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

List of Tables

List of Figures

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 [21] and Polaris [18] (now Tableau).

Visual encodings are defined by composing graphical primitives called *marks* [4], 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, 14], existing declarative visualization models, including widely used tools such as D3 [5] and ggplot2 [20], do not offer composable primitives for interaction design. Instead, if they support interaction, they do so through either a palette of standard techniques [4, 5] or *imperative* event handling callbacks. While the former restricts expressivity, the later 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 [**?**, **?**, 13].

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

1.1 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 (**??**) inspired by CSS selectors. While prior work has formulated regex-based symbols for event selection [**?**], we believe our approach yields operators that will be more familiar to visualization designers.

A basic event stream selector is specified by a particular event type (e.g., `mousemove`), optionally prepended with the source of the events — either a mark type (e.g., `rect:`) or 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 2000 milliseconds). All operators are composable. For instance, `[mousedown[event.shiftKey], window:mouseup] > window:mousemove100, 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 [9]. 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.

Intensionall

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 [9]. Scale functions are a critical component in visualization design [21] as they transform data values into

visual values such as pixels or colours. 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 colour using scale-transformed data if predicate `A` is true, set it to yellow is predicate `B` is true, or otherwise set the colour to grey by default.

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. **??** illustrates how a brush interaction, extracted from **??**can be applied to a scatterplot matrix to produce a brushing & linking interaction.

BrushingSP

SelectionSn

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 [23]. 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

??provides a snippet of Reactive Vega JSON to highlight points that a user clicks. Figure A signal constructed over a click stream feeds a data transform that toggles values in a data source (named `selected_pts`). An intensional predicate test whether the shift key is pressed and, if not, clears the data source prior to inserting the clicked values. An extensional predicate is used within a production rule to set the fill colour selected points.

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

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 **??**. 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 — as shown in **??**, brushing over orange (`versicolor`) points would also highlight red (`virginica`) and blue (`setosa`) points. 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.

SPLOM

1.2.3 Abstract/Elaborate: Overview + Detail

With our brush interactor, we can also create the overview + detail visualization shown in Figure **??**. 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 draws a brush, signals update the horizontal range predicate, which in turn reactively filters points in the data source, updates scale functions and re-renders the detail view.

1.2.4 Explore & Encode: Panning & Zooming

1.2.5 Reconfigure: Index Chart

1.2.6 Filter: Control Widgets

Chapter 2

A Streaming Dataflow Architecture

Chapter 3

A Grammar of Interactive Graphics

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:

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

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

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

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

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

An encoding may also specify properties of a *scale* that maps from the data domain to a visual range, and a *guide* (axis or legend) for visualizing the scale. If not specified, Vega-Lite will automatically populate default properties based on the *channel* and *data-type*. For x and y channels, either a linear scale (for quantitative data) or an ordinal scale (for ordinal and nominal data) is instantiated, along with an axis. For *color*, *size*, and *shape* channels, suitable palettes and legends are generated. For example, quantitative color encodings use a single-hue luminance ramp, while nominal color encodings use a categorical palette with varied hues. Our default assignments largely follow the model of prior systems [18, 22].

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"
    }
  }
}
```

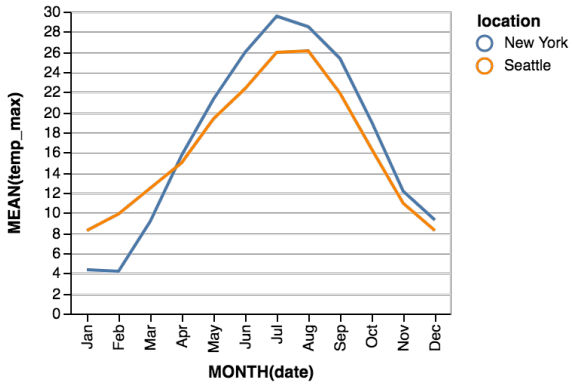


Figure 3.1: A unit specification that uses a *line* mark to visualize the *mean* temperature for every *month*.

```
{
  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"
    }
  }
}
```

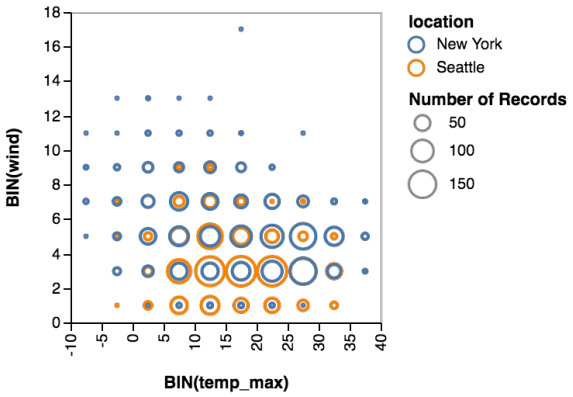


Figure 3.2: A *binned* scatterplot visualizes correlation between wind and temperature.

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

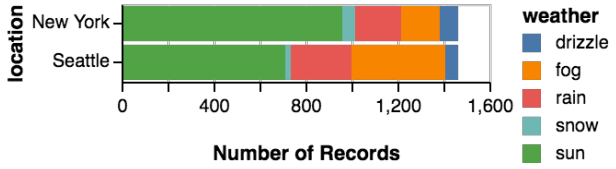


Figure 3.3: A *stacked* bar chart that sums the various weather types by location.

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 behaviours 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

layer([unit$_1$, unit$_2$, ...], resolve)

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

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

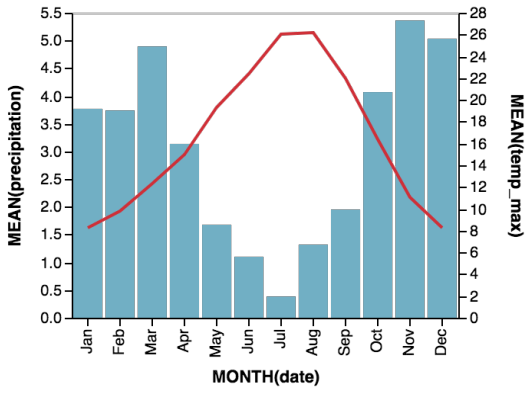


Figure 3.4: A dual axis chart that *layers* a line for the monthly mean temperature on top of bars for monthly mean precipitation. Each layer uses an *independent* y-scale.

3.2.2 Concatenation

$$hconcat([view_1, view_2, ...], resolve)$$
$$vconcat([view_1, view_2, ...], resolve)$$

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

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

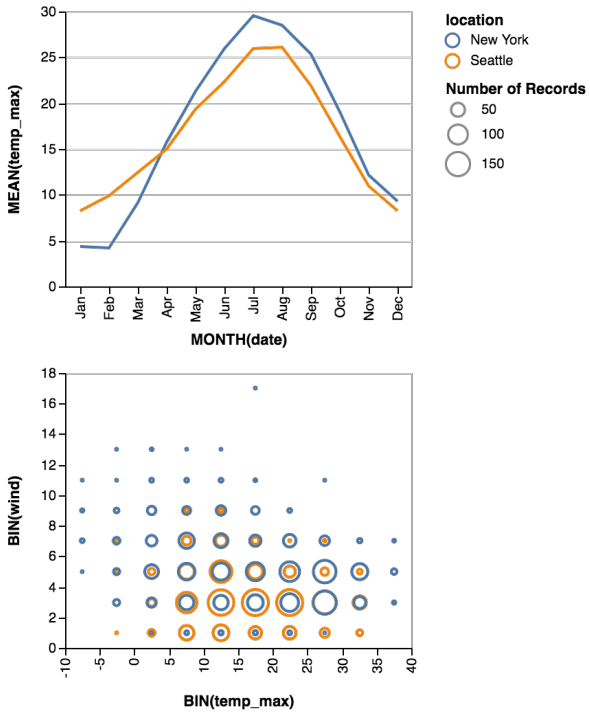


Figure 3.5: The Figs. 3.1 and 3.2 unit specifications *concatenated* vertically; scales and guides for each plot are independent by default.

3.2.3 Facet

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

The *facet* operator produces a trellis plot [2] 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.

3.2.4 Repeat

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

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

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

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

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 faceting 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:

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

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

As its name suggests, a single selection is backed by one datum, and its predicate tests for an exact match against properties of this datum. It can also function like a dynamic variable (or *signal* in Vega [16]), 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. **??**illustrates how points are highlighted in a scatterplot using point PointSelect and list selections.

Intervals are similar to multi selections. They are backed by datasets, but their

predicates determine whether an argument falls within the minimum and maximum extent defined by the backing points. Thus, they express continuous selections. The compiler automatically adds a rectangle mark, as shown in **??**, 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.

IntervalSele

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 [16]. For example, **??**demonstrates how `mouseover` events are used to populate a list 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,`

PointSelect

`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).

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 behaviours.

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 transformations). For such cases, a list of *channels* may also be specified (e.g., `color`, `size`). **??**demonstrate how *project* can be used to select all points with matching `Origin` fields, for example. This transform is also used to restrict interval selections to a particular dimension (**??**).

PointSelect
e)

IntervalSele

Toggle

toggle(event)

The *toggle* transform is automatically instantiated for uninitialized multi selections. When the *event* occurs, the corresponding data value is added or removed from the multi selection's backing dataset. By default, the toggle *event* corresponds to the selection's triggering event, but with the shift key pressed. For example, in **??**, additional points are added to the list selection on shift-click (where `click` is the default event for list selections). The selection in **??**, 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.

PointSelect

PointSelect

Bind

bind(widgets|scales)

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

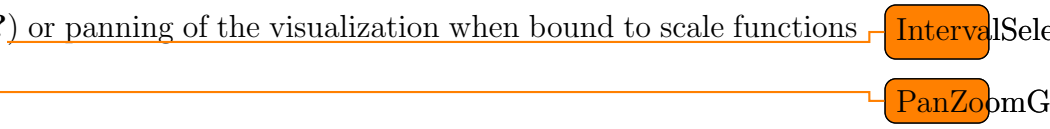
When a single selection is bound to query widgets, one widget per projected field is generated and may be used to manipulate the corresponding predicate clause. When triggering events occur to update the selected points, the widgets are updated as well. Control widgets, in addition to direct manipulation interaction, allow for more rapid and exhaustive querying of the backing data [17]. 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.

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 transforms, enabling movement of brushed regions (**??**) or panning of the visualization when bound to scale functions (**??**).



Zoom

zoom(event, factor)

The *zoom* transform applies a scale factor, determined by the *event* to the spatial properties (or corresponding data fields) of backing points. A *factor* must be specified if it cannot be determined from the events (e.g., when arrow keys are pressed). As with the *translate* transform, the *project* transform is respected, allowing for uni-dimensional zooming.

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 [8]). 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 **??**, 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 (**??**) and overview + detail (**??**).

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. Properties of 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. For example, the position of the red rule in **??** is set to 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 **??**, 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.

ResolveSele

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 in **??**.Each cell would display its own brush, which would determine how only its points would be highlighted.

ResolveSele (b)

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 **??**.We also support

ResolveSele d)

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.

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. Handcrafted rules are applied to produce perceptually effective visualizations. For example, if the color channel is mapped to an nominal field, and the user has not specified a scale domain, a categorical color palette is inferred. If the color is mapped to a quantitative field, a sequential color palette is chosen instead.

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

or should it be loaded from a URL, and in what format), which fields should be aggregated or binned, and what filters and calculations should be performed.

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

3. *Merge* — once the necessary components have been built, the compiler performs a bottom-up traversal of the model tree to merge redundant components. This step is critical for ensuring that the resultant Vega specification does not perform unnecessary computation that might hinder interactive performance. To determine whether components can be merged, the compiler computes a hash code and compares components of the same type. For example, when a scatterplot matrix is specified using the *repeat* operator, merging ensures that we only produce one scale for each row and column rather than two scales per cell ($2N$ versus $2N^2$ scales). Merging may introduce additional components if doing so results in a more optimal representation. For example, if multiple units within a composite specification load data from the same URL, a new `DataComponent` is created to load the data and the units are updated to inherit from it instead. This step also unions scale domains and resolves `SelectionComponents`.

4. *Assemble* — the final phase assembles the requisite Vega specification. In particular, `SelectionComponents` produce signals to capture events and the necessary backing points, and list and intervals 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 [23] 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 [15, 16, 19]. 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

??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 parameterized with conditional encoding logic.

PointSelect

Modifying a single property, *type*, as in **??**, 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* (**??**) 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

PointSelect

PointSelect

composed as shown in **??**.

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

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 **??**. 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 (**??**). 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 **??**) but also necessary. We override the default events that trigger the two interactions using Vega's event selector syntax [16]. As **??**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 [5]), 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 **??**, in a *repeat* operator to construct a scatterplot matrix (SPLOM) as shown in **??**. 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.

PanZoomG

ResolveSele

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 (**??**). With the *resolve* property, users can specify alternate disambiguation methods including creating an independent brush for each cell, unioning the brushes, or intersecting them (**??**respectively). If selections are bound to scales or parameterize them, only a global selection is supported for consistency with the composition algebra.

ResolveSele

ResolveSele c, d)

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. **??**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.

ODIndexCh

3.5.5 Reconfigure: Index Chart

??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

IndexChart

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.

3.5.6 Filter: Cross Filtering

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

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 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 (). 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.

3.6 Limitations & Future Work

The previous section demonstrates 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 list 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 [7]). 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.

An early version of Vega-Lite is used to automatically recommend static plots as part of the Voyager browser [22]. Voyager leverages perceptual *effectiveness criteria* [3, 6, 12] to rank candidate visual encodings. One promising avenue for future work is to develop models and techniques to analogously recommend suitable interaction methods for given visualizations and underlying data types. Here, the human-computer interaction literature [1] may offer insights on when and why users benefit

from certain interaction techniques.

Vega-Lite is an open source system available at `http://vega.github.io/vega-lite/`. By offering a multi-view grammar of graphics tightly integrated with a grammar of interaction, Vega-Lite facilitates rapid exploration of design variations. Ultimately, 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] Michel Beaudouin-Lafon. Instrumental interaction: an interaction model for designing post-wimp user interfaces. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 446–453. ACM, 2000.

[2] 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.

[3] Jacques Bertin. *Semiology of graphics: diagrams, networks, maps*. University of Wisconsin press, 1983.

[4] Michael Bostock and Jeffrey Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans. Visualization & Comp. Graphics*, 15(6):1121–1128, 2009.

[5] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics*, 17(12):2301–2309, 2011.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[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] Brad A Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proc. ACM UIST*, pages 211–220. ACM, 1991.

[14] William A Pike, John Stasko, Remco Chang, and Theresa A O'Connell. The science of interaction. *Information Visualization*, 8(4):263–274, 2009.

[15] 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.

[16] 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.

[17] Ben Shneiderman. Dynamic queries for visual information seeking. *IEEE software*, 11(6):70–77, 1994.

[18] 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.

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

[20] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2009.

[21] Leland Wilkinson. *The Grammar of Graphics*. Springer, 2 edition, 2005.

[22] 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.

[23] 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.