THE REACTIVE VEGA STACK: DECLARATIVE INTERACTION DESIGN FOR DATA VISUALIZATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Arvind Satyanarayan
August 2017

Abstract

Interactive visualization is an increasingly popular medium for analysis and communication as it allows readers to engage data in dialog. Hypotheses can be rapidly generated and evaluated in situ, facilitating an accretive construction of knowledge and serendipitous discovery. Crafting effective visualizations, however, remains difficult. Programming toolkits are typically required for custom visualization design, and impose a significant technical burden on users. Moreover, existing models of visualization relegate interaction to a second-class citizen: imperative event handling callbacks that are difficult to specify, and even harder to reason about.

This thesis introduces the *Reactive Vega stack*: two new declarative languages for *interactive* visualization that decouple specification (the *what*) from execution (the *how*). At the foundation is *Reactive Vega*, an expressive representation that models user input as streaming data. Its underlying dataflow runtime handles the complexity of event propagation and state management, freeing users to focus on interaction design decisions. *Vega-Lite* builds on Vega to provide a high-level grammar for rapidly authoring interactive graphics for exploratory analysis. Its concise specification format decomposes interaction design into semantic units that can be systematically enumerated.

Critically, these languages offer JSON syntaxes to simplify programmatic generation of interactive visualization and enable novel interactive data systems. This thesis develops one such system, *Lyra*, a direct manipulation tool for visualization design. Drag-and-drop operations in Lyra generate statements in Vega and Vega-Lite, allowing users to author a diverse range of visualizations without any textual specification.

These systems have been released as open-source projects, have been widely adopted, and have given rise to an *ecosystem* of interactive visualization tools. Users can author an exploratory visualization in the Jupyter Notebook, export it to Lyra via Vega-Lite and add an explanatory annotation layer, and then embed the resultant Reactive Vega visualization within a Wikipedia article. As a result, rather than a single monolithic system, the Reactive Vega stack facilitates development of targeted applications, and allows users to work at the level of abstraction most suited for the task at hand.

Acknowledgments

By every measure, Jeffrey Heer has been a consummate advisor. Early in my graduate student career, I was confronted with a fork in the road: remain at Stanford, a place I had worked hard to get to, or follow Jeff up to the University of Washington. Though I had to leave behind a significant other and a fledgling start-up, the choice was clear. It was Jeff's warmth, insight, and vision that had attracted me to Stanford; if he was moving, so must I! This dissertation more than validates that decision. Jeff has helped me not only grow as a researcher but, by frequently diving deep into the Vega code base, hone my software engineering skills as well. His commitment to producing high-caliber research *and* ensuring it is released as open-source work remains an ongoing source of inspiration. If I am half the advisor he is, my future students will be very luck indeed!

I have also been fortunate to have had the guidance of several other mentors. My undergraduate advisor, Jim Hollan, fueled my interest in human-computer interaction. A summer in his lab, where he gave me free reign to explore questions in multimodal interaction, launched my research career and his continued counsel to *"never let the urgent drive out the important"* kept me grounded. Wendy Mackay and Michel Beaudouin-Lafon have been critical for understanding the broader implications of my work (and for giving me ample excuses to visit Paris!). I am particularly excited about working with them on some of the future research directions we brainstormed over drinks or in the back of their car! Thank you also to Maneesh Agrawala, James Landay, Chris Ré, and Jeff Hancock for their time, generosity, and sage advice about the academic job market.

I am also indebted to my incredible collaborators and colleagues at both Stanford and the University of Washington. Little of the work described in this thesis would have been possible without the talent and dedication of Kanit Wongsuphasawat, Dominik Moritz, Jane Hoffswell, and Ryan Russell. The warm welcome from Danielle Bragg, Caitlin Bonnar, Felicia Cordeiro, and Daniel Epstein quickly made me feel like a member of the UW community. And, the 48-hour, caffeine-fueled, pre-CHI-deadline Webzeitgeist session with Ranjitha Kumar, Jerry Talton, Maxine Lim, and Cesar Torres remains one of my favorite graduate school memories. Thank you also to staff in both departments, including Jillian Lentz, Jayanthi Subramanian, Diane Rosano, Monica Niemiec, and Andrea Kuduk, for their patience with my uncountable administrative questions.

My work has been generously supported first by an SAP Stanford Graduate Fellowship, and later by a graduate fellowship from Google. I credit the widespread adoption of the Reactive Vega stack, in part, to the opportunities we have had to spread the word. To that end, I am grateful to our external collaborators including Irene Ros, Lynn Cherny, K. Adam White, Sue Lockwood, Jake Vanderplas, Brian Granger, and Yuri Astrakhan.

I am fortunate that graduate school has mostly offered me very high highs. But, for the occasional low low, I am thankful to have been able to commiserate, rant, and drink with a fantastic group of friends including Marianne Lontoc, Patrick Mutchler, Rachel Midura, Lauren Howe, Ben Poole, Megan Lin, Austin Gibbons, and Jessie Schroeder.

Finally, I dedicate this thesis to my family. To my parents, Manju and Satya, for fostering my interest in computing, often "forgetting" I had used up my alloted screen time for a particular day! Your unconditional love gave me the support I needed to forge an identity and career half a world away. And to the love of my life, Maeva Fincker, for being my rock these past four years. I am so very lucky to have you on my team, to celebrate my successes, and to confide in you my fears.

Thank you.

Contents

List of Figures

1 Introduction

> *A graphic is not "drawn" once and for all; it is "constructed" and reconstructed until it reveals all the relationships constituted by the interplay of the data. The best graphic operations are those carried out by the decision-maker himself.*
>
> *Graphics and Graphic Information Processing*
> Jacques Bertin, 1981

Data visualization has gone mainstream. From business intelligence to data journalism, society has embraced visualization as a medium for recording, analyzing, and communicating about data. This explosion of use is reflected in the proliferation of visualization tools, which span the gamut of expressivity. At one end are chart templates (e.g., as found in spreadsheet packages such as Microsoft Excel). Users can easily generate recognizable output by choosing from a pre-defined palette of chart types, but only a handful of properties may be customized. At the other end are vector graphics programs (e.g., Adobe Illustrator), and low-level programming APIs (e.g., Processing or OpenGL). Users have complete control but the added expressivity trades-off ease-of-use.

Advances in visualization toolkits have popularized *grammar-based* approaches for visual encodings [16, 17, 92, 105, 107]. These models are more expressive than chart templates as they decompose the design space into combinatorial primitives for data transformation and visual encoding. Moreover, they embody *declarative design* — the languages describe *what* the visualization should look like, rather than *how* it should be rendered. As a result, users are free to focus on visual encoding and design decisions while the underlying runtime is responsible for execution concerns and performance optimization [47].

However, key challenges remain. Interaction is critical for effective visualization, allowing users to interrogate data and iteratively refine their mental models [79, 111]. Yet, existing declarative visualization models provide poor support for interaction techniques. Simple "interaction typologies" (e.g., brushing, panning, etc.) are available, but limit customization. For custom interaction, users must author *imperative* event handling callbacks that undo the benefits of declarative design. Users are forced to manually maintain state [29] and coordinate interleaved execution — a complex and error-prone task colloquially called "callback hell" [34]. As a result, users must either invest significant effort in interaction design, or choose to forgo it and risk missing critical insights.

Moreover, most existing declarative models (including ggplot2 [105] and D3 [17]) are instantiated via complex APIs embedded in programming languages. This approach imposes a non-trivial *articulatory distance* [55] as users must map their desired *visual* output to *textual* commands — a fundamental mismatch in representations.

1.1 Thesis Contributions and Outline

This dissertation is divided into seven chapters, and contributes a layered stack of new declarative languages for *interactive* visualization as well as a new interactive system for visualization design through direct manipulation.

Chapter 2 surveys prior work that this dissertation builds on, including a panoply of visualization toolkits and systems, methods from functional programming languages and streaming database systems, and generative models of user interfaces.

Chapter 3 introduces Reactive Vega: a declarative language for interactive data visualization that models user interaction as *streaming data*. Alongside established declarative visual encoding primitives, Reactive Vega introduces event streams and signals, two constructs from Functional Reactive Programming. Event streams, defined with a novel CSS-inspired selector syntax, abstract the complexity of capturing and sequencing input events. Event streams drive signals: dynamic expressions that are automatically reevaluated when new events fire. Signals parameterize visual encoding primitives, thereby endowing them

with reactive semantics — when a new event fires, it is propagated to corresponding signals; dependent visual encodings are reevaluated and the visualization is automatically re-rendered. Additional primitives allow interaction techniques to be generalized and reused across distinct visualizations. This chapter demonstrates Reactive Vega's expressivity through example visualizations that cover an established interaction taxonomy.

Chapter 4 studies the implications of declarative interaction design on the architecture of visualization systems. The Reactive Vega architecture adapts and extends techniques from streaming database systems. In particular, an input declarative specification is parsed to construct a dataflow graph. Dataflow operators perform computations on *changesets* of tuples or scene graph elements, and are *replayed* if signal parameter values change. To support data-driven multi-view displays, the dataflow graph is conditioned on data values. As new tuples are observed, or as interaction events occur, the graph *dynamically rewrites itself* at runtime by extending or pruning branches. Performance optimizations are described and comparative benchmark studies conducted to evaluate Reactive Vega's performance. These studies find that Reactive Vega meets or exceeds the performance of D3 [17], the current state-of-the-art system.

Reactive Vega's primitives, however, are relatively low-level. They yield verbose specifications that impede a rapid authoring process and hinder exploration of alternative designs. Chapter 5 introduces Vega-Lite: a higher-level grammar that builds on Reactive Vega. Vega-Lite extends a traditional grammar of graphics with a *view composition algebra* for layered and multi-view displays. A novel *grammar of interaction* features *selections*: an abstraction that encapsulates input event processing, points of interest, and a predicate function for inclusion testing. Selections parameterize visual encodings by serving as input data, defining scale extents, or by driving conditional logic. Vega-Lite specifications are concise through ambiguity: lower-level details (e.g., the events that trigger a selection) can be omitted and are filled in by the Vega-Lite compiler. The smaller language surface area also facilitates higher-level reasoning tasks, such as systematic enumeration of alternative designs. By recreating the examples from Chapter 3, this chapter demonstrates how Vega-Lite enables succinct specification and broader exploration of the design space.

Reactive Vega and Vega-Lite provide JSON syntaxes to facilitate the programmatic generation of visualizations in novel higher-level interactive data systems. Chapter 6 describes one such system, Lyra, that enables direct manipulation visualization design. Through drag-and-drop interactions, users bind data values to mark properties. Data transformations and layout algorithms are visually specified and inspected with a "data pipeline" interface. Critically, Lyra allows users to fluidly move between the two levels of abstraction. Direct manipulation operations generate Vega-Lite statements, which are compiled, and merged into a backing Reactive Vega specification; the visual inspectors provide complete control over the latter. Thus, users can iterate between rapidly creating recognizable output and making fine-grained customizations — an approach that yields diverse range of visualizations without writing a single line of code.

Finally, Chapter 7 sketches new research directions that Reactive Vega and Vega-Lite enable. These systems have been released as open-source projects, and have already been used to power the Voyager visualization recommendation browser [108, 109, 110], develop a model for sequencing visualizations [58], and reverse-engineer visualizations from chart images [80]. Moreover, the broader adoption both tools have seen — Vega is used to embed interactive visualizations within Wikipedia articles, and Vega-Lite can be used within Jupyter notebooks — provide a growing, engaged user community to study their use with.

1.2 Prior Publications and Authorship

Although I am the principal author of the research detailed in this dissertation, it is also the product of years of collaboration with my primary advisor, Jeffrey Heer, and my colleagues at the University of Washington Interactive Data Lab. The Reactive Vega language (Chapter 3) appeared at ACM UIST 2014 [88] with Kanit Wongsuphasawat contributing a prototype implementation of the reactive runtime. The Reactive Vega architecture (Chapter 4) was published at IEEE VIS 2015 [87], with Ryan Russell and Jane Hoffswell contributing example visualizations and figures. Vega-Lite was published at IEEE VIS 2016 [86]. It is joint work with Dominik Moritz and Kanit Wongsuphasawat, who led the design and implementation of the visual encodings. Finally, Lyra was published at EuroVis 2014 [85]. To reflect my collaborators' contributions, I will use the first-person plural in these chapters.

2 Related Work

This dissertation builds on prior work in visualization systems, programming languages, streaming databases, and human-computer interaction. In this chapter, I survey pertinent work in each domain and describe how this thesis extends them.

2.1 Visualization Toolkits & Systems

Chart templates, as found in spreadsheets and online services (e.g., Many Eyes [101] and Google Fusion Tables), are a common means for creating visualizations. These *chart typologies* are popular because they are simple to use: users select from a pre-defined palette of chart types, and produce recognizable output with only a few clicks. This approach, however, is closed. Users are restricted to only the available types, and may customize only a small handful of parameters.

To enable custom visualization design, researchers have developed programming toolkits that offer a number of visualization components. For instance, the InfoVis Toolkit [35] provides a class hierarchy of visualization widgets, where new visualizations are created by subclassing existing components or writing new ones. In contrast, Prefuse [48] and Flare [36] offer composable operators for data transform, layout, and encoding. While either approach offers fine-grained control over visual output, they both constrain expressivity much like chart typologies. Novel designs require new widgets or operators, and significant software engineering effort.

2.1.1 Grammar-Based Visual Encoding

In 1999, Leland Wilkinson introduced an alternative approach with *The Grammar of Graphics* [107]. Eschewing chart typologies in favor of combinatorial building blocks, Wilkinson's grammar yields a more expressive design space and allows users to rapidly construct visualizations. Provided abstractions include primitives for data modeling and transformation, visual encodings defined as mappings between data fields and channels (e.g., position, color, and size), and scales and guides (i.e., axes and legends). His work was quickly followed by the Stanford Polaris system [92] (commercialized as Tableau), and Hadley Wickham's popular ggplot2 [105] package implements a variant of this model in R.

These tools focus on statistical graphics for exploratory analysis and, to that end, favor concise specification languages. Analysts may omit details such as scale transforms (e.g., linear or log) or color palettes, which are then filled in by the grammar using a rule-based system of smart defaults. As a result, analysts are able to rapidly generate visualizations, analyze them, and continue with their analysis process.

Through the lower-level Protovis [16, 47] and D3 [17] languages, Bostock and Heer describe how a similar approach can enable an expressive space of custom, explanatory graphics as well. They also articulate the advantages of declarative visualization design: separating specification from execution facilitates an iterative development process, simplifies retargeting across platforms, and enables unobtrusive runtime optimizations [47].

The design of Reactive Vega and Vega-Lite is heavily influenced by these works. Reactive Vega's visual encoding primitives and implementation pipeline closely mirror Protovis'. Marks including *arcs*, *areas*, *bars*, *lines*, plotting *symbols* and *text* are processed through *bind*, *build*, and *evaluate* stages to generate a scene graph. Protovis' *panel* marks are renamed to *group* marks, and enable the repeated and nested structures found in small multiple displays [94]. Marks can be positioned relative to one-another with Vega's *reactive geometry*, which provides a more unified approach to layout as compared with Protovis' *anchors*.

Similarly, Vega-Lite represents basic plots using a set of encoding definitions that map data attributes to visual channels, and may include common data transformations. Vega-Lite differs most from other high-level grammars in its approach to multiple view displays. Each of these grammars supports faceting (or nesting) to construct trellis plots [12] in which each cell visualizes a different partition of the data. Both Wilkinson's grammar and Polaris/Tableau achieve this through a *table algebra* over data fields, which in turn determines spatial subdivisions. Tableau additionally supports the construction of multi-view dashboards via a different mechanism, with each view backed by a separate specification. In contrast, Vega-Lite contributes a *view algebra*: starting with unit specifications that define a single plot, composite views are expressed using operators for layering, horizontal or vertical concatenation, faceting, and parameterized repetition. When applicable, these operators merge scale domains and properly align constituent views. Disparate views can also be combined into arbitrary dashboards, all within a unified algebraic model.

2.1.2 Dataflow Visualization Systems

Dataflow architectures are common in scientific visualization systems, such as IBM Data Explorer [3, 67] and VTK [89]. Developers must manually specify and connect each required operator into a network, with updates supported in a demand-driven fashion (e.g., as data is modified) or an event-driven fashion (e.g., in response to user input). These systems expose fine-grained control over the dataflow graph. For example, VTK developers can choose to favor memory efficiency over processing speed, which causes dataflow operators to delete their output after computation. While Reactive Vega shares some dataflow strategies with these systems — for example, using pass-by-reference for unchanged tuples to reduce memory consumption — it abstracts such execution concerns away from the user. The dataflow graph is automatically assembled based on definitions found in a declarative specification, and optimizations are transparently performed such that output data is only stored when needed by downstream operators and shared whenever possible.

Within the domain of information visualization, Stencil [30] is also grounded in reactive semantics and uses a dataflow model. Like Reactive Vega, it provides a unified data model where both input data and interaction events are modeled as first-class streaming data

sources. However, Reactive Vega is more expressive than Stencil in two important ways. First, Stencil does not provide primitives, beyond event streams, to support interaction design; Reactive Vega, on the other hand, offers predicates and scale inversions to lift interactive selections to data queries, and coordinate multiple displays. Moreover, Reactive Vega's graphical primitives can be arbitrarily nested, drawing from either hierarchical or distinct data sources. This ability is critical to concisely specifying small multiples displays, and requires Reactive Vega's dataflow graph to dynamically rewrite itself at runtime. To the best of our knowledge, Stencil does not support self-instantiating dataflows.

Improvise [104] features active variables called "live properties," which provide a basic form of reactivity — they may be bound to control widgets and parameterize a visualization. Using an expression language, live properties are assembled into a coordination graph to dynamically evaluate visual encodings and generate views of data. While Improvise and Reactive Vega share some conceptual underpinnings, Improvise places a higher burden on users to correctly construct the necessary graph. As Reactive Vega takes a declarative approach to visualization design, users need only compose the necessary primitives into a specification; the system parses it to build the requisite dataflow graph.

2.1.3 Interactive Visualization Systems

Besides text-based toolkits, researchers have also developed graphical user interfaces for constructing visualizations. Polaris/Tableau [92], for instance, displays a "schema" panel that lists available data fields. Data groupings and visual encodings are specified by dragging these fields onto "shelves" that array the periphery of the visualization. Lyra extends this mode of interaction onto the visualization canvas itself — property *drop zones* overlay marks during drag operations, providing a more direct target for data binding operations.

Roth et al.'s SageBrush [82] supports similar interactions. Users can drag-and-drop "partial prototypes" for spatial encodings and "grapheme" (mark) primitives such as lines and bars. Custom grapheme properties are manually selected via menus, then exposed as drop-target icons. Lyra refines the SageBrush model in several ways. Partial prototypes

in SageBrush implicitly define scale transforms and layout; by leveraging Vega and Vega-Lite, Lyra decouples these components to support richer designs. Lyra's data pipelines provide an extensible set of data transformations, and different marks can be driven by unique pipelines and composed. Finally, Lyra eschews iconic abstractions in favor of direct drop zones for mark properties, without need of explicit enumeration via menus.

More recently, Bret Victor demonstrated a data-driven drawing tool [100] (now developed as Apparatus [5]) that combines geometric constraints with *imperative* procedures over data. Alongside an interactive canvas and data table viewer, the tool includes programming constructs such as lexical scope, looping control flow, and conditionals. For a basic bar chart, designers must define loops to create and position each bar. The tool supports purely geometric constructions, rendering some layouts inexpressible. By building on Vega and Vega-Lite, Lyra takes a *declarative* approach to design. When a designer associates a mark with a dataset, one mark instance per datum is automatically produced rather than requiring explicit instantiation. More advanced layouts (e.g., cartographic projections, force-directed layout, etc.) are available through Lyra's graphical data pipeline.

2.2 Specifying Interactions in Visualizations

Despite the central role of interaction in effective data visualization [50, 79], little work has been done to develop a grammar for specifying interaction techniques. Wilkinson's grammar includes no notion of interaction. Tableau supports common interaction techniques, but relies on mechanisms external to the visual encoding grammar. Early systems like GGobi [93] support common techniques as well, and provide imperative APIs for custom methods. However, such APIs make easy tasks needlessly complex, burdening developers with learning low-level execution details. More recent systems, including Protovis, D3, and VisDock [26], offer a typology of common techniques that can be applied to a visualization. Such top-down approaches, however, limit customization and composition. For example, D3's interactors encapsulate event processing, making it difficult to combine them if their events conflict (e.g., if dragging triggers brushing *and* panning).

2.2.1 Interactive Selection & Querying

Selection (e.g., users clicking or lassoing visual items) is a fundamental operation in user interfaces and has been well-studied in data visualization. For example, in Snap-Together Visualization [75], multiple views are coordinated via "primary-" and "foreign-key actions," which propagate selected data tuples from one view to others. Wilhelm [106] describes the need for such "indirect object manipulation" methods as an axiom of interactive data displays. Chen's compound brushing [25] provides a visual dataflow language for specifying a rich space of transformations of brush selections. More recently, Brunel [19] provides a special `#selection` data field that is dynamically populated with the elements a user interacts with, and can be used to link multiple views or filter input data. Similarly, RStudio's Shiny [81], an imperative web application layer, provides `brushedPoints` and `nearestPoints` functions which can be used to manipulate selected elements.

Other systems have studied formally representing selections as data queries [106]. For example, brushing interactions in VQE [32] generate *extensional* queries that enumerate all items of interest; a form-based interface enables specification of *intensional* (declarative) queries. Individual point and brush selections in DEVise [66], known as *visual queries*, map to a declarative structure and are used to link together multiple views. Rectangular "rubber band" selections in VIQING [76] are modeled as range extents, and views can be dropped on top of each other to join their underlying datasets. Heer et al. [46] demonstrate that declarative selections can be interactively "relaxed" to capture more items of interest.

Reactive Vega and Vega-Lite both build on this work with abstractions for constructing selections. In Reactive Vega, *predicates* express interactive selections in visual space, are lifted to data queries by inverting them through scale functions, and participate in if-then-else chains to manipulate visual encoding rules. Vega-Lite abstracts these primitives with a higher-level *selection*. Vega-Lite selections are populated with one or more points of interest, in response to user interaction. Their built-in predicates map to declarative queries, thereby allowing a minimal set of "backing" points to represent the full space of selected

points. Additional operators transform a selection's predicate or backing points (e.g., offsetting them to translate a brush or perform panning). Selections parameterize visual encodings by driving conditional logic, serving as input data, or defining scale extents.

2.3 Alternate Paradigms to Imperative Callbacks

For custom interaction design with existing visualization toolkits, users must typically author imperative event handling callbacks which present several development hurdles [74]. Callbacks are registered outside the visual encoding specification, breaking encapsulation and necessitating side-effects [28]. Users are forced to manipulate the visualization state externally, and manually update dependencies [29]. Moreover, callback execution order can be unpredictable, requiring users to coordinate interleaved calls [34]. As a result, callbacks make it difficult to reason about the current state of the visualization, which is now the product of both the original specification and the side-effects of interaction callbacks.

2.3.1 Constraint Programming

One alternative to callback-oriented imperative programming is constraint-based specification. Systems like Gilt [74] and μconstraints [54] implement one-way constraints: when the value of a constrained interface is modified, its dependents are automatically affected. Other systems, like Cassowary [8], implement two-way constraints using more complex solvers. Recently, ConstraintJS [77] and Bret Victor's prototype system [100] have shown that web applications and visualizations, respectively, can be authored with constraints. However, constraint systems do not involve general consideration of event handling — a crucial element for interaction design. In fact, the authors of ConstraintJS intend for their system to complement event architectures [77].

2.3.2 Functional Reactive Programming

Functional Reactive Programming (FRP) formalizes semantics similar to one-way constraints. Drawing from dataflow programming, FRP recasts mutable values as continuous, time-varying data streams [9]. We focus on a discrete variant called Event-Driven

FRP (E-FRP) [103]. To capture value changes as they occur, E-FRP provides *streams*, which are infinite time-ordered sequences of discrete events. Streams can be composed into *signals* to build expressions that react to events. The E-FRP runtime constructs the necessary dataflow graph such that, when a new event fires, it propagates to corresponding streams. Dependent signals are evaluated in two phases: signals first use the prior computed values of their dependents, which are subsequently updated in the second phase. E-FRP has been shown to be viable for authoring web applications [31, 71] and visualizations [30, 57].

Reactive Vega's declarative interaction primitives are grounded in E-FRP semantics, and they preserve the two-phase update: interdependent signals are updated in the order in which they are defined in the specification. However, naive applications of E-FRP to visualization tasks can result in wasteful recomputation. Traditional E-FRP primitives support only *scalar* values, whereas visualization pipelines must also process relational and hierarchical data. Modeling these latter data types as scalar values provides insufficient granularity to perform targeted recomputation. To efficiently support relational data, Reactive Vega integrates streaming database methods; and, to support streaming hierarchical data, the dataflow graph dynamically rewrites itself at runtime, instantiating new branches to process nested relations.

2.4 Data Stream Management

The problem of managing streaming data has been well studied in the database community. Researchers have developed an arsenal of techniques through systems such as Aurora [2], Eddies [7], STREAM [6], and TelegraphCQ [24]. As tuples are observed by these systems, they are flagged as new or removed. Tuples, rather than full relations, are passed between operators in a query plan (realized as a dataflow graph). As a result, operators inspect just the updated tuples to perform efficient computation. However, for some operations a set of changed tuples is insufficient. For example, a join of two relations requires access to all tuples within a specified window. In such cases, caches (sometimes called *views* [2] or *synopses* [6]) are used to materialize a relation, and shared among dependents.

Borealis [1] extends this work in two ways. To support streaming tuple modifications, it introduces a revision processing scheme. An operator can be *replayed* with revised tuples in place of the original data, and will then only emit corresponding revisions. Similarly, to enable dynamic operator parameters, Borealis introduces *time travel*. When an operator parameter changes, an `undo` is issued to the nearest cache. The cache emits tuple deletions, effectively "rewinding" the system to a previous time. A subsequent `replay` triggers recomputation with the new parameter.

However, existing streaming data systems concern flat relations. Reactive Vega instantiates these techniques, alongside E-FRP, within a visualization pipeline and extends them to support streaming nested data. To do so, Reactive Vega's dataflow graph dynamically rewrites itself at runtime with new branches. These branches unpack nested relations, enabling downstream operators to remain agnostic to higher-level structure while supporting arbitrary levels of nesting.

2.5 Generative Models of User Interfaces

Michel Beaudoin-Lafon's instrumental interaction paradigm [11] helped motivate this dissertation's approach of considering declarative interaction design at varying levels of abstraction. He identifies three properties of interaction models for post-WIMP interaction: *descriptive* of existing and new techniques; *comparative* to evaluate alternative designs; and, *generative*, to facilitate the identification and creation of new points in the design space. Reactive Vega only offers the first property — an expressive design space — but is too low-level to be generative or facilitate evaluation of similar behaviors. Vega-Lite, on the other hand, offers both an expressive and generative design space, suitable for exploring alternative techniques as demonstrated in § 5.5. Kim et al. also demonstrate its comparative powers, using Vega-Lite to model optimal sequences of visualizations [58].

Moreover, our articulation of Vega-Lite as a parametric design language is inspired by Mackinlay's APT [68] and his follow-on work with Card and Robertson conducting a morphological analysis of input devices [21]. In particular, we sought to identify and decouple orthogonal axes of the design space. While § 5.5 begins to systematically consider points along these axes, a formal analysis of the Vega-Lite design space is left to future work.

3 Declarative Interaction Primitives

Reactive Vega builds on a long-running thread of research on declarative visualization design, popularized by the Grammar of Graphics [107] and Polaris/Tableau [92]. In this chapter, we describe the design of the Reactive Vega specification language. Discussion of its underlying streaming dataflow architecture is deferred to the subsequent chapter.

Visual encodings are defined by composing graphical primitives called *marks* [16], which include *arcs*, *areas*, *bars*, *lines*, plotting *symbols* and *text*. Marks are associated with datasets, and their specifications map tuple values to visual properties such as position and color. Scales and guides (i.e., axes and legends) are provided as first-class primitives for mapping a domain of data values to a range of visual properties. Special *group* marks serve as containers to express nested or small multiple displays. Child marks and scales can either inherit a group's data or draw from independent datasets.

Although interaction is a crucial component of effective visualization [65, 79], existing declarative visualization models, including widely used packages such as D3 [17] and ggplot2 [105], do not offer composable primitives for interaction design. Instead, if they support interaction, they do so with either a palette of standard techniques [16, 17] or *imperative* event handling callbacks. While the former restricts expressivity, the later undoes many of the benefits of declarative design. Users are forced to contend with complex interaction execution details, such as coordinating state and interleaved events [29, 34, 74].

In response, Reactive Vega introduces a model for *declarative* interaction design.

```
{
  data: [{
    name: "table",
    url: "data/letter_counts.json"
  }],

  scales: [{
    name: "xscale", type: "band",
    domain: {data: "table", field: "category"},
    range: "width"
  }, {
    name: "yscale", type: "linear",
    domain: {data: "table", field: "amount"},
    range: "height"
  }],

  axes: [
    {orient: "bottom", scale: "xscale"},
    {orient: "left", scale: "yscale"}
  ],

  marks: [{
    type: "rect",
    from: {data: "table"},
    encode: {
      enter: {
        x: {scale: "xscale", field: "category"},
        width: {scale: "xscale", band: 1},
        y: {scale: "yscale", field: "amount"},
        y2: {scale: "yscale", value: 0},
        fill: {value: "steelblue"}
      }
    }
  }]
}
```

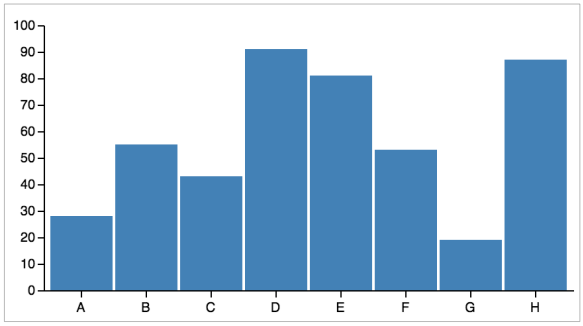


Figure 3.1: A JSON specification for a bar chart, demonstrating Vega's abstractions for visual encoding. Data is imported from a URL. Scales transform data values to visual values. Properties of graphical marks (in this case rectangles) are determined by scale mappings. Guides (here, axes) are instantiated as well.

3.1 Interaction Language Design

Reactive Vega models low-level events as composable data streams from which higher-level semantic *signals* can be constructed. Signals feed *predicates* and *scale inversions*, to generalize interactive selections at the level of item geometry (pixels) into interactive queries over the data domain. *Production rules* then use these queries to manipulate the visualization's appearance. To facilitate reuse and sharing, these constructs can be encapsulated as named *interactors*: standalone, purely declarative specifications of interaction techniques.

3.1.1 Event Streams and Signals

Reactive Vega adapts the semantics of Event-Driven Functional Reactive Programming [103]. Low-level input events (e.g., mouse events) are captured as time-varying *data streams*, rather than event callbacks. This abstraction reduces the burden of composing and sequencing events — operations that would require several callbacks and some external state under an imperative paradigm. To this end, we introduce a syntax for specifying event streams (Fig. 3.2). While prior work has formulated regex-based symbols for event selection [59] we believe our approach, by mimicking CSS selectors, will be more familiar to designers.

A basic event stream selector is specified by a particular event type (e.g., `mousemove`), optionally prepended with the source of the events — either a mark type (e.g., `rect:`) or name (e.g., `@cell:`). The comma operator (`,`) merges streams to produce a single stream with interleaved events. Square brackets (`[]`) filter events based on their properties. When followed by the right-combinator (`>`), the brackets indicate a "between filter," defining

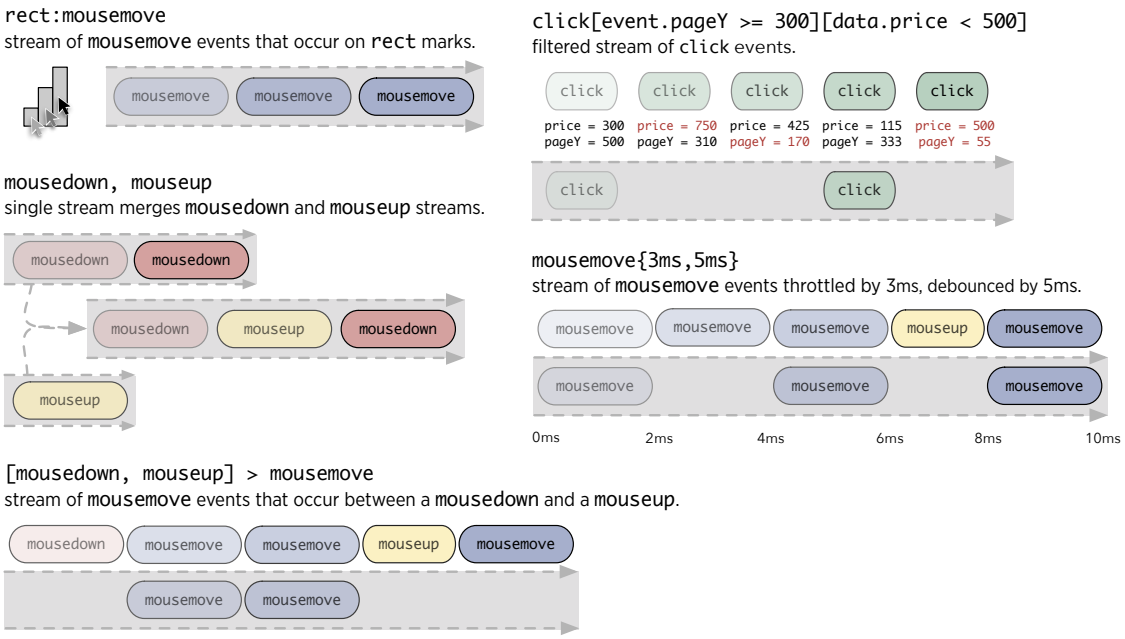


Figure 3.2: Reactive Vega provides an event stream selector syntax, inspired by CSS selectors, to compose, filter, and sequence input events.

bounding events for the stream. For instance, `[mousedown, mouseup] > mousemove` captures `mousemove` events that occur between a `mousedown` and `mouseup` (i.e., "drag" events). To throttle or debounce an event stream, timing information can be specified between curly braces (e.g., `{100, 200}` throttles a stream by 100ms and debounces it by 200ms). All operators are composable. For instance, `[mousedown[event.shiftKey], window:mouseup] > window:mousemove{100, 200}` specifies a stream of throttled and debounced drag events that are only triggered when the shift key is pressed.

With Reactive Vega, interaction events are a first-class data source. They can be run through the full gamut of data transformations and can drive visual encoding primitives. While doing so can usefully visualize a user's interaction, for added expressivity, event streams can also be composed into reactive expressions called *signals*. By default, signals are evaluated using the most recent event from a stream. However, signals can also define finite-state machines by drawing from multiple streams — each stream triggers a state transition.

Signals can be used to directly specify visual encoding primitives (e.g., a mark's fill color) thereby endowing them with reactive semantics. When an event fires, it enters appropriate streams and is propagated to corresponding signals; signals are re-evaluated and dependent visual encodings re-rendered automatically.

Upon definition, signals must be given unique names. These named entities are then used to define the rest of an interaction technique, thereby decoupling input events from downstream application logic. Thus, an interaction can be triggered by a different set of events by simply rebinding signal declarations. As we later demonstrate, rebinding is particularly useful for retargeting interactions or for combining otherwise conflicting interactions.

3.1.2 Predicates and Scale Inversion

Selection is a fundamental operation in interactive visualization design [46]. Once a selection is made, subsequent operators can be applied to manipulate the selected items. For visual design, it can be sufficient to make a predetermined selection (e.g., "select all rectangles"). With interaction design, however, selections are driven by user input — brushing over points of interest, or adjusting a slider to filter data.

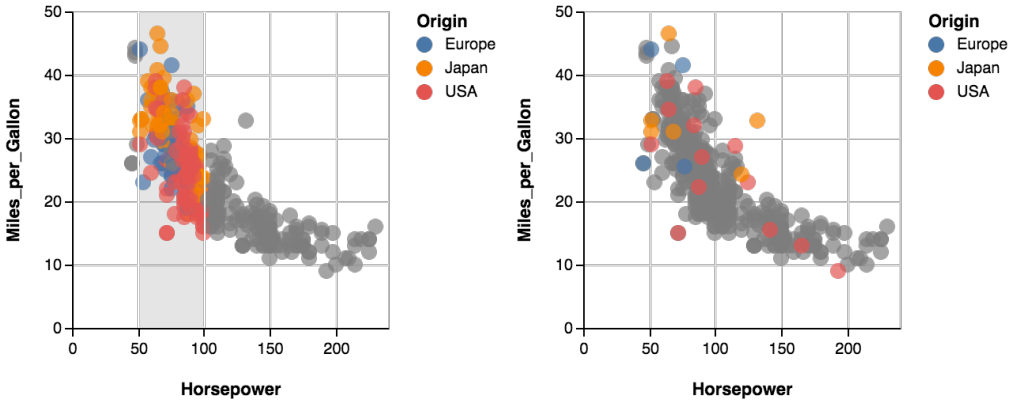


Figure 3.3: Points are highlighted using (left) an intensional predicate $50 \leq$ `Horsepower` ≤ 100 or (right) an extensional predicate with members `#56, #110, #79, #95, #40, #120, ...`.

To express interactive selections, we introduce reactive *predicates*. As shown in Fig. 3.3, predicates can be constructed either with an *intensional* definition — specifying conditions over properties of selected members — or an *extensional* one — explicitly enumerating all members of a selection.

Predicate operands are typically signals and, as signals drawn from input event streams, predicates express interactive selections at the visual (or pixel) level by default. However, pixel-level selection is often insufficient. A single visualization may have multiple distinct visual spaces, or an interactive technique may wish to coordinate several visualizations. In such cases, it is necessary to generalize an interactive selection into a query over the data domain [46]. Scale functions are a critical component in visualization design [107] as they transform data values into visual values such as pixels or colors. By applying an *inverted* scale function to predicate operands, we can lift a predicate to the data domain. Figure 3.4 demonstrates how a predicate defines an interactive selection and, when coupled with scale inversions, an interactive query to drive an overview + detail visualization.

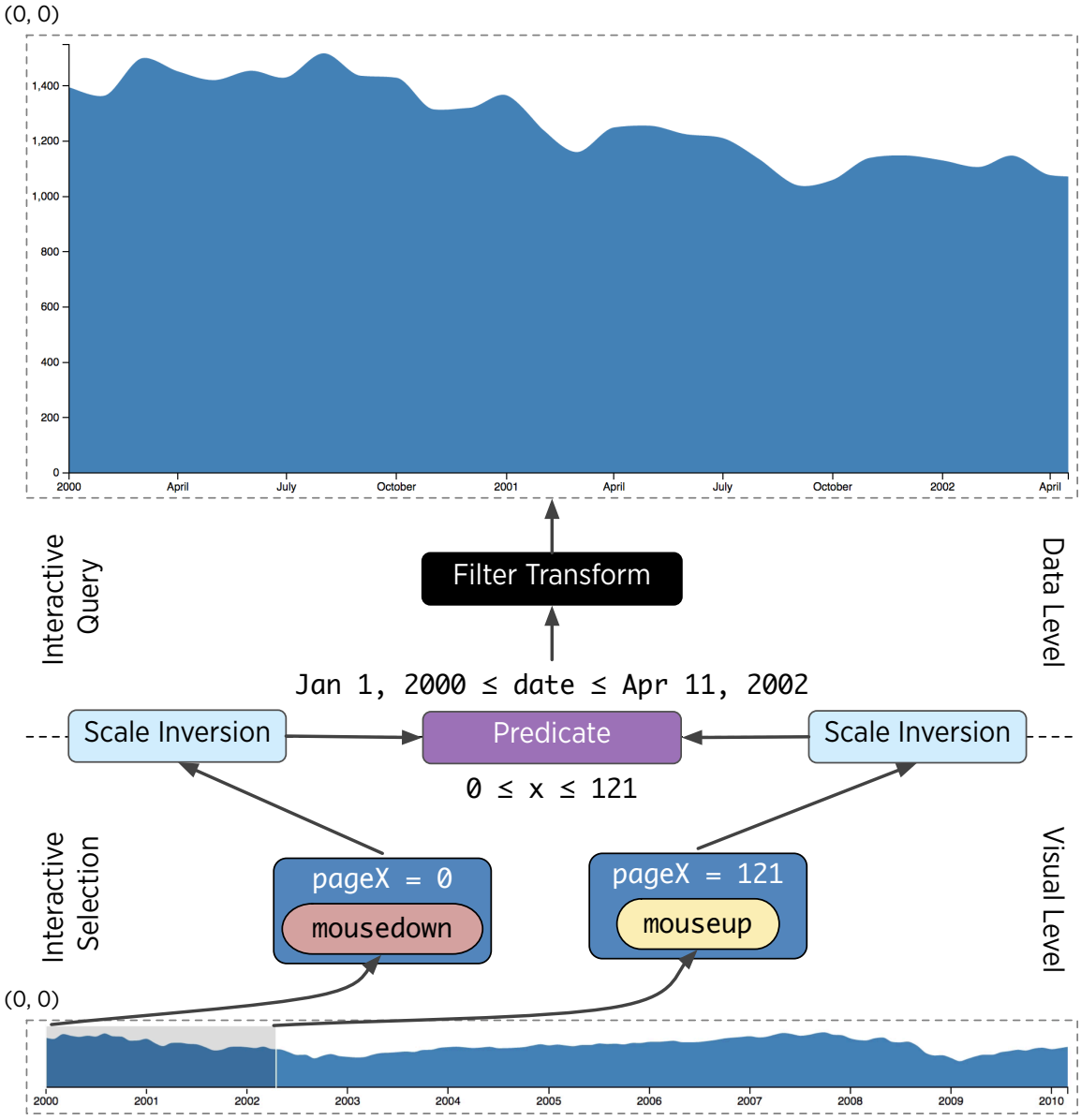


Figure 3.4: *Predicates* use signal values to define interactive selections of elements. Using *scale inversions*, predicates can be generalized to define interactive queries, and thus operate across different coordinate spaces: overview (bottom) and detail (top).

3.1.3 Production Rules

Production rules are an established design pattern for visualization specification [45] that we endow with reactive semantics. A rule defines the outcome of evaluating an `if-then-else` chain to set property values. For example, a rule might set a mark's fill color using scale-transformed data if predicate `A` is true, set it to yellow is predicate `B` is true, or otherwise set the color to grey by default.

3.1.4 User-Defined Functions

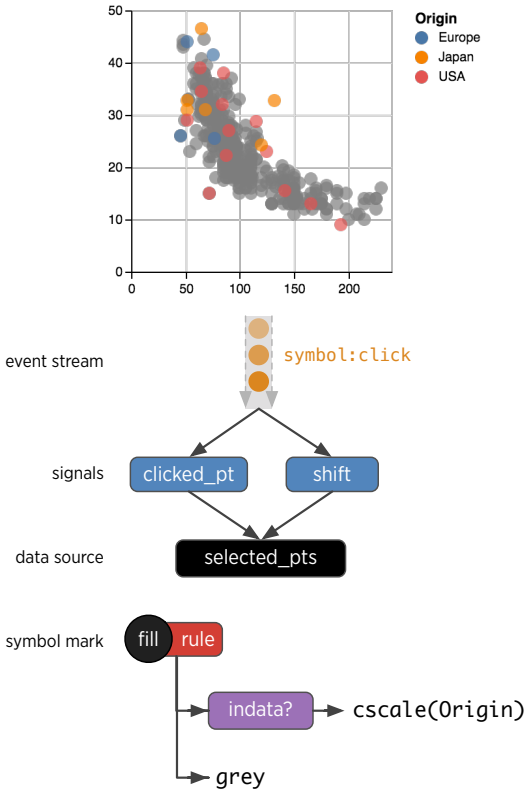
During our design process, we encountered visualizations in which interactions trigger custom data transforms. For example, sorting a co-occurrence matrix by frequency or querying time-series data via relaxed selections [53]. It is not feasible for a declarative language to natively support all possible functions, yet custom operations must still be expressible. Following the precedent of languages such as SQL, we provide *user-defined functions*. Such functions must be defined and registered with the system at runtime, and can subsequently be invoked declaratively within the specification. User-defined functions ensure that the language remains concise and domain-specific, while allowing idiosyncratic operations via extensions.

3.1.5 Encapsulated Interactors

To enable reuse of custom interaction techniques, Reactive Vega's interaction primitives can be parameterized and encapsulated as named *interactors*. Inspired by Garnet's component of the same name [73], a Reactive Vega interactor can subsequently be applied to a visualization and functions as a mixin. Its specification is merged into the host's and, to prevent conflicts, its components are addressable only under its namespace. Figures 3.6 and 3.7 illustrate how a brushing interaction (within a single scatterplot) can be extracted to a standalone interactor, and reapplied to brush & link a scatterplot matrix.

3.2 Example Interactive Visualizations

To evaluate the expressivity of our language, we present a range of examples and demonstrate coverage over Yi et al.'s interaction taxonomy [111]. Yi et al. identify seven categories based on user intent: *select*, to mark items of interest; *connect*, to show related items; *abstract/elaborate*, to show more or less detail; *explore*, to examine a different subset of data; *reconfigure*, to show a different arrangement of data; *filter*, to show something conditionally; and, *encode*, to use a different visual encoding. It is important to note that these categories are not mutually exclusive, and an interaction technique can be classified under several categories. We choose example interactive visualizations to demonstrate that our model can express interactions across all seven categories and how, through composition of its primitives, supports the accretive design of richer interactions.



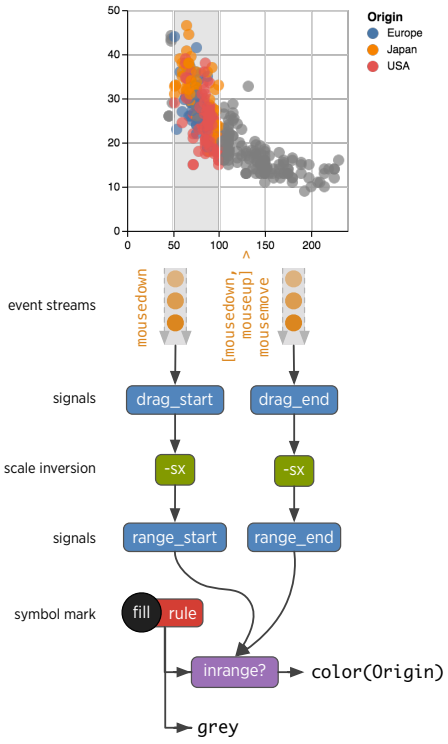

```
signals: [{
  name: "clicked_pt", value: null,
  on: [{
    events: "symbol:click",
    update: "datum", force: true
  }]
}, {
  name: "shift", value: false,
  on: [{
    events: "symbol:click",
    update: "event.shiftKey", force: true
  }]
}],

data: [{
  name: "selected_pts",
  on: [
    {trigger: "!shift", remove: true},
    {trigger: "!shift && clicked", insert: "clicked"},
    {trigger: "shift && clicked", toggle: "clicked"}
  ]
}],

marks: [{
  type: "symbol",
  encode: {
    update: {
      fill: [{
        test: "indata('selected_pts', 'id', datum.id)",
        scale: "cscale", field: "Origin"
      }, {
        value: "grey"
      }]
```

Figure 3.5: Reactive Vega JSON for a click-to-highlight interaction. Signals over a click stream feed data transform to toggle values in a data source. A production rule uses a predicate to set marks' fill color.

3.2.1 Selection: Click/Shift-Click and Brushing

Figure 3.5 provides a snippet of Reactive Vega JSON for a click-to-highlight interaction. Signals constructed over a click stream feed data transforms that toggle values in the `selected_pts` data source. An intensional predicate tests for the shift key. If it is not pressed, the data source is cleared prior to inserting the clicked values. A production rule sets the fill color of selected points using an extensional predicate.

Similarly, Fig. 3.6 demonstrates the Reactive Vega JSON necessary to enable brush selections. Signals are registered to capture the start and end positions of the brush, by default `mousedown` and `[mousedown, mouseup] > mousemove`, respectively. Scale inversions are invoked to calculate the data extents of the brush, which are used to define an intensional predicate to express the brushed data range. As before, the predicate is used within a production rule to set the fill color of selected points.




```
scales: [{name: "sx", type: "linear", ...}],

signals: [{
  name: "drag_start", value: 0,
  on: [{
    events: "mousedown",
    update: "x()"
  }]
}, {
  name: "drag_end", value: 0,
  on: [{
    events: "[mousedown, mouseup] > mousemove",
    update: "x()"
  }]
}, {
  name: "range_start", value: 0,
  on: [{
    events: {signal: "drag_start"},
    update: "invert('sx', drag_start)"
  }]
}, {
  name: "range_end", value: 0,
  on: [{
    events: {signal: "drag_end"},
    update: "invert('sx', drag_end)"
  }]
}],

marks: [{
  type: "symbol",
  encode: {
    update: {
      fill: [{
        test: "inrange(datum.Horsepower, range_start, range_end)",
        scale: "color", field: "Origin"
      }, {
        value: "grey"
      }]
```

Figure 3.6: A JSON snippet for one-dimensional brushing. Signals over drag events are inverted through the x-scale to construct a data query over the `Horsepower` field.

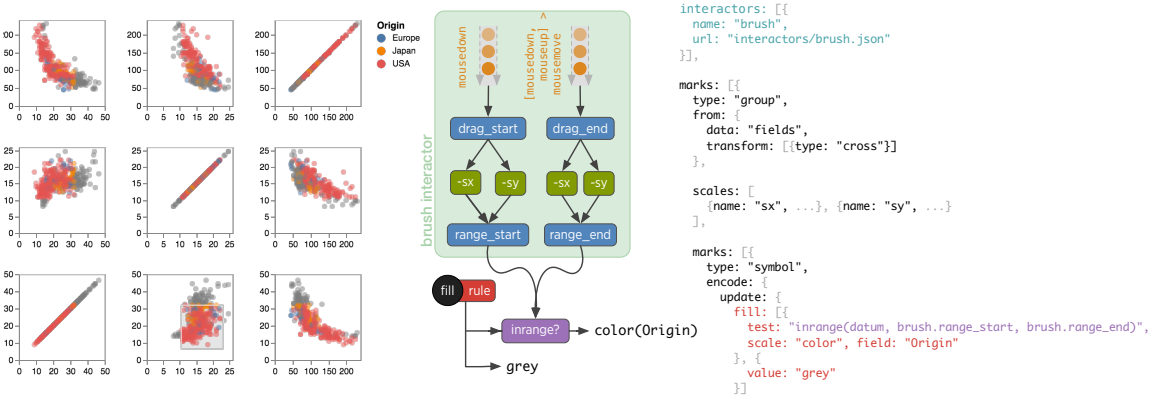
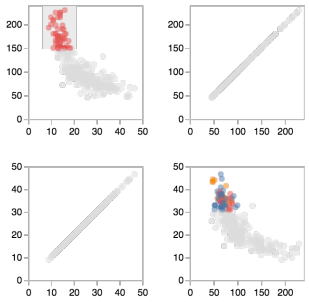


Figure 3.7: We can extract the brushing interaction from Fig. 3.6 into a standalone interactor, and reapply it to a scatterplot matrix to perform brushing & linking.

3.2.2 Connect: Brushing & Linking



We can extract the previous interaction into a standalone "brushing" interactor, and then apply it to brush & link a scatterplot matrix as shown in Fig. 3.7. Each cell of the matrix is an instance of a group mark with its own coordinate space. The plotting symbol and spatial scale functions are defined within these groups. Had the interactor's predicates defined selections over pixel space, the production rule would highlight points that fall along the same horizontal and vertical pixel regions — for example, as shown in the inset figure, brushing over red points would highlight blue points in the other cells. Instead, the interactor uses scale inversions to lift the predicate to the data domain, and the production rule correctly links brushed points across scatterplots.

3.2.3 Abstract/Elaborate: Overview + Detail

With our brush interactor, we can also create the overview + detail visualization in Fig. 3.4. In this case, brushing is restricted to the horizontal dimension by overriding the `height` property of the interactor's mark, and ignoring the vertical range predicates it populates.

We use the horizontal range predicate with a filter transformation, to filter points for display in the detail plot. As a user brushes, signals update the range predicate, which in turn filters points in the data source, updates scale functions and re-renders the detail view.

3.2.4 Explore & Encode: Panning & Zooming

Figure 3.8 shows pan and zoom interactions for a scatter plot. By default, scale functions calculate their domain automatically from a data source. For this interaction, however, we must parameterize the domain using reactive signals. For panning, a `start` signal captures an initial `(x,y)` position on `mousedown`, and subsequent pan signals calculate a delta on drag (`[mousedown, mouseup] > mousemove`). This delta is used to offset the scale domains. Similarly, when `wheel` events occur, a `zoom` signal applies a scale factor to the domains depending on the zoom direction.

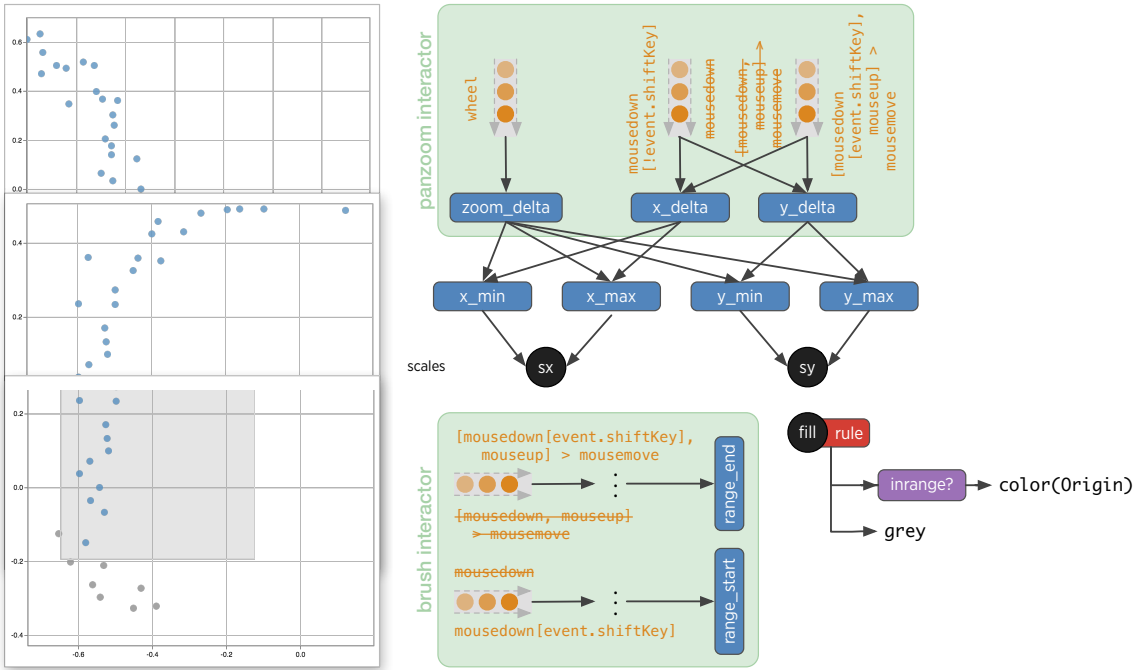


Figure 3.8: Panning & zooming a scatterplot. Brushing is accretively added with the brush interactor (Figs. 3.6 and 3.7), and conflicts are resolved by rebinding event streams (indicated with strikethroughs).

If we were to also add a brushing interaction to this visualization, a naïve application would yield a conflicting interaction: on drag, both panning and brushing would occur. One option to resolve this conflict is to begin brushing only when the shift key is pressed. If we try combining these interactions using D3 [17], which offers brushing and panning as part of its interactor typology, specifying this resolution scheme can be onerous. Additional callbacks must be registered that either instantiate or destroy a particular interaction depending on the state of the shift key.

With Reactive Vega, the brush and pan signals can be rebound without modifying the interactor definitions. Instead, we provide alternate source event streams when instantiating the interactor — `mousedown[event.shiftKey]` for brushing, and `mousedown[!event.shiftKey]` for panning.

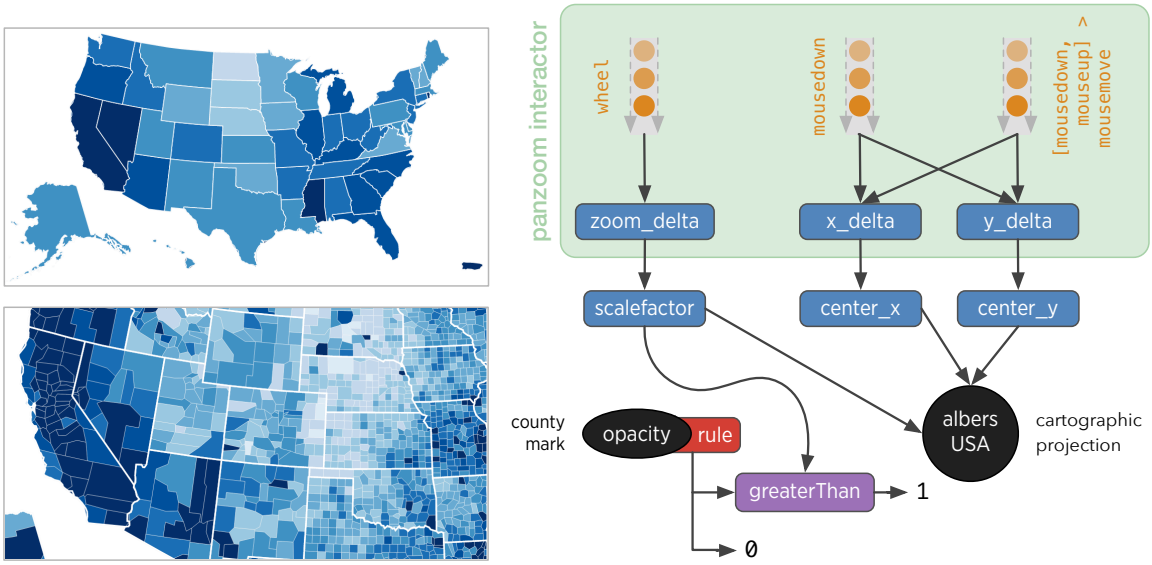


Figure 3.9: Extracting the pan & zoom interaction from Fig. 3.8 into an interactor, and repurposing it to perform *semantic* zooming on a cartographic map. Initially, a choropleth of state-level unemployment in the United States is shown. Zooming past a threshold, states break up into counties, and show county-level unemployment instead.

Moreover, by extracting panning & zooming into a standalone interactor, we can repurpose the behavior to instead trigger semantic zooming [78], an *encoding* interaction technique shown in Fig. 3.9. At the top-level, the visualization shows a choropleth map of state-level unemployment. After crossing a specified zoom threshold, states subdivide to show a choropleth map of country-level unemployment. Here, the pan signals drive the geographic projection's translation and the zoom signals drive the projection's scale parameter. By default, both maps are drawn with states overlaying counties. A production rule uses a predicate to test whether the zoom signal is above a specified threshold; if it is, the state-level map is rendered transparently, displaying the county-level map underneath it.

3.2.5 Reconfigure: Index Chart

Figure 3.10 shows an index chart: a line chart that interactively normalizes time series to show percentage change based on the current index point. To calculate the index point, a signal captures the x coordinate of `mousemove`events, and drives it through a scale inversion. As it is a quantitative scale, scale inversion produces a value from a continuous

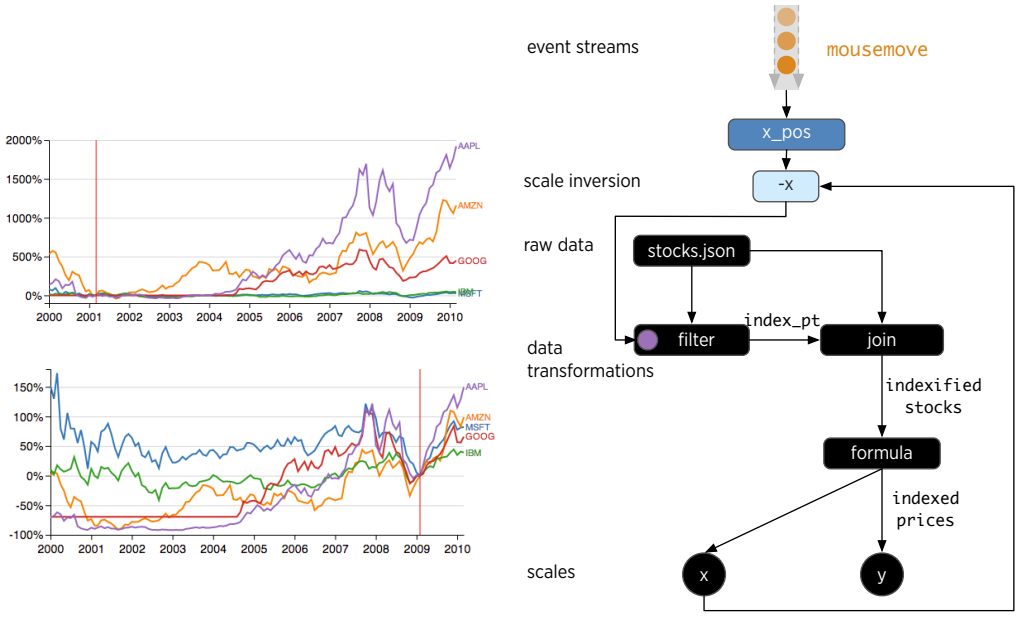
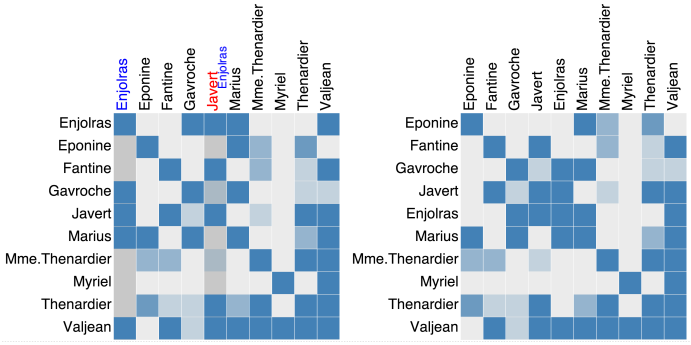


Figure 3.10: An index chart shows the percentage changes for time-series data. The index point (red vertical line) is determined by the x position of a `mousemove` signal, which filters points using a predicate.

domain (i.e., any date/time from Jan 1, 2000–Dec 31, 2010). However, our dataset only contains stock prices for the start of every month. We use a predicate to "snap to" the closest value for each time series, and use this as our index point. Using Vega's data transformations, we join the index point against the original data set and normalize the data values. Scale functions are defined in terms of the normalized data.

3.2.6 Reconfigure: Reordering Columns of a Matrix



The figure to the left shows a co-occurrence matrix of Les Misérables characters. To reorder the columns of the matrix, we first construct a data source that computes the sort order of characters and initialize it to an alphabetical ordering. A signal on `@col_label:mousedown` captures the source column to be reordered, while a signal on `[@col_label:mousedown, mouseup] > mousemove` updates the target column location. On `mouseup`, the data source is updated to swap the two sorting indices.

3.2.7 Filter: Control Widgets

Figure 3.11 shows the Job Voyager [51] visualization with control widgets to filter the visualized data. A textbox allows users to enter search terms to filter job titles, while the radio buttons allow users to filter by gender. We bind signals to the value of these control widgets, and then construct predicates attached to filter data transformations. For the textbox signal, a regular expression tests terms against job titles, while an equality test filters by gender based on the radio button signal. This example illustrates how external widgets can easily be bound to our reactive model.

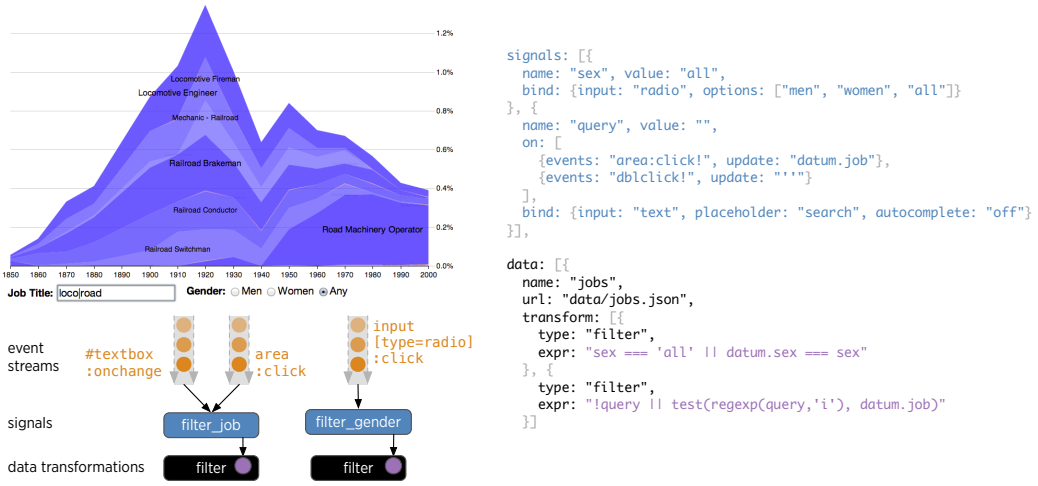


Figure 3.11: The job voyager can be filtered using signals bound to control widgets. The textbox pattern matches against job titles while radio buttons filter by gender.

3.2.8 DimpVis: Touch Navigation with Time-Series Data

DimpVis [60] is a recently introduced interaction technique that allows direct manipulation navigation of time-series data. Starting with a scatterplot depicting data at a particular time slice, users can touch plotted points to reveal a "hint path": a line graph that displays the trajectory of the selected element over time. Dragging the selected point along this path triggers temporal navigation, with the rest of the points updating to reflect the new time. In evaluation studies, users reported feeling more engaged when exploring their data using DimpVis [60].

As shown in Fig. 3.12, we can recreate this technique with Reactive Vega's declarative interaction primitives and the GapMinder country-fertility-life-expectancy dataset used by the original. Input data is passed through a `Window` transform, such that every tuple contains references to the tuples that come before and after it in time, and filtered to remove triplets that span multiple countries. Signals constructed over mouse and touch events capture the selected point, and downstream signals calculate distances between the user's current position and the previous and next points. A scalar projection over these distances gives us scoring functions that determine whether the user is moving forwards or backwards in time. Scores feed a signal that is used in a derived data source to calculate new

interpolated properties for the remaining points in the dataset. These interpolated properties determine the position of plotted points, producing smooth transitions as the user drags back-and-forth. To draw the hint map, a derived data source filters data tuples for the selected country across all years.

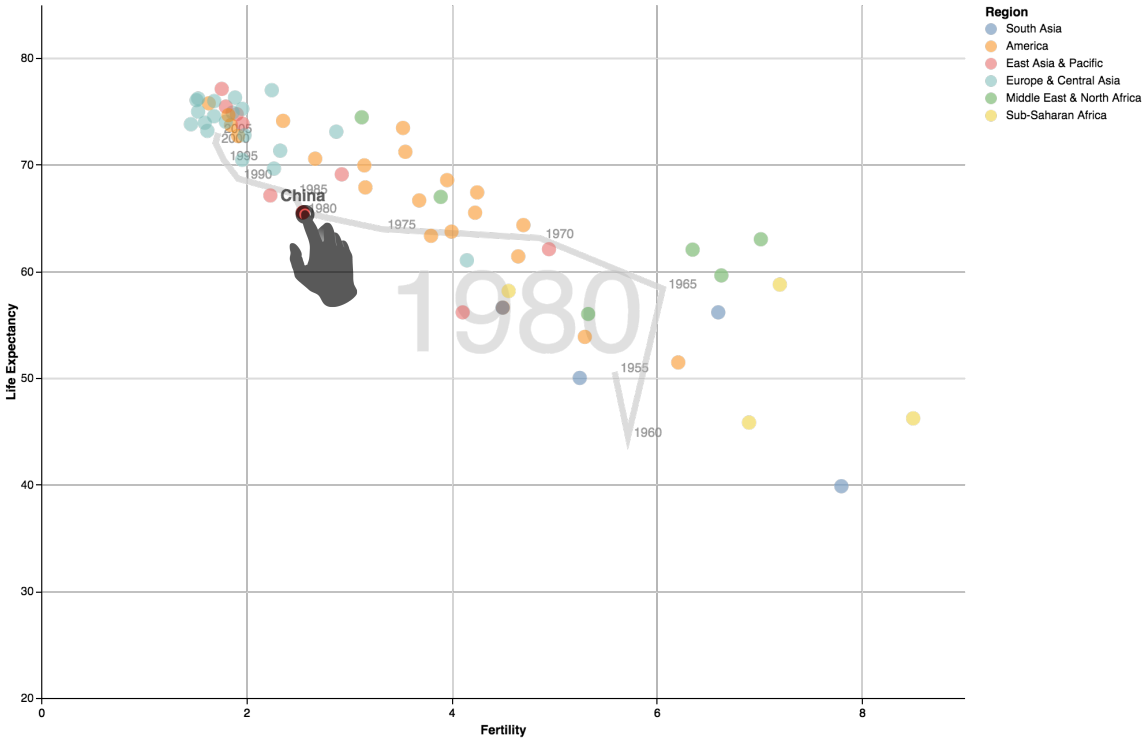


Figure 3.12: DimpVis [60], touch-based navigation of time-series data recreated with Reactive Vega.

3.2.9 Reusable Touch Interaction Abstractions

With the proliferation of touch-enabled devices, particularly smartphones and tablets, supporting touch-based interaction has become an increasingly important part of interactive visualization design. However, HTML5 only provides a low-level API for touch events, with three event types broadly supported — `touchstart`, `touchmove`, and `touchend`. On multitouch devices these events contain an array of touch points. The application developer is responsible for the bookkeeping involved with tracking multiple points across interactions, a cumbersome and difficult process.

Declarative interaction design enables us to abstract low-level details away, building reusable interactors that expose higher-level, semantic events as signals instead. For example, an interactor could define signals that perform the necessary logic for common multitouch gestures. Once such an interactor is included in a host visualization, the visualization designer can then safely ignore lower-level events, and instead build interactions driven by the interactor's signals — using `twotouchmove` and `pinchDelta` signals to drive panning and zooming behaviors, for instance.

3.3 Discussion: Cognitive Dimensions of Notation

The previous section demonstrated the expressivity of Reactive Vega's model of declarative interaction design. Here, we evaluate it from the perspective of a visualization designer using the Cognitive Dimensions of Notation [15], a set of heuristics for evaluating the efficacy of notational systems such as programming languages and interfaces. Of the 14 dimensions, we evaluate Reactive Vega against a relevant subset and compare it against current common practice: declarative specification of visual encodings using D3 [17] and imperative event handling callbacks for interaction.

Abstractions (types and availability of abstraction mechanisms) and **Viscosity** (resistance to change). Streams and signals abstract the low-level events that trigger interactions and decouple them from downstream logic. This approach can facilitate rapid iteration: the result of an interaction can be designed (for example, highlighting points), and then a variety of different event triggers can be prototyped by simply rebinding the appropriate signals. As our examples demonstrate, rebinding signals reduces the burden for resolving conflicting interactions or retargeting to different platforms. By comparison, iterating with event callbacks can be more difficult. A particular sequence of events may require a specific ordering of callbacks, and coordinating the visualization state across these functions falls to the designer.

Premature Commitment (constraints on the order of doing things). Abstracting streams into signals does impose a premature commitment. Users must define them before they are able to use any lower-level events to trigger state changes. This requirement could be

relaxed: users could use event streams inline, for example within a predicate or production rule. However, we believe such inline references are a poor design pattern as they make an interaction technique dependent on a specific set of events — a common problem with existing interactor typologies. Moreover, reuse would be hampered as it would become more difficult to resolve conflicting interactions (e.g., brushing and panning) or retarget techniques across input modalities.

Hidden Dependencies (important links between entities are not visible). Signals do introduce hidden dependencies as they obscure which input events trigger particular changes to the state of the visualization. However, we believe that the gains in viscosity outweigh the complexities of these hidden dependencies, particularly as the latter can be ameliorated by naming signals appropriately. Furthermore, as our code examples illustrate, all the factors that directly affect a particular state are captured within a single Reactive Vega specification. For example, a signal definition specifies all events or signals that may affect its value; similarly, a visual property may use a rule which enumerates all the values it may take. With D3, the visual encoding specification may not completely define all states. Instead, the user must trace a flow through event callbacks, a process further exacerbated by an unpredictable execution order. The user is forced to coordinate interleaved callbacks, introducing **hard mental operations** and **error-proneness**.

Consistency (similar semantics are expressed in similar syntactic forms). Our interaction model is best suited for systems that declaratively specify visual encodings, and would feel foreign in imperative systems. However, given widespread adoption of D3, and Vega's increasing integration in systems [70, 85, 108] we believe this is not a significant liability. By contrast, registering event callbacks on D3 visualizations breaks consistency: visual design is declarative while interaction design is imperative. Users must think in two notational systems, and the underlying implementation and execution concerns are exposed.

Visibility (ability to view components easily). One of the primary advantages of using D3, and registering event callbacks, is being able to debug code directly within a web browser [17]: the generated visualization can be inspected, while the JavaScript console can be used to interactively debug event callbacks. Reusing such existing scaffolding is more difficult

with Reactive Vega as its runtime, which parses and renders a specification, introduces its own stack of abstractions. However, we believe this gap offers a promising avenue for future work in new development environments that can leverage Vega's reactive semantics. For instance, initial work by Hoffswell et al. [52] has devised a "time-travelling" debugger for Reactive Vega specifications. In particular, the authors note that signals are a critical abstraction barrier, providing a meaningful entry-point into an interaction specification. To construct a similar debugger for imperative event handling callbacks would require complex static analysis to identify and surface relevant program state.

In summary, Reactive Vega introduces hidden dependencies between state changes and their triggering input events and, without additional tooling, decreases visibility into runtime execution. However, we believe these drawbacks are outweighed by the increase in specification consistency between visual encodings and interaction, and a decrease in viscosity, allowing users to iterate more quickly over interaction design.

3.4 Summary

In this chapter, we demonstrate that the advantages of declarative specification extend to interaction design as well. With event streams and signals, users need only describe the relationship between input events and interactive state respectively. Reactive semantics ensure that this state no longer needs to be manually maintained by users, but is automatically updated when new events occur. Moreover, signals provide a powerful abstraction barrier that simplifies retargeting and has spurred development of a new "time-traveling" debugger [52]. Reusing and repurposing interactive behaviors is achieved by passing predicates through scale inversions, and extracting specifications to standalone interactors.

4 A Streaming Dataflow Architecture

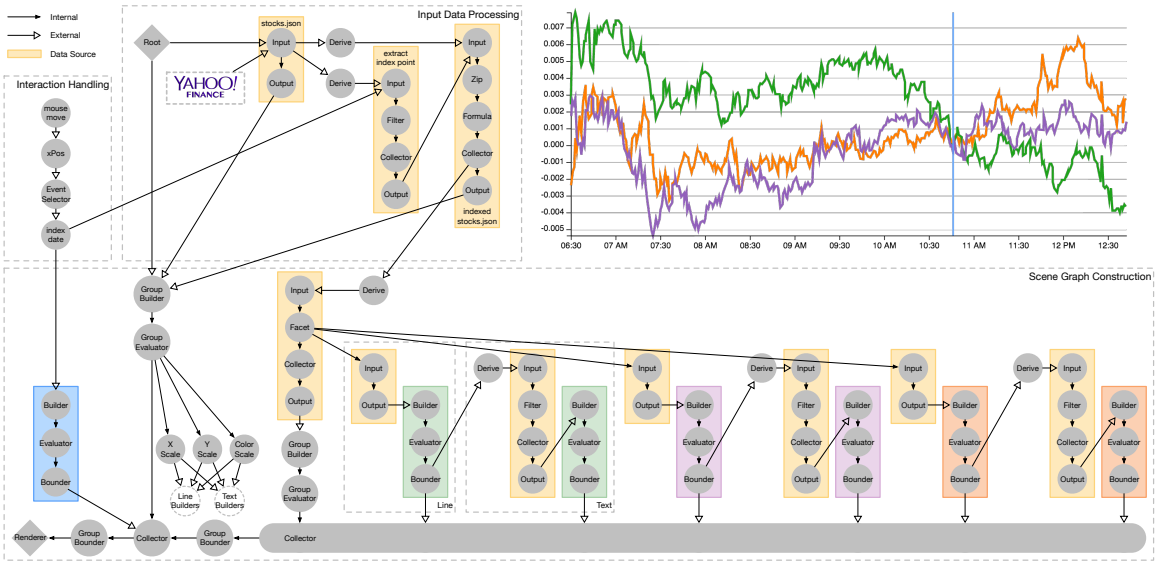


Figure 4.1: The Reactive Vega dataflow graph for a interactive index chart of streaming financial data. As data arrives from Yahoo! Finance, or as a user moves their mouse cursor across the chart, an update cycle propagates through the graph and triggers an efficient update and re-render of the visualization.

While the previous chapter describes the design of Reactive Vega's declarative interaction design model, we now turn to the system architecture needed to support it. Our architecture design is motivated by four primary goals:

1. **A Unified Data Model**. Existing reactive visualization toolkits (e.g., Model.js [57]) feature fragmented architectures where only interaction events are modeled as time-varying. Other input datasets remain static and batch-processed. This artificial disconnect restricts expressivity and can result in wasteful computation. For example, interaction events that manipulate only a subset of input tuples may trigger recomputation over the entire dataset. In contrast, Reactive Vega features a unified

data model in which input data, scene graph elements, and interaction events are all treated as first-class streaming data sources.

2. **Streaming Relational Data**. Modeling input relational data with Event-Driven Functional Reactive Programming (E-FRP) [103] semantics alone does not supply sufficient granularity for targeted recomputation. As E-FRP semantics consider only time-varying scalar values, operators would observe an entire relation as having changed and so would need to reprocess all tuples. Instead, Reactive Vega integrates techniques from streaming databases [1, 2, 6, 7, 24] alongside E-FRP, including tracking state at the tuple-level and only propagating modified tuples.

3. **Streaming Nested Data**. Interactive visualizations, particularly those involving small multiples, often require hierarchical structures. Processing such data poses an additional challenge not faced by prior reactive or streaming database systems. To support streaming nested data, Reactive Vega's dataflow graph rewrites itself in a data-driven fashion at runtime: new branches are extended, or existing branches pruned, in response to observed hierarchies. Each dataflow branch models its corresponding part of the hierarchy as a standard relation, enabling operators to remain agnostic to higher-level structure.

4. **Interactive Performance**. Reactive Vega performs both compile- and run-time optimizations to increase throughput and reduce memory footprint, including tracking metadata to prune unnecessary computation, and optimizing scheduling by inlining linear chains of operators. We conduct benchmark studies of streaming and interactive visualizations and find that Reactive Vega meets or exceeds the performance of both D3 and the original, unreactive Vega system.

Reactive Vega is implemented in the JavaScript programming language, and is intended to run either in a web browser or server-side using Node.js. By default, Reactive Vega renders to an HTML5 Canvas element; however, it also supports Scalable Vector Graphics (SVG) and server-side image rendering.

4.1 The Dataflow Graph Design

Dataflow operators are instantiated and connected by the Reactive Vega *parser*, which traverses a declarative specification containing definitions for input datasets, visual encoding rules, and interaction primitives as described in Chapter 3. When data tuples are observed, or when interaction events occur, they are propagated (or "*pulsed*") through the graph with each operator being evaluated in turn. Propagation ends at the renderer.

4.1.1 Data, Interaction, and Scene Graph Operators

Reactive Vega's dataflow operators fall into one of three categories: input data processing, interaction handling, or scene graph construction.

Processing Input Data

Reactive Vega parses each dataset definition and constructs a corresponding branch in the dataflow graph. These branches comprise input and output nodes connected by a pipeline of data transformation operators. Input nodes receive raw tuples as a linear stream (tree and graph structures are supported via parent-child or neighbor pointers, respectively). Upon data source updates, tuples are flagged as either *added*, *modified*, or *removed*, and each tuple is given a unique identifier. Data transformation operators use this metadata to perform targeted computation and, in the process, may derive new tuples from existing ones. Derived tuples retain access to their "parent" via prototypal inheritance — operators need not propagate unrelated upstream changes.

Some operators require additional inspection of tuple state. Consider an aggregate operator that calculates running statistics over a dataset (e.g., mean and variance). When the operator observes added or removed tuples, the statistics can be updated based on the current tuple values. With modified tuples, the previous value must be subtracted from the calculation and the new value added. Correspondingly, tuples include a `previous` property. Writes to a tuple attribute are done through a setter function that copies current values to the `previous` object.

Handling Interaction

Reactive Vega instantiates an event listener node for each low-level event type required by the visualization (e.g., `mousedown` or `touchstart`). These nodes are directly connected to dependent signals as specified by event selectors [88]. In the case of ordered selectors (e.g., "drag" events specified by `[mousedown, mouseup] > mousemove`), each constituent event is connected to an automatically created anonymous signal; an additional anonymous signal connects them to serve as a gatekeeper, and only propagates the final signal value when appropriate. Individual signals can be dependent on multiple event nodes and/or other signals, and value propagation follows E-FRP's two-phase update (§4.1.3).

Constructing the Scene Graph

To construct the scene graph, Reactive Vega follows a process akin to the Protovis bind-build-evaluate pipeline [47]. When a declarative specification is parsed, Reactive Vega traverses the mark hierarchy to *bind* property definitions: property sets are compiled into encoding functions and stored with the specification. At run-time, *build* and *evaluate* operators are created for each bound mark. The build operator performs a data join [17] to generate one scene graph element (or "mark") per tuple in the backing dataset, and the evaluate operator runs the appropriate encoding functions. A downstream *bounds* operator calculates the bounding boxes of generated marks. For a nested scene graph to be rendered correctly, the order of operations is critical: parent marks must be built and encoded before their children, but the bounds of the children must be calculated before their parents. The resultant scene graph, as seen in Fig. 4.2, exhibits an alternating structure, with individual mark elements grouped under a sentinel mark specification node.

Scene graph elements are also modeled as data tuples and can serve as the input data for downstream visual encoding primitives. This establishes a *reactive geometry* that accelerates common layout tasks, such as label positioning, and expands the expressiveness of the specification language. As marks can be run through subsequent data transformations, higher-level layout algorithms (e.g., those that require a pre-computed initial layout [112]) are now supported in a fully declarative fashion.

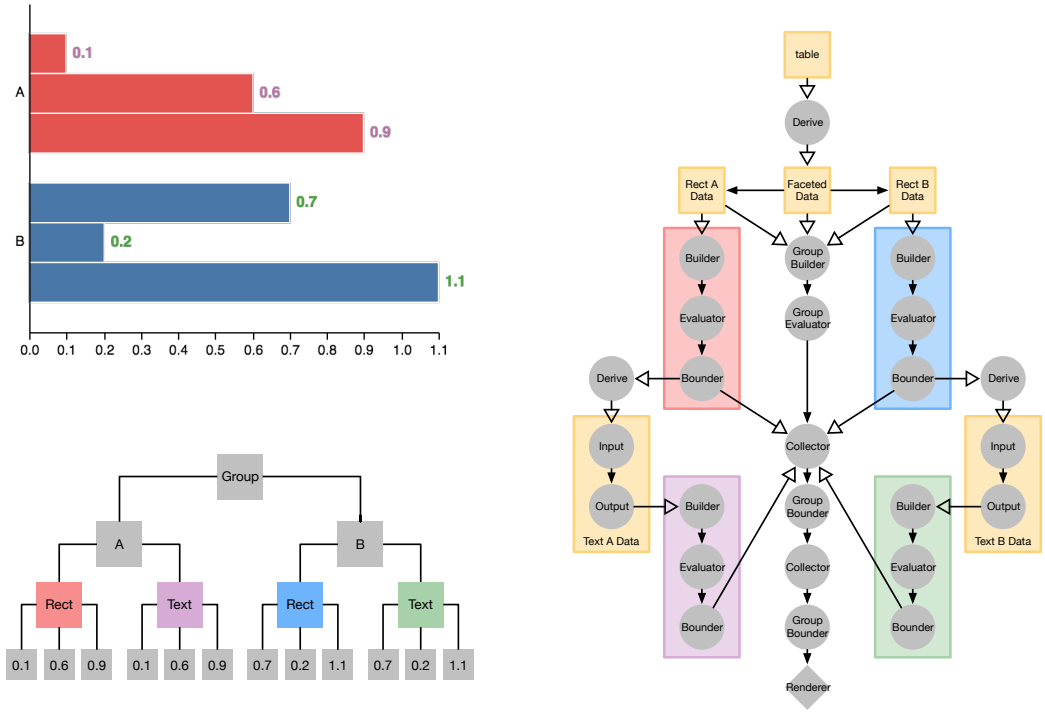


Figure 4.2: A grouped bar chart (top), with the underlying scene graph (bottom), and corresponding portion of the dataflow graph (right).

4.1.2 Changesets and Materialization

All data does not flow through the system at all times. Instead, operators receive and transmit *changesets*. A changeset consists of observed tuples, new signal values, and updates to other dependencies that have transpired since the last render event. The propagation of a changeset begins in response to streaming tuples or user interaction. The corresponding input node creates a fresh changeset, and populates it with the detected update. As the changeset flows through the graph, operators use it to perform targeted recomputation, and may augment it in a variety of ways. For example, a `Filter` operator might remove tuples from a changeset if they do not meet the filter predicate, or may mark modified tuples as added if they previously had been filtered. A Cartesian product operator, on the other hand, replaces incoming tuples with the cross-product with another data stream.

Some operators, however, require a complete dataset. For example, a windowed-join requires access to all tuples in the current windows of the joined data sources. For such scenarios, special *collector* operators (akin to *views* [2] or *synopses* [6] in streaming databases) materialize the data currently in a branch. In order to mitigate the associated time and memory expenses, Reactive Vega automatically shares collectors between dependent operators. Upon instantiation, such operators must be annotated as requiring a collector; at run-time they can then request a complete dataset from the scheduler.

Finally, if animated transitions are specified, a changeset contains an interpolation queue to which mark evaluators add generated mark instances; the interpolators are then run when the changeset is evaluated by the renderer.

4.1.3 Coordinating Changeset Propagation

A centralized dataflow graph scheduler is responsible for dispatching changesets to appropriate operators. The scheduler ensures that changeset propagation occurs in topological order so that an operator is only evaluated after all of its dependencies are up-to-date. This schedule prevents wasteful intermediary computation or momentary inconsistencies, known as *glitches* [29]. Centralizing this responsibility, rather than delegating it to operators, enables more aggressive pruning of unnecessary computation as described in §4.2.2. The scheduler has access to the full graph structure and, thus, more insight into the state of individual operators and propagation progress.

When an interaction event occurs, however, an initial non-topological update of signals is performed. Dependent signals are reevaluated according to their specification order. As a result, signals may use prior computed values of their dependencies, which will subsequently be updated. This process mimics E-FRP's two-phase update [103], and is necessary to enable expressive signal composition. Once all necessary signals have been reevaluated, a changeset with the new signal values is propagated to the rest of the dataflow graph.

4.1.4 Pushing Internal and Pulling External Changesets

Two types of edges connect operators in the dataflow graph. The first connects operators that work with the same data; for example a pipeline of data transformation operators for the same data source, or a mark's build and evaluate operators. Changesets are pushed along these edges, and operators use and augment them directly.

The second type of edge connects operators with external dependencies (e.g., other data sources, signals, and scale functions). As these edges connect disparate data spaces, they cannot directly connect operators with their dependencies. To do so would result in operators performing computation over mismatched data types. Instead, external dependencies are connected to their dependents' nearest upstream `Collector` node, and changesets that flow along these edges are flagged as *reflow changesets*. When a `Collector` receives a reflow changeset, it propagates its tuples forward, flagging them as modified. The dependents now receive correct input data.

Reflow changes also flow along edges that connect signals to other signals. However, as these edges exist in scalar data space, `Collector` nodes are not needed.

This hybrid push/pull system enables a complex web of interdependent operators while reducing the complexity of individual elements. For example, regardless of whether a signal parameterizes data transforms or visual encoding primitives, it simply needs to output a reflow changeset. Without such a system in place, the signal would instead have to construct a different changeset for each dependency edge it was a part of, and determine the correct dataset to supply. Figures 4.1 to 4.3 use filled and unfilled arrows for internal and external connections, respectively.

4.1.5 Dynamically Restructuring the Graph

To support streaming nested data structures, operators can dynamically restructure the graph at runtime by extending new branches, or pruning existing ones, based on observed data. These dataflow branches model their corresponding hierarchies as standard relations, thereby enabling subsequent operators to remain agnostic to higher-level structure.

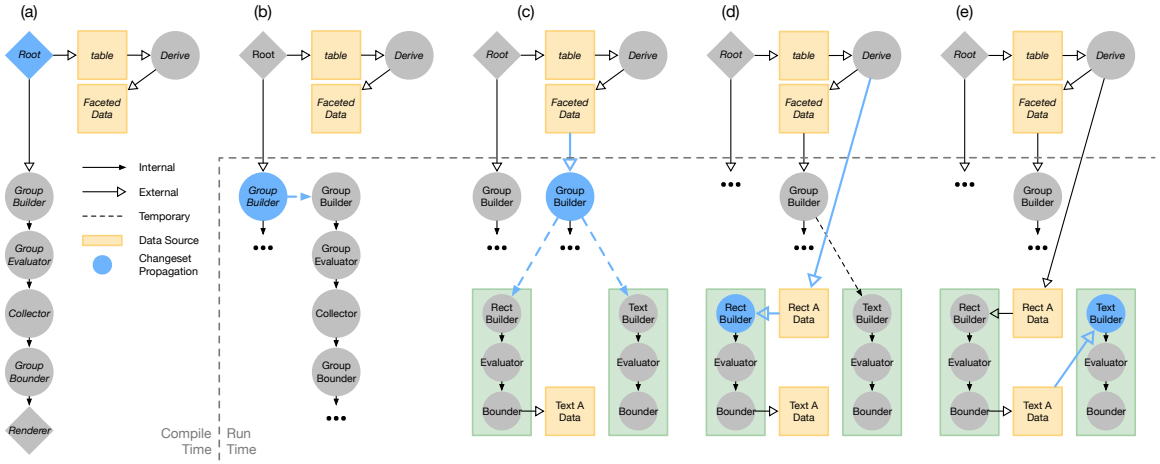


Figure 4.3: Dataflow operators responsible for scene graph construction are dynamically instantiated at run-time, a process that results in Fig. 4.2. (a) At compile-time, a branch corresponding to the root scene graph node is instantiated. (b-c) As the changeset (blue) propagates through nodes, group-mark builders instantiate builders for their children. Parent and child builders are temporarily connected (dotted lines) to ensure children are built in the same cycle. (d-e) When the changeset propagates to the children, the temporary connection is replaced with a connection to the mark's backing data source (also blue).

For example, a `Facet` operator partitions tuples by key fields; each partition then propagates down a unique, dynamically-constructed dataflow branch, which can include other operators such as `Filter` or `Sort`.

In order to maintain interactive performance, new branches are queued for evaluation as part of the same cycle they were created in. To ensure changeset propagation continues to occur in topological order, operators are given a *rank* upon instantiation to uniquely identify their place in the ordering. When new edges are added to the dataflow graph, the ranks are updated such that an operator's rank is always greater than those of its dependencies. When the scheduler queues operators for propagation, it also stores the ranks it observes. Before propagating a changeset to an operator, the scheduler compares the operator's current rank to the stored rank. If the ranks match, the operator is evaluated; if the ranks do not match, the graph was restructured and the scheduler requeues the operator.

Scene graph operators are the most common source of graph restructuring, as building a nested scene graph is entirely data-driven. Dataflow branches for child marks (consisting of build-evaluate-bound chains) cannot be instantiated until the parent mark instances

have been generated. As a result, only a single branch, corresponding to the root node of the scene graph, is constructed at compile-time. As data streams through the graph, or as interaction events occur, additional branches are created to build and encode corresponding nested marks. To ensure their marks are rendered in the same propagation cycle, new branches are temporarily connected to their parents. These connections are subsequently removed so that children marks will only be rebuilt and re-encoded when their backing data source updates. Figure 4.3 provides a step-by-step illustration of how scene graph operators are constructed during a propagation cycle for the grouped bar chart in Fig. 4.2.

4.2 Performance Optimizations

Declarative language runtimes can transparently optimize performance [47] and Reactive Vega uses several strategies to increase throughput and reduce memory usage. In this section, we describe these strategies and evaluate their effect through benchmark studies. Each benchmark was run with datasets sized at N = 100, 1,000, 10,000, and 100,000 tuples. For ecological validity, benchmarks were run with Google Chrome 42 (64-bit) and, to prevent confounds with browser-based just-in-time (JIT) optimizations, each iteration was run in a fresh instance. All tests were conducted on a MacBook Pro running Mac OS X 10.10.2, with a quad-core 2.5GHz Intel Core i7 processor and 16GB of 1600 MHz DDR3 RAM.

4.2.1 On-Demand Tuple Revision Tracking

Some operators (e.g., statistical aggregates) require both a tuple's current and previous values. Tracking prior values can affect both running time and memory consumption. One strategy to minimize this cost is to track tuple revisions only when necessary. Operators must declare their need for prior values. Then, when tuples are ingested, their previous values are only tracked if the scheduler determines that they will flow through an operator that requires revision tracking.

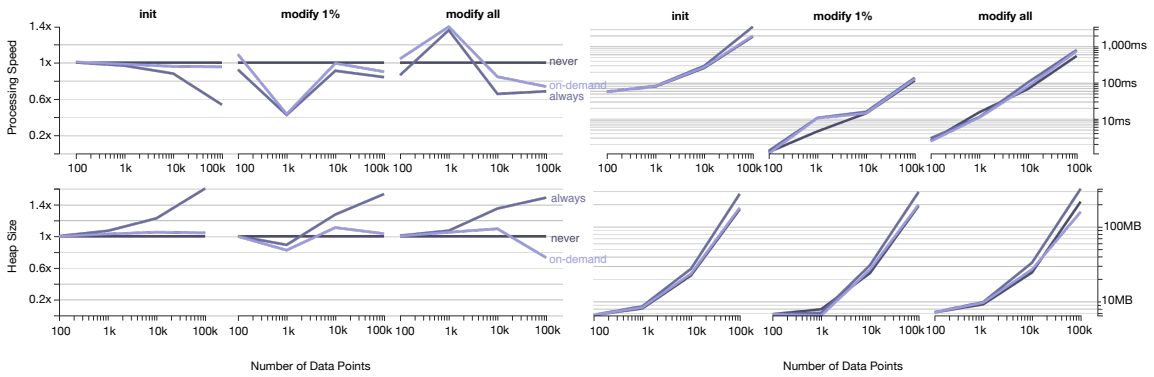


Figure 4.4: Effects of tuple revision optimizations on average processing speed (top) and memory footprint (bottom). Left-hand figures show relative changes using no-tracking as a baseline (closer to 1.0 are better), and right-hand figures show the absolute values on a $\log_{10}$ scale (lower is better).

We ran a benchmark comparing three conditions: always track revisions, never track revisions, and on-demand tracking. Although the "never" condition produces incorrect results, it provides a lower-bound for performance. We measured the system's throughput as well as memory allocated when initializing a scatterplot specification, and after modifying either 1% or 100% of input tuples. The scatterplot features two symbol marks fed by two distinct dataflows, A and B. Both branches ingest the same set of tuples, and include operators that derive new attributes. However, B includes additional aggregation operators that require revision tracking.

The results are shown in Figure 4.4, with the effects of revision tracking most salient at larger dataset sizes. Always tracking revisions can require 20-40% more memory, and can take up to 50% longer to initialize a visualization due to object instantiation overhead for storing previous values. Our on-demand strategy effectively reduces these costs, requiring only 5-10% more memory and taking 5% longer to initialize than the "never" condition.

4.2.2 Pruning Unnecessary Recomputation

By centralizing responsibility for operator scheduling and changeset dispatch, we can aggressively prune unnecessary recomputation. The dataflow graph scheduler knows the current state of the propagation, and dependency requirements for each queued operator, allowing us to perform two types of optimizations:

1. *Pruning multiple reflows of the same branch.* As the scheduler ensures a topological propagation ordering, a branch can be safely pruned for the current propagation if it has already been reflowed.

2. *Skipping unchanged operators.* Operators identify their dependencies — including signals, data fields, and scale functions — and changesets maintain a tally of updated dependencies as they flow through the graph. The scheduler skips evaluation of an individual operator if it is not responsible for deriving new tuples, or if a changeset contains only modified tuples and no dependencies have been updated. Downstream operators are still queued for propagation.

To measure the impact of these optimizations, we created a grouped bar chart with five data transformation operators: `Derive(signal1)` → `Fold` → `Derive(signal2)` → `Filter(signal2)` → `Facet`. We then benchmarked the effect of four conditions (processing all recomputations, pruning multiple reflows only, skipping unchanged operators only, and applying both optimizations) across four tasks (initializing the visualization, updating each signal in turn, and updating both signals together).

Results are shown in Figure 4.5. Preventing multiple reflows is the most effective strategy, increasing throughput 1.4 times on average. Skipping unchanged operators sees little benefit by itself as, in our benchmark setup, only the two operators following a fold are skipped when changing `signal1`, and only the first derivation operator is skipped when changing `signal2`. When the two strategies are combined, however, we see a 1.6x increase in performance. This result was consistent across multiple benchmark trials. After

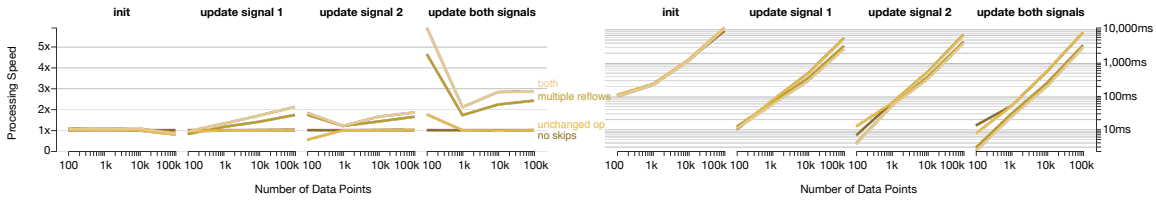


Figure 4.5: The effects of pruning unnecessary computation on average processing speed. (a) A relative difference between conditions (higher is better). (b) Absolute values for time taken, plotted on a $\log_{10}$ scale (lower is better).

careful hand-verification to ensure no additional nodes were erroneously skipped, we hypothesize that the JavaScript runtime is able to perform just-in-time optimizations that it is unable to apply to the other conditions.

4.2.3 Inlining Sequential Operators

To propagate changesets through the dataflow graph, the scheduler adds operators to a priority queue, backed by a binary heap sorted in topological order. This incurs an $O(logN)$ cost for enqueueing and dequeueing operators, which can be assessed multiple times per operator if the graph is dynamically restructured. However, branching only occurs when operators register dependencies, and dependencies are only connected to `Collector` nodes. As a result, much of the dataflow graph comprises linear paths. This is particularly true for scene graph operators, which are grouped into hundreds (or even thousands) of independent mark build-evaluate-bound branches.

We explore the effect of inline evaluation of linear branches, whereby operators indicate that their neighbors can be called directly rather than queued for evaluation. The scheduler remains responsible for propagating the changeset, and thus can continue to apply the optimizations previously discussed. Although inline evaluation can be applied in a general fashion by coalescing linear branches into "super nodes," for simplicity we only evaluate inlining of scene graph operators here. Mark builders directly call evaluators and bounders, and group mark builders directly call new child mark builders rather than forming a temporary connection.

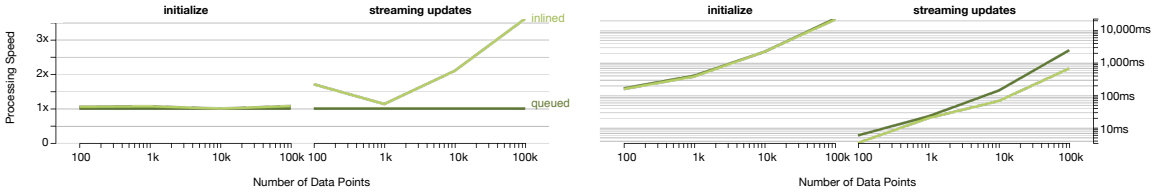


Figure 4.6: The effects of inlining sequential operators on average processing speed. (a) A relative difference between conditions (higher is better). (b) Absolute values for time taken, plotted on a $\log_{10}$ scale (lower is better).

Figure 4.6 shows the results of this optimization applied to a parallel coordinates plot. The plot uses a nested scene graph in which each line segment is built by a dedicated build-evaluate-bound branch. As we can see, inlining does not have much impact on the initialization time. This is unsurprising, as the largest initialization cost is due to unavoidable graph restructuring. However, inlining improves streaming operations by a 1.9x factor on average. As streaming updates only propagate down specific branches of the dataflow graph, inline evaluation results in at least 4 fewer queuing operations by the scheduler.

4.3 Comparative Performance Benchmarks

To evaluate the performance of Reactive Vega against D3 [17] and the origin, non-reactive Vega system (v1.5.0), we use the same setup described in the previous section.

4.3.1 Streaming Visualizations

Figure 4.7 shows the average performance of uninteractive streaming scatter plots, parallel coordinates plots, and trellis plots. We first measured the average time to initially parse and render the visualizations. To gauge streaming performance, we next measured the average time taken to update and re-render upon adding, modifying, or removing 1% of tuples. We ran 10 trials per dataset, sized 100–100,000 tuples.

Reactive Vega has the greatest effect with the parallel coordinates plot, displaying 2x and 4x performance increases over D3 and Vega 1.5, respectively. This effect is due to each plotted line being built and encoded by its own dataflow branch. Across the other two examples, and averaging between the Canvas and SVG renderers, we find that although Reactive Vega takes 1.7x longer to initialize the visualizations, subsequent streaming operations are 1.9x faster than D3. Against Vega 1.5, Reactive Vega is again 1.7x slower at initializing visualizations; streaming updates perform roughly op-par with the Canvas renderer, but are 2x faster with the SVG renderer.

Slower initialization times for Reactive Vega are to be expected. D3 does not have to parse and compile a JSON specification, and a streaming dataflow graph is a more complex execution model, with higher overheads, than batch processing. However, with streaming

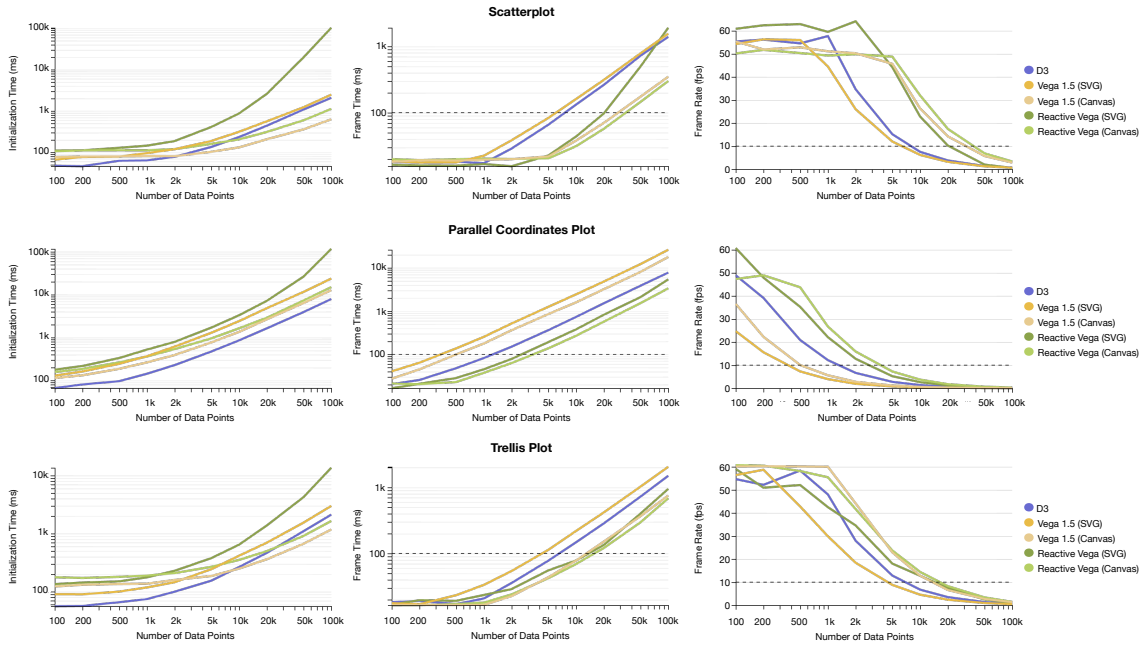


Figure 4.7: Average performance of rendering (non-interactive) streaming visualizations: (top-bottom) scatterplot, parallel coordinates, and trellis plot; (left-right) initialization time, average frame time, and average frame rate. Dashed lines indicate the threshold of interactive updates [22].

visualizations this cost amortizes and performance in response to data changes becomes more important. In this case, Reactive Vega makes up the difference in a single cycle.

4.3.2 Interactive Visualizations

We evaluated the performance of interactive visualizations (measured in terms of interactive frame rate) using three common examples: brushing & linking a scatterplot matrix, a time-series overview+detail visualization, and panning & zooming a scatterplot. We chose these examples as they all leverage interactive behaviors supported by D3, with canonical implementations available for each[1,2,3]. For Reactive Vega, we expressed these visualizations with a single declarative specification. For D3 and Vega 1.5, we use custom

[1]Brushing & Linking: http://bl.ocks.org/mbostock/4063663
[2]Overview + Detail: http://bl.ocks.org/mbostock/1667367
[3]Pan & Zoom: http://bl.ocks.org/mbostock/3892919

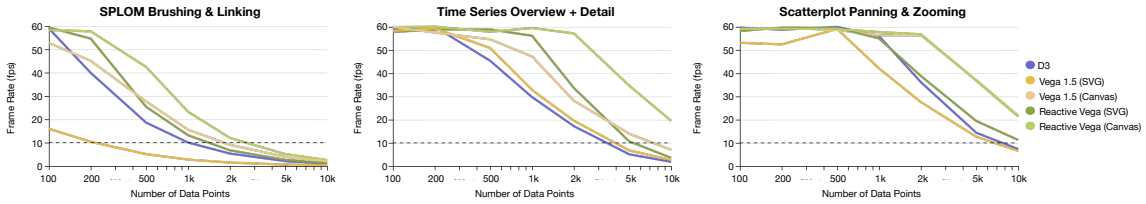


Figure 4.8: Average frame rates for three interactive visualizations: (left-right) brushing and linking on a scatterplot matrix; brushing and linking on an overview+detail visualization; panning and zooming on a scatterplot. Dashed lines indicate the threshold of interactive updates [22].

event handling callbacks. The Vega 1.5 callbacks mimic the behavior of the fragmented reactive approach used in prior work [88]. We tested these visualizations with datasets sized between 100 and 10,000 tuples.

Reactive Vega has the greatest effect with the parallel coordinates plot, displaying 2x and 4x performance increases over D3 and Vega 1.5, respectively. This effect is due to each plotted line being built and encoded by its own dataflow branch. Across the other two examples, and averaging between the Canvas and SVG renderers, we find that although Reactive Vega takes 1.7x longer to initialize the visualizations, subsequent streaming operations are 1.9x faster than D3. Against Vega 1.5, Reactive Vega is again 1.7x slower at initializing visualizations; streaming updates perform roughly op-par with the Canvas renderer, but are 2x faster with the SVG renderer.

4.4 Conclusion & Future Work

Declarative languages are a popular means of authoring visualizations [16, 17, 47], but have lacked first-class support for interaction design. In response, we contribute Reactive Vega, the first language and system architecture to support declarative visualization and interaction design in a comprehensive and performant fashion.

It is important to note that although Reactive Vega provides an complete end-to-end system — whereby users invoke the parser to traverse an input declarative specification and instantiate the necessary architecture components to render a visualization — this process can be decoupled. Reactive Vega's declarative model can be implemented as extensions to D3, and higher-level tools can manually construct and connect dataflow operators.

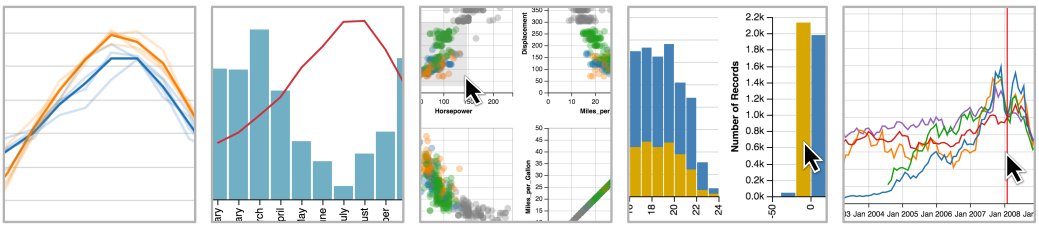
By facilitating programmatic generation of visualization, Reactive Vega's declarative JSON syntax has led to a growing ecosystem of higher-level visualization systems. For example, MapD [69] has integrated Vega with their GPU-powered database; custom SQL queries can be embedded within the JSON specification, which are dispatched to the backend server, rendered, and returned to the client as a PNG image. Similarly, Wikipedia, a security-concious environment where it would be difficult to allow users to write imperative visualization code, has integrated Vega [70] to embed interactive visualizations within articles.

Still, improved support for authoring and debugging Vega specifications remains a promising avenue for future work. For instance, Hoffswell et al. [52] developed a "time-traveling" debugger for Reactive Vega specifications and found that first-time Vega users were able to accurately trace errors through the specification. Further work, particularly by instrumenting Reactive Vega's dataflow graph to enable inspection and stepping through change-set propagation could aid in learnability [41]. Through the development of such tools, we can also assess the accessibility of the language. Can new users learn the declarative interaction model? Can experts, accustomed to callback-driven programming, adapt quickly?

Reactive Vega's architecture also offers opportunities to study scalable visualization design. Interactive visualization of large-scale datasets often requires offloading computation to server-side architectures. For example, Nanocubes [63] and imMens [64] assemble multi-dimensional data cubes that can be decomposed into smaller data tiles and pushed to the client. Such components could be integrated into a dataflow graph with execution distributed across server and client [72]. For example, as the dataflow graph scheduler is responsible for propagation, it might anticipate possible user interactions and prefetch data tiles in order to reduce latency [10].

Reactive Vega is an open source system available at `http://vega.github.io/vega/`.

5 | A Grammar of Interactive Graphics



Reactive Vega demonstrates that declarative interaction design is an expressive and performant alternative to imperative event handling callbacks. However, as its abstractions are relatively low-level, it is best described as an "assembly language" for interactive visualization most suited for *explanatory* visualization. In contrast, for *exploratory* visualization, higher-level grammars such as ggplot2 [105], and grammar-based systems such as Tableau (née Polaris [92]), are typically preferred as they favor conciseness over expressiveness. Analysts rapidly author partial specifications of visualizations; the grammar applies default values to resolve ambiguities, and synthesizes low-level details to produce visualizations.

High-level languages can also enable search and inference over the space of visualizations. For example, Wongsuphasawat et al. introduced Vega-Lite to power the Voyager visualization browser [108]. With a smaller surface area than Vega, Vega-Lite makes systematic enumeration and ranking of data transformations and visual encodings more tractable.

However, existing high-level languages provide limited support for interactivity. An analyst can, at most, enable a predefined set of common techniques (linked selections, panning & zooming, etc.) or parameterize their visualization with dynamic query widgets [81]. For custom, direct manipulation interaction they must turn to imperative event callbacks.

In this chapter, we extend Vega-Lite with a *multi-view* grammar of graphics and a *grammar of interaction* to enable concise, high-level specification of interactive data visualizations.

5.1 Visual Encoding

The simplest Vega-Lite specification — referred to as a *unit* specification — describes a single Cartesian plot with the following four-tuple:

unit := (data, transforms, mark-type, encodings)

The *data* definition identifies a data source: a relational table consisting of records (rows) with named attributes (columns). This table can be subject to a set of *transforms*, including filtering and adding derived fields via formulas. The *mark-type* specifies the geometric object used to visually encode the data records. Legal values include *bar*, *line*, *area*, *text*, *rule* for reference lines, and plotting symbols (*point* & *tick*). The *encodings* determine how data attributes map to the properties of visual marks. Formally, an encoding is a seven-tuple:

encoding := (channel, field, data-type, value, functions, scale, guide)

Available visual encoding *channels* include spatial position (*x*, *y*), *color*, *shape*, *size*, and *text*. An *order* channel controls sorting of stacked elements (e.g., for stacked bar charts and the layering order of line charts). A *path* order channel determines the sequence in which points of a line or area mark are connected to each other. A *detail* channel includes additional group-by fields in aggregate plots.

The *field* string denotes a data attribute to visualize, along with a given *data-type* (one of *nominal*, *ordinal*, *quantitative* or *temporal*). Alternatively, one can specify a constant literal *value* to serve as the data field. The data field can be further transformed using *functions* such as binning, aggregation (e.g., mean), and sorting.

An encoding may also specify properties of a *scale* that maps from the data domain to a visual range, and a *guide* (axis or legend) for visualizing the scale. If not specified, Vega-Lite will automatically populate default properties based on the *channel* and *data-type*. For *x* and *y* channels, either a linear scale (for quantitative data) or an ordinal scale (for ordinal

and nominal data) is instantiated, along with an axis. For *color*, *size*, and *shape* channels, suitable palettes and legends are generated. For example, quantitative color encodings use a single-hue luminance ramp, while nominal color encodings use a categorical palette with varied hues. Our default assignments largely follow the model of prior systems [92, 108].

Unit specifications are capable of expressing a variety of common, useful plots of both raw and aggregated data. Examples include bar charts, histograms, dot plots, scatter plots, line graphs, and area graphs. Our formal definitions are instantiated in a JSON (JavaScript Object Notation) syntax, as shown in Figs. 5.1 to 5.3.

```
{
  data: {url: "data/weather.csv", ...},
  mark: "line",
  encoding: {
    x: {
      field: "date", type: "temporal",
      timeUnit: "month"
    },
    y: {
      field: "temp_max", type: "quantitative",
      aggregate: "mean"
    },
    color: {
      field: "location", type: "nominal"
    }
  }
}
```

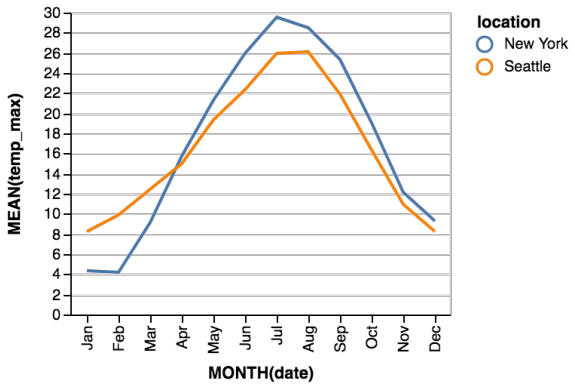


Figure 5.1: A unit specification that uses a *line* mark to visualize the *mean* temperature for every *month*.

```
{
  data: {url: "data/weather.csv", ...},
  mark: "bar",
  encoding: {
    x: {
      aggregate: "count", field: "*",
      type: "quantitative"
    },
    y: {
      field: "location", type: "nominal"
    },
    color: {
      field: "weather", type: "nominal"
    }
  }
}
```

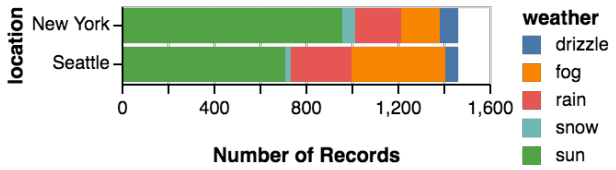


Figure 5.2: A *stacked* bar chart that sums the various weather types by location.

```
{
  data: {url: "data/weather.csv", ...},
  mark: "point",
  encoding: {
    x: {
      field: "temp_max", type: "quantitative",
      bin: true
    },
    y: {
      field: "wind", type: "quantitative",
      bin: true
    },
    size: {
      aggregate: "count", field: "*",
      type: "quantitative"
    },
    color: {
      field: "location", type: "nominal"
    }
  }
}
```

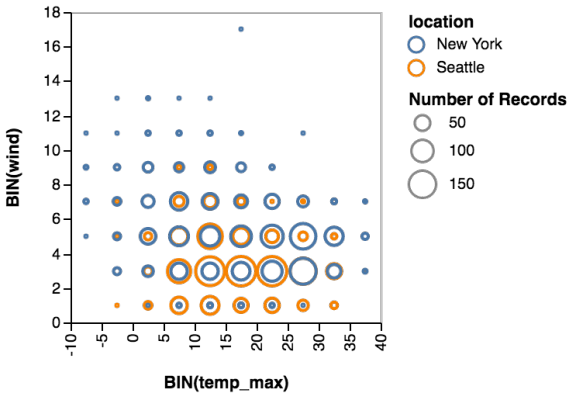


Figure 5.3: A *binned* scatterplot visualizes correlation between wind and temperature.

5.2 View Composition Algebra

Given multiple *unit* specifications, *composite* views can be constructed using the following operators. Each operator provides default strategies to *resolve* scales, axes, and legends across views. A user can choose to override these default behaviors by specifying tuples of the form *(channel, scale|axis|legend, union|independent)*. We use *view* to refer to any Vega-Lite specification, be it a *unit* or *composite* specification.

5.2.1 Layer

$$layer([unit_1, unit_2, ...], resolve)$$

The *layer* operator produces a view in which subsequent charts are plotted on top of each other. To produce coherent and comparable layers, we share scales (if their types match) and merge guides by default. For example, we compute the union of the data domains for the *x* or *y* channel, for which we then generate a single scale. However, Vega-Lite can not enforce that a unioned domain is *semantically* meaningful. To prohibit layering of incongruent composite views, the *layer* operator restricts its operands to be *unit* views.

```
{
  data: {url: "data/weather.csv", ...},
  transform: [{filter: "datum.location === 'Seattle'"}],
  layer: [{
    mark: "bar",
    encoding: {
      x: {
        field: "date", type: "temporal",
        timeUnit: "month"
      },
      y: {
        field: "precipitation", type: "quantitative",
        aggregate: "mean", axis: {grid: false}
      },
      color: {value: "#77b2c7"}
    }
  }, {
    mark: "line",
    encoding: {
      x: {
        field: "date", type: "temporal",
        timeUnit: "month"
      },
      y: {
        field: "temp_max", type: "quantitative",
        aggregate: "mean", axis: {grid: false},
      },
      color: {value: "#ce323c"},
    }
  }],
  resolve: {
    y: {scale: "independent"}
  }
}
```

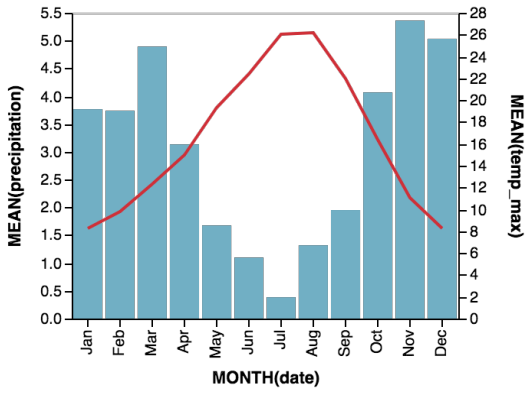


Figure 5.4: A dual axis chart that *layers* a line for the monthly mean temperature on top of bars for monthly mean precipitation. Each layer uses an *independent* y-scale.

5.2.2 Concatenation

$$\textit{hconcat}([view_1, view_2, ...], \textit{resolve})$$
$$\textit{vconcat}([view_1, view_2, ...], \textit{resolve})$$

The *hconcat* and *vconcat* operators place views side-by-side horizontally or vertically, respectively. If aligned spatial channels have matching data fields (e.g., the *y* channels in an *hconcat* use the same field), a shared scale and axis are used. Axis composition facilitates comparison across views and optimizes the underlying implementation.

```
{
  data: {url: "data/weather.csv", ...},
  vconcat: [{
    mark: "line",
    encoding: {
      x: {
        field: "date", type: "temporal",
        timeUnit: "month"
      },
      y: {
        field: "temp_max", type: "quantitative",
        aggregate: "mean"
      },
      color: {
        field: "location", type: "nominal"
      }
    }
  }, {
    mark: "point",
    encoding: {
      x: {
        field: "temp_max", type: "quantitative",
        bin: true
      },
      y: {
        field: "wind", type: "quantitative",
        bin: true
      },
      size: {
        aggregate: "count", field: "*",
        type: "quantitative"
      },
      color: {
        field: "location", type: "nominal"
      }
    }
  }]
}
```

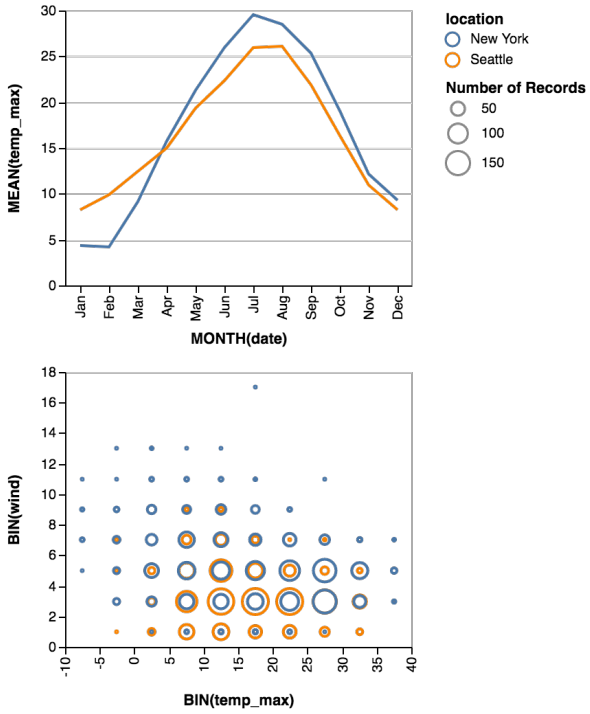


Figure 5.5: The Figs. 5.1 and 5.3 unit specifications *concatenated* vertically; scales and guides for each plot are independent by default.

5.2.3 Facet

facet(channel, data, field, view, scale, axis, resolve)

The *facet* operator produces a trellis plot [12] by subsetting the *data* by the distinct values of a *field*. The *view* specification provides a template for the sub-plots, and inherits the backing *data* for each partition from the operator. The *channel* indicates if sub-plots should be laid out vertically (*row*) or horizontally (*column*), and the *scale* and *axis* parameters enable further customization of sub-plot layout and labeling.

To facilitate comparison, scales and guides for quantitative fields are shared by default. This ensures that each facet visualizes the same data domain. However, for ordinal scales we generate independent scales by default to avoid unnecessary inclusion of empty categories, akin to Polaris' *nest* operator. When faceting by fiscal quarter and visualizing per-month data in each cell, one likely wishes to see three months per quarter, not twelve months of which nine are empty.

```
{
  data: {url: "data/weather.csv", ...},
  facet: {
    row: {field: "location", type: "nominal"}
  },
  spec: {
    mark: "line",
    encoding: {
      x: {
        field: "date", type: "temporal",
        timeUnit: "month"
      },
      y: {
        field: "temp_max", type: "quantitative",
        aggregate: "mean"
      },
      color: {
        field: "location", type: "nominal"
      }
    }
  }
}
```

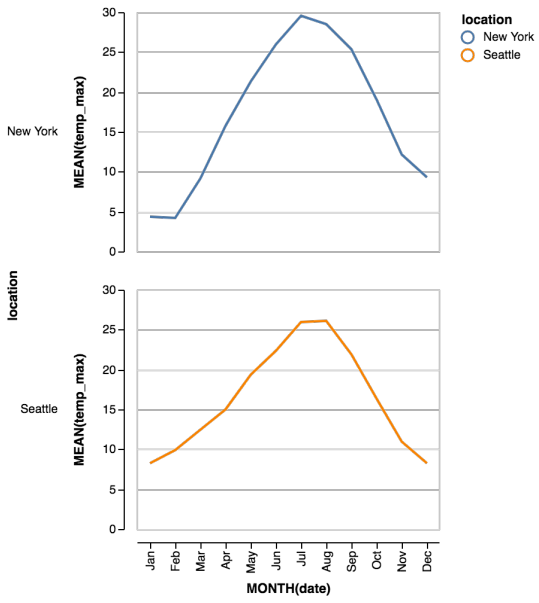


Figure 5.6: The line chart from Fig. 5.1 *faceted* vertically by location; the x-axis is shared, and the underlying scale domains unioned, to facilitate easier comparison.

5.2.4 Repeat

repeat(channel, values, scale, axis, view, resolve)

The *repeat* operator generates one plot for each entry in a list of *values*. The *view* specification provides a template for the sub-plots, and inherits the full backing dataset. Encodings within the repeated *view* specification can refer to this provided *value* to parameterize the plot[1]. As with *facet*, the *channel* indicates if plots should divide by *row* or *column*, with further customization possible via the *scale* and *axis* components. By default, scales and axes are independent, but legends are shared when data fields coincide.

```
{
  data: {url: "data/weather.csv", ...},
  repeat: {
    row: ["temp_max", "precipitation"]
  },
  spec: {
    mark: "line",
    encoding: {
      x: {
        field: "date", type: "temporal",
        timeUnit: "month"
      },
      y: {
        field: {"repeat": "row"},
        type: "quantitative",
        aggregate: "mean"
      },
      color: {
        field: "location", type: "nominal"
      }
    }
  }
}
```

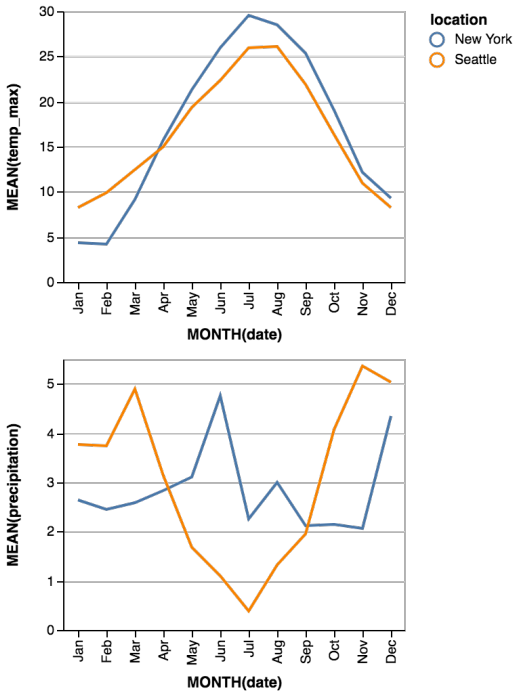


Figure 5.7: *Repetition* of different measures across rows; the y-channel references the `row` template parameter to vary the encoding.

[1]As the *repeat* operator requires parameterization of the inner view, it is not strictly algebraic. It is possible to achieve algebraic "purity" via explicit repeated concatenation or by reformulating the repeat operator (e.g., by including rewrite rules that apply to the inner view specification). However, we believe the current syntax to be more usable and concise than these alternatives.

5.2.5 Dashboards and Nested Views

These view composition operators form an algebra: the output of one operator can serve as input to a subsequent operator. As a result, complex dashboards and nested views can be concisely specified. For instance, a layer of two unit views might be repeated, and then concatenated with a different unit view. The one exception is *layer*, which, as previously noted, only accepts unit views to ensure consistent plots. For concision, two dimensional faceted or repeated layouts can be achieved by applying the operators to the *row* and *column* channels simultaneously. When faceting a composite view, only the dataset targeted by the operator is partitioned; any other datasets specified in sub-views are replicated.

5.3 Interactive Selections

To support specification of interaction techniques, we extend the definition of unit specifications to also include a set of *selections*. Selections identify the set of points a user is interested in manipulating, and is formally defined as an eight-tuple:

$$selection := (name, type, predicate, domain|range, event, init, transforms, resolve)$$

When an input *event* occurs, the selection is populated with *backing points* of interest. These points are the minimal set needed to identify all *selected points*. The selection *type* determines how many backing values are stored, and how the *predicate* function uses them to determine the set of selected points. Supported types include a *single* point, *multiple* discrete points, or a continuous *interval* of points.

As its name suggests, a single selection is backed by one datum, and its predicate tests for an exact match against properties of this datum. It can also function like a dynamic variable (or *signal* in Vega [88]), and can be invoked as such. For example, it can be referenced by name within a filter expression, or its values used directly for particular encoding channels. *Multi* selections, on the other hand, are backed by datasets into which data values are inserted, modified or removed as events fire. They express discrete selections, as their predicates test for an exact match with at least one value in the backing dataset. The order of points in a multi selection can be semantically meaningful, for example when a

multi selection serves as an ordinal scale domain. Figures 5.8 and 5.9 illustrate how points are highlighted in a scatterplot using single and multi selections.

Intervals are similar to multi selections. They are backed by datasets, but their predicates determine whether an argument falls within the minimum and maximum extent defined by the backing points. Thus, they express continuous selections. The compiler automatically adds a rectangle mark, as shown in Fig. 5.10, to depict the selected interval. Users can customize the appearance of this mark via the `mark` keyword, or disable it altogether when defining the selection.

```
{
  data: {url: "data/cars.json"},
  mark: "circle",
  selection: {
    id: {type: "single"}
  },
  encoding: {
    x: {field: "Horsepower", type: "Q"},
    y: {field: "MPG", type: "Q"},
    color: {
      condition: {
        selection: "id",
        field: "Origin", type: "N"
      },
      value: "grey"
    }
  }
}
```

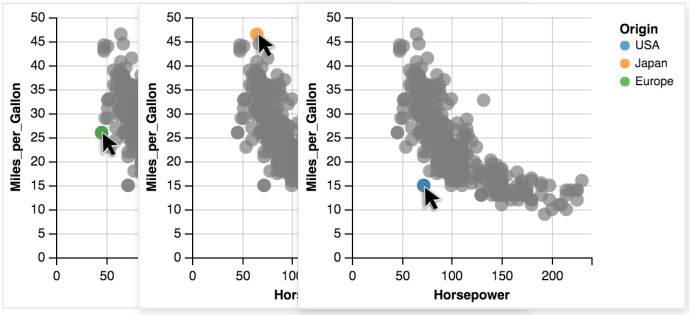


Figure 5.8: Adding a *single* selection to parameterize the fill color of a scatterplot's circle mark.

```
{
  data: {url: "data/cars.json"},
  mark: "circle",
  selection: {
    id: {type: "multi"}
  },
  encoding: {
    x: {field: "Horsepower", type: "Q"},
    y: {field: "MPG", type: "Q"},
    color: {
      condition: {
        selection: "id",
        field: "Origin", type: "N"
      },
      value: "grey"
    }
  }
}
```

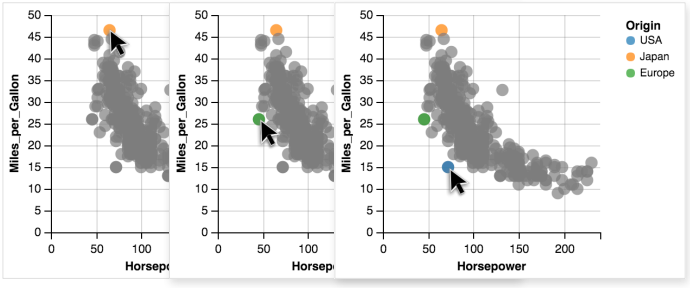


Figure 5.9: Switching from a *single* to *multi* selection. The first value is selected on click, and additional values on shift-click.

```
{
  data: {"url": "data/cars.json"},
  mark: "circle",
  selection: {
    region: {"type": "interval"}
  },
  encoding: {
    x: {field: "Horsepower", type: "Q"},
    y: {field: "MPG", type: "Q"},
    color: {
      condition: {
        selection: "region",
        field: "Origin", type: "N"
      },
      value: "grey"
    }
  }
}
```

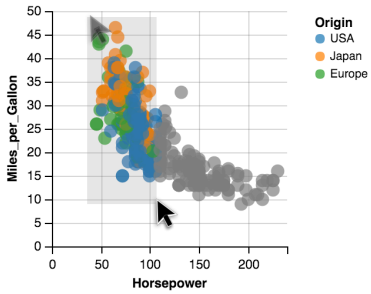


Figure 5.10: Highlight a continuous range of points using an *interval* selection. A rectangle mark is automatically added to depict the interval extents.

Predicate functions enable a minimal set of backing points to represent the full space of selected points. For example, with predicates, an interval selection need only be backed by two points: the minimum and maximum values of the interval. While selection types provide default definitions, predicates can be customized to concisely specify an expressive space of selections. For example, a single selection with a custom predicate of the form `datum.binned_price == selection.binned_price` is sufficient for selecting all data points that fall within a given bin.

By default, backing points lie in the data *domain*. For example, if the user clicks a mark instance, the underlying data tuple is added to the selection. If no tuple is available, event properties are passed through inverse scale transforms. For example, as the user moves their mouse within the data rectangle, the mouse position is inverted through the x and y scales and stored in the selection. Defining selections over data values, rather than visual properties, facilitates reuse across distinct views; each view may have different encodings specified, but are likely to share the same data domain. However, some interactions are inherently about manipulating visual properties — for example, interactively selecting the colors of a heatmap. For such cases, users can define selections over the visual *range* instead. When input events occur, visual elements or event properties are then stored.

The particular events that update a selection are determined by the platform a Vega-Lite specification is compiled on, and the input modalities it supports. By default we use mouse events on desktops, and touch events on mobile and tablet devices. A user can specify alternate events using Vega's event selectors (§3.1.1). For example, Fig. 5.11 demonstrates how `mouseover` events are used to populate a multi selection. With the event selector syntax, multiple events are specified using a comma (e.g., `mousedown, mouseup` adds items to the selection when either event occurs). A sequence of events is denoted with the between-filter. For example, `[mousedown, mouseup] > mousemove` selects all `mousemove` events that occur between a `mousedown` and a `mouseup` (otherwise known as "drag" events). Events can also be filtered using square brackets (e.g., `mousemove [event.pageY > 5]` for events at the top of the page) and throttled using braces (e.g., `mousemove{100}` populates a selection at most every 100 milliseconds).

```
{
  data: {url: "data/cars.json"},
  mark: "circle",
  selection: {
    id: {type: "multi", on: "mouseover"}
  },
  encoding: {
    x: {field: "Horsepower", type: "Q"},
    y: {field: "MPG", type: "Q"},
    color: {
      condition: {
        selection: "id",
        field: "Origin", type: "N"
      },
      value: "grey"
    }
  }
}
```

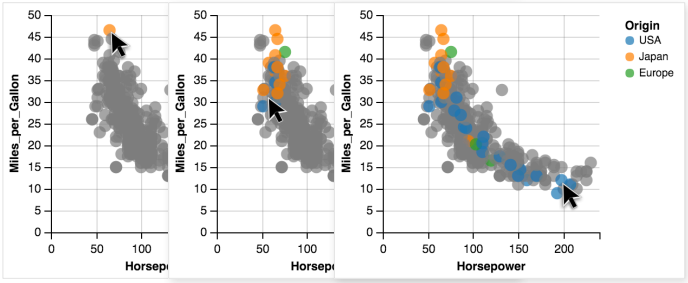


Figure 5.11: Specifying a custom event trigger for a *multi* selection: the first point is selected on `mouseover` and subsequent points when the shift key is pressed.

5.3.1 Selection Transforms

Analogous to data transforms, selection transforms manipulate the components of the selection they are applied to. For example, they may perform operations on the backing points, alter a selection's predicate function, or modify the input events that update the selection. Unlike data transforms, however, specifying an ordering to selection transforms is not necessary as the compilation step ensures commutativity. All transforms are

first parsed, setting properties on an internal representation of a selection, before they are compiled to produce event handling and interaction logic.

We identify the following transforms as a minimal set to support both common and custom interaction techniques. Additional transforms can be defined and registered with the system, and then invoked within the specification. In this way, the Vega-Lite language remains concise while ensuring extensibility for custom behaviors.

Project

project(fields, channels)

The *project* transform alters a selection's predicate function to determine inclusion by matching only the given *fields*. Some fields, however, may be difficult for users to address directly (e.g., new fields introduced due to inline binning or aggregation). For such cases, a list of *channels* may also be specified (e.g., `color`, `size`).

```
{
  ...
  selection: {
    id: {
      type: "single",
      fields: ["Origin"]
    }
  },
  ...
}
```

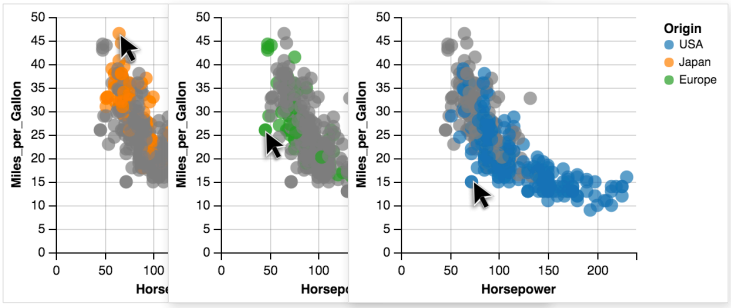


Figure 5.12: Using the *project* transform to highlight a *single* `Origin`.

```
{
  ...
  selection: {
    id: {
      type: "multi",
      fields: ["Origin"]
    }
  },
  ...
}
```

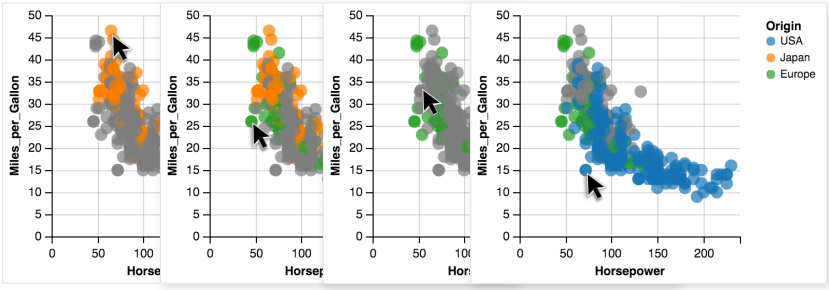


Figure 5.13: Using the *project* transform to highlight *multiple* `Origins`.

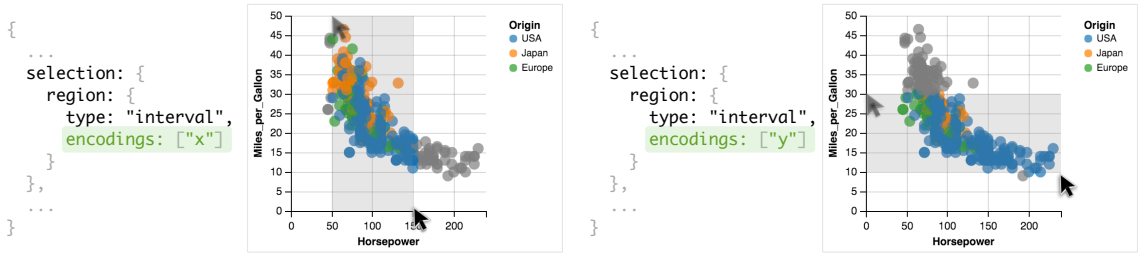


Figure 5.14: *Projecting* an interval selection to restrict it to a single dimension.

Toggle

toggle(event)

The *toggle* transform is automatically instantiated for uninitialized multi selections. When the *event* occurs, the corresponding data value is added or removed from the multi selection's backing dataset. By default, the toggle *event* corresponds to the selection's triggering event, but with the shift key pressed. For example, in Fig. 5.9, additional points are added to the multi selection on shift-click (where `click` is the default event for multi selections). The selection in Fig. 5.11, however, specifies a custom `mouseover` event. Thus, additional points are inserted when the shift key is pressed and the mouse cursor hovers over a point.

Translate

translate(events, by)

The *translate* transform offsets the spatial properties (or corresponding data fields) of backing points by an amount determined by the coordinates of the sequenced *events*. For example, on the desktop, drag events (`[mousedown, mouseup] > mousemove`) are used and the offset corresponds to the difference between where the `mousedown` and subsequent `mousemove` events occur. If no coordinates are available (e.g., as with keyboard events), a *by* argument should be specified. This transform respects the *project* transform as well, restricting movement to the specified dimensions. This transform is automatically instantiated for interval selections.

```
{
  ...
  selection: {
    region: {
      type: "interval",
      translate: true
    }
  },
  ...
}
```

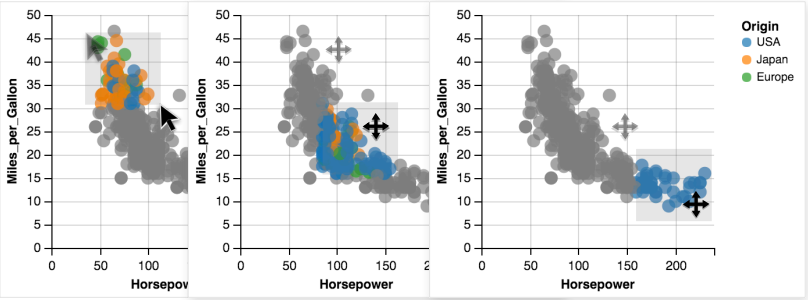


Figure 5.15: The *translate* transform enables movement of the brushed region. It is automatically invoked for interval selections but is explicitly depicted here for clarity.

Zoom

zoom(event, factor)

The *zoom* transform applies a scale factor, determined by the *event* to the spatial properties (or corresponding data fields) of backing points. A *factor* must be specified if it cannot be determined from the events (e.g., when arrow keys are pressed). When combined with the *project* transform, an interval can be zoomed uni-dimensionally.

Bind

bind(widgets|scales)

The *bind* transform establishes a two-way binding between control widgets (e.g., sliders, textboxes, etc.) or scale functions for single and interval selections respectively.

When a single selection is bound to query widgets, one widget per projected field is generated and may be used to manipulate the corresponding predicate clause. When triggering events occur to update the selected points, the widgets are updated as well. Control widgets, in addition to direct manipulation interaction, allow for more rapid and exhaustive querying of the backing data [91]. For example, scrubbing a slider back and forth can quickly reveal a trend in the data or highlight a small number of selected points that would otherwise be difficult to pick out directly.

Interval selections can be bound to the scales of the unit specification they are defined in. Doing so *initializes* the selection, populating it with the given scales' domain or range, and

parameterizes the scales to use the selection instead. Binding selections to scales allows scale extents to be interactively manipulated, yet remain automatically initialized by the input data. By default, both the x and y scales are bound; alternate scales are specified by *projecting* over the corresponding channels.

```
{
  ...
  selection: {
    region: {
      type: "interval",
      on: "[mousedown[event.shiftKey], mouseup] > mousemove"
    },
    grid: {
      type: "interval",
      bind: "scales", zoom: true,
      translate: "[mousedown[event.shiftKey], mouseup] > mousemove"
    }
  },
  ...
}
```

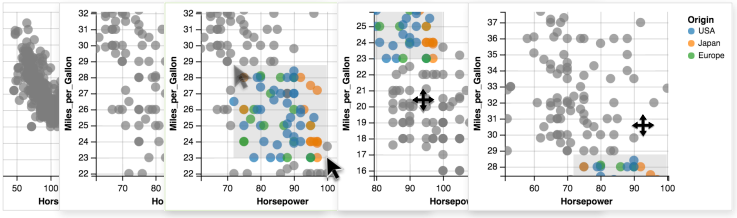


Figure 5.16: Panning and zooming the scatterplot is achieved by first *binding* an interval selection to the x- and y-scale domains, and then applying the *translate* and *zoom* transforms. Alternate events prevent collision with the brushing interaction, previously defined in Fig. 5.10

Nearest

nearest()

The *nearest* transform computes a Voronoi decomposition, and augments the selection's event processing. The data value or visual element nearest the triggering *event* is now selected (approximating a Bubble Cursor [39]). Currently, the centroid of each mark instance is used to calculate the Voronoi diagram but we plan to extend this operator to account for boundary points as well (e.g., rectangle vertices).

5.3.2 Selection-Driven Visual Encodings

Once selections are defined, they parameterize visual encodings to make them interactive — visual encodings are automatically reevaluated as selections change. First, selections can be used to drive *conditional* encoding rules. Each data tuple participating in the

encoding is evaluated against the selection's predicate, and properties are set based on whether it belongs to the selection or not. For example, as shown in Figs. 5.8 and 5.9, the fill color of the scatterplot circles is determined by a data field if they fall within the `id` selection, or set to grey otherwise.

Next, selected points can be explicitly materialized and used as input data for other encodings within the specification. By default, this applies a selection's predicate against the data tuples (or visual elements) of the unit specification it is defined in. To materialize a selection against an arbitrary dataset, a *map* transform rewrites the predicate function to account for differing schemas. Using selections in this way enables linked interactions, including displaying tooltips or labels, and cross-filtering.

Besides serving as input data, a materialized selection can also define scale extents. Binding a selection to scales offers a concise way of specifying this behavior within the same unit specification. For multi-view displays, selection names can be specified as the domain or range of a particular channel's scale. Doing so constructs interactions that manipulate viewports, including panning & zooming (Fig. 5.16) and overview + detail (Fig. 5.21).

In all three cases, selections can be composed using logical `OR`, `AND`, and `NOT` operators. As previously discussed, single selections offer an additional mechanism for parameterizing encodings. The backing point can be directly referenced within the specification, for example as part of a filter or calculate expression, or to determine a visual encoding channel without the overhead of a conditional rule. In Fig. 5.22, for instance, the red rule is positioned using the `date` value of the `indexPt` selection.

5.3.3 Disambiguating Composite Selections

Selections are defined within unit specifications to provide a default context — a selection's events are registered on the unit's mark instances, and materializing a selection applies its predicate against the unit's input data by default. When units are composed, however, selection definitions and applications become ambiguous.

Consider Fig. 5.17, which illustrates how a scatterplot matrix (SPLOM) is constructed by repeating a unit specification. To brush, we define an interval selection (`region`) within the unit, and use it to perform a linking operation by parameterizing the color of the circle marks. However, there are several ambiguities within this setup. Is there one `region` for the overall visualization, or one per cell? If the latter, which cell's `region` should be used to highlight the points? This ambiguity recurs when selections serve as input data or scale extents, and when selections share the same name across a layered or concatenated views.

Several strategies exist for resolving this ambiguity. By default, a *global* selection exists across all views. With our SPLOM example, this setting causes only one brush to be populated and shared across all cells. When the user brushes in a cell, points that fall within it are highlighted, and previous brushes are removed.

```
{
  data: {url: "data/cars.json"},
  repeat: {
    row: ["Displacement", "Miles_per_Gallon"],
    column: ["Horsepower", "Miles_per_Gallon"]
  },
  spec: {
    mark: "circle",
    selection: {
      region: {
        type: "interval", resolve: "global",
        on: "[mousedown[event.shiftKey], mouseup] > mousemove"
      },
      grid: {
        type: "interval",
        bind: "scales", zoom: true,
        translate: "[mousedown[event.shiftKey], mouseup] > mousemove"
      }
    },
    encoding: {
      x: {field: "Horsepower", type: "Q"},
      y: {field: "MPG", type: "Q"},
      color: {
        condition: {
          selection: "region",
          field: "Origin", type: "N"
        },
        value: "grey"
      }
    }
  }
}
```

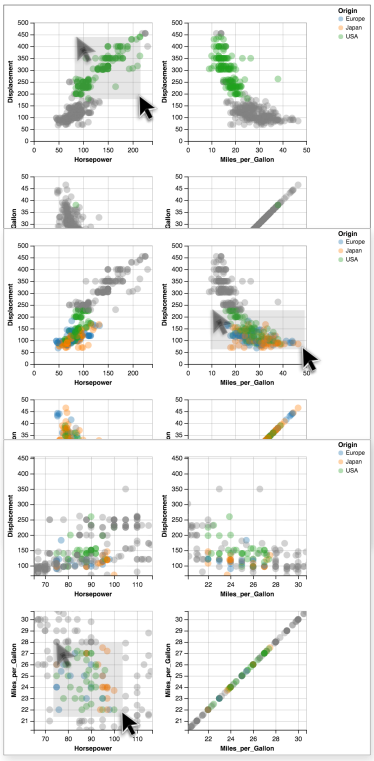


Figure 5.17: By adding a *repeat* operator, we compose the encodings and interactions from Fig. 5.16 into a scatterplot matrix. Users can brush, pan, and zoom within each cell, and the others update in concert. By default, a *global* composite selection is created: brushing in a cell replaces previous brushes.

Users can specify an alternate ambiguity resolution when defining a selection. These schemes all construct one instance of the selection per view, and define which instances are used in determining inclusion. For example, resolving a selection to *independent* creates one instance per view, and each unit uses only its own selection to determine inclusion. With our SPLOM example, this would produce the interaction shown below. Each cell would display its own brush, which would determine how only its points would be highlighted.

Selections can also be resolved to *union* or *intersect*. Here, all instances of a selection are considered in concert: a point falls within the overall selection if it is included in, respectively, at least one of the constituents or all of them. More concretely, with the SPLOM example, these settings would continue to produce one brush per cell, and points would highlight when they lie within at least one brush (*union*) or if they are within every brush (*intersect*) as shown in Figs. 5.19 and 5.20 respectively. We also support *union others* and *intersect others* resolutions, which function like their full counterparts except that a unit's own selection is not part of the inclusion determination. These latter methods support cross-filtering (Fig. 5.23) where interactions within a view should not filter itself.

```
{
  ...
  selection: {
    region: {
      type: "interval", global: "independent",
      on: "[mousedown[event.shiftKey], mouseup] > mousemove"
    },
    ...
  },
  ...
}
```

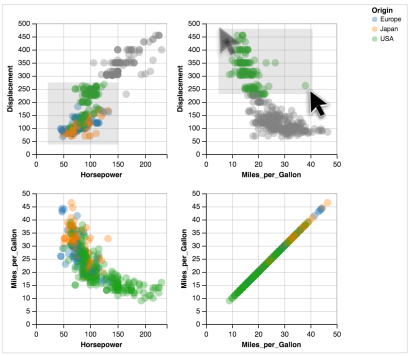


Figure 5.18: Resolving the `region` selection to *independent* produces a brush in each cell, and points only highlight based on the selection in their own cell.

```
{
  ...
  selection: {
    region: {
      type: "interval", global: "union",
      on: "[mousedown[event.shiftKey], mouseup] > mousemove"
    },
    ...
  },
  ...
}
```

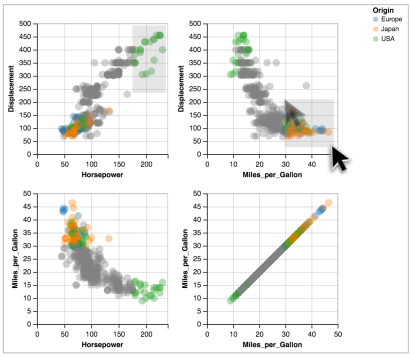


Figure 5.19: Resolving the `region` selection to *union* produces a brush in each cell, and points highlight if they fall within any of the selections.

```
{
  ...
  selection: {
    region: {
      type: "interval", global: "intersect",
      on: "[mousedown[event.shiftKey], mouseup] > mousemove"
    },
    ...
  },
  ...
}
```

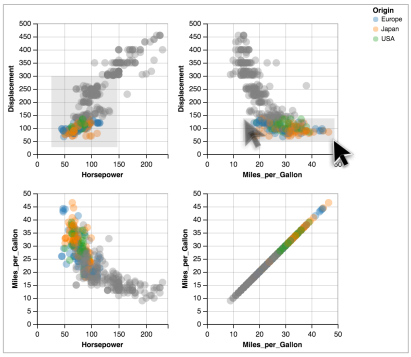


Figure 5.20: Resolving the `region` selection to *intersect* produces a brush in each cell, and points only highlight if they lie within all of the selections.

5.4 Compilation

The Vega-Lite compiler ingests a JSON specification and outputs a lower-level Reactive Vega specification (also expressed as JSON). However, there is no one-to-one correspondence between components of the Vega-Lite and Vega specifications. For instance, the compiler has to synthesize a single Vega data source, with transforms for binning and aggregation, from multiple Vega-Lite encoding definitions. Conversely, for a single definition of a Vega-Lite selection, the compiler might generate multiple Vega signals, data sources, and even parameterize scale extents. Moreover, to facilitate rapid authoring of visualizations, Vega-Lite specifications omit lower-level details including scale types and

the properties of the visual elements such as the font size. The compiler must resolve the resulting ambiguities.

To overcome these challenges, the compiler moves through four phases:

1. *Parse* — the compiler parses and disambiguates an input specification. Hand-crafted rules are applied to produce perceptually effective visualizations. For example, if the color channel is mapped to an nominal field, and the user has not specified a scale domain, a categorical color palette is inferred. If the color is mapped to a quantitative field, a sequential color palette is chosen instead.

2. *Build* — the compiler builds an internal representation to map between Vega-Lite and Vega primitives. A tree of *models* is constructed; each model corresponds to a unit or composite view, and stores a series of *components*. Components are data structures that loosely correspond to Vega primitives (such as data sources, scales, and marks). For example, the `DataComponent` details how the dataset should be loaded (e.g., is it embedded directly in the specification, or should it be loaded from a URL, and in what format), which fields should be aggregated or binned, and what filters and calculations should be performed.

   In this step, compile-time selection transforms (those not parameterized by events) are applied to the requisite components. For example, the *project* transform overrides the `SelectionComponent`'s predicate function, while the *nearest* transform augments the `MarkComponent` with a Voronoi diagram. This phase also constructs a special `LayoutComponent` to calculate suitable spatial dimensions for views. This component emits Vega data sources and transforms to calculate a bottom-up view layout at runtime.

3. *Merge* — once the necessary components have been built, the compiler performs a bottom-up traversal of the model tree to merge redundant components. This step is critical for ensuring that the resultant Vega specification does not perform unnecessary computation that might hinder interactive performance. To determine

whether components can be merged, the compiler computes a hash code and compares components of the same type. For example, when a scatterplot matrix is specified using the *repeat* operator, merging ensures that we only produce one scale for each row and column rather than two scales per cell ($2N$ versus $2N^2$ scales). Merging may introduce additional components if doing so results in a more optimal representation. For example, if multiple units within a composite specification load data from the same URL, a new `DataComponent` is created to load the data and the units are updated to inherit from it instead. This step also unions scale domains and resolves `SelectionComponents`.

4. *Assemble* — the final phase assembles the requisite Vega specification. In particular, `SelectionComponents` produce signals to capture events and the necessary backing points, and multi and interval selections construct data sources as well to hold multiple points. Each run-time selection transform (i.e., those that are triggered by an event) generates signals as well, and may augment the selection's data source with data transformations. For example, the *translate* transform adds a signal to capture an "anchor" position, to determine where panning begins, and another to calculate a "delta" from the anchor. These two signals then feed transforms that offset the backing points stored in the selection's data source, thereby moving the brush or panning the scales.

5.5 Example Interactive Visualizations

Vega-Lite's design is motivated by two goals: to enable rapid yet expressive specification of interactive visualizations, and to do so with concise primitives that facilitate systematic enumeration and exploration of design variations. In this section, we demonstrate how these goals are addressed using a range of example interactive visualizations. To evaluate expressivity, we once again choose examples that cover Yi et al.'s [111] taxonomy of interaction methods. Recall, the taxonomy identifies seven categories of techniques: *select*, to mark items of interest; *explore* to examine subsets of the data; *connect* to highlight related items within and across views; *abstract/elaborate* to vary the level of detail; *reconfigure* to

show different arrangements of the data; *filter* to show elements conditionally; and, *encode*, to change the visual representations used. To assess authoring speed, we compare our specifications against canonical Reactive Vega examples [87, 88, 95]. Where applicable, we also show how construction of our examples can be systematically varied to explore alternate points in the design space.

5.5.1 Selection: Click/Shift-Click and Brushing

Fig. 5.8 provides the full Vega-Lite specification for a scatterplot where users can mark individual points of interest. It includes the simplest definition of a selection — a name and type — and illustrates how the mark color is determined conditionally.

Modifying a single property, *type*, as in Fig. 5.9, allows users to mark multiple points (*toggle* is automatically instantiated by the compiler). We can instead add *project* (Fig. 5.12) such that marking a single point of interest highlights all other points that share particular data values — a *connect*-type interaction. Such changes to the specification are not mutually exclusive, and can be composed as shown in Fig. 5.13.

By using the *interval* type, users can mark items of interest within a continuous region. As shown in Fig. 5.10, the compiler automatically adds a rectangle mark to depict the selection, and instantiates *translate* to allow it to be repositioned (Fig. 5.15). In this context, *project* restricts the interval to a single dimension (Fig. 5.14).

These specifications are an order of magnitude more concise than their Vega counterparts. With Vega-Lite, users need only specify the semantics of their interaction and the compiler fills in appropriate default values. For example, by default, individual points are selected on click and multiple points on shift-click. Users can override these defaults, sometimes producing a qualitatively different user experience. For example, one can instead update selections on `mouseover` to produce a "paint brush" interaction, as in Fig. 5.11. In contrast, with Vega, users must manually author all the components of an interaction, including determining if event properties need to be passed through scale inversions, creating necessary backing data structures, and adding marks to represent brushed extents.

5.5.2 Explore & Encode: Panning & Zooming

Vega-Lite's selections also enable accretive design of interactions. Consider our previous example of brushing a scatterplot. We can define an additional interval selection and *bind* it to the unit's scale functions (Fig. 5.16). The compiler populates the selection with the x and y scale domains, parameterizes them to use it, and instantiates the *translate* and *zoom* transforms. Users can now brush, pan, and zoom the scatterplot. However, the default definitions of the two interval selections collide: dragging produces a brush and pans the plot. This example illustrates that concise methods for overriding defaults can not only be useful (as in Fig. 5.11) but also necessary. We override the default events that trigger the two interactions using Vega's event selector syntax [88]. As Fig. 5.16 shows, we specify that brushing only occurs when the user drags with the shift key pressed.

The Vega-Lite specification for panning and zooming is, once again, more succinct than the corresponding Vega example. However, it is more interesting to compare the latter against the output specification produced by the Vega-Lite compiler. The Vega example requires users to manually specify their initial scale extents when defining the interaction. On the other hand, to enable data-driven initialization of interval selections, the Vega-Lite output calculates scale extents as part of a derived dataset, with additional transformations to offset these calculations for the interaction. Such a construction is not idiomatic Vega, and would be unintuitive for users to construct manually. Thus, Vega-Lite's higher-level approach not only offers more rapid specification, but it can also enable interactions that a user may not realize are expressible with lower-level representations.

Moreover, by enabling this interaction through composable primitives (rather than a single, specific "pan and zoom" operator [17]), Vega-Lite also facilitates exploring related interactions in the design space. For example, using the *project* transform, we can author a separate selection for the x and y scales each, and selectively enable the *translate* and *zoom* transforms. While such a combination may not be desirable — panning only one scale while zooming the other — Vega-Lite's selections nevertheless allow us to systematically identify it as a possible design. Similarly, we could project over the color or size channels, thereby allowing users to interactively vary the mappings specified by these scales. For

example, "panning" a heatmap's color legend to shift the high and low intensity data values. If the selections were defined over the visual *range*, users could instead shift the colors used in a sequential color scale.

5.5.3 Connect: Brushing & Linking

We can wrap our previous example, from Fig. 5.16, in a *repeat* operator to construct a scatterplot matrix (SPLOM) as shown in Fig. 5.17. With no further modifications, all our previous interactions now work within each cell of the SPLOM and are synchronized across the others. For example, dragging pans not only the particular cell the user is in, but related cells along shared axes. Similarly, dragging with the shift key pressed produces a brush in the current cell, and highlights points across all cells that fall within it.

As its name suggests, the repeat operator creates one instance of the child specification for the given parameters. By default, to provide a consistent experience when moving from a unit to a composite specification, Vega-Lite creates a *global* instance of the selection that is populated and shared between all repeated instances (Fig. 5.17). With the *resolve* property, users can specify alternate disambiguation methods including creating an independent brush for each cell, unioning the brushes, or intersecting them (Figs. 5.18 to 5.20 respectively). If selections are bound to scales or parameterize them, only a global selection is supported for consistency with the composition algebra.

With this example, it is more instructive to compare the amount of effort required, with Vega-Lite and Vega, to move from a single interactive scatterplot to an interactive SPLOM. While the Vega specifications for the two are broadly similar, the latter requires an extra level of indirection to identify the specific cell a user is interacting in, and to ensure that the correct data values are used to determine inclusion within the brush. In Vega-Lite, this complexity is succinctly encapsulated by the *resolve* keyword which, as discussed, can be systematically varied to explore alternatives. Mimicing Vega-Lite's *union* and *intersect* behaviors is not trivial, and requires unidiomatic Vega once more. Users cannot simply duplicate the interaction logic for each cell manually, as the dimensions of the SPLOM are determined by data.

5.5.4 Abstract/Elaborate: Overview + Detail

Thus far, selections have parameterized scale extents through the *bind* transform and previous examples have demonstrated how visualized data can be abstracted/elaborated via zooming. The figure below shows how a selection defined in one unit specification can be explicitly given as the scale domain of another in a concatenated display. Doing so creates an overview + detail interaction: brushing in the bottom (overview) chart displays only selected items at a higher resolution in the larger (detail) chart at the top.

```
{
  data: {url: "data/sp500.csv", ...},
  vconcat: [
    {
      mark: "area",
      encoding: {
        x: {
          field: "date", type: "temporal", ...,
          scale: {domain: {selection: "region"}}
        },
        y: {field: "price", type: "quantitative"}
      }
    },
    {
      mark: "area",
      selection: {
        region: {
          type: "interval", "encodings": ["x"]
        }
      },
      encoding: {
        x: {field: "date", type: "temporal", ...},
        y: {field: "price", type: "quantitative", ...}
      }
    }
  ]
}
```

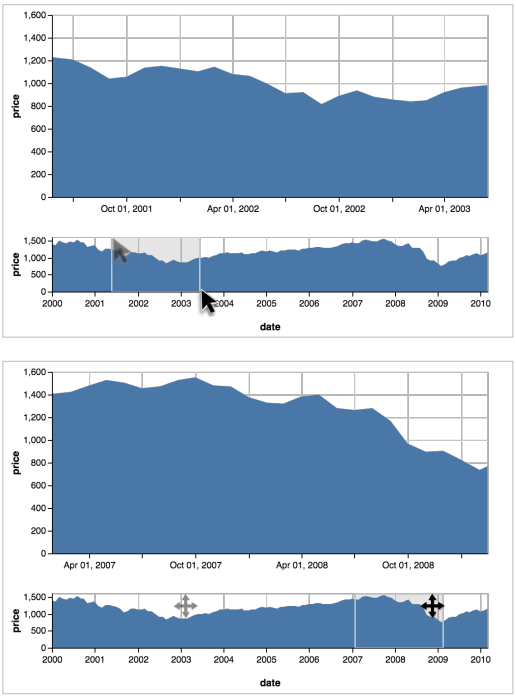


Figure 5.21: An overview + detail visualization concatenates two unit specifications, with a selection in the second one parameterizing the x-scale domain in the first.

5.5.5 Reconfigure: Index Chart

Figure 5.22 uses a single selection to interactively normalize stock price time series data as the user moves their mouse across the chart. We apply the *nearest* transform to accelerate the selection using an invisible Voronoi diagram. By projecting over the `date` field, the selection represents both a single data value as well a set of values that share the selected `date`. Thus, we can reference the single selection directly, to position the red vertical rule, and also materialize it as part of the *lookup* data transform.

```
{
  data: {url: "data/stocks.csv"},
  transform: [
    {
      lookup: "symbol",
      from: {
        selection: "indexPt", keys: ["symbol"],
        fields: ["price"], as: ["idx_price"]
      }
    },
    {
      calculate: "(datum.price - datum.idx_price) / datum.idx_price"
      as: "norm_price"
    }
  ],
  layer: [{
    selection: {
      indexPt: {
        type: "single", fields: ["date"],
        on: "mousemove", nearest: true
      }
    },
    mark: "line",
    encoding: {
      x: {field: "date", type: "temporal", ...},
      y: {field: "norm_price", type: "quantitative", ...},
      color: {field: "symbol", type: "nominal"}
    }
  }, {
    mark: "rule",
    encoding: {
      x: {selection: "indexPt.date", type: "temporal"},
      color: {value: "red"}
    },
  }]
}
```

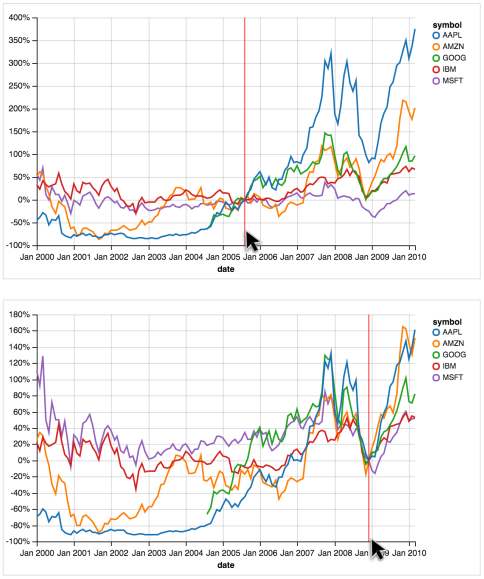


Figure 5.22: An index chart uses a single selection to renormalize data based on the index point nearest the mouse cursor.

5.5.6 Filter: Cross Filtering

As selections provide a predicate function, it is trivial to use them to filter a dataset. Figure 5.23, for example, presents a concise specification to enable filtering across three distinct binned histograms. It uses a *repeat* operator with a uni-dimensional interval selection over the bins set to *intersect others*. The *filter* data transform applies the selection against the backing datasets such that only data values that fall within the selection are displayed. Thus, as the user brushes in one histogram, the datasets that drive each of the other two are filtered, the data values are re-aggregated, and the bars rise and fall. As with other interval selections, the Vega-Lite compiler automatically instantiates the *translate* transform, allowing users to drag brushes around rather than having to reselect them from scratch.

```
{
  data: {url: "data/flights.json"},
  repeat: {
    column: ["hour", "delay", "distance"]
  },
  spec: {
    selection: {
      region: {
        type: "interval", encodings: ["x"],
        resolve: "intersect_others"
      }
    },
    transform: [
      {filter: {selection: "region"}}
    ],
    mark: "bar",
    encoding: {
      x: {field: {repeat: "column"}, bin: true, ...},
      y: {aggregate: "count", field: "*", ...},
      color: {value: "steelblue"},
    }
  }
}
```

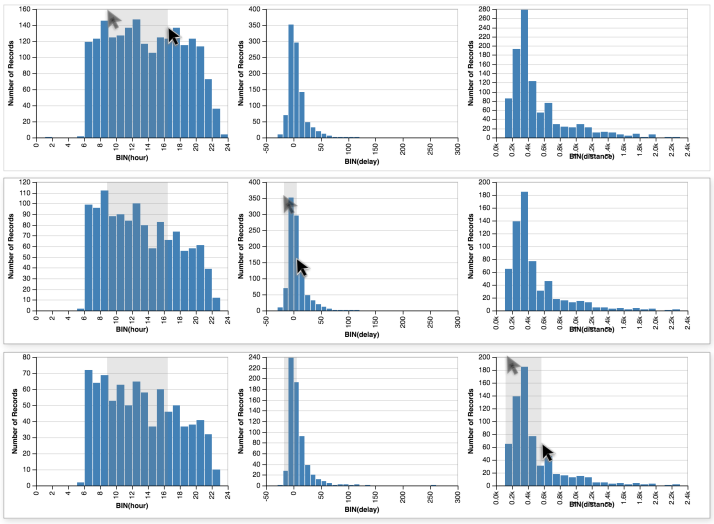


Figure 5.23: An interval selection, resolved to *intersect others*, drives a cross filtering interaction. Brushing in one histogram filters and reaggregates the data in the others, observable by the varying y-axis labels in the screenshots.

The *filter* data transform can also be used to materialize the selection as an input dataset for secondary views. For instance, one drawback of cross-filtering as in Fig. 5.23 is that users only see the selected values, and lose the context of the overall dataset. Instead of applying the selection back onto the input dataset, we can instead materialize it as an overlay

(Fig. 5.24). Now, as the user brushes in one histogram, bars highlight to visualize the proportion of the overall distribution that falls within the brushed region(s). With this setup, it is necessary to change the selection's resolution to simply *intersect*, such that bars in the brushed plot also highlight during the interaction.

```
{
  data: {url: "data/flights.json"},
  repeat: {
    column: ["hour", "delay", "distance"]
  },
  spec: {
    layer: [{
      selection: {
        region: {
          type: "interval", encodings: ["x"],
          resolve: "intersect"
        }
      },
      mark: "bar",
      encoding: {
        x: {field: {repeat: "column"}, type: "Q", bin: true},
        y: {aggregate: "count", field: "*", type: "Q"},
        color: {value: "steelblue"}
      },
    }, {
      transform: [
        {filter: {selection: "selectedBins"}}
      ],
      mark: "bar",
      encoding: {
        x: {field: {repeat: "column"}, type: "Q", bin:  true},
        y: {aggregate: "count", field: "*", type: "Q"},
        color: {value: "goldenrod"}
      }
    }]
  }
}
```

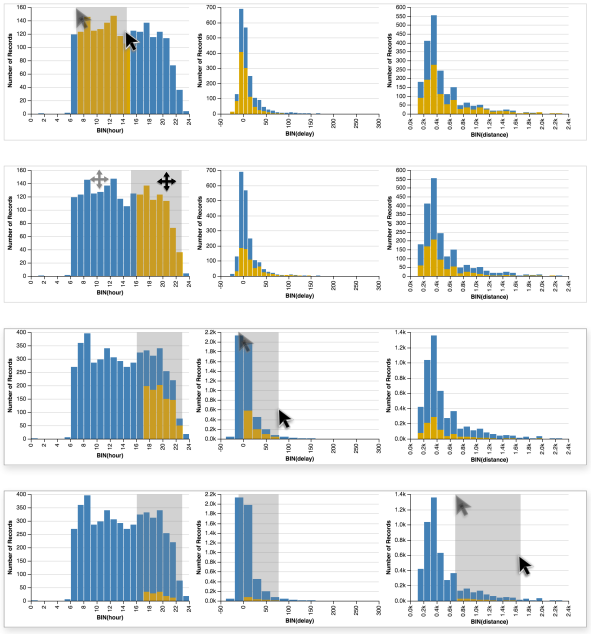


Figure 5.24: A layered cross filtering interaction is constructed by resolving the interval selection to *intersect*, and then materializing it to serve as the input data for a second layer. Highlights indicate changes to the specification from Fig. 5.23.

5.5.7 Limitations

The previous examples demonstrate that Vega-Lite specifications are more concise than those of the lower-level Vega language, and yet are sufficiently expressive to cover an interactive visualization taxonomy. Moreover, we have shown how primitives can be systematically enumerated to facilitate exploration of alternative designs. Nevertheless, we identify two classes of limitations that currently exist.

First, there are limitations that are a result of how our formal model has been reified in the current Vega-Lite implementation. In particular, components that are determined at

compile-time cannot be interactively manipulated. For example, a selection cannot specify alternate fields to bin or aggregate over. Similarly, more complex selection types (e.g., lasso selections) cannot be expressed as the Vega-Lite system does not support arbitrary path marks. Such limitations can be addressed with future versions of Vega-Lite, or alternate systems that instantiate its grammar. For example, rather than a *compiler*, interactions could parameterize the entirety of a specification within a Vega-Lite *interpreter*.

The second class of limitations are inherent to the model itself. As a higher-level grammar, our model favors conciseness over expressivity. The available primitives ensure that common methods can be rapidly specified, with sufficient composition to enable more custom behaviors as well. However, highly specialized techniques, such as querying time-series data via relaxed selections [53], cannot be expressed by default. Fortunately, our formulation of selections, which decouple backing points from selected points via a predicate function, provide a useful abstraction for extending our base semantics with new, custom transforms. For example, the aforementioned technique could be encapsulated in a *relax* transform applicable to multi selections.

While our selection abstraction supports *interactive* linking of marks, our view algebra does not yet provide means of *visually* linking marks across views (e.g., as in the Domino system [38]). Our view algebra might be extended with support for connecting corresponding marks. For example, points in repeated dot plots could be visually linked using line segments to produce a parallel coordinates display.

5.6 Conclusion

To our knowledge, Vega-Lite is the first high-level visualization language to offer a multi-view grammar of graphics tightly integrated with a grammar of interaction. The resulting concise specifications facilitate rapid exploration of design variations.

An early version of Vega-Lite has already been well-received by the broader community. Third-party bindings have been created for a number of environments including Python [4],

R [96, 98], Scala [99], Julia [97], and a REPL client for Clojure [102]. In a widely-shared review of Python visualization libraries, community member Dan Saber noted that *"it is this type of 1:1:1 mapping between thinking, code, and visualization that is my favourite thing about [Vega-Lite]"* [84]. Moreover, members of the Jupyter team have called Vega and Vega-Lite *"perhaps the best existing candidates for a principled lingua franca of data visualization"* [4].

Vega-Lite has also had an impact in the research community. It has been used to reverse-engineer visualizations from chart images [80], build a model for sequencing visualizations [58], and powers the CompassQL recommendation engine [108, 109, 110]. Such work is possible, in part, due to well-tested *effectiveness criteria* [13, 27, 68] for visual encodings. One promising avenue for future work is to use Vega-Lite to derive analogous criteria for interaction techniques.

Vega-Lite is an open source system available at `http://vega.github.io/vega-lite/`. We hope that it enables analysts to produce and modify interactive graphics with the same ease with which they currently construct static plots.

6 | A Visualization Design Environment

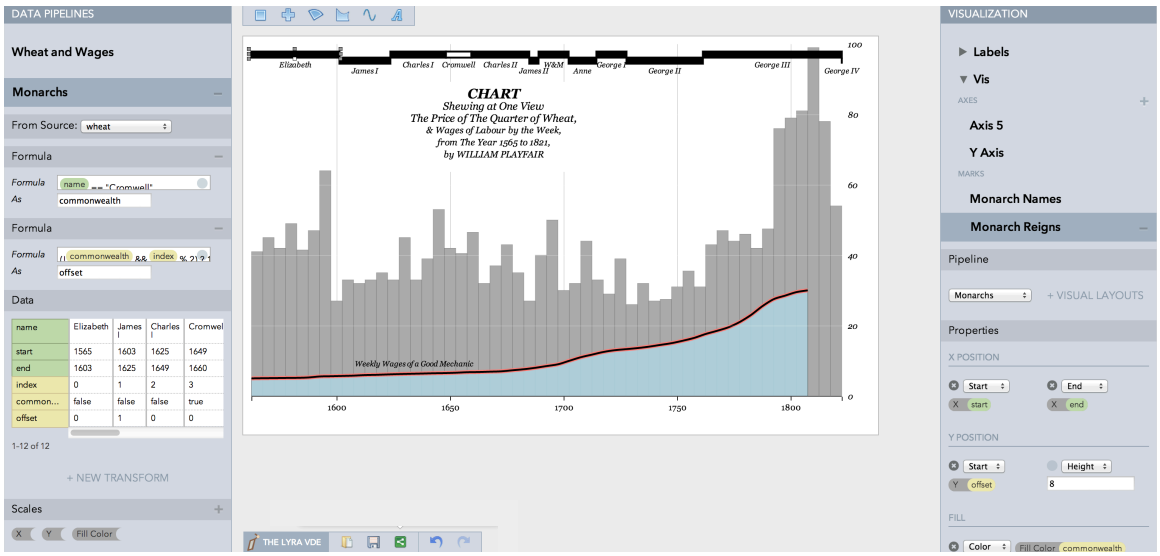


Figure 6.1: The Lyra visualization design environment, here used to recreate William Playfair's classic chart comparing the price of wheat and wages in England. Lyra enables the design of custom visualizations without writing code.

Reactive Vega and Vega-Lite offer JSON syntaxes to facilitate programmatic generation of interactive visualizations within higher-level interactive applications. In this chapter, I explore this nascent space with Lyra: an interactive visualization design environment (VDE). Lyra is motivated by recognizing that Reactive Vega and Vega-Lite present a fundamental mismatch in representations — using *textual* languages to express *visual* output. As a result, with a commensurately poor "closeness of mapping," [15] these languages impose a wide gulf of execution [55]. It can be difficult for users to map desired graphical outputs with the required textual specifications.

Existing graphical applications for visualization design, however, have limited expressivity. At one end are *chart typologies* [107]: pre-defined palettes of chart types (bar charts, line charts, etc.) that make numerous design decisions on behalf of the user. At the other end, vector graphics packages offer designers complete flexibility but provide few (if any) data-driven abstractions [14].

6.1 User Interface Design

Lyra was developed through an iterative user-centered design process. We held formative interviews with representative users, such as visualization designers and journalists, to understand their design process and the limitations of their existing tools. These users evaluated low-fidelity prototypes and later interactive prototypes.

The Lyra interface, as shown in Fig. 6.1, is split into three sections. The left-hand panel (Fig. 6.2(a)) depicts *data pipelines*: chains of data transformations applied to a data source. A pipeline's inspector provides a paginated data table showing the output of the pipeline, buttons to add new transformations, and a list of scales defined over data fields. The right-hand panel (Fig. 6.2(c)) contains inspectors for graphical elements such as marks, axes, and legends. These elements are grouped into *layers* to determine coordinate spaces and z-ordering. These inspectors list all visual properties (position, fill color, angle, etc.) along with widgets to manipulate them. The central panel contains the visualization canvas where graphical elements may be directly manipulated.

6.1.1 Data Pipelines

Lyra's left-hand panel contains *data pipelines*: workflows of transforms applied to input data. Clicking a pipeline reveals an inspector that lists applicable transforms and presents a paginated table view of transformed data.

Data Table View. Data pipelines include a data table view, using a layout inspired by Bret Victor [100]. The first column in the table view lists field names, enabling vertical scanning. Subsequent columns display individual records (Fig. 6.2(a)). Field names in the first

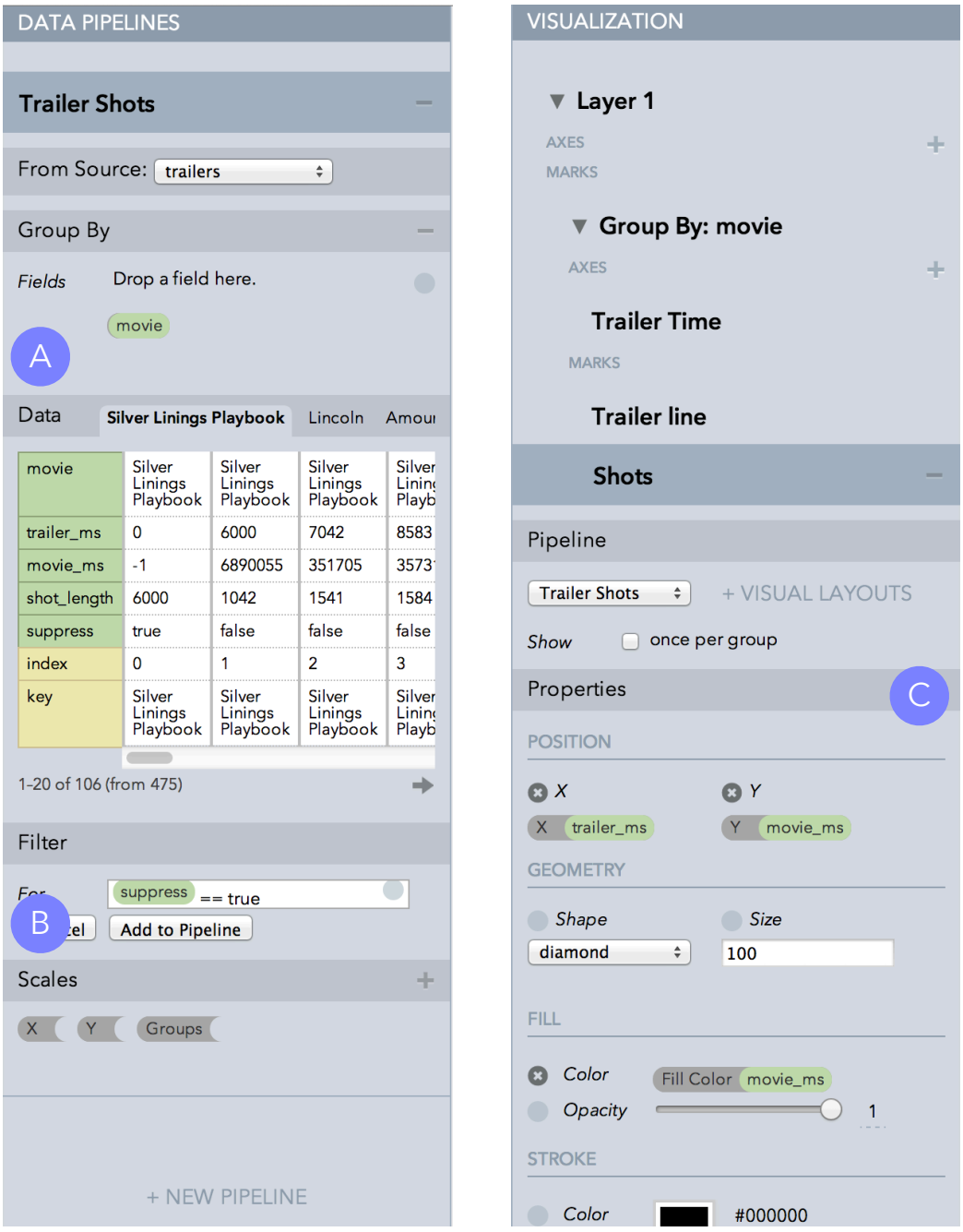


Figure 6.2: Lyra's side panels for data pipelines (left), and visual properties (right). (a) Data table showing the current output of the pipeline; (b) Scale transforms defined over fields in the pipeline. (c) A property inspector for a symbol mark type; two properties have been mapped to data fields.

column are interactive: clicking a field sorts the table by that dimension, drag-and-drop can be used to bind fields to mark properties. Fields are colored by their source: green for fields in the original data and yellow for fields derived by a transform. For example, the *formula* transforms adds new fields based on mathematical expressions. When *group by* transforms are applied, one tab for each group appears above the table view.

Authoring Transforms. Users can add a transform by clicking the corresponding icon and configuring its parameters. Users may preview the effect of applying a transformation in a popover. Once a transformation is added to the pipeline, adjusting its properties is reflected in real-time across the table view and the visualization.

Scales. The inspector also lists all scales defined over data fields in the pipeline (Fig. 6.2(b)). Lyra automatically instantiates scales when a field is associated with a mark property. The scale domain is defined over the field values; the range is determined using production rules described below. Users can also create scales manually. Users can drag scales onto mark properties to apply a scale transform, or click a scale to access an editor dialog (see Fig. 6.3(e)). When editing a scale that is not represented by an axis or legend, a transient guide is shown in the canvas to convey the effect of scale changes.

Design Rationale

Our initial prototypes hid raw data values in favor of exposing only the table schema. However, evaluations indicated this was insufficient. Users noted that it was difficult to determine the effect of a data transformation based solely on the visualization. Moreover, the incremental nature of visualization design can lead to unexpected intermediate output, for example setting the `height` of a rectangle mark can cause all mark instances to overlap if no `x` or `width` property has been set. Later prototypes introduced a full data table view, to enable inspection of raw values and expose the current data organization.

Similarly, early prototypes masked the presence of scales: mapping data to visual properties automatically instantiated a scale, but they were not explicitly exposed in the interface. When users attempted to construct visualizations, we found that this significantly restricted their expressiveness. For example, it is often necessary to specify custom ranges

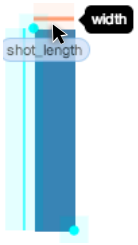
for scales rather than rely on preset ranges. Such modification is difficult to do without surfacing scales as a first-class construct. Later evaluations found that users additionally had trouble identifying the purpose of scales, or the effects of scale modification, if the scales were not explicitly represented on the visualization by an axis or legend guide. In response, we introduced transient guides.

6.1.2 Composing Visual Elements

Visualizations in Lyra are compositions of visual elements: graphical *marks* and *guides*. Elements are grouped together into *layers*, which define local coordinates and establish z-ordering. Lyra's right-hand panel lists the layers and their elements (Fig. 6.2(c)). Elements are added to a visualization by creating them within this panel or by dragging a mark from the mark palette. When an element is selected, an inspector presents all the element's associated properties. Property values may be edited directly or set via drag-and-drop of data fields with changes reflected on the visualization in real-time. Hovering over a property displays a guide overlaid on the visualization to illustrate how that particular property affects the rendered output. Visual elements can also be manipulated directly on the visualization canvas.

Handles in the canvas can be used to interactively move, rotate and resize selected elements. A mark definition will typically render one mark instance per datum in the visualization. To reduce visual clutter, selecting a mark displays handles only on the instance that was clicked. However, when a user adjusts the handles, the change is reflected simultaneously across all mark instances.

Connectors can be used to position marks relative to one-another. Dragging a target mark onto a host mark's connector establishes a connection: the target mark's position is now determined by the host's properties. Changes to the host mark automatically propagate to all connected targets. Connectors are particularly useful for positioning text labels relative to other marks.



Drop zones are used to associate data fields with mark properties. When dragging a data field, drop zones overlay the visualization canvas. Each drop zone comprises a shaded region and a guide line or point to indicate the corresponding mark property (e.g., `x`, `width`, etc.). Hovering on a drop zone highlights it and shows the property name in a tooltip. Dropping a field then establishes a mapping between the data field and the mark property. To avoid clutter, Lyra shows drop zones only for the currently selected item. When dragging a data field, users can hover and pause over a mark instance to make it the selected item.

Design Rationale

Surfacing all properties in the inspector was an immediate first step to ensure that Lyra maintained Vega's expressivity. Users noted that these inspectors were akin to Tableau's "shelves," a familiar interaction paradigm for many of them. However, there remained a clear opportunity to further narrow the gulf of execution [55] by pushing interactions to the visualization canvas itself. For example, we observed users attempting to select, move, or resize marks currently visualized on the canvas.

As users cited familiarity with drawing tools, we sought to reuse familiar interaction mechanisms with *handles* and *connectors*. However, there is not a similarly established interaction mechanism for data-property bindings. We ultimately arrived at our *drop zones* design by prototyping a number of alternatives. One such alternative incorporated flow menus [40]. When dragging a data field over a mark on the canvas, a flow menu would appear listing all mappable visual properties. When dragging the field over a property, a submenu would appear listing appropriate scale types given the type of the data field, and the particular property. For example, for fields with numeric data, this submenu offered all quantitative scale types including linear, logarithmic, and so forth. Dropping the field over a particular scale type established a mapping and instantiated the appropriate scale.

In addition to testing designs with users, we analyzed them using the Cognitive Dimensions of Notation heuristics [15]. Data mapping through flow menus, for example, provided a *visible* and *consistent* interface—regardless of the mark type, all properties were

consistently ordered within the top-level menu. Although exposing scale types in the submenu arguably reduced *error-proneness* (as Lyra need not infer a scale type), it increased the *diffuseness* (or verbosity) of the interface. User feedback also revealed that selecting an option from this submenu was a *hard mental operation* as it forced them to select a particular scale type up front. Many users perceived this as a *premature commitment*. Perhaps most troublesome, given our goal of reducing the gulf of execution, was the lack of a *closeness of mapping*: properties were listed as menu items, one after another.

Drop zones, on the other hand, achieve a high *closeness of mapping* as they overlay the canvas in a way that corresponds to the property they represent. For example, a rectangle's `x2` drop zone is shown extending from the left edge of the canvas to the right-most edge of the rectangle. Dropping a field over a drop zone performs *scale inference* (described below) to reuse an existing scale definition or instantiate a new one. Although this may increase *error-proneness*, it decreases *diffuseness* and reduces the *hard mental operations* flow menus presented. One limitation of drop zones is a subtle lack of *consistency*; for example, a tall rectangle mark will present a larger `height` drop zone than a shorter one. We mitigate this issue by showing drop zones only for the currently selected mark.

6.1.3 Scale Inference and Production Rules

When a user binds a data field to a mark property, Lyra performs *scale inference* in an attempt to reuse existing scale definitions. Lyra searches for an existing scale with the field as its domain. If a scale is found, it is reused if its range type is appropriate (e.g., spatial or color values). If no scale is found or the range type does not match, Lyra instantiates a new scale: ordinal for categorical data or linear for quantitative data, along with a default range based on the property type (e.g. `width` for `x` properties).

To accelerate common encoding decisions, Lyra also uses a set of context- and mark-specific *production rules* to determine intelligent defaults. These production rules may set additional properties of the mark or add new graphical elements to the canvas. For example, dropping a field over a rectangle mark's `width` drop zone automatically binds the `x` property as well to correctly position each rectangle. Dropping a field over a spatial property

may add an axis; dropping a field over a color property may add a legend. A user can customize these defaults using the property inspector. However, users may occasionally wish to sidestep the production rules. Accordingly, Lyra evaluates production rules only when a mapping is established using drop zones. If the user instead drops a data field onto the property inspector panel, no production rules are evaluated and so no defaults are added.

Design Rationale

Scale inference and production rules were informed primarily by early user feedback. Without these features, users had to manually create every aspect of the visualization, which they found to be tedious. Users did not expect to have to specify a scale definition on every data mapping operation, and expected axes or legends to be automatically added as appropriate. We found that the features did alleviate this tedium but, interestingly, users subsequently requested a method of circumventing them *"if they knew better"*. As a result, although Lyra performs scale inference on every data field mapping, production rules are only evaluated if the data field is dropped over a drop zone. Users may sidestep the rules by working directly with property inspector instead. We fully enumerate Lyra's scale inference procedure and production rules in supplementary material.

6.1.4 Saving and Exporting Visualizations

Visualizations built in Lyra can be exported as static PNG or SVG files, or as Vega JSON specifications. With Vega's JavaScript runtime, the JSON file can be parsed and rendered on a web page, dynamically bound to new input data, and extended with interactions.

6.2 Implementation Details

Lyra is a web-based HTML5 application built using AngularJS[1]. Vega is used extensively throughout the system both to parse data transformations and to represent and generate visualizations. While close, the mapping between Lyra and Vega abstractions is not one-to-one. To reduce complexity, Lyra consolidates some Vega mark types. A line mark in

[1] http://angularjs.org

Lyra translates to one of Vega's line, path, or rule marks based on which properties and fields the designer chooses to map. Similarly, Vega's image marks are simply rectangle marks in Lyra with an "image fill."

Lyra also simplifies Vega's handling of hierarchical data. In a Vega specification, each level of a hierarchy is assigned to a "group" mark, with children marks inheriting the corresponding subset of data. Lyra insulates users from this scheme by automatically performing *group injection*: when a user assigns a mark to a pipeline containing hierarchical data, Lyra automatically adds and nests the necessary group marks for each level of the hierarchy in the resulting Vega specification. Group injection makes building small multiples easy: an exposed `layout` property lets designers choose between having groups overlap, layout horizontally, or layout vertically.

Lyra's direct manipulation interactors (handles, connectors, drop zones) are also generated using Vega, using a separate specification rendered *over* the visualization. Mark geometry serves as input data for this interactive layer. Decoupling the visualization and interactors ensures that Lyra features do not interfere with the designer's visualization.

6.3 Usage Scenario

Dissecting a Trailer [23] is a visualization from The New York Times that illustrates how scenes from five Best Picture Oscar nominees were edited into trailers. It is an example of a visualization that cannot be built using existing chart typologies or high-level grammars. The visualization layers several mark types and uses a non-standard "scatterplot" with rectangles of varying widths. The original consists of over 350 lines of JavaScript/D3 [17] code. Figure 6.3 demonstrates how it can be recreated in Lyra.

We introduce lines by dragging a *line mark* from the palette at the top of the screen and dropping it on the canvas (Fig. 6.3(a)). This adds a new line (backed by a single datum) to the current layer and associates it with a new data pipeline. The default pipeline is empty, as we have not yet specified a data source. To register a new source, we provide a name and

Figure 6.3: Using Lyra to recreate the New York Times' Dissecting a Trailer. (a) Drag a line *mark* onto the canvas. (b) Drag a field from a *pipeline*'s data table to a *drop zone* to map it to a mark property. (c) Add a "group by" *data transform* to create a hierarchy. (d) Edit a *scale* definition to reverse the range. (e) Use a *connector* to anchor text marks to the rectangles.

a URL to our trailer shots data and, on load, review the inferred data types (number, string, etc.) for each data field. The output of the pipeline is shown in the data table (Fig. 6.3(b)).

We bind data in the pipeline to visual properties of the line mark via drag-and-drop of data fields. Dragging triggers the display of *drop zones* overlaid on the visualization canvas. Dropping a data field on a zone establishes a visual encoding (Fig. 6.3(b)). We drag `trailer_ms` and drop it over the line's `x` property, then drop `movie_ms` over `y`. These actions result in line segments connecting every data point.

However, we desire separate line charts per film. To divide the data, we add a *group by* transformation to our pipeline (Fig. 6.3(c)), keyed on the movie title. The data table reflects the result via tabbed groups, and the mark's property inspector now offers an option for group `layout`. We choose a `vertical` layout to produce one line per movie, arrayed down the canvas. We now see that the data includes shots that appear in trailers but not in movies (identified by the `suppress` field); we add a *filter* transform to remove them (Fig. 6.3(d)). We want film timelines oriented top-to-bottom, but our lines curently show the reverse. We adjust the orientation of the y-axis *scale* by reversing its range (Fig. 6.3(e)).

To add the small rectangle plots, we drag a *rectangle mark* onto the canvas. Lyra automatically associates the mark with the current pipeline and the visualization now shows one rectangle per shot. As with the line mark, we drag `trailer_ms` and `movie_ms` to the rectangle's `x` and `y` property drop zones. We size the rectangles by dragging `shot_length` over the `width` drop zone and dragging the bottom *handle* to manually set the desired height (Fig. 6.3(f)). To color rectangles based on when the shot occurs in the film, we drag `movie_ms` over the `fill` property drop zone, and choose custom colors in the resulting scale definition dialog.

The final step is to create labels and background rectangles to identify the movies and their beginning, middle, and end sections. As these elements should reside in the background, we create a new layer in Lyra. Movie section information is drawn from a different data source, so we create a new pipeline. We then drag a rectangle mark onto the canvas, and drag the section `start` and `end` fields onto the rectangle `y` and `y2` properties, resulting in a vertical layout for beginning, middle, and end sections. We bind the `label` field to the

fill color property and select custom colors for the resulting scale mapping. To label each section, we drag a *text mark* from the palette and drop it on a rectangle's diamond-shaped *connector* (Fig. 6.3(g)). Doing so anchors the text mark coordinates to the rectangle. Finally, we bind the `label` field as textual content and set the text fill color. We now can export the visualization as either a Vega specification or an image file.

6.4 Example Visualizations

One of Lyra's primary goals is to enable an expressive design space. With Lyra, it should be possible to create visualizations that would have previously required programming. To assess the extent to which this goal has been met, we constructed a diverse collection of example visualizations, including those shown below. These examples compose multiple mark types, and many require multiple data pipelines. For example, Fig. 6.7 uses line, symbol, and text marks to convey two datasets: train routes and stations. Fig. 6.11 demonstrates that Lyra's integration of data pipelines and graphical manipulation is necessary to maintain expressiveness: shading the bars requires a data pipeline with *Group By* and *Formula* data transformations applied.

Similarly, exposing visual layout inspectors allows for rapid design iteration. In Fig. 6.6, for example, the *force-directed layout* inspector exposes parameters such as link distance, strength and gravity; adjusting them re-renders the layout in real-time. The layout also augments direct manipulation on the canvas: designers can brush to select nodes and double-click to pin them. Together, these facilitate a converging process: pinning satisfactory nodes, adjusting layout properties, and re-running the layout to reposition unpinned nodes. This process would be cumbersome using only D3's force-directed layout [17]: after programming the layout, adjusting parameters requires editing the code and refreshing the browser. Pinning nodes then requires inspecting the properties of each rendered node individually and copying the x and y positions into the raw dataset.

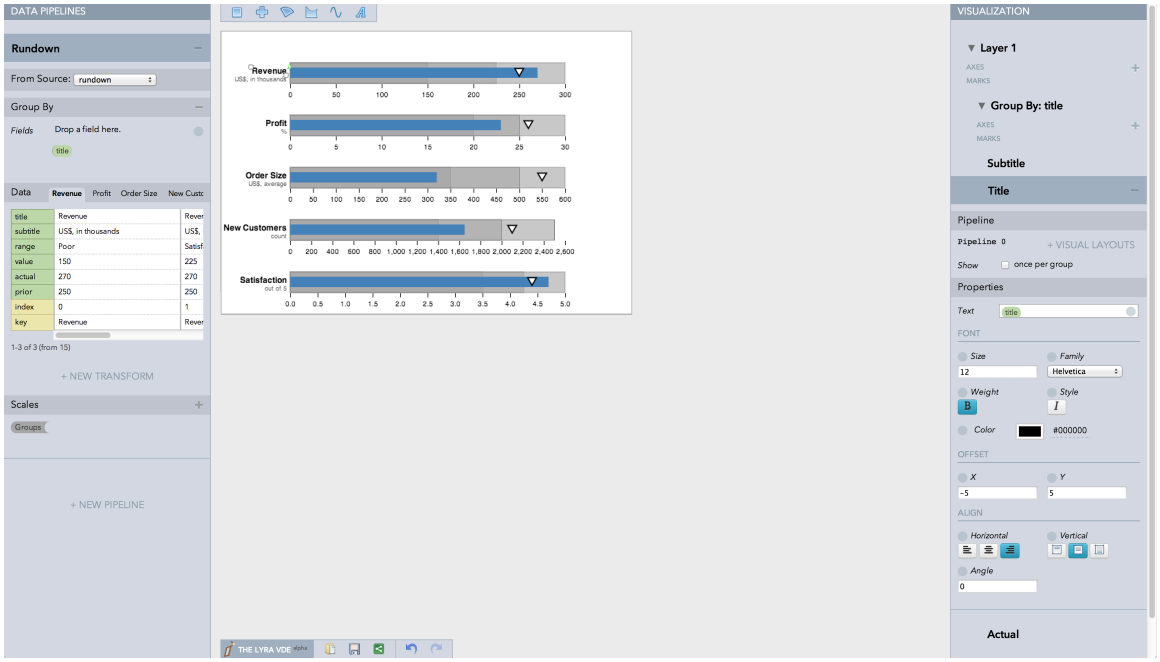


Figure 6.4: Bullet chart using rectangle and symbol marks grouped by category. Labels are positioned via a left-edge connector on rectangles.

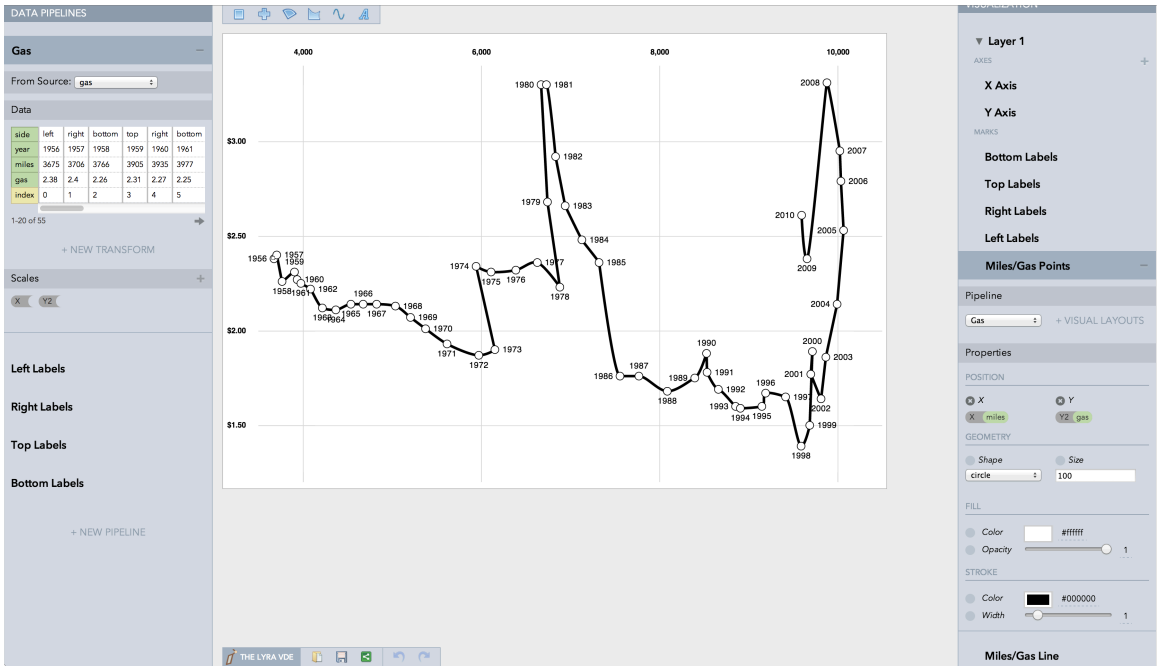


Figure 6.5: A recreation of *Driving Shifts Into Reverse* by Hannah Fairfield from The New York Times, originally published May 2, 2010.

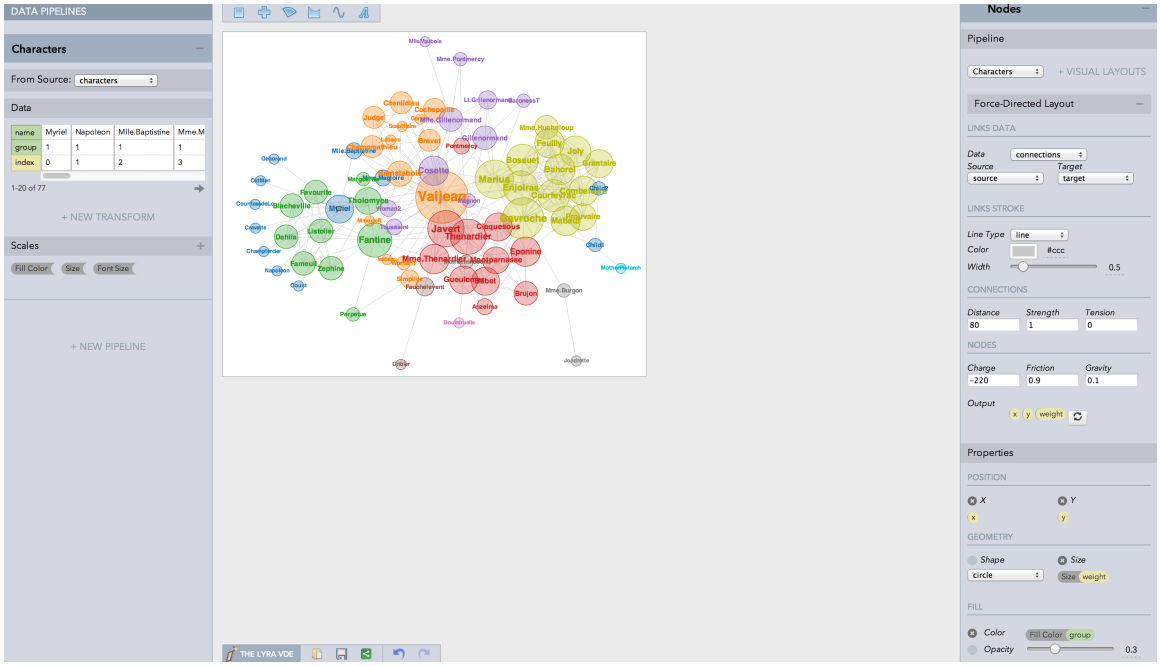


Figure 6.6: Character co-occurrences in Les Misérables. Colors represent cluster memberships computed by a community-detection algorithm.

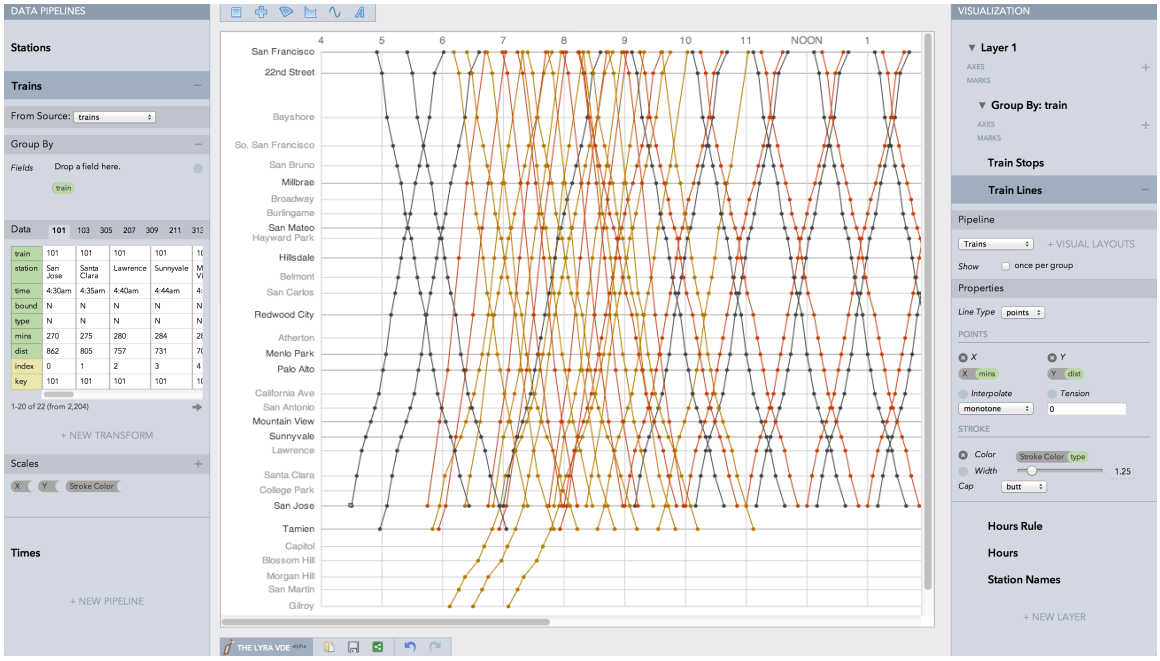


Figure 6.7: The schedule of the San Francisco Bay Area's CalTrain service in the style of E. J. Marey's Paris train schedule.

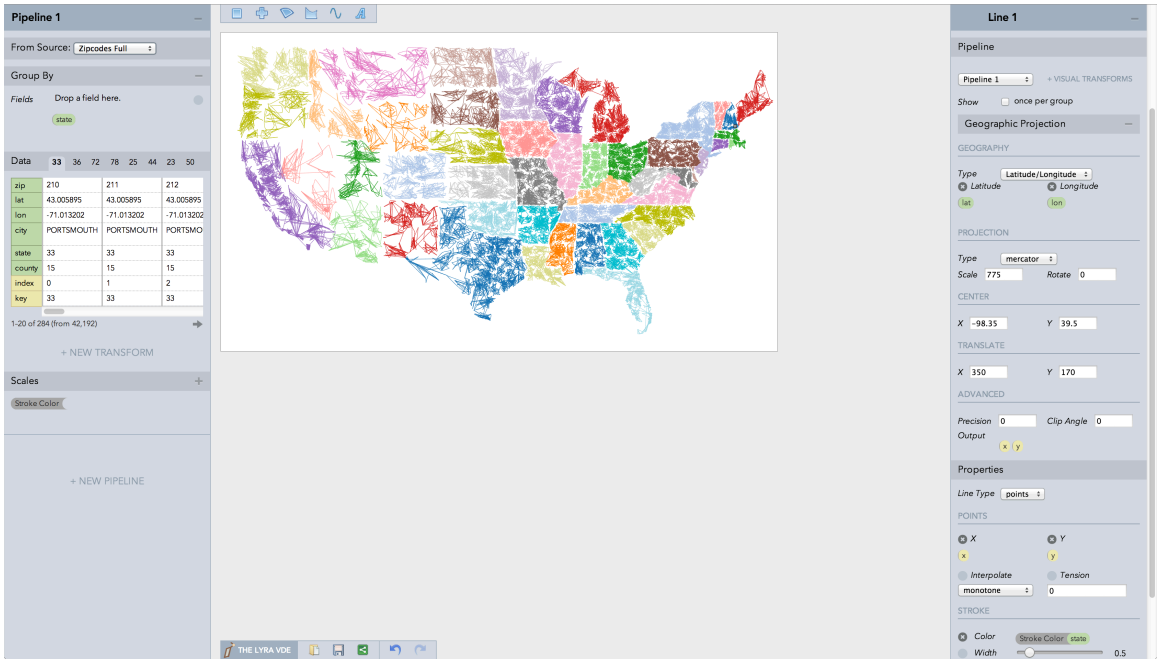


Figure 6.8: ZipScribble by Kosara [61]. A *geo* layout encoder is used with line marks to connect latitude and longitudes of zip codes.

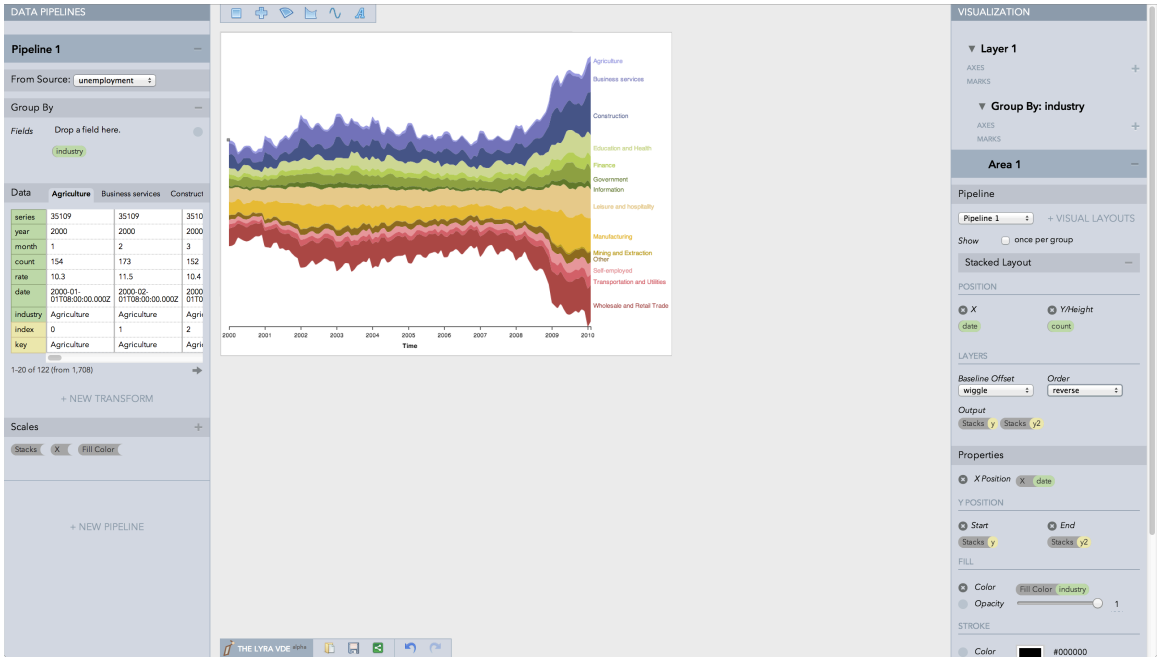


Figure 6.9: A streamgraph of unemployed U.S. workers by industry, using a *stack* layout with a `wiggle` offset [20].

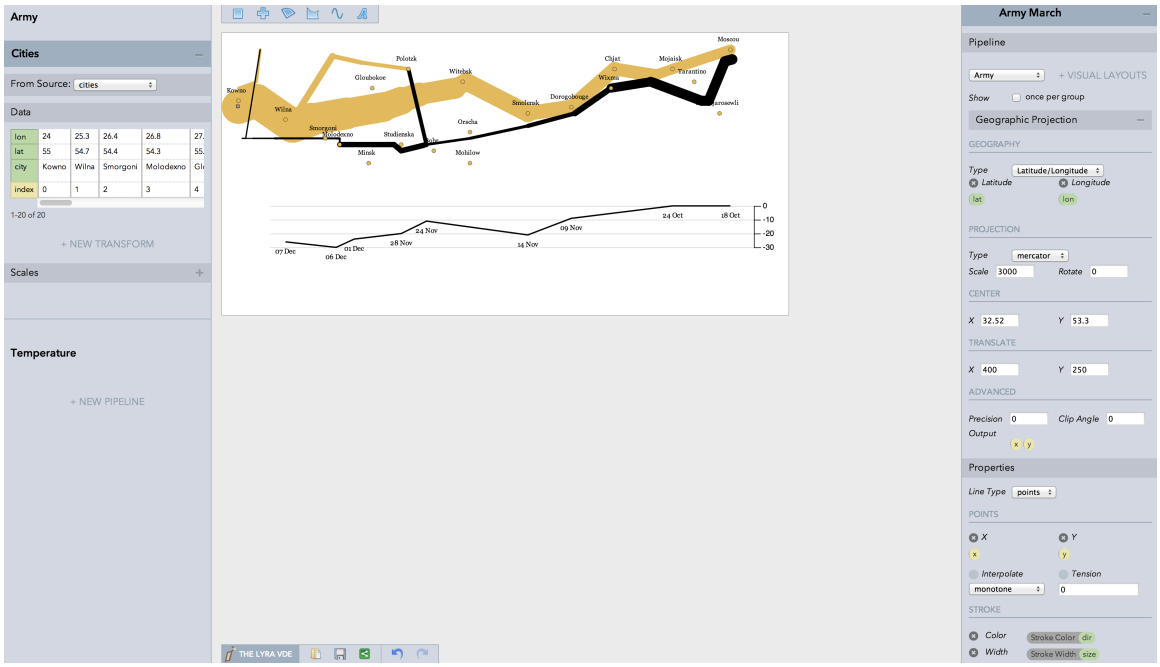


Figure 6.10: Minard's map of Napoleon's Russian campaign. A *geo* transform encodes spatial positions; army size maps to line stroke width.

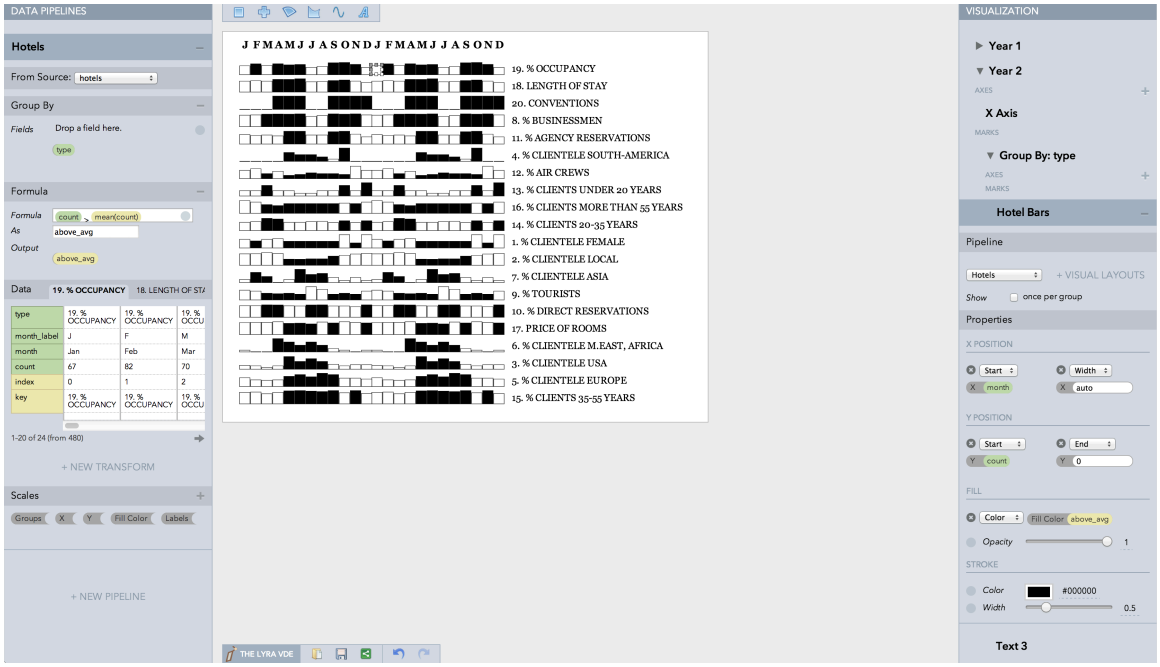


Figure 6.11: Jacques Bertin's analysis of hotel patterns. *Group by* and *formula* transforms are used to shade bars with values above the mean.

6.4.1 Limitations

Figures 6.3 to 6.11 demonstrate that Lyra enables an expressive design space, but creating these examples also reveals some limitations. Vega currently lacks support for polar coordinates. As a result, Lyra cannot (yet) provide *arc* mark connectors or produce radial axes, making it difficult to recreate classic visualizations such as Nightingale's Rose or Burtin's antibiotics chart. Additionally, Lyra only supports the RGB color space, while Vega also supports HSL, LAB, and HCL. These color spaces facilitate perceptually-sound designs. We plan to address these limitations in future versions of Vega and Lyra.

6.5 Formative User Evaluations

Lyra was designed to support both expressive and *accessible* visualization design: users should not require coding expertise to be able to construct custom visualizations. To evaluate Lyra's accessibility, we conducted first-use studies with 15 representative users including 6 data analysts / visualization designers, 5 data journalists, and 4 graduate students in data visualization. On 10-point scales, the median self-reported visualization design expertise was 7, while programming expertise range between 2–8. These users all use visualization as a communicative medium but their processes for creating them vary. The visualization designers and grad students were more technically proficient and typically use D3, whereas the data journalists rely on chart typologies (Microsoft Excel) or grammar-based systems (Tableau) that do not require programming. Some journalists also reported eschewing visualization systems in favor of drawing programs such as Adobe Illustrator.

6.5.1 Methods

We began each study with a 10 minute tutorial. We then asked participants to design three graphics: a bar chart of medal count by country at the 2012 Olympics (T1), a grouped or stacked bar chart of medal counts by medal type and country (T2), and a trellis plot of barley yields (T3, Fig. 6.12). These tasks were designed to ensure participants interacted with all aspects of Lyra. Each task was more difficult than the previous, intending to first

familiarize participants with the Lyra design process, and then challenge them. Participants were encouraged to think-aloud and were de-briefed at the end. Sessions lasted approximately 45 minutes, after which we administered a post-study survey.

6.5.2 Successes

Users quickly learned Lyra's interaction model and all users, regardless of their technical expertise, successfully completed all three tasks with minimal guidance (100% task completion rate). Users completed the first two tasks in just a few minutes, the more complex third task took longer (T1: median time = 1:33, inter-quartile range (IQR) = 0:51; T2: median = 2:43, IQR = 2:57; T3: median = 10:24, IQR = 4:00). In a post- study survey, users rated Lyra's interface highly: drop zones felt natural to use ($\mu = 4.4$, $\sigma = 0.57$ on a 5-point Likert scale), connectors helped to relatively position marks ($\mu = 4.3$, $\sigma = 0.49$), and a pipeline's data table helped evaluate context ($\mu = 4.4$, $\sigma = 0.51$). Handles were found useful for resizing and positioning ($\mu = 3.8$, $\sigma = 0.45$) but users noted that the properties they control are typically mapped to data. When asked to recount their experience, users described drop zones as

Figure 6.12: Study participants recreated the barley yields Trellis display [12].

"natural" and *"intuitive."* One user stated, *"it's like literally saying 'put that there.'"* Others drew comparisons to Tableau's shelves: *"[shelves] don't always behave like I expect them to but [drop zones] make me feel more in control."* One participant ended his session by saying that *"there's a real joy in using Lyra."*

Two journalists who participated lead data visualization teams in their organizations. They appreciated that Lyra took cues from familiar drawing tools. They welcomed Lyra's image export options, particularly SVG export, as the visualizations they produce are often reutilized in print media. One suggested that Lyra could be a powerful training tool that could help familiarize his team with the process of designing visualizations from the ground-up.

Users familiar with lower-level tools such as D3 found Lyra useful for rapidly prototyping and iterating on their design. For example, one user (self-rated expertise with D3 as 6.5/10) used Lyra with her own data. By her estimate, she had previously spent between 4–6 hours repurposing an existing D3 example to create a custom visualization. She was able to create a close approximation in Lyra, shown in Fig. 6.13, in only 10 minutes.

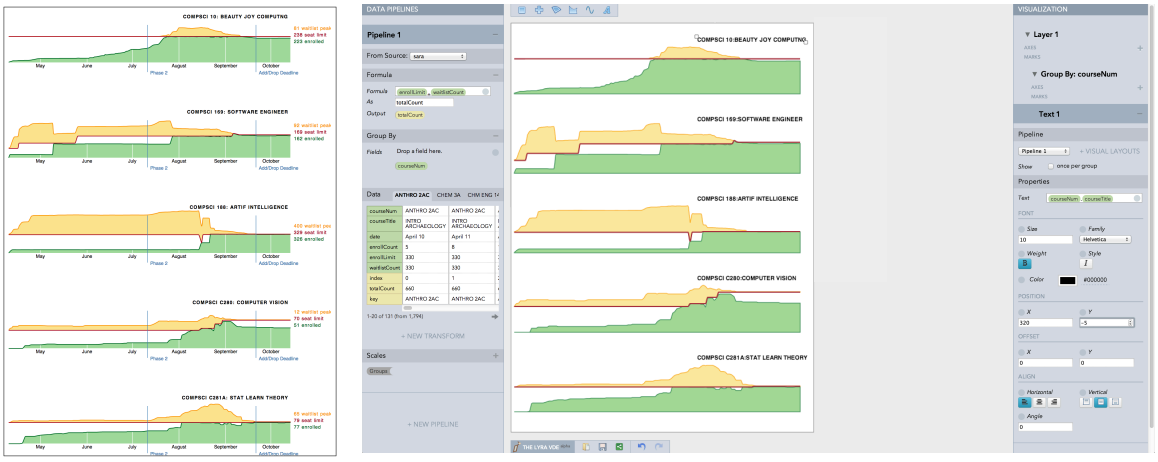


Figure 6.13: A study participant approximately recreated a D3 visualization (left, requiring 4-6 hours) in Lyra (right, requiring only 10 minutes).

6.5.3 Shortcomings

We also observed that Lyra posed certain challenges for our participants. Although users found drop zones natural and intuitive, they noted problems with the current implementation. First, when users missed a drop zone by a few pixels, they expected Lyra to infer their intent. Second, when users successfully dropped a field, they would lose track of the currently selected mark if it was repositioned. A third shortcoming was Lyra's lack of support for undo, which led users to become more hesitant to freely explore. Undo support has since been added to Lyra, and we plan to address the remaining issues in future versions. For example, a Voronoi diagram could be used to find the nearest drop zone [39], and staggered animated transitions could help users better track changes [49] to marks.

Finally, several users mentioned that learning from and repurposing existing visualizations is an important part of their design process. They found the blank canvas to be an intimidating starting point. We anticipate that providing a gallery of examples (including those in this paper) that users can import, reuse and modify would mitigate this issue.

6.6 Reflections & Future Work

Chronologically, Lyra was the first project to form part of this dissertation. At the time of writing, a new version, that leverages Reactive Vega and Vega-Lite, is under development. Thus, it is worth reflecting on how its role in the ecosystem has evolved.

Much of Lyra's existing functionality can now be driven purely via a Reactive Vega specification rather than through external callbacks. For instance, direct manipulation operations on handles are now driven entirely via signals, rather than event handling callbacks. Similarly, connectors make use of reactive geometry directly. A bubble cursor [39] UDF (see § 3.1.4) is registered to accelerate drop zone selection, as shown in Fig. 6.14, addressing a major shortcoming users identified previously.

Moreover, Lyra's built-in production rule system has been subsumed by Vega-Lite. Now, when users drag a data field onto a drop zone, a Vega-Lite unit specification is compiled,

the resultant Vega specification is statically analyzed, and components selectively merged or updated in Lyra's backing Vega specification.

With these changes, Lyra's place in the tool stack comes into sharper relief: it *bridges* two levels of abstraction within a single, cohesive environment. Direct manipulation operations occur at the Vega-Lite level, and allow users to rapidly generate recognizable visualizations. For more fine-grained manipulation, or for custom design elements, users drop down to the Vega level by interacting with the visual inspectors. A critical advantage of this approach is that users can fluidly work with the level of abstraction best suited for the task at hand. In fact, they may be altogether unaware of the separate roles played by Vega and Vega-Lite under-the-hood!

Looking ahead, an exciting challenge is how Lyra might be extended to support the design of *interactive* visualizations. In-keeping with the above approach, perhaps direct manipulation interactions (or demonstrations) should generate Vega-Lite selections, which can be further modified with signal and predicate inspectors.

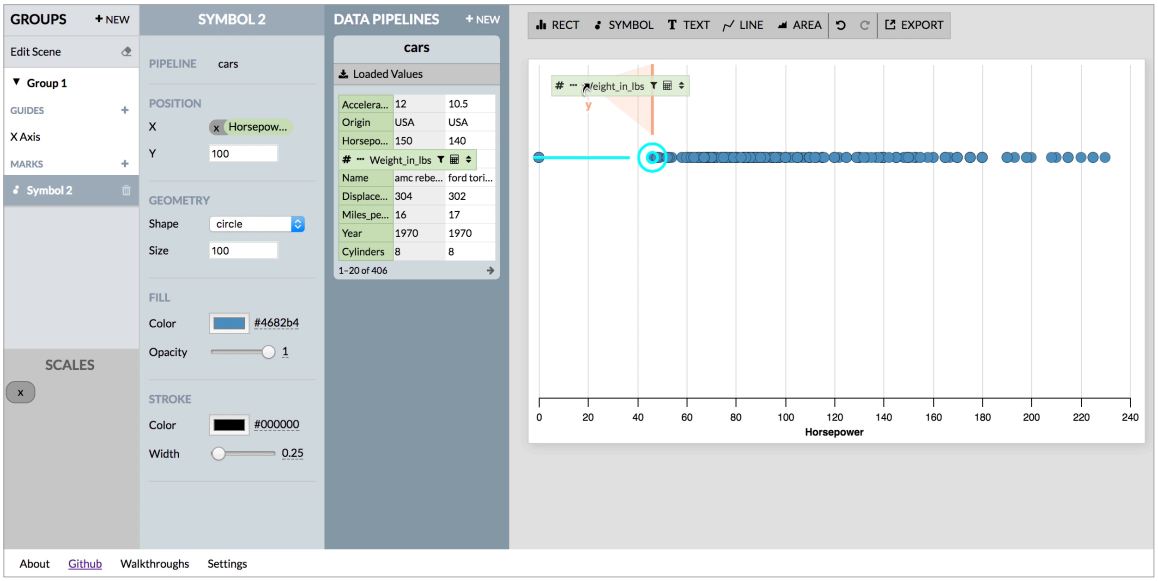


Figure 6.14: The new Lyra interface. The side panels from Fig. 6.2 have been redesigned and consolidated to the left-hand side. As the user drags a data field across the canvas, the nearest drop zone (highlighted in orange) is automatically selected — a technique known as a bubble cursor [39].

Similarly, while Lyra's primary focus is currently on design tasks, data visualization inevitably requires data cleaning and transformation. Lyra's data pipelines offer sufficient flexibility to support analytic tasks, but require familiarity with transformation operators. Can direct manipulation methods, akin to those in Data Wrangler [56], be further incorporated to specify more complex data transformations?

7 | Conclusion

Existing systems have popularized *declarative* visualization design — users describe *what* a visualization should look like rather than *how* it should be computed and rendered. This decoupling of specification and execution facilitates a more iterative design process, as users can focus on design decisions while deferring implementation concerns to the system runtime. However, existing systems provide poor support for interaction, despite it being central to effective visualization [79, 111]. Palettes of common techniques are sometimes offered but limit customization and composition, and users are forced to program complex event handling callbacks for custom behaviors. As a result, users must either invest significant effort in interaction design, or forgo it and risk missing critical insights. Moreover, existing declarative visualization models are typically embedded within programming languages. This approach exhibits a poor *closeness of mapping* [15] as users must map *textual* commands to their desired *visual* output.

This dissertation addresses these issues through two new declarative visualization languages, Reactive Vega and Vega-Lite, and a new interactive design system, Lyra. Reactive Vega reifies four key insights: modeling user interaction (e.g., mouse clicks or touch gestures) as composable data streams; constructing named reactive expressions, called signals, from these streams; lifting signals from the pixel level to the data domain, to coordinate multiple visualizations; and, finally, endowing existing visual encoding primitives with reactive semantics. This approach is suitably expressive to cover an established taxonomy of interaction methods for data visualization [111] and extends the benefits of declarative specification to interaction design as well. Users need only describe how input

events map to interactive state changes. The Reactive Vega runtime constructs the requisite dataflow graph and transparently optimizes processing to yield performance that meets or exceeds the current state of the art. And, by decoupling input events from downstream logic, signals simplify retargeting an interaction technique to alternate modalities.

Vega-Lite moves a step higher in the ladder of abstraction. Whereas Reactive Vega offers separate primitives for capturing events, building reactive expressions, and constructing data queries, Vega-Lite encapsulates all this functionality within a single abstraction called a selection, and offers extensible selection transformations for further customization. As a result, Vega-Lite specifications are at least an order of magnitude more concise than their Reactive Vega counterparts. Moreover, selections and transformations decompose interaction design into semantic units that can be systematically enumerated. Thus, not only can users rapidly explore alternative designs, but higher-level reasoning tasks (e.g., visualization recommendation [109]) are now more tractable as well.

Critically, Reactive Vega and Vega-Lite are instantiated through JSON syntaxes to facilitate programmatic generation of interactive visualizations in novel interactive data systems. This dissertation explores this higher-level application space with Lyra, an interactive visualization design environment (VDE). Users bind data values to mark properties through drag-and-drop operations, with each such interaction generating a statement in either Vega-Lite or Reactive Vega. Users are able to author a diverse range of visualizations without writing a single line of code and, during formative studies, reported *"[feeling] more in control"* of the design process and *"a real joy in using Lyra"*.

7.1 Future Directions in Interactive Data Systems

The Reactive Vega stack has been released as open-source software, and has seen widespread adoption among the visualization and data science communities — Vega-Lite can be used within a Jupyter notebook [4] and Reactive Vega visualizations can be embedded within Wikipedia articles [70]. As they share the same underlying foundation, these tools form a new *interoperable ecosystem* of interactive visualization systems. Users can, for instance,

construct an exploratory visualization in a Jupyter notebook, export it to Lyra via Vega-Lite, add an explanatory annotation layer, and embed the resultant Reactive Vega specification within a Wikipedia article. Researchers have also been adding to this ecosystem with the Voyager visualization recommendation browser [108, 109, 110], and tools for sequencing visualizations [58], and reverse-engineering them from chart images [80].

Thus, rather than a single monolithic system, the Reactive Vega stack supports the development of applications targeted towards specific tasks, and allows users to work at the level of abstraction most suited for the task at hand.

7.1.1 Automated Design & Inference over Interactive Visualizations

These systems are only an initial exploration of the space of higher-level interactive data systems, and there is a fertile ground to study how inference procedures can be used to accelerate analysis and design. An immediate next step is to study how Lyra can be extended to support interaction design by direct manipulation. Analogous to Lyra's existing drag-and-drop interactions, perhaps users *demonstrate* a desired interactive behavior. A new *interaction inference* architecture will need to be developed, akin to Lyra's existing scale inference procedures, to interpret these demonstrations. Initially, heuristics may suffice to simply map demonstrations to Vega-Lite selections; users would still be responsible for parameterizing visual encodings with the inferred selections. For instance, if a user clicked multiple points that shared a common data value, Lyra might suggest a `multi` selection with a `project` transform, which could then be dragged-and-dropped over the `color` dropzone to establish a linking interaction. Over time, Lyra would learn from user behavior to improve this inference — for example, suggesting commonly used selections after fewer demonstrations, or even recommending fully formed interaction techniques such as the aforementioned linking.

Similarly, Reactive Vega and Vega-Lite provide representations that are readily amenable for design mining techniques [62]. Corpora of interactive visualization designs can be

assembled via reverse-engineering [80] or deconstruction [43, 44], and then mined to codify best practices and identify design trends. This information could then enable "autocomplete" design suggestions (e.g., automatically coloring axis labels based on mark colors in a dual-axis chart), or improve designs with a visualization "linter" (e.g., ensuring bar charts always begin at a zero baseline). Once deployed, such systems could use *online* design mining algorithms to surface *emergent* trends.

7.1.2 Scalable Interactive Visualization

Besides insights on data stream management, the database literature offers additional methods that can be applied to Reactive Vega's architecture to better support interactive visualization of large-scale data. Namely, Reactive Vega's dataflow graph instantiates a query plan that can be optimized — a well-studied subject in database systems [37, 90]. With visualization workloads, it is likely that queries issued as a result of user interaction share common structure or can reuse previously calculated results. Thus, techniques in multi-query optimization [83] (e.g., rewriting queries using views [42]) and adaptive query processing [7, 33] seem most promising.

Moreover, a key property of the Reactive Vega stack is that its abstraction models are *layered*: high-level Vega-Lite specifications map to low-level dataflow computational units. As a result, new architectures, such as the one proposed by Moritz et al. [72], can be developed to dynamically partition computation between a client and server, enabling low-latency interactive visualization of large-scale data. For example, by reasoning about Vega-Lite semantics, such an architecture could automatically reuse data already loaded on the client for a zooming interaction, prefetch the previous/next batches of data for panning operations, or adapt the resolution of server-side aggregation when the viewport size is changed.

7.1.3 Evaluating Expressivity and Usability

This dissertation evaluated the expressiveness of the Reactive Vega stack through example interactive visualizations. A more rigorous analysis of the design space, like the morphological analysis of input devices conducted by Card et al. [21], would identify new points

(e.g., panning color legends as described in §5.5.2) and highlight opportunities to further refine the models. For example, statistical displays such as error bars and box plots can now be constructed by composing several Vega-Lite unit specifications. To simplify these common use cases, one could extend Vega-Lite's existing mark types with new *composite* types that function like macros. However, the semantics of selections against composite mark types remains an open question. For instance, what does a single selection of a box plot mean? Is the whole box plot itself selected, or the underlying aggregated values? How might we specify selecting constituent components of the box plot, (e.g., a specific quartile or whisker) when they cannot be addressed with the project transform?

Similarly, while the Cognitive Dimensions of Notation [15] provide a useful set of heuristic to bootstrap evaluating the usability of the systems, more formal and longitudinal user studies are necessary to assess the cognitive burden these tools impose. Are new users able to learn the declarative interaction design model? Can expert users, long-accustomed to event handling callbacks, adapt quickly? What pitfalls do the two user classes experience? Such studies can also lead to the development of new scaffolding to support users. For instance, the Reactive Vega dataflow graph offers an execution model that can be readily visualized, and new instrumentation to enable inspection and stepping through changeset propagation could aid learnability [41]. Initial work here has been encouraging: a "time-traveling" debugger, developed by Hoffswell et al. [52], was found to help first-time Vega users accurately identify the source of errors, with some event attempting to fix them.

7.1.4 A Science of Interaction

Developing a generalized theory of interaction — one that answers questions such as what makes an interaction technique more effective than another, or what are principles for combining multiple techniques that preserve their individual advantages — has been difficult because existing empirical evaluations of interactions have been conducted largely in an ad-hoc manner. This is due, in part, to representations of interaction that have obscured how to isolate properties of a behavior as experimental variables.

The Reactive Vega stack offers a compelling opportunity here. For a constant Vega-Lite visual encoding, we can systematically generate interaction techniques and vary their constituent properties. These alternative designs could then be classified using a taxonomy of analytic tasks [18] and tested with human subjects. The results will inform our understanding of the costs and benefits of interactive methods — for example, are specific interactive formulations better suited for particular tasks — and spur the development of design guidelines, much as graphical perception studies have done for visual encodings.

7.2 Concluding Remarks

Solving a problem simply means representing it so as to make the solution transparent.

The Sciences of the Artificial
Herbert Simon, 1981

The lack of unified support for interaction design in declarative visualization systems has not only imposed an artificially high technical burden on end-users, but has also hindered researchers' ability to study interaction as a first-class citizen of visualization design, and develop interactive data systems. This dissertation charts a way forward with the Reactive Vega stack, available as open-source software at `http://vega.github.io/`.

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, volume 5, pages 277–289, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal–The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[3] G. Abram and L. Treinish. An Extended Data-Flow Architecture for Data Analysis and Visualization. In *Proceedings of the 6th conference on Visualization*, page 263. IEEE Computer Society, 1995.

[4] Altair: Declarative Visualization in Python. `https://altair-viz.github.io`, August 2017.

[5] Apparatus: A hybrid graphics editor and programming environment for creating interactive diagrams. `http://aprt.us/`, August 2017.

[6] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. STREAM: The Stanford Data Stream Management System. In *Data Stream Management*, pages 317–336. Springer, 2016.

[7] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. *ACM SIGMOD Record*, 29(2):261–272, 2000.

[8] G. J. Badros, A. Borning, and P. J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, 2001.

[9] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive programming. *ACM Computing Surveys (CSUR)*, 45(4):52, 2013.

[10] L. Battle, R. Chang, and M. Stonebraker. Dynamic generation and prefetching of data chunks for exploratory visualization. In *IEEE InfoVis Posters Track*, 2014.

[11] M. Beaudouin-Lafon. Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 446–453. ACM, 2000.

[12] R. A. Becker, W. S. Cleveland, and M.-J. Shyu. The Visual Design and Control of Trellis Display. *Journal of Computational and Graphical Statistics*, 5(2):123–155, 1996.

[13] J. Bertin. *Semiology of Graphics: Diagrams, Networks, Maps*. University of Wisconsin press, 1983.

[14] A. Bigelow, S. Drucker, D. Fisher, and M. Meyer. Reflections on how designers design with data. In *Proceedings of the international working conference on Advanced Visual Interfaces*, pages 17–24. ACM, 2014.

[15] A. F. Blackwell, C. Britton, A. Cox, T. R. Green, C. Gurr, G. Kadoda, M. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, et al. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Cognitive Technology: Instruments of Mind*, pages 325–341. Springer, 2001.

[16] M. Bostock and J. Heer. Protovis: A Graphical Toolkit for Visualization. *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Proc. InfoVis)*, 15(6):1121–1128, 2009.

[17] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-Driven Documents. *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Proc. InfoVis)*, 17(12):2301–2309, 2011.

[18] M. Brehmer and T. Munzner. A Multi-Level Typology of Abstract Visualization Tasks. *IEEE Transactions Visualization and Computer Graphics (TVCG) (Proc. InfoVis)*, 19(12):2376–2385, 2013.

[19] Brunel Visualization. `https://developer.ibm.com/open/brunel-visualization/`, June 2016.

[20] L. Byron and M. Wattenberg. Stacked graphs – geometry & aesthetics. *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Proc. InfoVis)*, 14(6):1245–1252, 2008.

[21] S. K. Card, J. D. Mackinlay, and G. G. Robertson. A morphological analysis of the design space of input devices. *ACM Transactions on Information Systems (TOIS)*, 9(2):99–122, 1991.

[22] S. K. Card, T. P. Moran, and A. Newell. An engineering model of human performance. *Ergonomics: Psychological mechanisms and models in ergonomics*, 3:382, 2005.

[23] S. Carter, A. Cox, and M. Bostock. Dissecting a Trailer: The Parts of the Film That Make the Cut. `http://www.nytimes.com/interactive/2013/02/19/movies/awardsseason/oscar-trailers.html`, 2013.

[24] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of Data*, pages 668–668. ACM, 2003.

[25] H. Chen. Compound brushing [dynamic data visualization]. In *Information Visualization, 2003. INFOVIS 2003. IEEE Symposium on*, pages 181–188. IEEE, 2003.

[26] J. Choi, D. G. Park, Y. L. Wong, E. Fisher, and N. Elmqvist. VisDock: A Toolkit for Cross-Cutting Interactions in Visualization. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 21(9):1087–1100, 2015.

[27] W. S. Cleveland and R. McGill. Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods. *Journal of the American Statistical Association*, 79(387):531–554, 1984.

[28] G. H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-order Call-by-Value Language*. PhD thesis, Brown University, 2008.

[29] G. H. Cooper and S. Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Programming Languages and Systems*, pages 294–308. Springer, 2006.

[30] J. Cottam and A. Lumsdaine. Stencil: A Conceptual Model for Representation and Interaction. In *Information Visualisation*, pages 51–56. IEEE, 2008.

[31] E. Czaplicki and S. Chong. Asynchronous Functional Reactive Programming for GUIs. In *ACM SIGPLAN Notices*, volume 48, pages 411–422. ACM, 2013.

[32] M. Derthick, J. Kolojejchick, and S. F. Roth. An Interactive Visual Query Environment for Exploring Data. In *Proceedings of the 10th annual ACM symposium on User Interface Software and Technology*, pages 189–198. ACM, 1997.

[33] A. Deshpande, Z. Ives, V. Raman, et al. Adaptive Query Processing. *Foundations and Trends® in Databases*, 1(1):1–140, 2007.

[34] J. Edwards. Coherent Reaction. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 925–932. ACM, 2009.

[35] J.-D. Fekete. The InfoVis toolkit. In *IEEE Symposium on Information Visualization*, pages 167–174, 2004.

[36] Flare. `http://flare.prefuse.org/`, September 2013.

[37] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Data Engineering, 1993. Proceedings. Ninth International Conference on*, pages 209–218. IEEE, 1993.

[38] S. Gratzl, N. Gehlenborg, A. Lex, H. Pfister, and M. Streit. Domino: Extracting, Comparing, and Manipulating Subsets across Multiple Tabular Datasets. *IEEE Transactions on Visualization and Computer Graphics (TVG) (Proc. InfoVis)*, 20(12):2023–2032, 2014.

[39] T. Grossman and R. Balakrishnan. The Bubble Cursor: Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 281–290. ACM, 2005.

[40] F. Guimbretiére and T. Winograd. FlowMenu: Combining Command, Text, and Data Entry. In *Proceedings of the 13th annual ACM symposium on User Interface Software and Technology*, pages 213–216. ACM, 2000.

[41] P. J. Guo. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584. ACM, 2013.

[42] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[43] J. Harper and M. Agrawala. Deconstructing and Restyling D3 Visualizations. In *Proceedings of the 27th annual ACM symposium on User Interface Software and Technology*, pages 253–262. ACM, 2014.

[44] J. Harper and M. Agrawala. Converting Basic D3 Charts into Reusable Style Templates. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 2017.

[45] J. Heer and M. Agrawala. Software Design Patterns for Information Visualization. *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Proc. InfoVis)*, 12(5):853–860, 2006.

[46] J. Heer, M. Agrawala, and W. Willett. Generalized Selection via Interactive Query Relaxation. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 959–968. ACM, 2008.

[47] J. Heer and M. Bostock. Declarative Language Design for Interactive Visualization. *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Proc. InfoVis)*, 16(6):1149–1156, 2010.

[48] J. Heer, S. K. Card, and J. A. Landay. Prefuse: a toolkit for interactive information visualization. In *Proc. ACM CHI*, pages 421–430. ACM, 2005.

[49] J. Heer and G. G. Robertson. Animated Transitions in Statistical Data Graphics. *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Proc. InfoVis)*, 13(6):1240–1247, 2007.

[50] J. Heer and B. Shneiderman. Interactive Dynamics for Visual Analysis. *Communications of the ACM*, 55(4):45–54, 2012.

[51] J. Heer, F. B. Viégas, and M. Wattenberg. Voyagers and Voyeurs: Supporting Asynchronous Collaborative Visualization. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 1029–1038. ACM, 2007.

[52] J. Hoffswell, A. Satyanarayan, and J. Heer. Visual Debugging Techniques for Reactive Data Visualization. *Computer Graphics Forum (Proc. EuroVis)*, 2016.

[53] C. Holz and S. Feiner. Relaxed Selection Techniques for Querying Time-Series Graphs. In *Proceedings of the 22nd annual ACM symposium on User Interface Software and Technology*, pages 213–222. ACM, 2009.

[54] S. E. Hudson and I. Smith. Ultra-Lightweight Constraints. In *Proceedings of the 9th annual ACM symposium on User Interface Software and Technology*, pages 147–155. ACM, 1996.

[55] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. *Human-Computer Interaction*, 1(4):311–338, 1985.

[56] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 3363–3372. ACM, 2011.

[57] C. Kelleher and H. Levkowitz. Reactive data visualizations. In *IS&T/SPIE Electronic Imaging*, pages 93970N–93970N. International Society for Optics and Photonics, 2015.

[58] Y. Kim, K. Wongsuphasawat, J. Hullman, and J. Heer. GraphScape: A Model for Automated Reasoning about Visualization Similarity and Sequencing. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 2017.

[59] K. Kin, B. Hartmann, T. DeRose, and M. Agrawala. Proton++: A Customizable Declarative Multitouch Framework. In *Proceedings of the 25th annual ACM symposium on User Interface Software and Technology*, pages 477–486. ACM, 2012.

[60] B. Kondo and C. Collins. DimpVis: Exploring Time-varying Information Visualizations by Direct Manipulation. *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Proc. InfoVis)*, 20(12):2003–2012, 2014.

[61] R. Kosara. The US ZIPScribble Map. `http://eagereyes.org/zipscribble-maps/united-states/`, December 2006.

[62] R. Kumar, A. Satyanarayan, C. Torres, M. Lim, S. Ahmad, S. R. Klemmer, and J. O. Talton. Webzeitgeist: Design Mining the Web. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 3083–3092. ACM, 2013.

[63] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Proc. InfoVis)*, 19(12):2456–2465, 2013.

[64] Z. Liu, B. Jiang, and J. Heer. imMens: Real-time Visual Querying of Big Data. *Computer Graphics Forum (Proc. EuroVis)*, 32, 2013.

[65] Z. Liu and J. T. Stasko. Mental Models, Visual Reasoning and Interaction in Information Visualization: A Top-down Perspective. *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Proc. InfoVis)*, 16(6):999–1008, 2010.

[66] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. DEVise: Integrated Querying and Visual Exploration of Large Datasets. *ACM SIGMOD Record*, 26(2):301–312, 1997.

[67] B. Lucas, G. D. Abram, N. S. Collins, D. A. Epstein, D. L. Gresh, and K. P. McAuliffe. An Architecture for a Scientific Visualization System. In *Proceedings of the 3rd conference on Visualization*, pages 107–114, 1992.

[68] J. Mackinlay. Automating the Design of Graphical Presentations of Relational Information. *ACM Transactions on Graphics (TOG)*, 5(2):110–141, 1986.

[69] Vega makes visualizing BIG data easy. `https://www.mapd.com/blog/2017/07/22/vega-makes-visualizing-big-data-easy/`, July 2017.

[70] MediaWiki Extension:Graph. `https://www.mediawiki.org/wiki/Extension:Graph`, June 2015.

[71] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

[72] D. Moritz, J. Heer, and B. Howe. Dynamic Client-Server Optimization for Scalable Interactive Visualization on the Web. *Workshop on Data Systems for Interactive Analysis at InfoVis*, 2015.

[73] B. A. Myers. A New Model for Handling Input. *ACM Transactions on Information Systems (TOIS)*, 8(3):289–320, 1990.

[74] B. A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proceedings of the 4th annual ACM symposium on User Interface Software and Technology*, pages 211–220. ACM, 1991.

[75] C. North and B. Shneiderman. Snap-Together Visualization: A User Interface for Coordinating Visualizations via Relational Schemata. In *Proceedings of the working conference on Advanced Visual Interfaces*, pages 128–135. ACM, 2000.

[76] C. Olsten, M. Stonebraker, A. Aiken, and J. M. Hellerstein. VIQING: Visual Interactive QueryING. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 162–169. IEEE, 1998.

[77] S. Oney, B. Myers, and J. Brandt. ConstraintJS: Programming Interactive Behaviors for the Web by Integrating Constraints and States. In *Proceedings of the 9th annual ACM symposium on User Interface Software and Technology*, pages 229–238. ACM, 2012.

[78] K. Perlin and D. Fox. Pad: An alternative approach to the computer interface. In *Proceedings of the 20th annual conference on Computer Graphics and Interactive Techniques*, pages 57–64. ACM, 1993.

[79] W. A. Pike, J. Stasko, R. Chang, and T. A. O'Connell. The Science of Interaction. *Information Visualization*, 8(4):263–274, 2009.

[80] J. Poco and J. Heer. Reverse-Engineering Visualizations: Recovering Visual Encodings from Chart Images. *Computer Graphics Forum (Proc. EuroVis)*, 2017.

[81] Shiny by RStudio. `http://shiny.rstudio.com`, June 2016.

[82] S. F. Roth, J. Kolojejchick, J. Mattis, and J. Goldstein. Interactive Graphic Design Using Automatic Presentation Knowledge. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 112–117. ACM, 1994.

[83] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *ACM SIGMOD Record*, volume 29, pages 249–260. ACM, 2000.

[84] D. Saber. A Dramatic Tour through Python's Data Visualization Landscape (including ggplot and Altair). `https://dsaber.com/2016/10/02/a-dramatic-tour-`

through-pythons-data-visualization-landscape-including-ggplot-and-altair/, October 2016.

[85] A. Satyanarayan and J. Heer. Lyra: An Interactive Visualization Design Environment. *Computer Graphics Forum (Proc. EuroVis)*, 2014.

[86] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics (Proc. InfoVis)*, 2017.

[87] 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 (Proc. InfoVis)*, 2016.

[88] A. Satyanarayan, K. Wongsuphasawat, and J. Heer. Declarative Interaction Design for Data Visualization. In *Proceedings of the 27th annual ACM symposium on User Interface Software and Technology*, pages 669–678. ACM, 2014.

[89] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization. In *Proceedings of the 7th conference on Visualization*, pages 93–ff. IEEE Computer Society Press, 1996.

[90] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[91] B. Shneiderman. Dynamic Queries for Visual Information Seeking. *IEEE software*, 11(6):70–77, 1994.

[92] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases. *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Proc. InfoVis)*, 8(1):52–65, 2002.

[93] D. F. Swayne, D. T. Lang, A. Buja, and D. Cook. GGobi: evolving from XGobi into an extensible framework for interactive data visualization. *Computational Statistics & Data Analysis*, 43(4):423–444, 2003.

[94] E. R. Tufte. *Envisioning Information*. Graphics Press, 1990.

[95] Online Vega Editor. `http://vega.github.io/vega-editor/`, June 2016.

[96] vegalite: R ggplot2 "bindings" for Vega-Lite. `https://github.com/hrbrmstr/vegalite`, August 2017.

[97] VegaLite.jl: Julia bindings to Vega-Lite. `https://github.com/fredo-dedup/VegaLite.jl`, August 2017.

[98] vegaliteR: A Vega-Lite htmlwidget for R. `https://github.com/timelyportfolio/vegaliteR`, August 2017.

[99] Vegas: The missing MatPlotLib for Scala + Spark. `https://github.com/aishfenton/Vegas`, August 2017.

[100] B. Victor. Drawing Dynamic Visualizations. `http://vimeo.com/66085662`, February 2013.

[101] F. B. Viégas, M. Wattenberg, F. van Ham, J. Kriss, and M. McKeon. Many Eyes: A Site for Visualization at Internet Scale. *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Proc. InfoVis)*, 13(6):1121–1128, 2007.

[102] Vizard: Magic Visualization. `https://github.com/yieldbot/vizard`, August 2017.

[103] Z. Wan, W. Taha, and P. Hudak. Event-Driven FRP. In *Practical Aspects of Declarative Languages*, pages 155–172. Springer, 2002.

[104] C. Weaver. Building Highly-Coordinated Visualizations in Improvise. In *Proc. IEEE Information Visualization*, pages 159–166, 2004.

[105] H. Wickham. A Layered Grammar of Graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010.

[106] A. Wilhelm. User interaction at various levels of data displays. *Computational statistics & data analysis*, 43(4):471–494, 2003.

[107] L. Wilkinson. *The Grammar of Graphics*. Springer, 2 edition, 2005.

[108] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Proc. InfoVis)*, 2015.

[109] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Towards A General-Purpose Query Language for Visualization Recommendation. In *ACM SIGMOD Human-in-the-Loop Data Analysis (HILDA)*, 2016.

[110] K. Wongsuphasawat, Z. Qu, D. Moritz, R. Chang, F. Ouk, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager 2: Augmenting Visual Analysis with Partial View Specifications. In *ACM Human Factors in Computing Systems (CHI)*, 2017.

[111] J. S. Yi, Y. ah 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 (TVCG) (Proc. InfoVis)*, 13(6):1224–1231, 2007.

[112] Yoga. `https://facebook.github.io/yoga/`, March 2015.