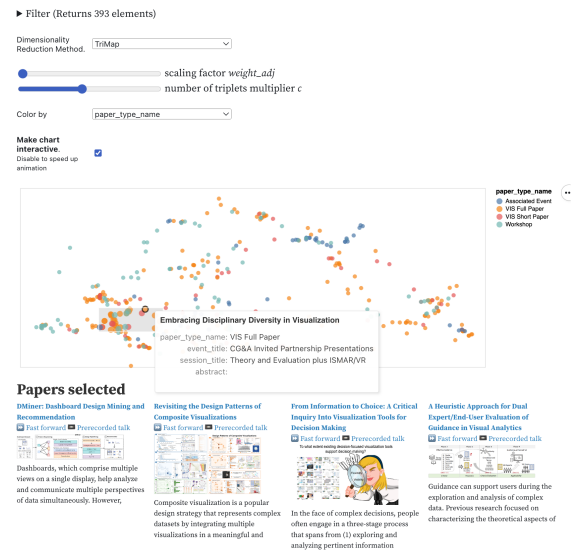


Figure 3: IEEE VIS 2023 Papers embeddings explorer interface. Built with the Reactive Widgets design pattern presented in this paper.

puted in the browser, even the embeddings. It features widgets for filtering the papers, changing the models' parameters, and even filtering the dimensionality reduction space. Part of the application interface can be seen in Figure 3.

### 5.2 Creating reactive widgets with Vega

*Vega-Selected* <https://observablehq.com/@john-guerra/vega-selected> is a convenient function to facilitate the creation of reactive widgets using the pattern described in this paper. It accepts a any visualization described as a JSON Vega specification as an input, and returns a reactive widget that will expose all the user's selections made in the visualization. It works well both with Vega's *point* and *interval* selections and has been tested even with complex faceted and layered visualizations. Its main limitation is that it exposes the selected data using Vega's internal signals and data structures which can be overwhelming. It was designed this way to support any type of Vega specification, however, it should be straightforward for a programmer to extract the selected data once the specification is defined, as shown in the examples. It can be easily used with any derivative of Vega that can output a Vega JSON specification.

### 5.3 IEEEVIS Observable Tutorial

An initial version of the concepts described in this paper can be found in the *Quick and Effective Visualization Prototyping with Reactive Notebooks* IEEEVIS 2021 tutorial [9]. More information about reactive programming can be found there.

## 6 CONCLUSION

This paper presents a new reusable and reactive widget design pattern that aims to facilitate the dissemination and replication of technical research contributions in the visualization field. The presented design facilitates the development of larger applications while also allowing other researchers and the community as a whole to take advantage of our contributions. This proposal leverages the power web standards to guarantee its generalizability and perseverance over time. We hope that this contribution will enrich our community and motivate more dissemination of our contributions, while also improving the replicability of experimental results by other researchers who could more easily include widgets into their own applications.

## SUPPLEMENTAL MATERIALS

All supplemental materials are available on Observable at <https://johnguerra.co/reactiveWidgets/>, released under a MIT license. In particular, they include (1) The Reactive Widget code template, (2) a helper function to facilitate development of widgets, (3) D3 examples and templates, (4) Vega Examples and helper function, (5) Vanilla JavaScript example for usage outside Observable, (6) Other applications and widget examples.

## REFERENCES

- [1] M. Q. Baldonado, A. Woodruff, and A. Kuchinsky. Guidelines for using multiple views in information visualization. *Proceedings of the Workshop on Advanced Visual Interfaces*, pp. 110–119, 2000. doi: 10.1145/345513.345271 2
- [2] M. Bostock. A Better Way to Code. Introducing d3.express: the integrated discovery environment, 4 2017. 2
- [3] M. Bostock. A Brief Introduction to Viewof / Observable — Observable, 2019. 2
- [4] M. Bostock, V. Ogievetsky, and J. Heer. D3 Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 12 2011. doi: 10.1109/TVCG.2011.185 2
- [5] b. Chris North and B. Shneiderman. A taxonomy of multiple window coordination. Technical report, University of Maryland, College Park, 5 1997. 2
- [6] R. Cutura, C. Kralj, and M. Sedlmair. DRUIDJS - A JavaScript Library for Dimensionality Reduction. *Proceedings - 2020 IEEE Visualization Conference, VIS 2020*, pp. 111–115, 10 2020. doi: 10.1109/VIS47514.2020.00029 4
- [7] Google PAIR. PAIR-code/umap-js: JavaScript implementation of UMAP, 2019. 1, 4
- [8] J. A. Guerra-Gomez. navio — A d3 visualization widget to help summarizing, exploring and navigating large network visualizations, 2018. 1, 4
- [9] J. A. Guerra-Gomez. Quick and Effective Visualization Prototyping with Reactive Notebooks. Observable IEEEVIS 2021 tutorial, 2021. 4
- [10] J. Heer. Vega-Lite API — vega-lite-api, 2019. 2
- [11] M. Hognräfer and H. J. Schulz. ReVize: A library for visualization toolchaining with vega-lite. *Italian Chapter Conference 2019 - Smart Tools and Apps in computer Graphics, STAG 2019*, pp. 129–139, 2019. doi: 10.2312/STAG.20191375 2
- [12] HuggingFace. Transformers.js, 2022. 4
- [13] M. B. Kery, D. Ren, F. Hohman, D. Moritz, K. Wongsuphasawat, and K. Patel. Mage: Fluid moves between code and graphical work in computational notebooks. *UIST 2020 - Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, pp. 140–151, 10 2020. doi: 10.1145/3379337.3415842 2
- [14] T. Manz. manzt/anywidget: jupyter widgets made easy, 2023. 2
- [15] L. McInnes, J. Healy, and J. Melville. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction, 2 2018. doi: 10.48550/arXiv.1802.03426 1, 4
- [16] Meta. React, 2013. 2
- [17] Mike Bostock. Towards Reusable Charts, 2012. 2
- [18] Mozilla Developer Network. Element: input event - Web APIs — MDN, 2024. 2
- [19] Mozilla Developers Network. HTML elements reference - HTML: HyperText Markup Language — MDN, 2024. 2
- [20] Observable. inputs/src/bind.js at main · observablehq/inputs, 2021. 3
- [21] Observable. Synchronized Inputs / Observable — Observable, 2022. 2
- [22] Observable. Observable Inputs — Observable documentation, 2024. 2, 4
- [23] J. M. Perkel. Reactive, reproducible, collaborative: computational notebooks evolve. *Nature*, 593(7857):156–157, 5 2021. doi: 10.1038/d41586-021-01174-w 2
- [24] C. Plaisant, J. Grosjean, and B. Bederson. SpaceTree: supporting exploration in large node link tree, design evolution and empirical evaluation. In *IEEE Symposium on Information Visualization, 2002. INFOVIS 2002.*, pp. 57–64. IEEE Comput. Soc, 1998. doi: 10.1109/INFVIS.2002.1173148 2, 4
- [25] Rich Harris. Svelte • Cybernetically enhanced web apps, 2016. 2
- [26] J. C. Roberts. Exploratory Visualization with Multiple Linked Views. *Exploring Geovisualization*, pp. 159–180, 1 2005. doi: 10.1016/B978-008044531-1/50426-7 2
- [27] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 1 2017. doi: 10.1109/TVCG.2016.2599030 1, 2, 4
- [28] A. Satyanarayan, R. Russell, J. Hoffswell, and J. Heer. Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):659–668, 1 2016. doi: 10.1109/TVCG.2015.2467091 1, 2, 4
- [29] B. Shneiderman. The future of interactive systems and the emergence of direct manipulation†. *Behaviour & Information Technology*, 1(3):237–256, 7 1982. doi: 10.1080/01449298208914450 2
- [30] B. Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *Computer*, 16(8):57–69, 8 1983. doi: 10.1109/MC.1983.1654471 2
- [31] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*, vol. 0, pp. 336–343. IEEE Comput. Soc. Press, Los Alamitos, CA, USA, 1996. doi: 10.1109/VL.1996.545307 1
- [32] M. Sun, A. Namburi, D. Koop, J. Zhao, T. Li, and H. Chung. Towards Systematic Design Considerations for Visualizing Cross-View Data Relationships. *IEEE Transactions on Visualization and Computer Graphics*, 28(12):4741–4756, 12 2022. doi: 10.1109/TVCG.2021.3102966 2
- [33] A. R. S. US Department of Agriculture. Nutrient Data Laboratory, 2016. 4
- [34] J. VanderPlas, B. E. Granger, J. Heer, D. Moritz, K. Wongsuphasawat, A. Satyanarayan, E. Lees, I. Timofeev, B. Welsh, and S. Sievert. Altair: Interactive Statistical Visualizations for Python. *Journal of Open Source Software*, 3(32):1057, 12 2018. doi: 10.21105/joss.01057 2
- [35] H. Wickham. Tidy Data. *Journal of Statistical Software*, 59(10), 2014. doi: 10.18637/jss.v059.i10 2
- [36] K. Wongsuphasawat. Encodable: Configurable Grammar for Visualization Components. *Proceedings - 2020 IEEE Visualization Conference, VIS 2020*, pp. 131–135, 10 2020. doi: 10.1109/VIS47514.2020.00033 2
- [37] J. S. Yi, Y. A. Kang, J. T. Stasko, and J. A. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231, 11 2007. doi: 10.1109/TVCG.2007.70515 2