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

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 [5, 6].

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"
    }
  }
}
```

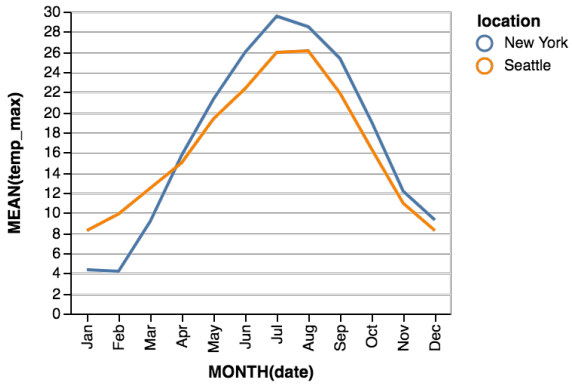

Figure 3.1: A unit specification that uses a *line* mark to visualize the *mean* temperature for every *month*.

```
{
  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"
    }
  }
}
```

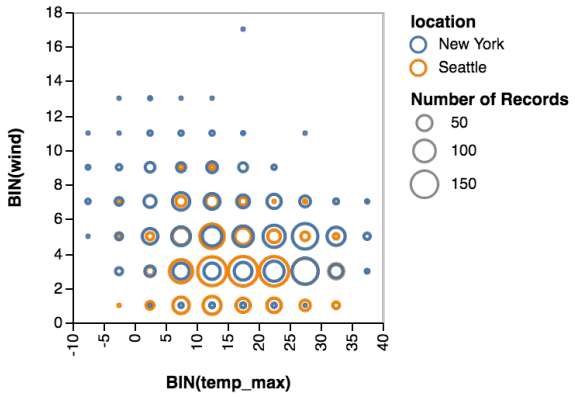

Figure 3.2: A *binned* scatterplot visualizes correlation between wind and temperature.

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

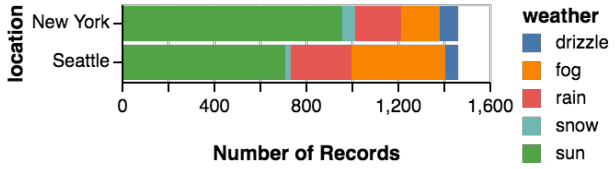

Figure 3.3: A *stacked* bar chart that sums the various weather types by location.

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"}
  }
}
```

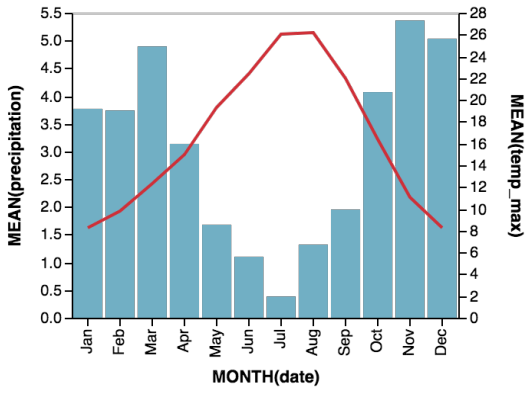

Figure 3.4: A dual axis chart that *layers* a line for the monthly mean temperature on top of bars for monthly mean precipitation. Each layer uses an *independent* y-scale.

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"
      }
    }
  }]
}
```

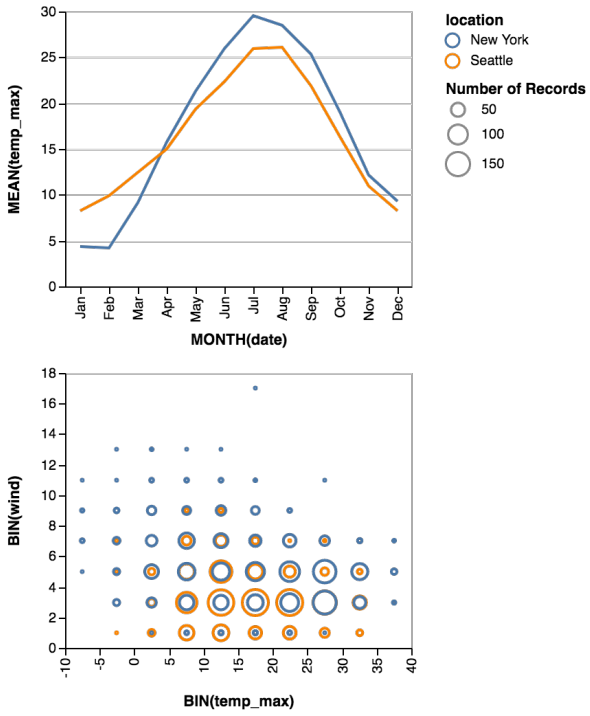

Figure 3.5: The Figs. 3.1 and 3.2 unit specifications *concatenated* vertically; scales and guides for each plot are independent by default.

3.2.3 Facet

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

The *facet* operator produces a trellis plot [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.

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 [3]), 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 and PointSelect 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 [3]. 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 [4]. 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 (). IntervalSele PanZoomG

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 single-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 [2]). 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.

PointSelect

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 ().

PanZoomG

ODIndexCh

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.

ODIndexCh

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.

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] 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.

[3] 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.

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

[5] 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.

[6] 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.