

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

This thesis tells you all you need to know about...

Acknowledgments

I would like to thank...

Contents

Abstract	iv
Acknowledgments	v
1 Declarative Primitives for Interaction Design	1
1.1 Interaction Language Design	2
1.1.1 Event Streams and Signals	2
1.1.2 Predicates and Scale Inversion	4
1.1.3 Production Rules	5
1.1.4 User-Defined Functions	6
1.1.5 Encapsulated Interactors	7
1.2 Example Interactive Visualizations	7
1.2.1 Selection: Click/Shift-Click and Brushing	7
1.2.2 Connect: Brushing & Linking	10
1.2.3 Abstract/Elaborate: Overview + Detail	10
1.2.4 Explore & Encode: Panning & Zooming	10
1.2.5 Reconfigure: Index Chart	12
1.2.6 Reconfigure: Reordering Columns of a Matrix	13
1.2.7 Filter: Control Widgets	14
1.2.8 DimpVis: Touch Navigation with Time-Series Data	15
1.2.9 Reusable Touch Interaction Abstractions	16
1.3 Discussion: Cognitive Dimensions of Notation	16
1.4 Summary	19

2 A Streaming Dataflow Architecture	20
2.1 The Dataflow Graph Design	21
2.1.1 Data, Interaction, and Scene Graph Operators	22
2.1.2 Changesets and Materialization	23
2.1.3 Coordinating Changeset Propagation	24
2.1.4 Pushing Internal and Pulling External Changesets	25
2.1.5 Dynamically Restructuring the Graph	26
2.2 Performance Optimizations	27
2.2.1 On-Demand Tuple Revision Tracking	27
2.2.2 Pruning Unnecessary Recomputation	28
2.2.3 Inlining Sequential Operators	29
2.3 Comparative Performance Benchmarks	30
2.3.1 Streaming Visualizations	30
2.3.2 Interactive Visualizations	31
2.4 Conclusion & Future Work	31
3 A Grammar of Interactive Graphics	34
3.1 Visual Encoding	35
3.2 View Composition Algebra	37
3.2.1 Layer	38
3.2.2 Concatenation	39
3.2.3 Facet	40
3.2.4 Repeat	41
3.2.5 Dashboards and Nested Views	42
3.3 Interactive Selections	42
3.3.1 Selection Transforms	45
3.3.2 Selection-Driven Visual Encodings	49
3.3.3 Disambiguating Composite Selections	50
3.4 Compilation	53
3.5 Example Interactive Visualizations	55
3.5.1 Selection: Click/Shift-Click and Brushing	56

3.5.2	Explore & Encode: Panning & Zooming	57
3.5.3	Connect: Brushing & Linking	58
3.5.4	Abstract/Elaborate: Overview + Detail	59
3.5.5	Reconfigure: Index Chart	60
3.5.6	Filter: Cross Filtering	60
3.5.7	Limitations	62
3.6	Conclusion	63
4	An Interactive Visualization Design Environment (VDE)	65

List of Figures

1.1	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) can be instantiated as well.	2
1.2	Reactive Vega provides an event stream selector syntax, inspired by CSS selectors, to compose, filter, and sequence input events.	3
1.3	Points are highlighted using (left) an intensional predicate $50 \leq \text{Horsepower} \leq 100$ or (right) an extensional predicate with members #56, #110, #79, #95, #40, #120,	5
1.4	<i>Predicates</i> use signal values to define interactive selections of elements. Using <i>scale inversions</i> , predicates can be generalized to define interactive queries, and thus operate across different coordinate spaces: overview (bottom) and detail (top).	6
1.5	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.	8
1.6	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.	9
1.7	We can extract the brushing interaction from Fig. 1.6 into a standalone interactor, and reapply it to a scatterplot matrix to perform brushing & linking.	9

1.8	Panning & zooming a scatterplot. Brushing can be added accretively by including the brush interactor (Figs. 1.6 and 1.7, and rebinding event streams (indicated with strikethroughs) to resolve conflicting events.	11
1.9	By extracting the pan & zoom interaction from Fig. 1.8 into an interactor, we can repurpose it to perform semantic zooming on a geographic map. A map of the United States shows a choropleth of state-level unemployment. Zooming past a threshold, states break up into counties, and show county-level unemployment.	12
1.10	An index chart that shows the percentage changes for a collection of time-series. The index point (red vertical line) is determined by the x position of a <code>mousemove</code> signal, which feeds a predicate within a filter data transform.	13
1.11	The job voyager can be filtered using signals bound to the state of control widgets. A textbox allows for pattern-based filtering of job titles while radio buttons allow gender-based filtering.	14
3.1	A unit specification that uses a <i>line</i> mark to visualize the <i>mean</i> temperature for every <i>month</i>	36
3.2	A <i>binned</i> scatterplot visualizes correlation between wind and temperature.	37
3.3	A <i>stacked</i> bar chart that sums the various weather types by location.	37
3.4	A dual axis chart that <i>layers</i> a line for the monthly mean temperature on top of bars for monthly mean precipitation. Each layer uses an <i>independent</i> y-scale.	38
3.5	The Figs. 3.1 and 3.2 unit specifications <i>concatenated</i> vertically; scales and guides for each plot are independent by default.	39
3.6	The line chart from Fig. 3.1 <i>faceted</i> vertically by location; the x-axis is shared, and the underlying scale domains unioned, to facilitate easier comparison.	40

3.7	<i>Repetition</i> of different measures across rows; the y-channel references the <code>row</code> template parameter to vary the encoding.	41
3.8	Adding a <i>single</i> selection to parameterize the fill color of a scatterplot's circle mark.	43
3.9	Switching from a <i>single</i> to <i>multi</i> selection. The first value is selected on click, and additional values on shift-click.	43
3.10	Highlight a continuous range of points using an <i>interval</i> selection. A rectangle mark is automatically added to depict the interval extents. . .	44
3.11	Specifying a custom event trigger for a <i>multi</i> selection: the first point is selected on <code>mouseover</code> and subsequent points when the shift key is pressed.	45
3.12	Using the <i>project</i> transform to highlight a <i>single Origin</i>	46
3.13	Using the <i>project</i> transform to highlight <i>multiple Origins</i>	47
3.14	<i>Projecting</i> an interval selection to restrict it to a single dimension. . .	47
3.15	The <i>translate</i> transform enables movement of the brushed region. It is automatically invoked for interval selections but is explicitly depicted here for clarity.	48
3.16	Panning and zooming the scatterplot is achieved by first <i>binding</i> an interval selection to the x- and y-scale domains, and then applying the <i>translate</i> and <i>zoom</i> transforms. Alternate events prevent collision with the brushing interaction, previously defined in Fig. 3.10	49
3.17	By adding a <i>repeat</i> operator, we compose the encodings and interactions from Fig. 3.16 into a scatterplot matrix. Users can brush, pan, and zoom within each cell, and the others update in concert. By default, a <i>global</i> composite selection is created: brushing in a cell replaces previous brushes.	51
3.18	Resolving the <code>region</code> selection to <i>independent</i> produces a brush in each cell, and points only highlight based on the selection in their own cell.	52
3.19	Resolving the <code>region</code> selection to <i>union</i> produces a brush in each cell, and points highlight if they fall within any of the selections.	53

3.20 Resolving the <code>region</code> selection to <i>intersect</i> produces a brush in each cell, and points only highlight if they lie within all of the selections.	53
3.21 An overview + detail visualization concatenates two unit specifications, with a selection in the second one parameterizing the x-scale domain in the first.	59
3.22 An index chart uses a single selection to renormalize data based on the index point nearest the mouse cursor.	60
3.23 An interval selection, resolved to <i>intersect others</i> , 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.	61
3.24 A layered cross filtering interaction is constructed by resolving the interval selection to <i>intersect</i> , and then materializing it to serve as the input data for a second layer. Highlights indicate changes to the specification from Fig. 3.23.	62

Chapter 1

Declarative Primitives for Interaction Design

Reactive Vega builds on a long-running thread of research on declarative visualization design, popularized by the Grammar of Graphics [24] and Polaris [20] (now Tableau).

Visual encodings are defined by composing graphical primitives called *marks* [3], 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 inherit a group’s data, or draw from independent datasets.

Although interaction is a crucial component of effective visualization [11, 15], existing declarative visualization models, including widely used tools such as D3 [4] and ggplot2 [22], do not offer composable primitives for interaction design. Instead, if they support interaction, they do so through either a palette of standard techniques [3, 4] or *imperative* event handling callbacks. While the former restricts expressivity, the latter undoes many of the benefits of declarative design. In particular, users are forced to contend with interaction execution details, such as interleaved events and coordinating external state, which can be complex and error-prone [14? ?].

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"}
        }
      }
    }
  ]
}
```

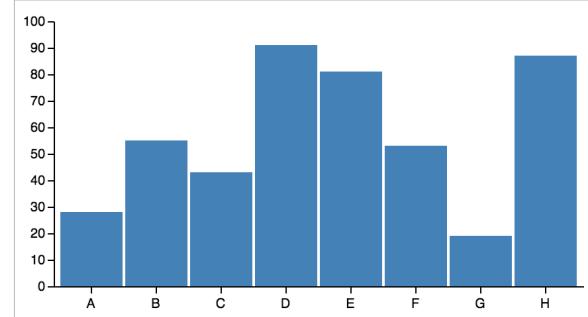


Figure 1.1: A JSON specification for a bar chart, demonstrating Vega’s abstractions for visual encoding. Data is imported from a URL. Scales transform data values to visual values. Properties of graphical marks (in this case rectangles) are determined by scale mappings. Guides (here, axes) can be instantiated as well.

1.1 Interaction Language Design

1.1.1 Event Streams and Signals

Reactive Vega adapts the semantics of Event-Driven Functional Reactive Programming (E-FRP) [?]. Low-level input events (e.g., mouse events and keystrokes) are captured as time-varying *streaming data*, 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. 1.2). While prior work has formulated regex-based symbols for event selection [?] we believe our approach, by drawing on CSS selectors for inspiration, will be more familiar to designers.

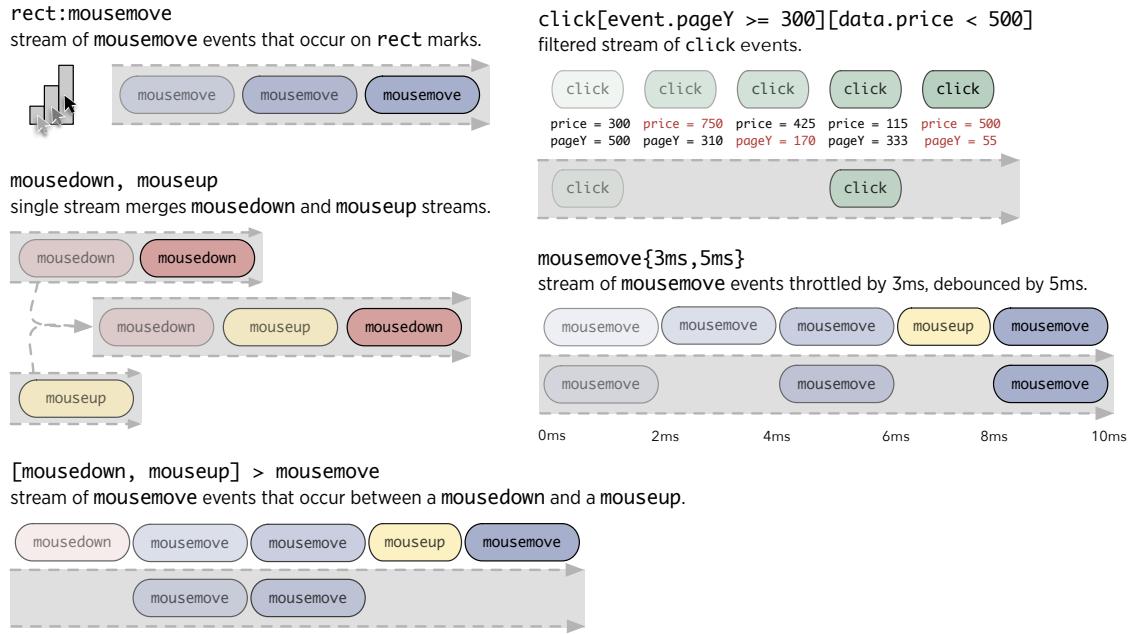


Figure 1.2: Reactive Vega provides an event stream selector syntax, inspired by CSS selectors, to compose, filter, and sequence input events.

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 mark 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 bounding events for the stream. For instance, `[mousedown, mouseup] >mousemove` is a single stream of `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 100 milliseconds and debounces it by 200 milliseconds). 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, by drawing from multiple event streams, signals can define finite-state machines with each stream triggering a transition between states.

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. This separation decouples 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.

1.1.2 Predicates and Scale Inversion

Selection is a fundamental operation in interactive visualization design [8]. 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.

To express interactive selections, we introduce reactive *predicates*. As shown below, 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.

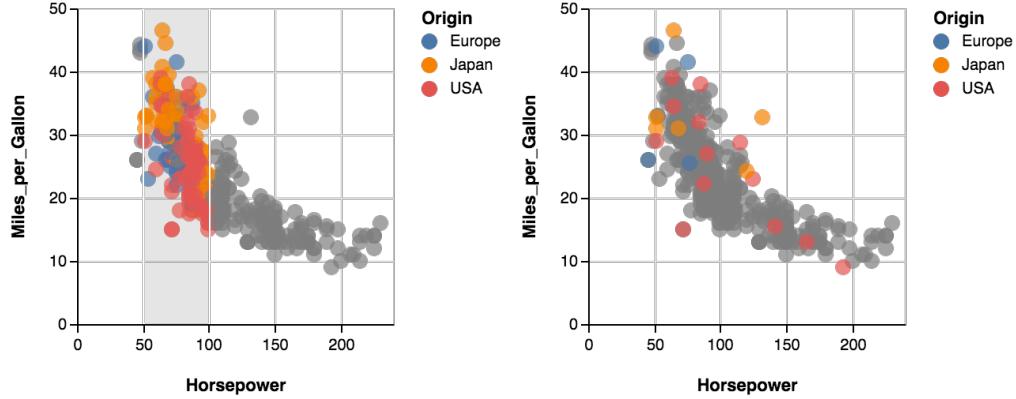


Figure 1.3: Points are highlighted using (left) an intensional predicate $50 \leq \text{Horsepower} \leq 100$ or (right) an extensional predicate with members #56, #110, #79, #95, #40, #120,

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 multiple distinct visualizations. In such cases, it is necessary to generalize an interactive selection into a query over the data domain [8]. Scale functions are a critical component in visualization design [24] 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 [?].

1.1.3 Production Rules

Production rules are an established design pattern for visualization specification [?] 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 if predicate B is true, or otherwise set the color to grey by default.

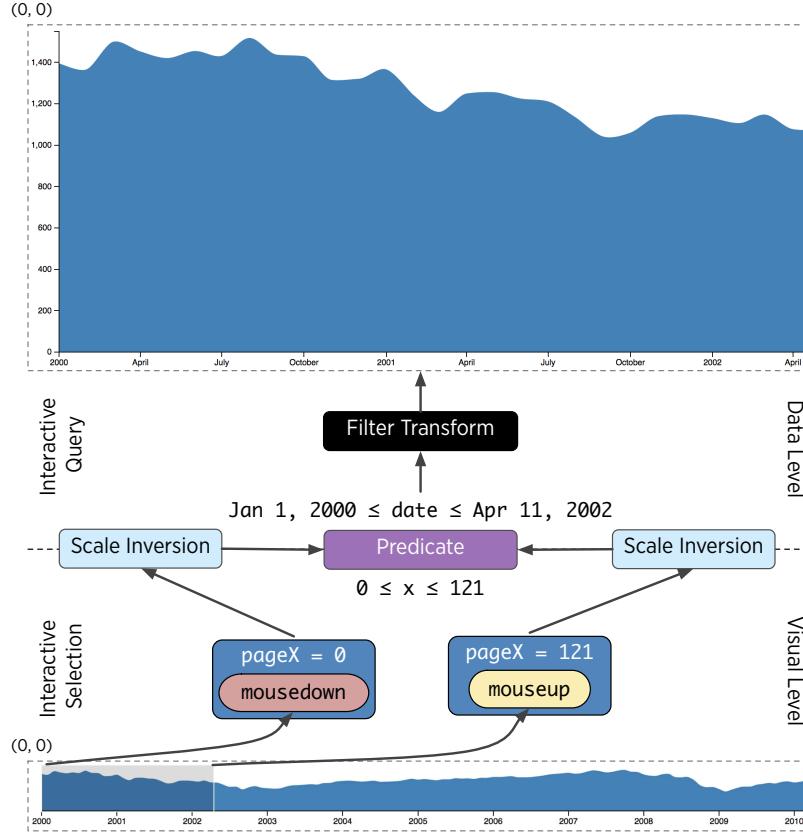


Figure 1.4: *Predicates* use signal values to define interactive selections of elements. Using *scale inversions*, predicates can be generalized to define interactive queries, and thus operate across different coordinate spaces: overview (bottom) and detail (top).

1.1.4 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 [10]. 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 ensuring extensibility to idiosyncratic operations.

1.1.5 Encapsulated Interactors

To allow reuse of custom interaction techniques, Reactive Vega’s interaction primitives can be parameterized and encapsulated as named *interactors*. An interactor can subsequently be applied to a visualization and functions like mixin. Its specification is merged into the host’s and, to prevent conflicts, its components are addressable only under its namespace. Figure 1.7 illustrates how a brush interaction, extracted from Fig. 1.6, can be applied to brush & link a scatterplot matrix.

1.2 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 [26]. 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.

1.2.1 Selection: Click/Shift-Click and Brushing

Figure 1.5 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 whether the shift key is pressed and, if not, clears the data source prior to inserting the clicked values. A production rule sets the fill color of selected points using an extensional predicate.

Similarly, Fig. 1.6 demonstrates the Reactive Vega JSON necessary to enable brush selections. Signals are registered to capture the start and end positions of the brush,

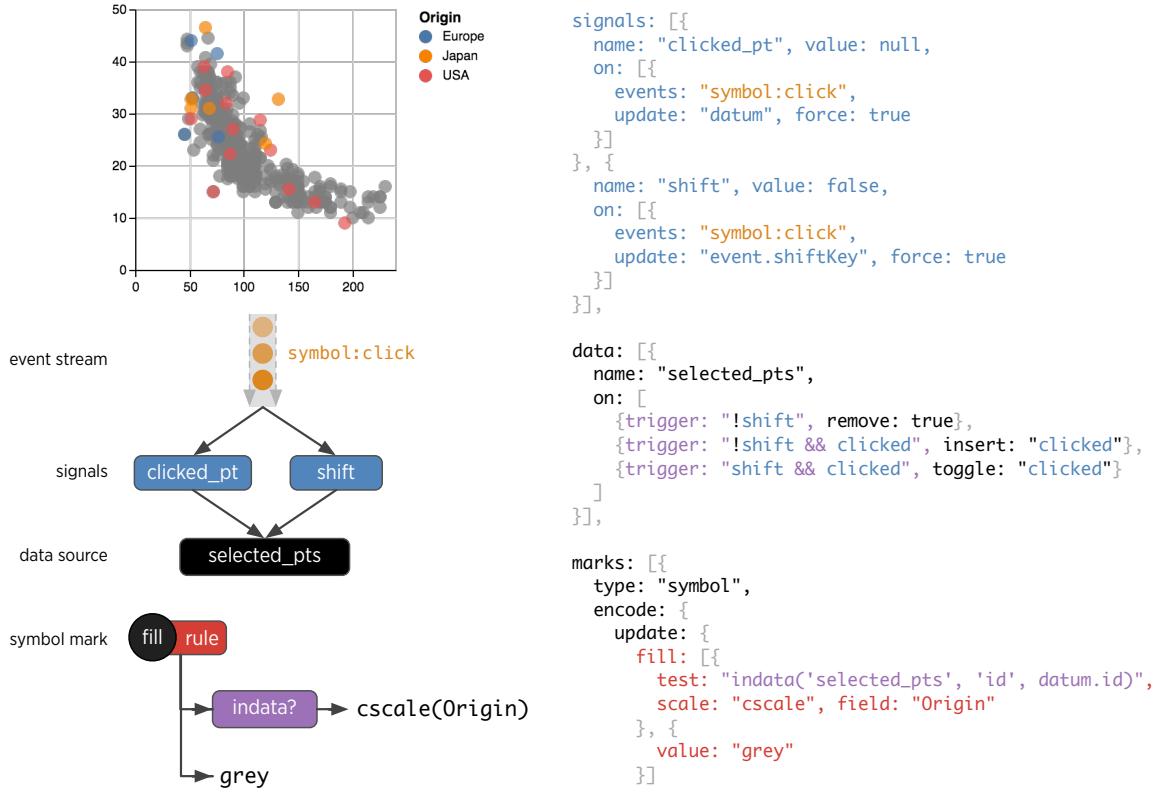


Figure 1.5: Reactive Vega JSON for a click-to-highlight interaction. Signals over a click stream feed data transform to toggle values in a data source. A production rule uses a predicate to set marks' fill color.

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.

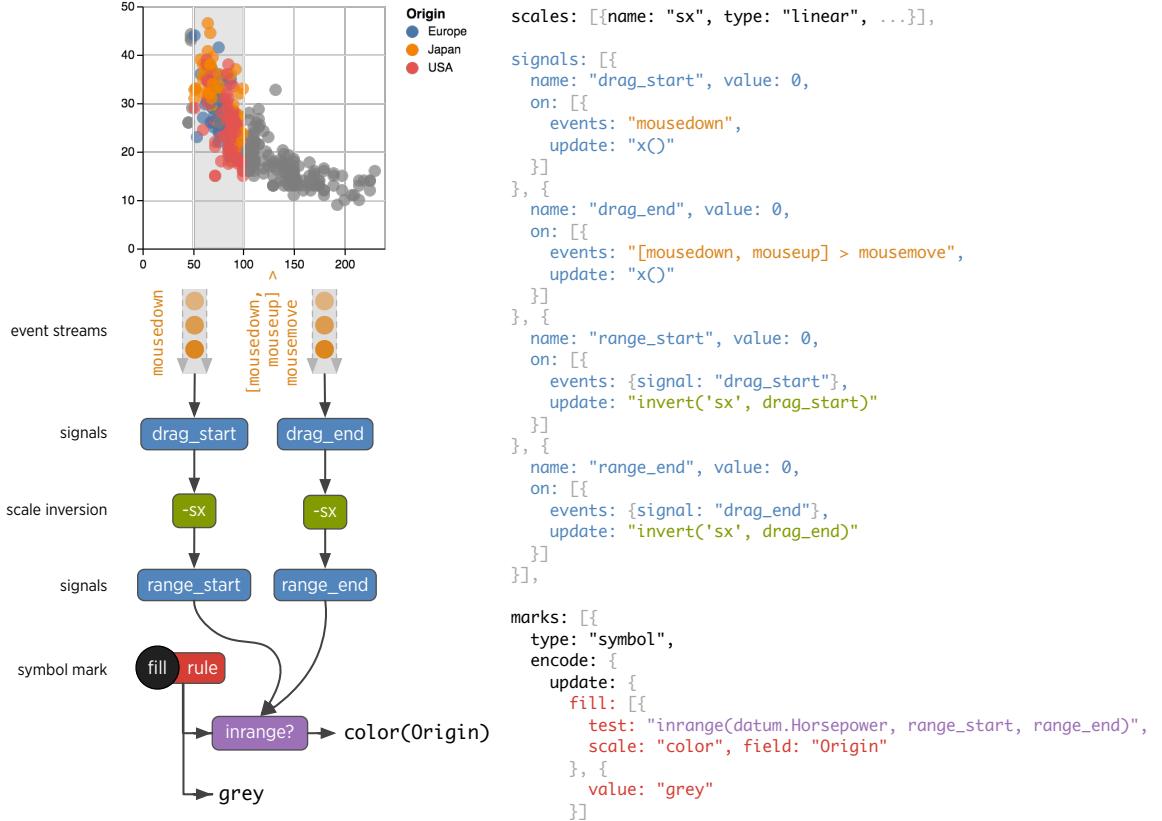


Figure 1.6: A JSON snippet for one-dimensional brushing. Signals over drag events are inverted through the x-scale to construct a data query over the `Horsepower` field.

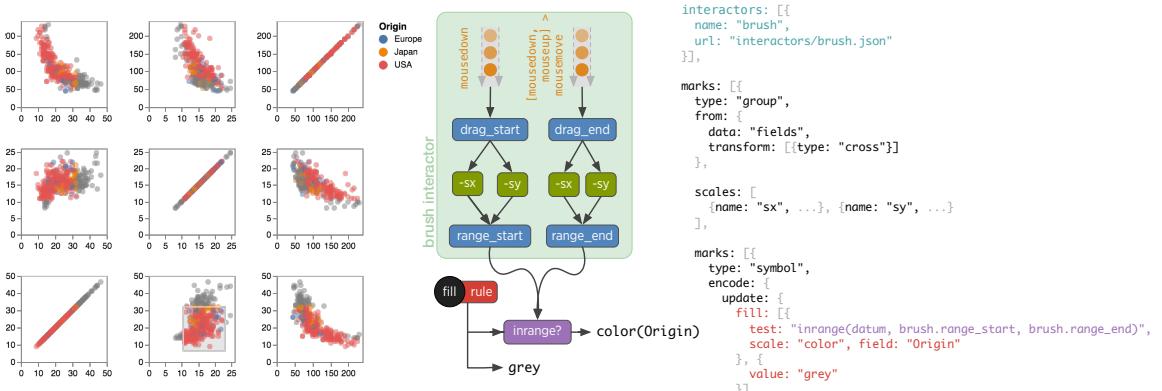
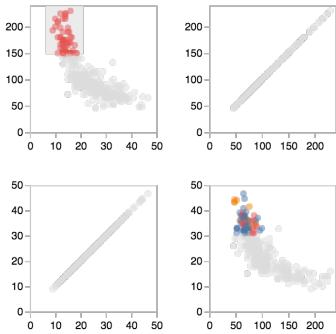


Figure 1.7: We can extract the brushing interaction from Fig. 1.6 into a standalone interactor, and reapply it to a scatterplot matrix to perform brushing & linking.

1.2.2 Connect: Brushing & Linking



We can extract the interaction from the previous example into a standalone “brushing” interactor, and then apply it to brush & link a scatterplot matrix as shown in Fig. 1.7. Each cell of the matrix is an instance of a group mark with its own coordinate space. The plotting symbol and necessary spatial scale functions are defined within this group. 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, in the inset figure, brushing over red points would highlight no or blue points in the other cells. Instead, the interactor uses scale inversions to lift the predicate to the data domain. Thus, the production rule correctly performs the linking operation across scatterplots.

1.2.3 Abstract/Elaborate: Overview + Detail

With our brush interactor, we can also create the overview + detail visualization shown in Fig. 1.4. In this case, brushing is restricted to the horizontal dimension. In our visualization, we override the `height` property of the visual brush added by the interactor, and ignore 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.

1.2.4 Explore & Encode: Panning & Zooming

Figure 1.8 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.

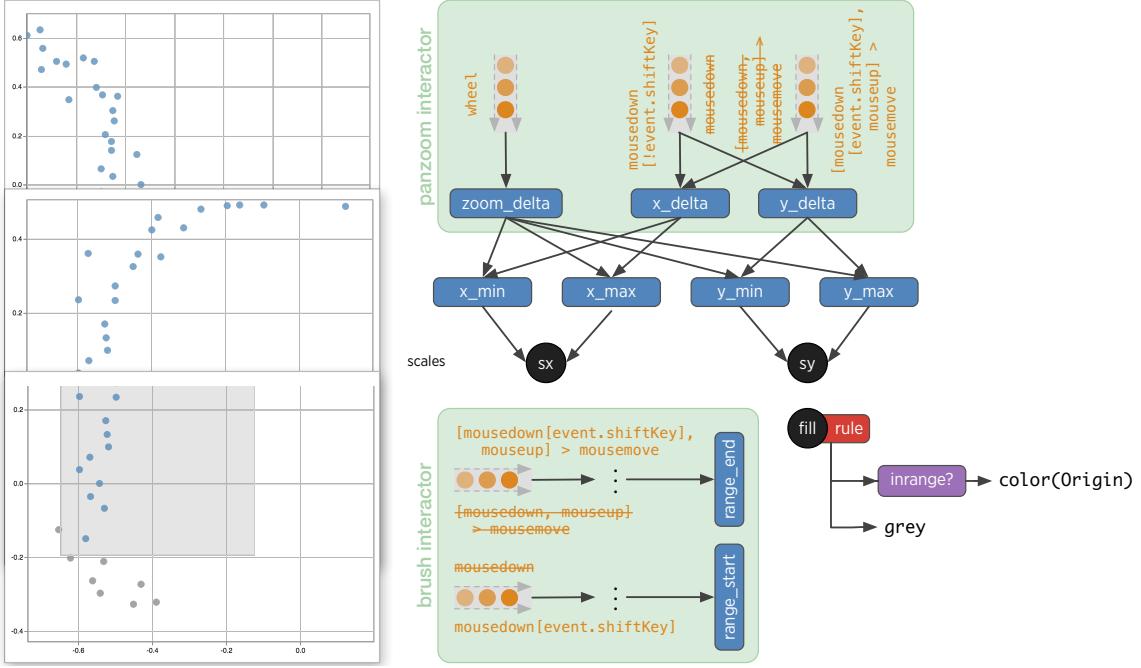


Figure 1.8: Panning & zooming a scatterplot. Brushing can be added accretively by including the brush interactor (Figs. 1.6 and 1.7, and rebinding event streams (indicated with strikethroughs) to resolve conflicting events.

If we were to also add a brushing interaction to this visualization, a naïve application would produce 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 [4], which offers brushing and panning as part of its interactor typology, the process 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.

Moreover, by extracting panning & zooming into a standalone interactor, we can

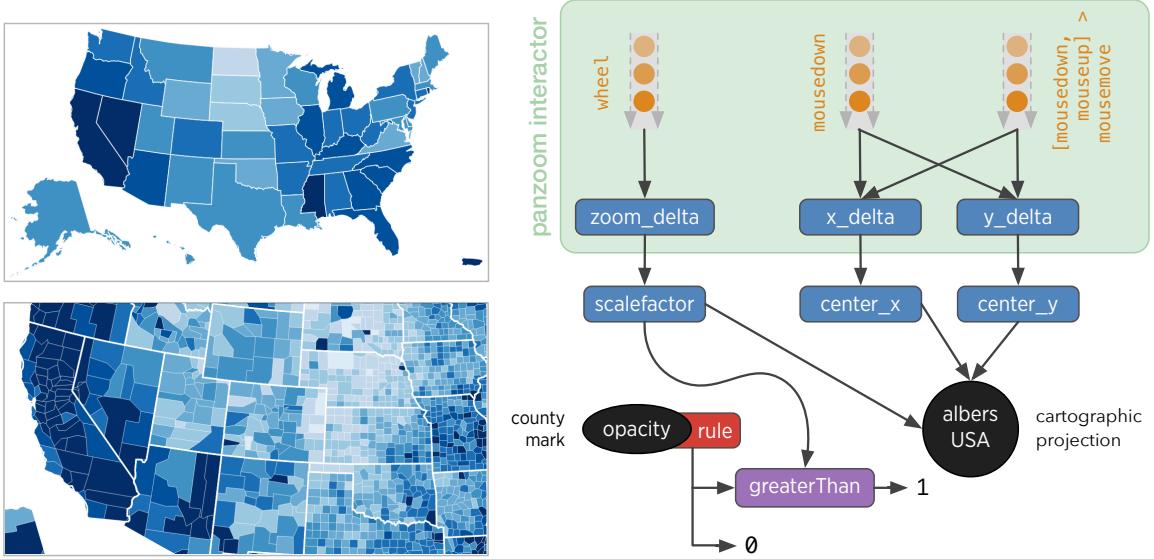


Figure 1.9: By extracting the pan & zoom interaction from Fig. 1.8 into an interactor, we can repurpose it to perform semantic zooming on a geographic map. A map of the United States shows a choropleth of state-level unemployment. Zooming past a threshold, states break up into counties, and show county-level unemployment.

repurpose the behavior to instead trigger semantic zooming [?], an *encoding* interaction technique shown in Fig. 1.9. 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.

1.2.5 Reconfigure: Index Chart

Figure 1.10 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, we construct a signal over `mousemove` events and then drive the `x` coordinate through a scale inversion. As it is a quantitative scale, scale inversion results in a

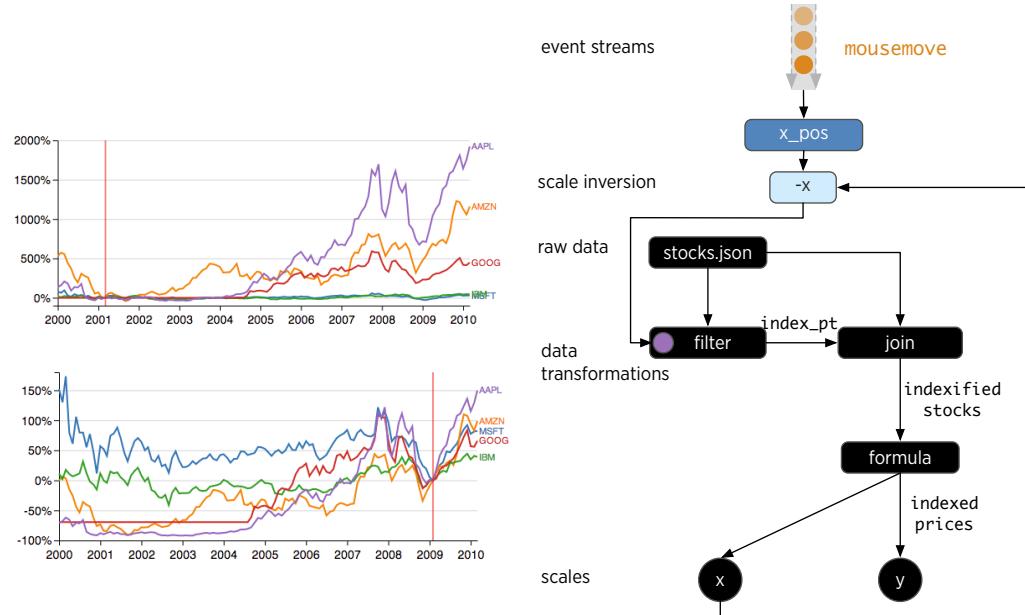
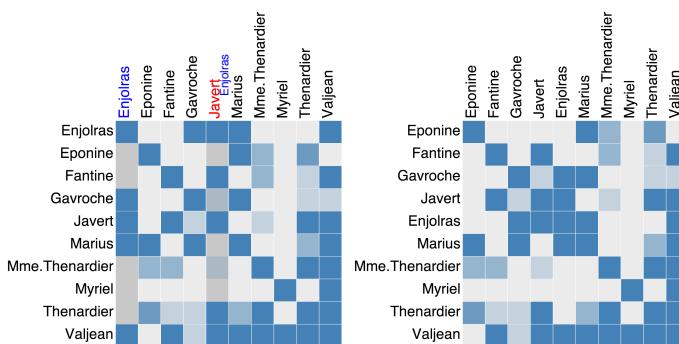


Figure 1.10: An index chart that shows the percentage changes for a collection of time-series. The index point (red vertical line) is determined by the x position of a `mousemove` signal, which feeds a predicate within a filter data transform.

value from a continuous 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, with the line interpolating between these points. 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.

1.2.6 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 sorting indices of the two columns.

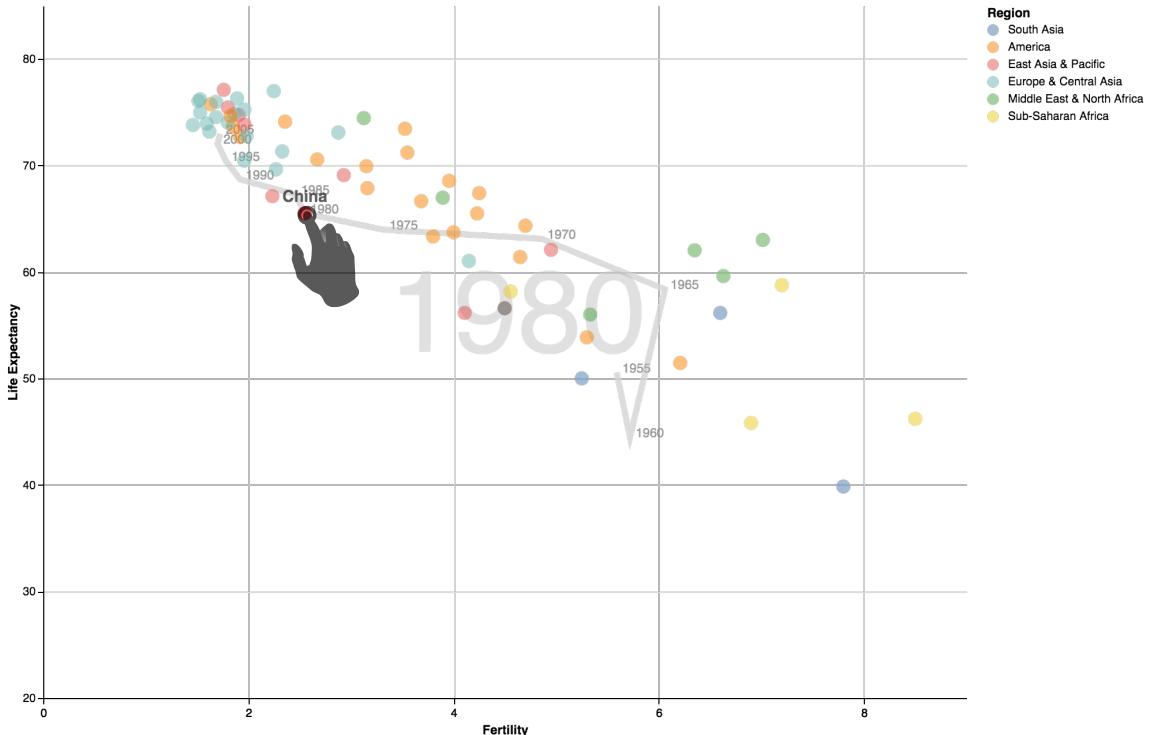
1.2.7 Filter: Control Widgets

Figure 1.11 shows the Job Voyager [?] 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 1.11: The job voyager can be filtered using signals bound to the state of control widgets. A textbox allows for pattern-based filtering of job titles while radio buttons allow gender-based filtering.

1.2.8 DimpVis: Touch Navigation with Time-Series Data



DimpVis [?] 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 [?].

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.

1.2.9 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. When 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—for instance, using `twotouchmove` and `pinchDelta` signals to drive panning and zooming behaviors.

1.3 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 [?], 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 [4] 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 the widespread adoption of D3, and Vega’s increasing integration in systems [25? ?] 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. It requires users to think in terms of different notational systems, and exposes underlying implementation and execution concerns.

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 [4]: 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. [9] 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.

1.4 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 [9]. Reusing and repurposing interactive behaviors is also supported by passing predicates through scale inversions, and extracting declarative specifications to standalone interactors.

Chapter 2

A Streaming Dataflow Architecture

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 [?]) 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) [?] 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 [? ? ? ? ?] alongside E-FRP, including tracking state at the tuple-level and only propagating modified tuples through the dataflow graph.

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.

2.1 The Dataflow Graph Design

Operators in Reactive Vega’s dataflow graph are instantiated and connected by its *parser*, which traverses a declarative specification containing definitions for input datasets, visual encoding rules, and interaction primitives as described in Chapter 1. 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 graph’s sole sink: the renderer.

2.1.1 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, as shown in Figure ???. 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. This relieves operators of the burden of propagating 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 in the dataflow graph 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 [?]. In the case of ordered selectors (e.g., a “drag” event 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 [?] as described in §2.1.3.

Constructing the Scene Graph

To construct the scene graph, Reactive Vega follows a process akin to the Protovis bind-build-evaluate pipeline [?]. 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 [4] 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 exhibits an alternating structure, with individual mark elements grouped under a sentinel mark specification node. Figure ?? illustrates this process for a grouped bar chart.

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 [?]) are now supported in a fully declarative fashion.

2.1.2 Changesets and Materialization

All data does not flow through the system at all times. Instead, operators receive and transmit *changesets*. A changeset consists of tuples that have been observed, 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, would replace all incoming tuples with the cross-product with another data stream.

While changesets only include updated data, some operators 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* [?] or *synopses* [?] in streaming databases) exist to 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 runtime they can then request a complete dataset from the dataflow graph 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.

2.1.3 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* [?]. Centralizing this responsibility, rather than delegating it to operators, enables more aggressive pruning of unnecessary computation as described in §2.2.2. The scheduler has access to the full graph structure and, thus, has 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 [?], and is necessary to enable expressive signal composition. Once all necessary signals have been reevaluated, a changeset with the new signal values is sent to the scheduler for propagation to the rest of the dataflow graph.

2.1.4 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 such as other data sources, signals, and scale functions. As these edges connect disparate data spaces, they cannot directly connect operators with their dependencies. To do otherwise 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 and request the latest values of their dependencies from the scheduler.

The only exception to this pattern is when signals rely on other signals. Reflow changesets still flow along these edges but, as they operate in scalar data space, they are not mediated by **Collectors**.

This hybrid push/pull system enables a complex web of interdependent operators while reducing the implementation 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 ?????????? use filled and unfilled arrows for internal and external connections, respectively.

2.1.5 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. 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 propagation in which they were created. 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.

The most common source of restructuring operations are scene graph operators, 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 ?? provides a step-by-step illustration of how scene graph operators are constructed during a propagation cycle for the grouped bar chart in Figure ??.

2.2 Performance Optimizations

Declarative language runtimes can transparently optimize performance [?] 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 system running Mac OS X 10.10.2, with a quad-core 2.5GHz Intel Core i7 processor and 16GB of 1600 MHz DDR3 RAM.

2.2.1 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.

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 ??, 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.

2.2.2 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 ???. 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 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.

2.2.3 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 $O(\log N)$ cost for enqueueing and dequeuing 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 ?? 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.

2.3 Comparative Performance Benchmarks

To evaluate the performance of Reactive Vega against D3 [4] and the origin, non-reactive Vega system (v1.5.0), we use the same setup described in the previous section.

2.3.1 Streaming Visualizations

Figure ?? shows the average performance of (non-interactive) 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 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 update cycle.

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

Figure ?? shows the results—on average, and across both Canvas and SVG renderers, Reactive Vega offers superior interactive performance to custom D3 and Vega event handling callbacks. This effect primarily stems from Reactive Vega’s unified data model, and is most noticeable with brushing & linking a scatterplot matrix and the time-series overview+detail visualization. In both examples, interactions manipulate only a subset of all data tuples. With Reactive Vega, only these tuples are processed, and their corresponding scene graph elements re-encoded and re- rendered. By comparison, with Vega 1.5’s fragmented reactive approach, the entire scene graph must be reconstructed and rendered in response to changes in input data.

2.4 Conclusion & Future Work

Declarative languages are a popular means of authoring visualizations [3, 4?], 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.

¹Brushing & Linking: <http://bl.ocks.org/mbostock/4063663>

²Overview + Detail: <http://bl.ocks.org/mbostock/1667367>

³Pan & Zoom: <http://bl.ocks.org/mbostock/3892919>

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 used to implement extensions to D3, and higher-level tools can opt to manually construct and connect required dataflow operators.

By simplifying programmatic generation of visualization, Reactive Vega’s declarative JSON syntax has led to a growing ecosystem of higher-level visualization systems. For example, MapD [?] has integrated Vega with their GPU-powered database; custom SQL queries can be embedded within the JSON specification, which is dispatched to the backend server, rendered, and returned to the client as a PNG image. Similarly, Wikipedia, a security-conscious environment where it would be difficult to allow users to write imperative visualization code, has recently integrated Vega [?] to enable visualization of data embedded in articles.

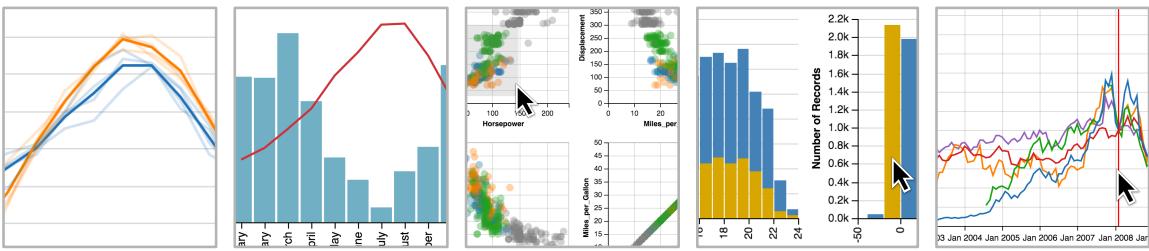
Still, improved support for authoring and debugging Vega specifications remains a promising avenue for future work. In recent work, Hoffswell et al. [9] developed a “time-traveling” debugger for Reactive Vega specifications and found that first-time Vega users were able to accurately trace errors through the specification. Further work, particularly by instrumenting Reactive Vega’s dataflow graph to enable inspection and stepping through changeset propagation could aid in learnability [?]. Through the development of such tools, we can also assess the accessibility of the language. Are new users able to learn the declarative interaction model? Can experts, accustomed to callback-driven programming, quickly transition as well?

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 [?] and imMens [?] 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 [13]. 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 [?].

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

Chapter 3

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 would be apt to describe it as an “assembly language” for interactive visualization most suited for *exploratory* visualization. In contrast, for *exploratory* visualization, higher-level grammars such as ggplot2 [23], and grammar-based systems such as Tableau (née Polaris [20]), 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 [25]. By providing 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 [16]. For custom, direct-manipulation interaction they must, once again, turn to imperative event handling callbacks.

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

3.1 Visual Encoding

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

$$\textit{unit} := (\textit{data}, \textit{transforms}, \textit{mark-type}, \textit{encodings})$$

The *data* definition identifies a data source, a relational table consisting of records (rows) with named attributes (columns). This data 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:

$$\textit{encoding} := (\textit{channel}, \textit{field}, \textit{data-type}, \textit{value}, \textit{functions}, \textit{scale}, \textit{guide})$$

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 [20, 25].

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. 3.1 to 3.3.

```
{
  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"
    }
  }
}
```

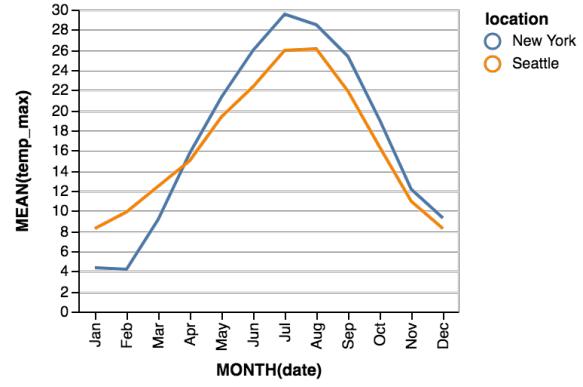


Figure 3.1: A unit specification that uses a *line* mark to visualize the *mean* temperature for every *month*.

```
{
  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"
    }
  }
}
```

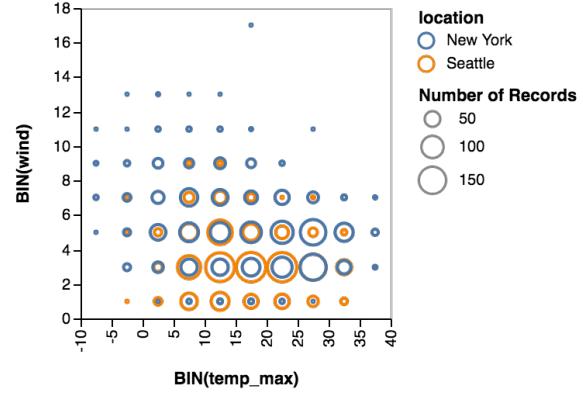


Figure 3.2: A *binned* scatterplot visualizes correlation between wind and temperature.

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

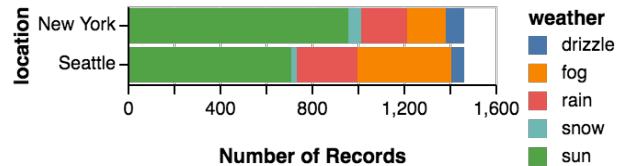


Figure 3.3: A *stacked* bar chart that sums the various weather types by location.

3.2 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.

3.2.1 Layer

$\text{layer}([\text{unit}_1, \text{unit}_2, \dots], \text{resolve})$

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 composite views with incongruent internal structures, the *layer* operator restricts its operands to be *unit* views.

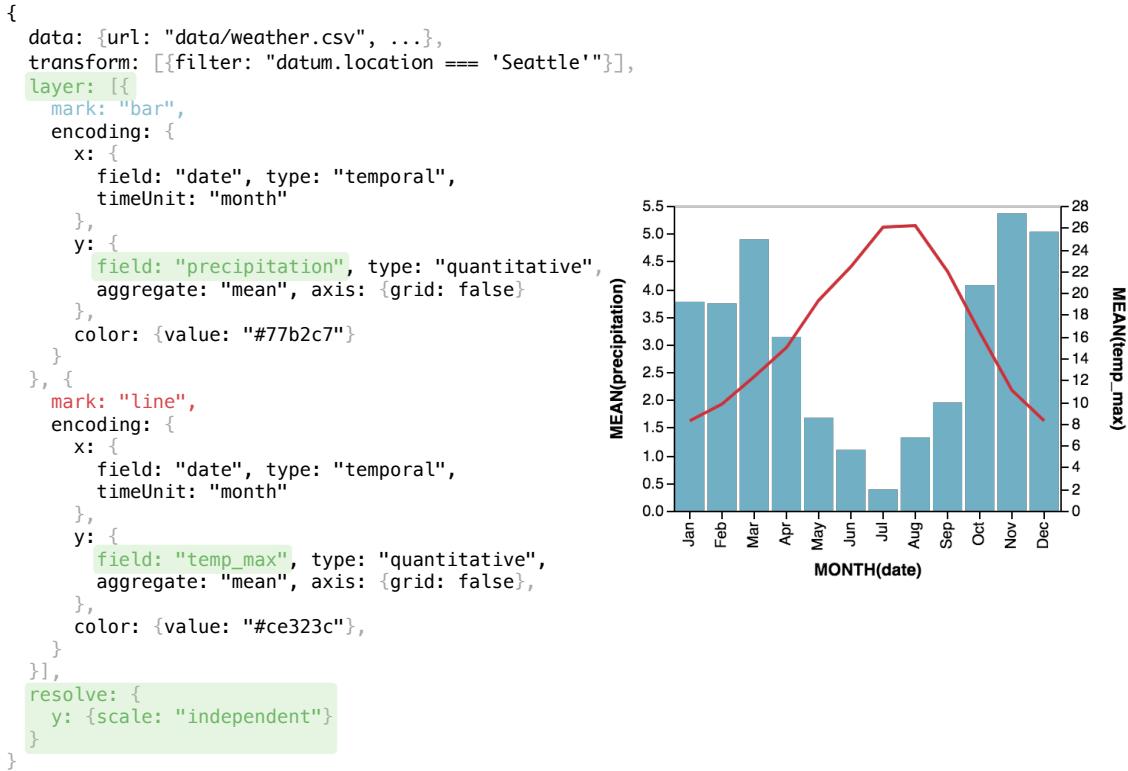


Figure 3.4: A dual axis chart that *layers* a line for the monthly mean temperature on top of bars for monthly mean precipitation. Each layer uses an *independent* y-scale.

3.2.2 Concatenation

```
hconcat([view1, view2, ...], resolve)
vconcat([view1, view2, ...], 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.

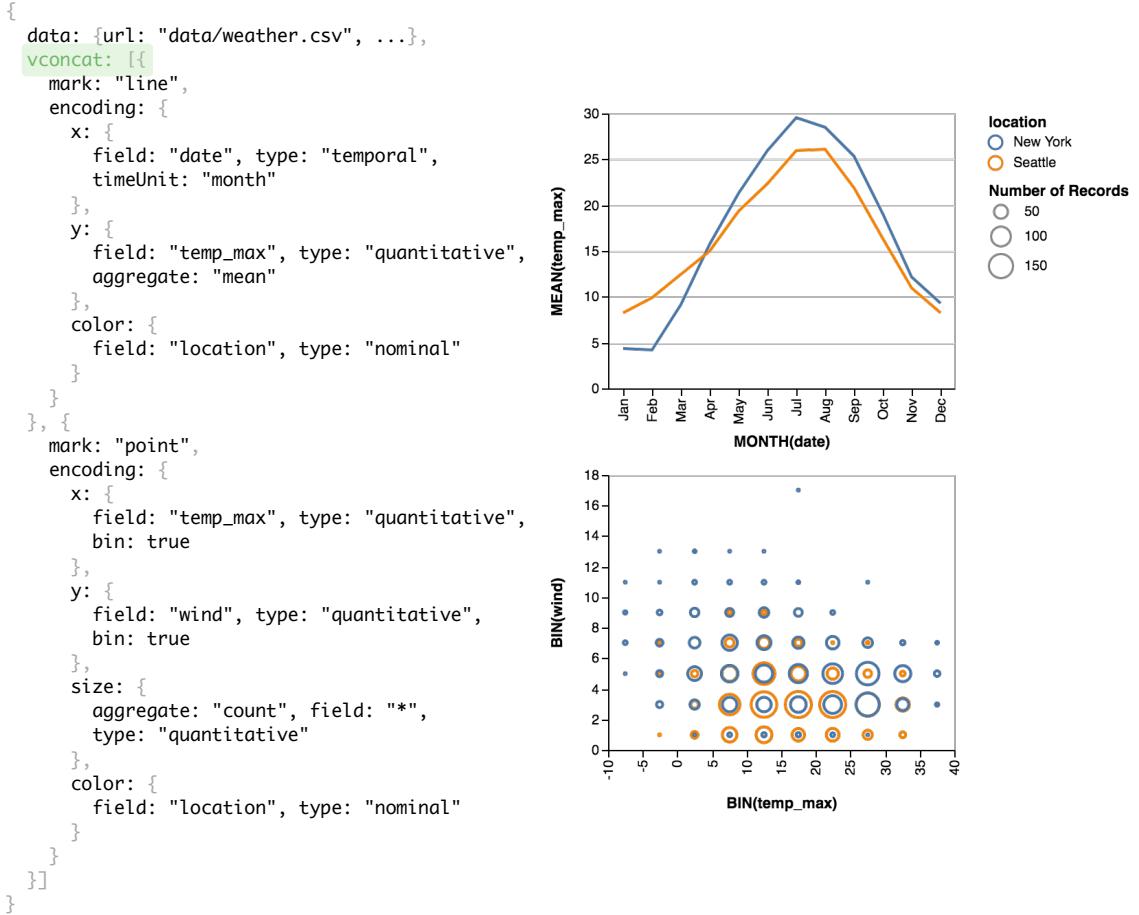


Figure 3.5: The Figs. 3.1 and 3.2 unit specifications *concatenated* vertically; scales and guides for each plot are independent by default.

3.2.3 Facet

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

The `facet` operator produces a trellis plot [1] 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"
      }
    }
  }
}
```

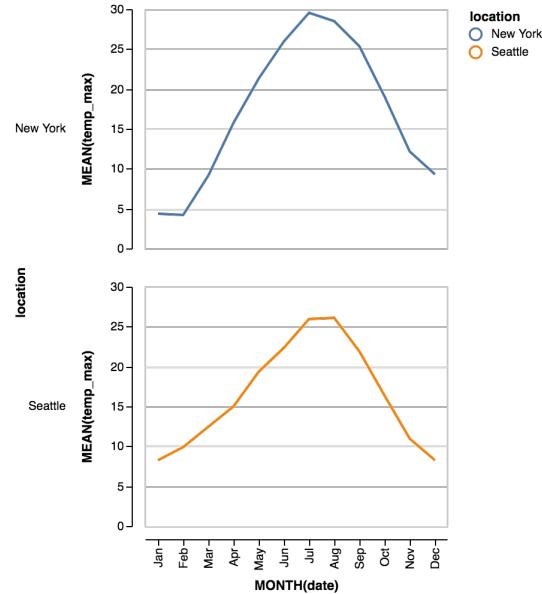


Figure 3.6: The line chart from Fig. 3.1 *faceted* vertically by location; the x-axis is shared, and the underlying scale domains unioned, to facilitate easier comparison.

3.2.4 Repeat

$\text{repeat}(\text{channel}, \text{values}, \text{scale}, \text{axis}, \text{view}, \text{resolve})$

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 plot¹. 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"
      }
    }
  }
}
```

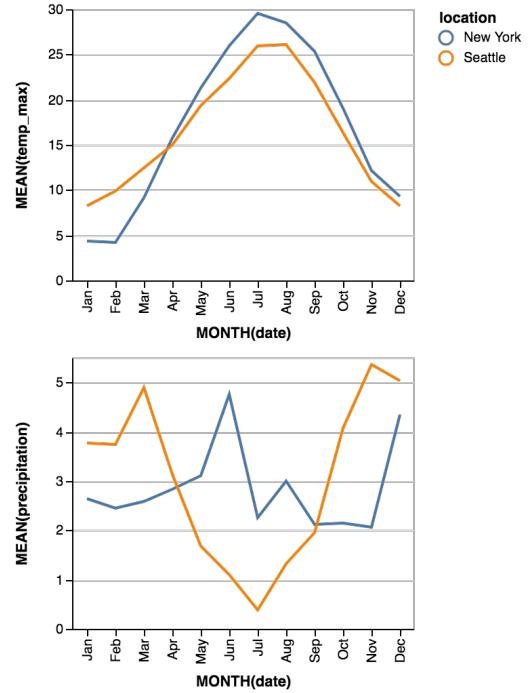


Figure 3.7: *Repetition* of different measures across rows; the y-channel references the *row* template parameter to vary the encoding.

¹As 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.

3.2.5 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 the *layer* operator, 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 facetting a composite view, only the dataset targeted by the operator is partitioned; any other datasets specified in sub-views are replicated.

3.3 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:

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

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 [18]), 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 3.8 and 3.9 illustrate how points are highlighted in a scatterplot using single and multi selections.

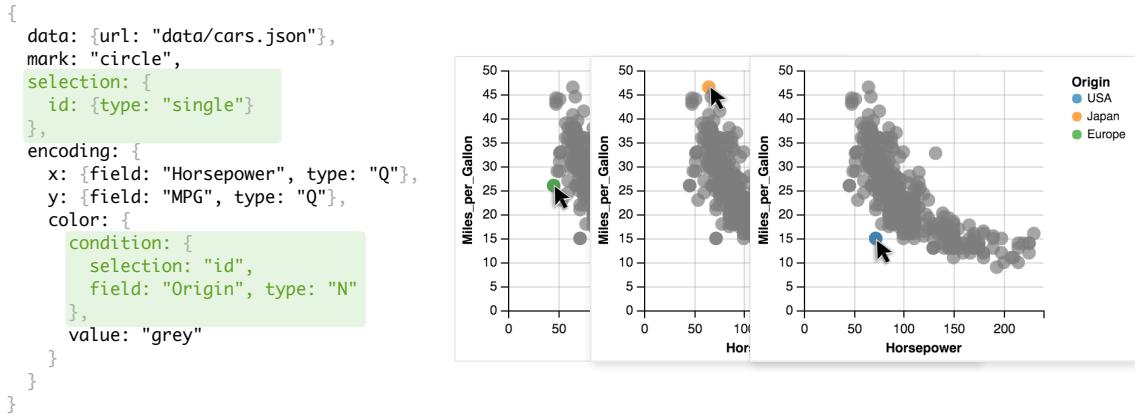


Figure 3.8: Adding a *single* selection to parameterize the fill color of a scatterplot's circle mark.

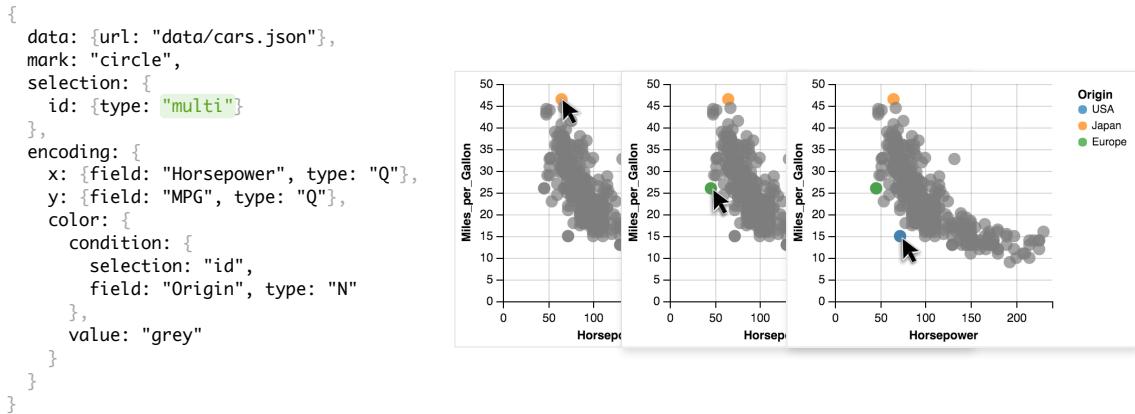


Figure 3.9: Switching from a *single* to *multi* selection. The first value is selected on click, and additional values on shift-click.

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. 3.10, 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: {
    region: {"type": "interval"}
  },
  encoding: {
    x: {field: "Horsepower", type: "Q"},
    y: {field: "MPG", type: "Q"},
    color: {
      condition: {
        selection: "region",
        field: "Origin", type: "N"
      },
      value: "grey"
    }
  }
}
```

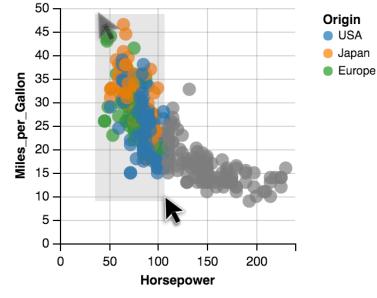


Figure 3.10: Highlight a continuous range of points using an *interval* selection. A rectangle mark is automatically added to depict the interval extents.

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 selector syntax [18]. For example, Fig. 3.11 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{100ms}` populates a selection at most every 100 milliseconds).

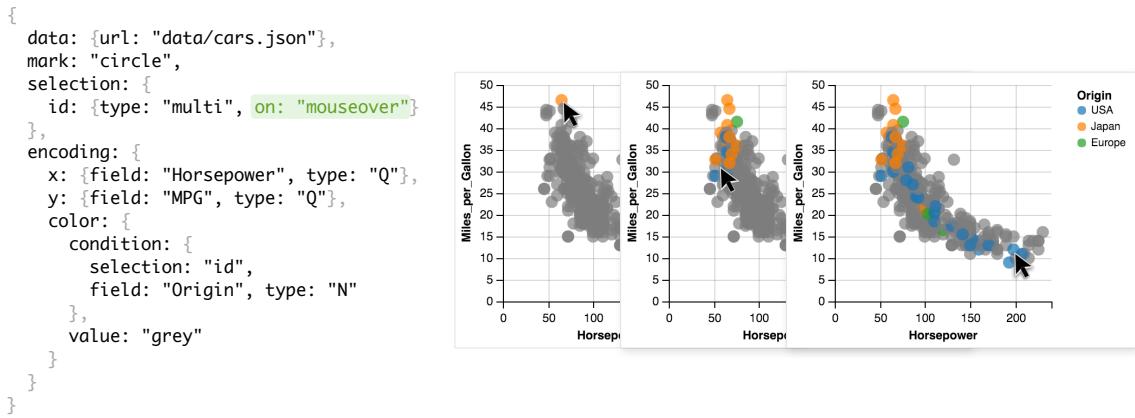


Figure 3.11: Specifying a custom event trigger for a *multi* selection: the first point is selected on `mouseover` and subsequent points when the shift key is pressed.

3.3.1 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"]
    }
  },
  ...
}
```

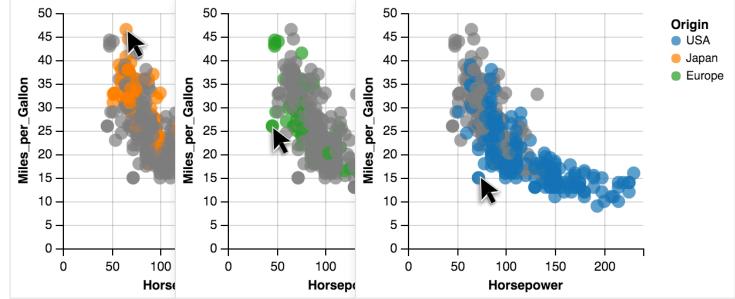
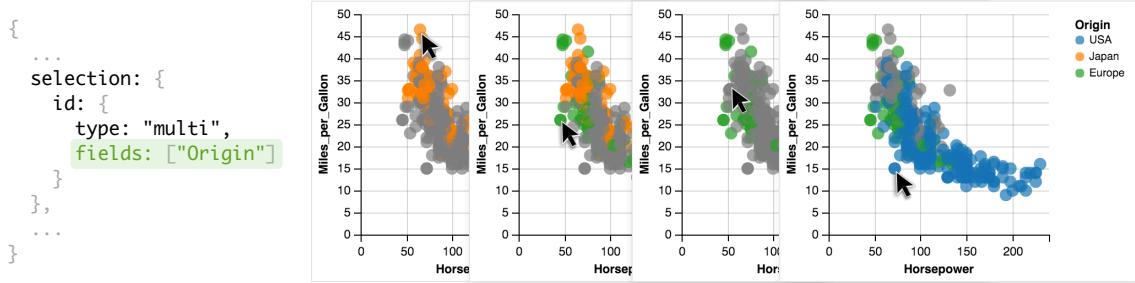
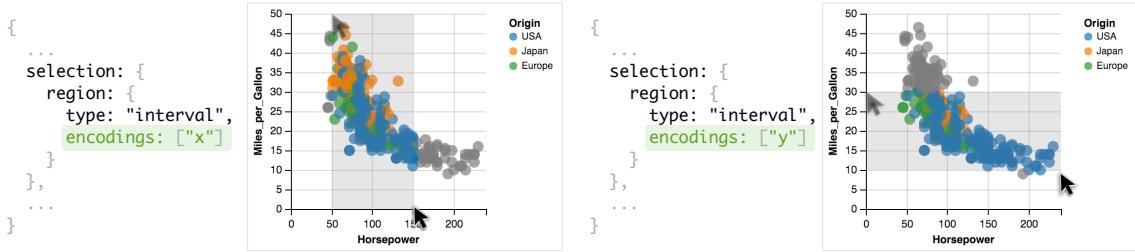


Figure 3.12: Using the *project* transform to highlight a *single* Origin.

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

Figure 3.13: Using the *project* transform to highlight *multiple* Origins.Figure 3.14: *Projecting* an interval selection to restrict it to a single dimension.

selection's triggering event, but with the shift key pressed. For example, in Fig. 3.9, additional points are added to the multi selection on shift-click (where `click` is the default event for multi selections). The selection in Fig. 3.11, 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.

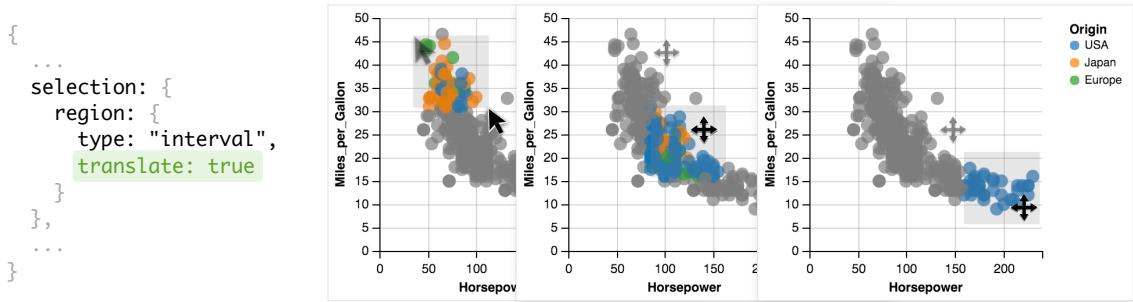


Figure 3.15: The *translate* transform enables movement of the brushed region. It is automatically invoked for interval selections but is explicitly depicted here for clarity.

Zoom

$$\text{zoom}(\textit{event}, \textit{factor})$$

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

$$\text{bind}(\textit{widgets}|\textit{scales})$$

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 [19]. 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.

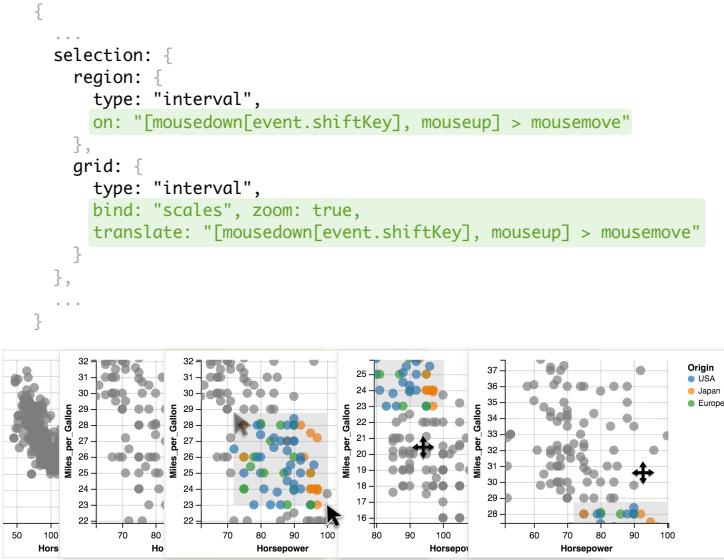


Figure 3.16: Panning and zooming the scatterplot is achieved by first *binding* an interval selection to the `x`- and `y`-scale domains, and then applying the *translate* and *zoom* transforms. Alternate events prevent collision with the brushing interaction, previously defined in Fig. 3.10

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 [7]). 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).

3.3.2 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. 3.8 and 3.9, 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. Initializing a selection with scale extents 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. 3.16) and overview + detail (Fig. 3.21).

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. 3.22, for instance, the red rule is positioned using the `date` value of the `indexPt` selection.

3.3.3 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. 3.17, 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.

```
{
  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"
      }
    }
  ]
}
```

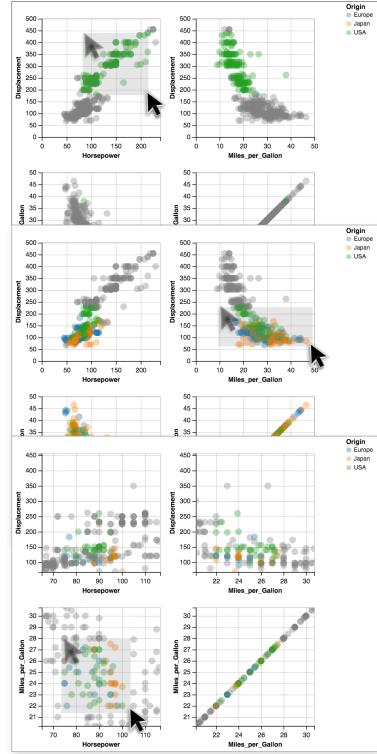


Figure 3.17: By adding a `repeat` operator, we compose the encodings and interactions from Fig. 3.16 into a scatterplot matrix. Users can brush, pan, and zoom within each cell, and the others update in concert. By default, a `global` composite selection is created: brushing in a cell replaces previous brushes.

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.

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, setting a selection to resolve 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.

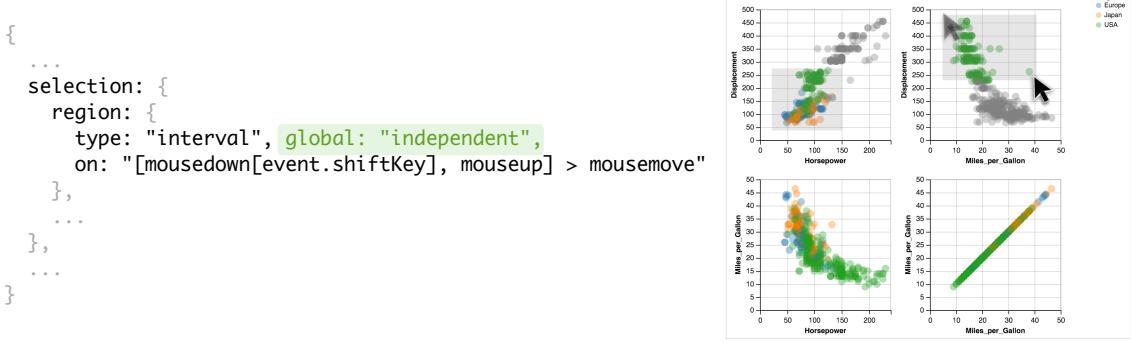


Figure 3.18: Resolving the `region` selection to *independent* produces a brush in each cell, and points only highlight based on the selection in their own cell.

Selections can also be resolved to *union* or *intersect*. In these cases, 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 Fig. 3.19 and Fig. 3.20 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 interactions, as in Figs. ?? & ??, where interactions within a view should not filter itself.

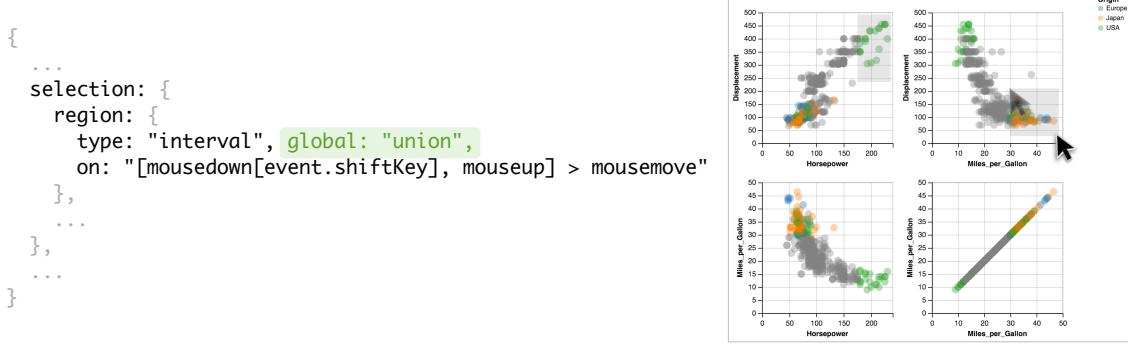


Figure 3.19: Resolving the `region` selection to `union` produces a brush in each cell, and points highlight if they fall within any of the selections.

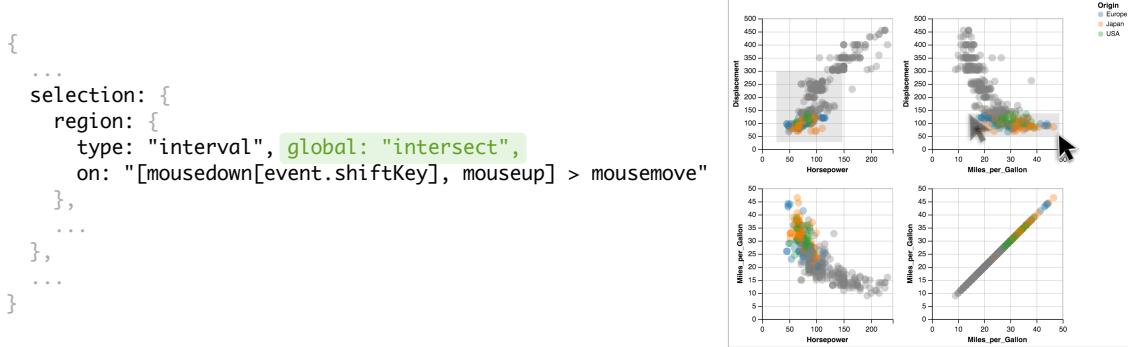


Figure 3.20: Resolving the `region` selection to `intersect` produces a brush in each cell, and points only highlight if they lie within all of the selections.

3.4 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 scatter-plot 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.

3.5 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 [26] 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 [17, 18, 21]. Where applicable, we also show how construction of our examples can be systematically varied to explore alternate points in the design space.

3.5.1 Selection: Click/Shift-Click and Brushing

Fig. 3.8 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. 3.9, allows users to mark multiple points (*toggle* is automatically instantiated by the compiler, but we explicitly specify it in the figure for clarity). We can instead add *project* (Fig. 3.12) 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. 3.13.

By using the *interval* type, users can mark items of interest within a continuous region. As shown in Fig. 3.10, the compiler automatically adds a rectangle mark to depict the selection, and instantiates *translate* to allow it to be repositioned (Fig. 3.15). In this context, *project* restricts the interval to a single dimension (Fig. 3.14).

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. 3.11. In contrast, with Vega, users need to manually author all the components of an interaction technique, including determining whether event properties need to be passed through scale inversions, creating necessary backing data structures, and adding marks to represent a brush component.

3.5.2 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. 3.16). 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. 3.11) but also necessary. We override the default events that trigger the two interactions using Vega’s event selector syntax [18]. As Fig. 3.16 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 in the output specification, 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 [4]), 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.

3.5.3 Connect: Brushing & Linking

We can wrap our previous example, from Fig. 3.16, in a *repeat* operator to construct a scatterplot matrix (SPLOM) as shown in Fig. 3.17. 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. 3.17). 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. 3.18 to 3.20 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.

3.5.4 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", ...}
      }
    }
  ]
}
```

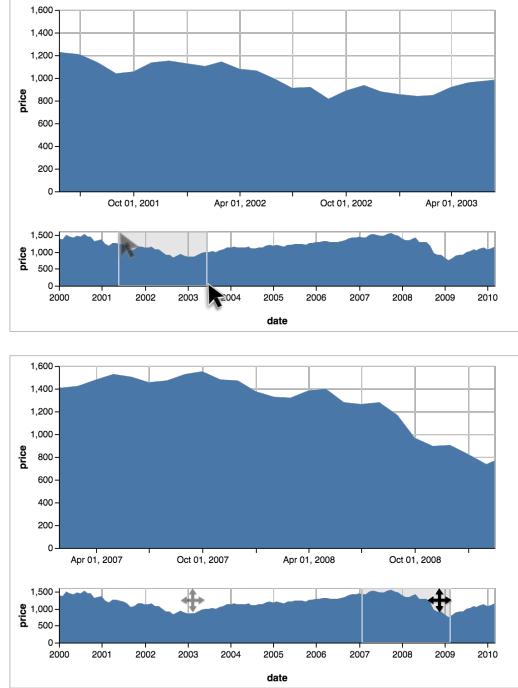


Figure 3.21: An overview + detail visualization concatenates two unit specifications, with a selection in the second one parameterizing the x-scale domain in the first.

3.5.5 Reconfigure: Index Chart

Figure 3.22 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"}
      }
    }
  ]
}
```

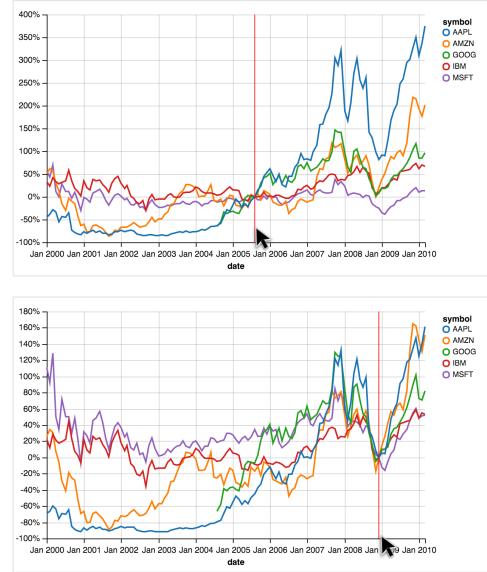


Figure 3.22: An index chart uses a single selection to renormalize data based on the index point nearest the mouse cursor.

3.5.6 Filter: Cross Filtering

As selections provide a predicate function, it is trivial to use them to filter a dataset. Figure 3.23, 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.

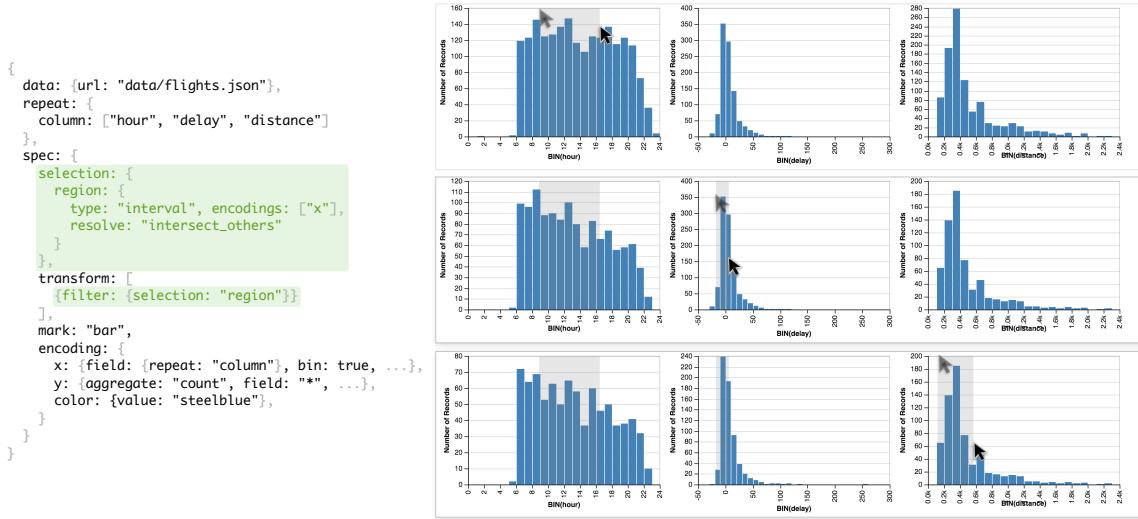


Figure 3.23: An interval selection, resolved to *intersect others*, drives a cross filtering interaction. Brushing in one histogram filters and reaggregates the data in the others, observable by the varying y-axis labels in the screenshots.

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. 3.23 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. 3.24). 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.

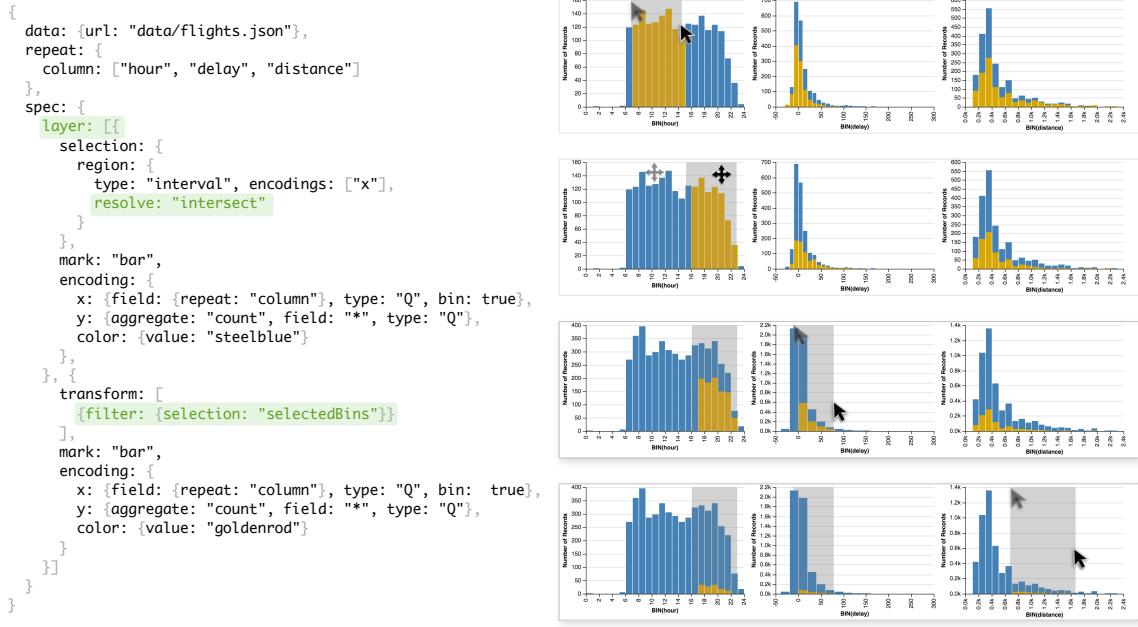


Figure 3.24: A layered cross filtering interaction is constructed by resolving the interval selection to *intersect*, and then materializing it to serve as the input data for a second layer. Highlights indicate changes to the specification from Fig. 3.23.

3.5.7 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 [10], 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 [6]). 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.

3.6 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 [?], R [? ?], Scala [?], Julia [?], and a REPL client for Clojure [?]. 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].*” 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.*”

Vega-Lite has also had an impact in the research community. It has been used

to reverse-engineer visualizations from chart images [?], build a model for sequencing visualizations [?], and powers the CompassQL recommendation engine [25?]. Such work is possible, in part, due to well-tested *effectiveness criteria* [2, 5, 12] 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.

Chapter 4

An Interactive Visualization Design Environment (VDE)

Bibliography

- [1] Richard A Becker, William S Cleveland, and Ming-Jen Shyu. The visual design and control of trellis display. *Journal of computational and Graphical Statistics*, 5(2):123–155, 1996.
- [2] Jacques Bertin. *Semiology of graphics: diagrams, networks, maps*. University of Wisconsin press, 1983.
- [3] Michael Bostock and Jeffrey Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans. Visualization & Comp. Graphics*, 15(6):1121–1128, 2009.
- [4] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics*, 17(12):2301–2309, 2011.
- [5] William S Cleveland and Robert McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554, 1984.
- [6] Samuel Gratzl, Nils Gehlenborg, Alexander Lex, Hanspeter Pfister, and Marc Streit. Domino: Extracting, comparing, and manipulating subsets across multiple tabular datasets. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2023–2032, 2014.
- [7] Tovi Grossman and Ravin 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.

- [8] Jeffrey Heer, Maneesh Agrawala, and Wesley Willett. Generalized selection via interactive query relaxation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 959–968. ACM, 2008.
- [9] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. Visual debugging techniques for reactive data visualization. *Computer Graphics Forum (Proc. EuroVis)*, 2016.
- [10] Christian Holz and Steven 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.
- [11] Zhicheng Liu and John T Stasko. Mental models, visual reasoning and interaction in information visualization: A top-down perspective. *IEEE Trans. Visualization & Comp. Graphics*, 16(6):999–1008, 2010.
- [12] Jock Mackinlay. Automating the design of graphical presentations of relational information. *Acm Transactions On Graphics (Tog)*, 5(2):110–141, 1986.
- [13] Dominik Moritz, Jeffrey Heer, and Bill Howe. Dynamic client-server optimization for scalable interactive visualization on the web. *Workshop on Data Systems for Interactive Analysis at InfoVis*, 2015.
- [14] Brad A Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proc. ACM UIST*, pages 211–220. ACM, 1991.
- [15] William A Pike, John Stasko, Remco Chang, and Theresa A O’Connell. The science of interaction. *Information Visualization*, 8(4):263–274, 2009.
- [16] Shiny by RStudio. <http://shiny.rstudio.com>, June 2016.
- [17] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. Reactive Vega: A streaming dataflow architecture for declarative interactive visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2016.

- [18] Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer. Declarative interaction design for data visualization. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pages 669–678. ACM, 2014.
- [19] Ben Shneiderman. Dynamic queries for visual information seeking. *IEEE software*, 11(6):70–77, 1994.
- [20] Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans. Visualization & Comp. Graphics*, 8(1):52–65, 2002.
- [21] Online Vega Editor. <http://vega.github.io/vega-editor/>, June 2016.
- [22] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2009.
- [23] Hadley Wickham. A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28, 2010.
- [24] Leland Wilkinson. *The Grammar of Graphics*. Springer, 2 edition, 2005.
- [25] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Trans. Visualization & Comp. Graphics*, 2015.
- [26] Ji Soo Yi, Youn ah Kang, John T Stasko, and Julie A Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1224–1231, 2007.