Dziban: Balancing Agency & Automation in Visualization Design via Anchored Recommendations

Halden Lin
University of Washington
haldenl@cs.washington.edu

Dominik Moritz
University of Washington
domoritz@cs.washington.edu

Jeffrey Heer
University of Washington
jheer@uw.edu

ABSTRACT

Visualization recommender systems attempt to automate design decisions spanning choices of selected data, transformations, and visual encodings. However, across invocations such recommenders may lack the context of prior results, producing unstable outputs that override earlier design choices. To better balance automated suggestions with user intent, we contribute Dziban, a visualization API that supports both ambiguous specification and a novel anchoring mechanism for conveying desired context. Dziban uses the Draco knowledge base to automatically complete partial specifications and suggest appropriate visualizations. In addition, it extends Draco with chart similarity logic, enabling recommendations that also remain perceptually similar to a provided "anchor" chart. Existing APIs for exploratory visualization, such as ggplot2 and Vega-Lite, require fully specified chart definitions. In contrast, Dziban provides a more concise and flexible authoring experience through automated design, while preserving predictability and control through anchored recommendations.

Author Keywords

visualization; recommendation; anchoring; language

CCS Concepts

•**Human-centered computing** → **Visualization systems and tools;** *Visualization toolkits;* •**Software and its engineering** → *Domain specific languages;*

INTRODUCTION

Data analysts must balance ease of use with control when choosing a visualization authoring tool. Visualization recommender systems [10, 13, 14, 22, 26] have the potential to provide more effective and efficient exploration of data by automating decisions over selected data, transformations, and visual encodings that are normally required from users of full-specification *APIs* (application programming interfaces) such as Vega-Lite [18] and ggplot2 [24]. These full-specification APIs, in contrast, offer tight control over output visualizations when recommendation systems may be forced to compromise in the face of ambiguous user intent (Figure 1). Indeed, many exploratory analysis tools (e.g., Voyager [27, 28], Tableau) mix the two authoring paradigms to marry agency and automation [9] and provide a more efficient and amenable visualization authoring experience.

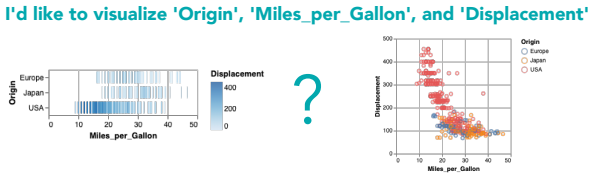


Figure 1. Which chart should a recommender suggest? Recommender systems are often forced to make decisions in the face of ambiguous user intent. Sometimes, these decisions will hamper exploration.

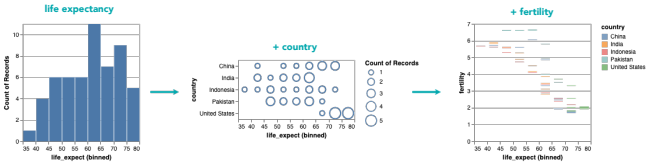


Figure 2. A series of recommendations by the Draco [14] recommender system. Inconsistency in channel assignments and marks can be found between the addition of each new field.

The mixture of manual and automated methods is particularly evident for *visualization query refinement*, when an analyst asks follow-on questions (e.g., adding fields to their query) or modifies a visualization to better answer an existing question (e.g., changing data transformations). Recommendations resulting from iteration on an ambiguous *partial* specification can lack stability or coherence relative to their context (e.g., when a user wishes to inspect fields A and B, but has not specified data transformations or encoding channel assignments, see Figure 2). In these cases, refinements to the specification may produce stark discontinuities among output charts. This difference can result in a high cognitive cost for people as they attempt to (1) make sense of the new visualization, which may possess inconsistent channel assignments, scales, or other graphical and data properties, and (2) elaborate the specification in order to match the context of their exploration.

Reliance on design decisions by people as a solution for resolving ambiguous intent creates a burden for users of recommendation systems, who may lack expertise or be averse to the tedium of this process. With respect to efficiency and approachability, tools built around recommendation systems can be bottle-necked by this crutch. Voyager [27] is an example of a tool that attempts a more elegant solution to the visualization refinement problem. Voyager allows users to "lock" the data and encoding properties of a recommended visualization

CHI '20, April 25–30, 2020, Honolulu, HI, USA.

ACM ISBN 978-1-4503-6708-0/20/04 ...$15.00.
http://dx.doi.org/10.1145/3313831.3376880

(among a set recommendations) before proceeding with refinement. However, this restriction can lead to ineffective (or even inexpressive) visualizations if data transformations or other graphical properties are subsequently modified.

Moreover, it is unclear how to design interactions for a handoff between recommender and user in programming environments. This may, in part, contribute to the lack of a mainstream recommender-powered visualization API. Instead, programmers rely on full-specification APIs for their visualization needs, without access to a dedicated recommender system.

To address these challenges, we present *Dziban*, a novel visualization API. Dziban differs from existing visualization APIs for exploratory visualization, such as Vega-Lite and ggplot2, by accepting *partial specifications* and incorporating recommendation, removing the need for complete specification of data transformations and visual encodings. In addition, Dziban contributes a novel *anchoring* mechanism for conveying desired context for recommendations as a solution for recommender-powered query refinement. With *anchored recommendations*, users provide Dziban with an "anchor" chart to supplement their specification query. Dziban then provides recommendations that are perceptually similar to the anchor. By not requiring manual specification or "locking" of field-channel assignments and other encoding properties, Dziban offers visualization refinement that is automated, yet flexible.

Internally, Dziban uses the Draco knowledge base [14] to generate recommendations. We contribute extensions to Draco to enable reasoning over multiple charts, and integrate the chart similarity logic of GraphScape [11] to support anchoring as an additional soft constraint. The Dziban API is implemented in Python and intended for use in Jupyter notebook environments, where interactive programming lends itself to exploration context and iteration.

We first present Dziban's design and implementation. Next, we demonstrate its usage, focusing on the merits of anchored recommendations. We argue that anchored recommendations are particularly beneficial when refinement of an existing visualization query is desired. Next, we present a benchmark evaluation of anchored recommendations, demonstrating a favorable trade-off between reduced context-free effectiveness and increased perceptual similarity. Finally, we discuss future work on Dziban and potential applications of its core ideas. Dziban is available as open source at github.com/uwdata/dziban.

BACKGROUND & RELATED WORK

Dziban draws on prior work in visualization specification, automated design, multi-view consistency, and chart sequencing.

Visualization Specification

Visualization authoring tools typically use domain-specific abstractions to formalize their design space. In particular, Wilkinson's Grammar of Graphics [25] has inspired many visualization grammars and specification APIs.

Vega-Lite [18] is a high-level grammar that maps data to visual properties. Dziban uses Vega-Lite as its output format. The Vega-Lite API [23] wraps this grammar in a convenient interface for use in JavaScript programming environments. Altair [21] is an analogous Python API for the Vega-Lite grammar. Both the Vega-Lite API and Altair provide interfaces that map nested objects to a resulting Vega-Lite JSON dictionary. Taking a slightly different approach, ggplot2 [24] abstracts visualizations as a sum of graphical layers. Its R API presents this paradigm in its language design, were the addition operator is used to add elements together.

These APIs require full specification of chart encodings; users can omit low-level details such as scales and guides (i.e., axes and legends) that are synthesized by the tool. In contrast, Dziban takes advantage of a visualization recommendation system to allow users to omit visualization properties as desired, increasing efficiency [13, 27]. Dziban searches a design space of possible visualizations, using a set of scoring functions to suggest one appropriate complete visualization specification for the user's query. The user can then modify the result if adjustments are required. While Dziban takes inspiration from the design of the Vega-Lite and Altair APIs, it flattens their specification interface to promote iterative refinement of visualizations rather than complete re-declaration.

Automated Visualization Design

Automated visualization design systems vary greatly in implementation and design. Dziban draws from a breadth of existing tools, both graphical and non-graphical.

Mackinlay's APT [12] asks users for a dataset and ranked list of fields of interest as input. It then enumerates candidate visualizations, prunes those that violate *expressiveness* criteria, and ranks remaining candidates using perceptual *effectiveness* criteria. Expressiveness refers to a graphic's ability to convey no more and no less than the facts provided by the data. Effectiveness refers to a graphic's ability to convey these facts in a form that is perceptually consumable by viewers. APT uses ranked lists of the presumed effectiveness of encoding channels (x, y, color, etc.) based on the field datatype (quantitative, ordinal, nominal), informed by the work of Bertin [3], Cleveland & McGill [5], and others. Unlike Dziban, APT recommends visualizations without context.

Dziban's goal of recommender-powered exploratory analysis takes inspiration from existing recommender-powered end-user interfaces. SeeDB [22] analyzes statistical properties of a dataset to suggest a set of visualizations that may be of interest. ZenVisage [19] recommends visualizations that match hand-drawn sketches of a pattern of interest. DIVE [10] is a mixed-initiative system that recommends lists of visualizations from selected fields. Users can then select charts of interest, perform statistical analysis, and construct stories. Using Voyager [27, 28], analysts specify data or visualizations of interest, and a gallery of recommended views is presented, facilitating more systematic exploration. Tableau's Show Me [13] suggests graphical encodings based on selected data. Tableau's Ask Data provides a natural language interface that recommends visualizations in response to user questions; Tory and Setlur [20] describe their process of designing recommendations that are context-sensitive and (like Dziban) enforce consistency to provide more goal-oriented graphics. Dziban aims to increase agency in visualization modification by allowing users to modify a breadth of visualization properties

(as compared to SeeDB, ZenVisage, DIVE, Ask Data), while removing reliance on manual specification (Show Me) or inflexible locking of visualization properties (Voyager).

CompassQL [26] is a visualization query language that powers recommendations in Voyager. Like CompassQL, Dziban uses Vega-Lite [18] to express output charts. However, CompassQL cannot recommend visual encoding and data transformation properties without explicit prompting. For example, one must specify a transformation (e.g., aggregation) as desired for an aggregation function to be chosen.

Draco

Draco [14] is a more recent design knowledge base that expands on of the principles of APT and CompassQL. It is more flexible in both user input and scoring logic to find appropriate recommendations. Draco is designed to accept arbitrary properties of a visualization specification (in contrast to APT) and other information, such as analysis task (in contrast to CompassQL), and recommends visualizations based on a system of weighted constraints over potential visualization properties. Draco is written in a logical programming language, Answer Set Programming (ASP) [4]. Answer set programming enables the encoding of design rules and preferences as logical statements. For example:

```
:~ type(E,nominal), channel(E,x). [1]
```

states a general preference against encoding nominal variables on the x channel (as rotated text is more difficult to read). Violating this constraint incurs a cost of 1.

Using these constructs, Draco has the expressive power to model a variety of complex inputs and utility functions. Dziban is built on top of Draco and exploits this flexibility. As described later, Dziban extends Draco with additional chart similarity logic to support anchored recommendations.

Multi-View Consistency and Visualization Sequencing

Dziban's anchored recommendations draw from a body of work on multi-view consistency for the purposes of both exploration and presentation. Baldanado et al. [1] present eight guidelines for the design of multiple view systems. For each, they discuss the impacts of these guidelines on cognitive utility for viewers; for example, "consistency" can "facilitate comparison and learning" [1]. Qu and Hullman [16] present a Wizard-of-Oz study to assess the role of multi-view consistency in exploratory visual analysis and presentation. In it, they document the trade-offs made by users to either achieve or ignore consistency between visualizations. They extend prior work on consistency guidelines [1, 6, 15] by proposing a set of constraints for use in consistency checking (such as "same field $\rightarrow$ same color mapping") and discuss design opportunities for visualization tools. Existing visual analysis tools also apply some consistency principles, such as the use of consistent scale domains across charts in Voyager [27, 28].

While these works focus on consistency between views in dashboard visualizations (where diverse views are adjacent to each other), Dziban is concerned with similar views across an interactive session. Moreover, Dziban does not contain explicit constraints for multi-view consistency. Rather, it uses perceptual similarity as a flexible measure to guide the modification of data transformations and graphical encoding properties. Nonetheless, the core objective is similar: anchored recommendations attempt to preserve consistency between visualizations authored by the user.

GraphScape

Dziban uses GraphScape [11] as its model for perceptual distance between arbitrary pairs of visualizations. Unlike APT, CompassQL, and Draco, GraphScape is an automated design system that focuses on the recommendation of *sequences* of visualizations, rather than single views. It contributes a model of chart similarity between Vega-Lite visualizations, based on atomic editing operations one can apply to a chart. In GraphScape, nodes represent visualizations and weighted edges represent modifications to them. A path from one visualization to another represents one possible sequence of edits that achieve the transformation. The weight of an edge represents the relative cognitive cost of that modification. For example, moving an encoded data field from the x-axis to the y-axis may induce less interpretation cost than changing to a color encoding. The cost of a path represents the perceptual cost of moving from its source to its destination. The weights of GraphScape are computed by solving a linear program, and do not represent an absolute measure of perceptual distance. They can, however, be used to rank order perceptual distance between pairs of visualizations. GraphScape was verified in user studies that asked participants to rank sequences of visualizations based on their ease of interpretation.

DESIGN CONSIDERATIONS

The design of Dziban was driven by several considerations:

C1. Facilitate iterative development of visualizations.

Exploratory visual analysis is a conversational process. An analyst begins by asking a question and authoring a visualization to answer it. The analyst learns from the result, which may prompt further questions, and the process repeats. Battle & Heer [2] found that exploratory analysis is made up of depth-oriented sessions: more often going down an exploration tree (iterating on existing questions) rather than across (asking entirely new questions). Building on this insight, Dziban adopts an iteration-oriented, functional programming paradigm. Instead of declaring entire specifications in a single invocation, edits are chained together to construct immutable chart objects. In contrast to existing APIs such as Altair [21] and Vega-Lite [18], which use nested objects for specification, Dziban has a flat, relational model of visualization properties.

C2. Guide recommendations towards user goals.

While recommendations can be powerful, the ambiguity of partial specifications can result in inconsistent outputs when iteratively updating queries (Figure 2). Dziban employs *anchored recommendations* to address this issue. An anchored recommendation accepts a supplementary chart (the "anchor") and presents a visualization that is similar. In this way, users guide Dziban via charts they have already constructed by simply referencing them as anchors.

C3. Flexibly apply automation, but uphold user agency. Properties explicitly specified by a user should be taken as-is. However, design decisions made by Dziban may be reconsidered in subsequent invocations, lest the system dig itself into an inexpressive or ineffective hole. To this end, Dziban's anchored recommendations allow visualization properties present in the "anchor" to be modified, so long as they are not explicitly specified by the user. In other words, anchors act as soft constraints on the recommendation process.

THE DZIBAN API

We now present the basics of query construction, modification, and anchored recommendation in the Dziban API. The examples below assume Dziban use within Jupyter notebooks.

Query Construction

Dziban exposes its API through the `Chart` object. `Charts` accept a Pandas data frame as input.

```
from dziban import Chart
from vega_datasets import data

movies = data('movies')  # a Pandas dataframe
base = Chart(movies)  # constructs an empty query
```

Editing a Query

Queries are modified by invoking functions on a `Chart` object. The `field` function accepts a column name to visualize.

```
# view the values of the IMDB_Rating column
imdb_ratings = base.field('IMDB_Rating')
```

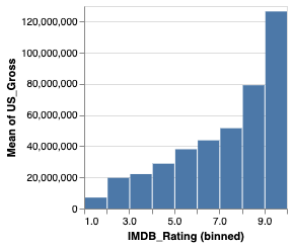
For iterative development (C1) users can update queries by building on top of existing ones. The `field` function also accepts graphical properties and data transformations as input.

```
# IMDB_Rating by mean(US_Gross)
imdb_by_us_gross = imdb_ratings.field(
    'US_Gross',
    aggregate='mean'
)
```

Rendering

To render a query, a `Chart` is listed as output for a notebook cell. Upon execution, Dziban's recommendation system is invoked and the top-ranked visualization is returned.

```
imdb_by_us_gross
```

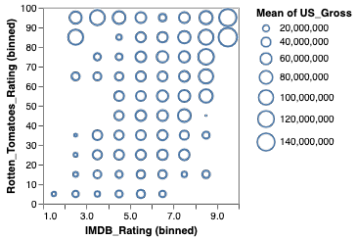


Note that `IMDB_Rating` was automatically binned by Dziban to preserve expressiveness, and mark and channel assignments were chosen by effectiveness criteria modeled in Draco [14].

Cold and Anchored Recommendations

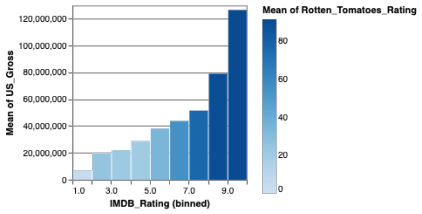
Dziban's default behavior is to provide *cold* (or context-free) recommendations. Here, we add a field, the Rotten Tomatoes Rating of movies, to our query:

```
# cold recommendation
imdb_by_us_gross.field('Rotten_Tomatoes_Rating')
```



Notice that this visualization, while effective if the analyst's intent is to gain an understanding of `US_Gross` as a function of the other variables, does not adhere to the encodings produced by the `imdb_by_us_gross` query. If users intend to refine their chart (rather than ask a new question), they can *anchor* on a previous chart to signal their intent and guide Dziban towards a similar design. This can be done by either invoking the `anchor` function on a previous chart or invoking the `anchor_on(...)` function with an arbitrary chart.

```
# anchored recommendation
imdb_by_us_gross.anchor().field(
    'Rotten_Tomatoes_Rating'
)
```



Users may also specify queries using channel functions (e.g., `chart.x(field='IMDB_Rating')`) and a breadth of other data transformation and scale properties as function arguments (e.g., `chart.field('Rotten_Tomatoes_Rating', scale='log')`). Further documentation is provided as supplemental material.

DZIBAN IMPLEMENTATION DETAILS

Draco's strength is its ability to reason about the effectiveness of visualizations. However, it cannot reason about multiple views. GraphScape is a model for reasoning about the perceptual distance between two visualizations. However, in the context of recommendation, optimizing for perceptual distance without taking to account visualization effectiveness will result in many unusable graphics. Dziban overcomes conceptual and technical challenges to combine Draco and GraphScape.

Knowledge Representation

Dziban uses the Draco [14] knowledge base to assess visualization expressiveness and effectiveness. Draco doubles as a recommendation system. Given a partial query, Draco uses the Clingo [7, 8] solver to search a design space of potential visualizations by reasoning through a set of hard and soft constraints,

assigning each visualization a *Draco cost* and picking the best one. This is sufficient to provide cold recommendations.

For the purpose of anchored recommendations, Dziban augments the Draco knowledge base by incorporating chart similarity logic. We accomplish this in two steps.

Multi-View Reasoning

First, we extend Draco to reason about multiple views. In Draco, visualization properties are defined using Answer Set Programming (ASP) syntax:

```
mark(bar).  % bar mark
encoding(e1).  % there is an encoding named e1
field(e1,"IMDB_Rating").  % e1 shows IMDB_Rating
...
```

For Dziban, we introduce an additional predicate, the `view`, and use it to qualify every encoding property:

```
view(view1).
mark(view1,bar).
encoding(view1,e1).
...
view(view2).
mark(view2,tick).
...
```

The `view` predicate allows us to introduce new constraints defined over multiple views. As an example, the statement `same_mark(V1,V2) :- mark(V1,M), mark(V2,M).` infers an atom with `same_mark` when two views have the same mark type. A slight modification to the Draco API gives us an interface for interacting with multiple views in Python.

GraphScape for View Similarity Logic

We translated Kim et al.'s GraphScape [11] model into a set of constraints and weights in ASP, similar to Draco's soft constraints. GraphScape describes asymmetrical edits from one visualization to another, so we introduce a `base` predicate to denote a source, or *anchor*, visualization.

We can then construct a set of constraints to express the chart edits modeled by GraphScape. As an example, a `mark` edit from `bar` to `area` with a weight of 3 looks like this in ASP:

```
edit(area_bar,V1,V2) :~ base(V1), mark(V1,area),
                        mark(V2,bar). [3]
```

We then add an optimization function to Draco that gives it the option to minimize GraphScape weights. Critically, this GraphScape optimization is able to leverage the Draco knowledge base's hard constraints, and thus the recommended visualizations do not include those that Draco deems inexpressive. The *GraphScape cost* of a chart is the sum of all edit weights between itself and an anchor.

Recommendation Execution

Rendering a `Chart` initiates compilation and execution of a recommendation query.

Compiling a Draco Specification

Dziban constructs a partial query from the `Chart`'s properties (expressing user-specified constraints) translated into ASP [4]. When a user invokes a `Chart`'s `field` function, a new `Chart` is constructed with an additional `Encoding` object in its internal representation. These objects hold (1) the field of interest, and (2) any properties specified by the user (e.g., `aggregate`, `bin`). Similarly, when the `mark` function is invoked, the resulting `Chart` holds a `mark` property.

For example, the Dziban `Chart`

```
Chart(
    movies
).mark('bar').field('Major_Genre', channel='y')
```

is translated into the following ASP statement (a chart's default view name is `v`, this is changed for anchors):

```
1 view(v).
2 encoding(v,e0).
3 mark(v,bar).
4 :- not field(v,E,"Major_Genre") : encoding(v,E).
5 :- field(v,E,"Major_Genre"), not channel(v,E,y).
```

In plain English, this states that there is a chart (named `v`) that has an encoding and uses a bar mark type (lines 1–3). Additionally, for some encoding, that encoding must express the `Major_Genre` field (line 4). Finally, the encoding with `Major_Genre` must be placed on the `y` channel (line 5).

Retrieving Cold Recommendations

For cold recommendations, the ASP query statement is combined with the data declaration and sent to Draco, to be optimized under Draco's standard optimization function. In return, we receive the "optimal" (least cost) completed specification. For the example above, the completed specification describes a horizontal bar chart measuring counts of records:

```
view(v).
encoding(v,e0).
mark(v,bar).
encoding(v,1).
channel(v,1,x).
channel(v,e0,y).
field(v,e0,"Major_Genre").
aggregate(v,1,count).
type(v,e0,nominal).
zero(v,1).
type(v,1,quantitative).
```

We then translate this logical representation into a Vega-Lite specification using Draco's existing conversion API.

Retrieving Anchored Recommendations

With anchored recommendations, an "anchor" `Chart` will be provided by the user. We assume here the anchor is not an anchored query itself. (If it is anchored, the process we describe is repeated recursively from the bottom up.) First, we retrieve its recommendation—in this case, a cold recommendation. To differentiate its specification, we change its view name to "`anchor`". Additionally, we mark it as a `base` specification, such that it is not modified by Draco when solving our subsequent query. This becomes our *anchored specification*.

Next, we combine this anchored specification with specification of the current chart to create an *anchored query*. In contrast to a cold recommendation, where a single visualization is retrieved from a Draco optimization function, an anchored query is passed to two top-k functions – (1) optimizing Draco costs, and (2) optimizing GraphScape costs – where k is an adjustable parameter discussed later in this section. Intuitively, (1) provides us the k "best" visualizations, according

to Draco, while (2) provides the k "most similar" according to GraphScape.

We compute the intersection between the Draco and GraphScape results. This list represents all visualizations that are both (to some degree) effective *and* similar to the anchor. If the intersection is empty, we use the Draco list, preferring effective charts over those that are only perceptually similar.

Next, we obtain GraphScape and Draco scores for each visualization in the intersection list. For Draco, these often fall between 0 and 100; for GraphScape, between 0 and upwards of 1000. These numeric scores vary according to the complexity of the query and its relation to the "anchor." Note that Draco and GraphScape scores were tuned independently to support ordinal comparisons. As a result, a linear combination of the two would provide inconsistent trade-offs. This is why we must combine two optimization queries. To reconcile the two measures, we normalize the scores to $[0,1]$ within their respective top-k groups: $(val - min)/(max - min)$. We select a "best" visualization by choosing the one with the lowest sum of normalized Draco and GraphScape scores (indicating the best tradeoff), breaking any ties by picking the more effective visualization (Draco score).

In practice, we observe that k=200 provides reasonable computation time (no more than a couple seconds, including rendering) without a noticeable change in recommendation quality relative to higher values. In development, we found that k values that were *too* high (e.g., 400) resulted in ineffective recommendations. We hypothesize that k=200 provides a near exhaustive search of the most reasonably effective and similar visualizations in Draco's design space, while a value such as k=400 allowed for less effective visualizations to be considered and picked in favor of a lower GraphScape cost.

ANCHORED RECOMMENDATION USE CASES

We now demonstrate the advantages of anchored recommendations in specific scenarios, increasing user agency overall. Our goal is *not* to show that anchored recommendations are *always* superior. Rather, we argue that they provide users an option for greater control in well-defined circumstances. In general, we argue that going *down* an exploratory branch [2] (e.g., iterating on an existing question) is a good use case for anchored recommendations, while cold recommendations may be more suited for lateral exploration moves (e.g., asking new questions). In this section we also further exhibit Dziban's language design and functionality.

Drilling Down

Imagine we are analyzing a dataset of movies and are curious about the distribution of Genre (Figure 3). We start with a univariate summary (a) and want to refine the resulting bar chart with a *further* breakdown by MPAA Rating. That is, our question goes from "What is the distribution of Genre?" to "What is the distribution of Genre *and* their MPAA Ratings?". In this case, a cold recommendation (b) changes the encoding of every field, redefining the focus of the chart. This change results in a graphic that both fails to answer our question and requires significant cognitive effort to make sense of. The anchored recommendation (c), meanwhile, answers our question

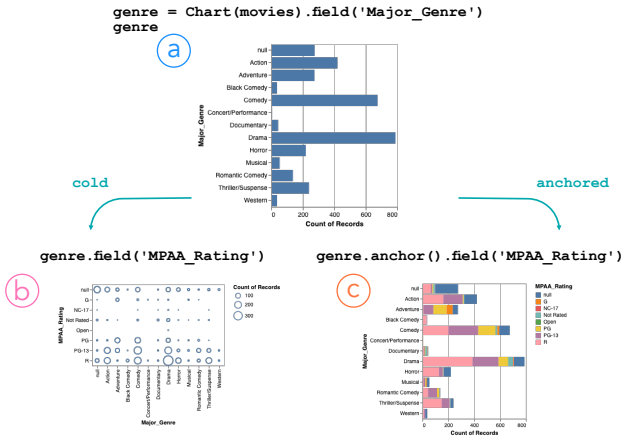


Figure 3. Anchored recommendation (c) from a prior (a) is better suited than cold recommendation (b) when elaborating an analysis question. This may be useful when adding supplementary information to an existing visualization. In this example, adding MPAA Rating as supplementary to the distribution of movies by Genre is better achieved by using an anchored recommendation, as the cold recommendation completely changes the relationship in focus by swapping channel assignments.

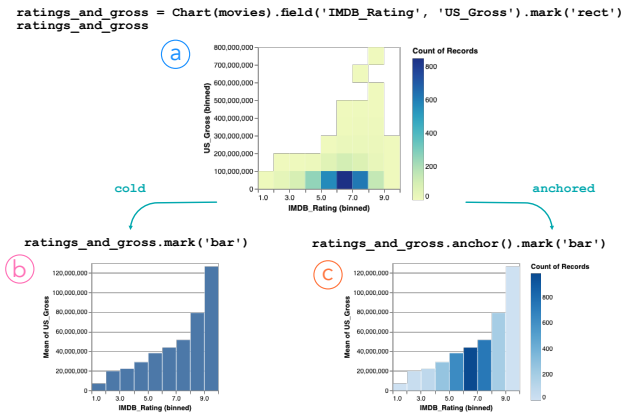


Figure 4. Anchored recommendation (c) from a prior (a) is better suited than cold recommendation (b) when pivoting to a modified hypothesis. This may be beneficial when an original property (in this case, count) is useful for maintaining context in analyzing a relationship between variables (here, understanding the density of movies per rating group).

effectively by *supplementing* the existing visualization and preserving original channel assignments.

Pivoting a Hypothesis

Suppose we are exploring this same dataset of movies, as in Figure 4. We wonder "How does IMDB Rating correlate with US Gross?" and review a heatmap (a). We realize that perhaps the relationship is beyond correlation; perhaps a movie's rating actively affects the number of movie-goers attending its showings. To illustrate this, we want to turn our heatmap into a bar chart, so we specify a `bar` mark. In this scenario, the cold recommendation (b) loses the count encoding, emphasizing the average gross of highly (9+) rated movies. A viewer may be confused initially, as the prior chart shows a lack of high grossing, 9+ rated movies. The anchored recommendation (c), however, retains the count aggregation from the anchor, making mapping between the two easy and reassuring the viewer that a few outliers must be the explanation.

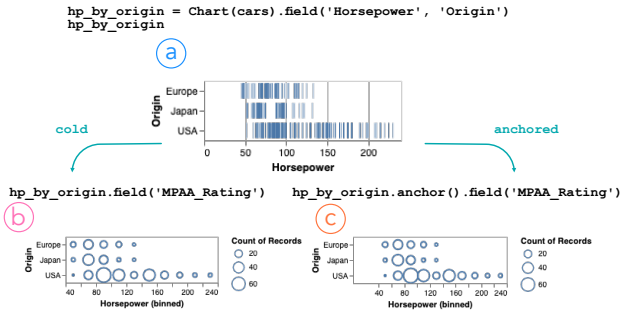


Figure 5. Anchored recommendation (c) can preserve the scale of the prior (a) when cold recommendation (b) does not. This may be useful in scenarios where comparison to an original chart (here, using the original as an unaggregated reference for sample awareness) is desired.

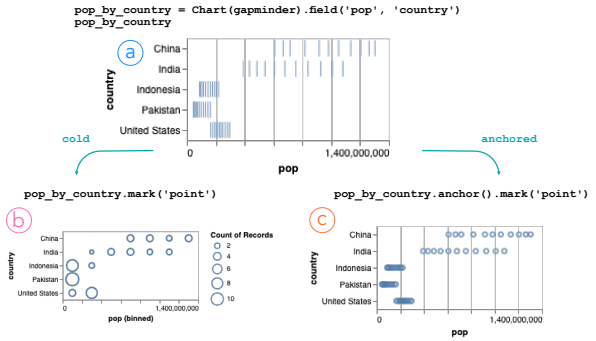


Figure 6. Anchored recommendation (c) can be used to fine-tune a prior (a) whereas cold recommendation may modify extraneous properties. If a user wants to adjust only the mark type (for visibility), anchored recommendations may be more effective. Here, changing the mark of the prior via cold recommendation results in unwanted binning.

This case also demonstrates the flexibility of anchored recommendations. Here, the prior recommendation (a) bins both axes. Changing the mark to a bar without modifying these properties would have resulted in an inexpressive visualization, leaving Dziban's user confused and frustrated. However, Dziban's anchored recommendations can override previous design decisions when necessary, so long as they have not been explicitly set by the user.

Edits for Effectiveness

Now suppose we are analyzing a dataset of cars from the 1970s and are curious as to the distribution of Horsepower across different manufacturing regions (Figure 5). We start with a bivariate query (a) but want to refine the resulting tick plot because of overplotting. To do so, we request that Horsepower be binned. In this case, the cold recommendation (b) loses its zero baseline. A user may be confused upon first glance. It's unclear, initially, if or how the distribution has shifted. The anchored recommendation (c) preserves the zero baseline of the original chart. The prior and anchored charts are immediately comparable: the prior provides context and can be used to spot gaps not visible in the binned plot.

This example shows how anchored recommendation can safeguard user agency. Dziban preserves a property the user *may not have known was present*. Upon noticing it, they, rather than the recommender, can decide whether to keep it.

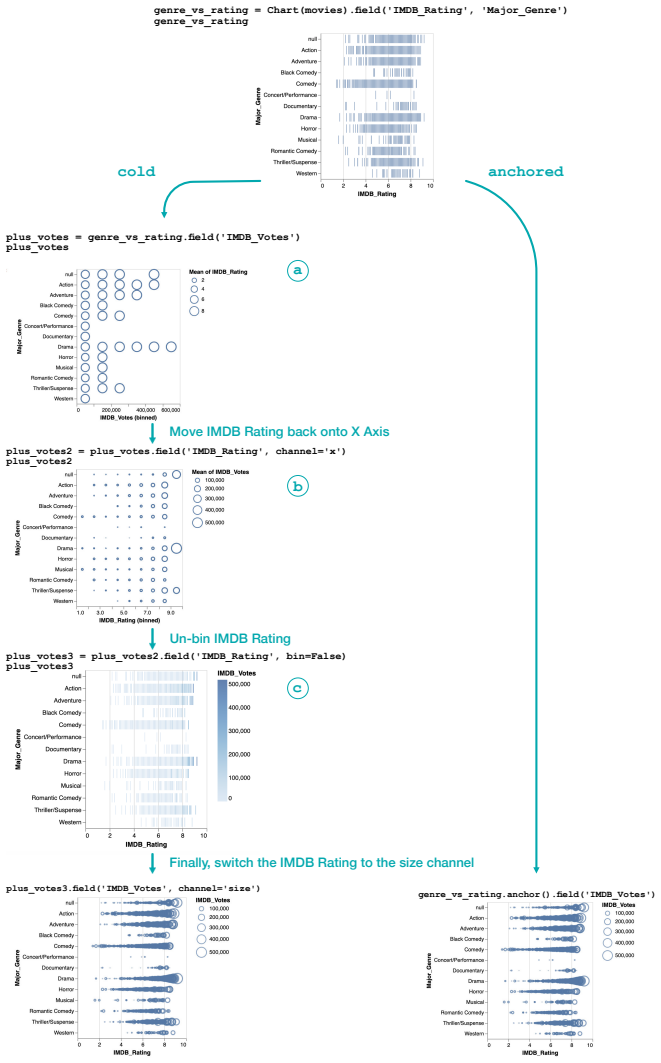


Figure 7. Trying to coerce cold recommendations towards a specific goal can result in a frustrating experience in which effectiveness is optimized at the expense of consistency. In this example, an anchored recommendation maintains the unaggregated view of movies plotted by their IMDB Rating and Genre. On the other hand, a cold recommendation initially swaps channel assignments (removing the original visualized relationship) and aggregates fields (removing sample awareness). Cold queries that attempt to correct these changes can result in further deviations (in this case, changing of mark and channels) that require further adjustments. Anchored recommendations ease this process by reducing the number of changes made to the prior.

Fine-Tuning

In cases where a small modification to a chart is desired, a cold recommendation may automate *too much*. Figure 6 shows a tick plot displaying multi-year population records for the five largest countries in the world. We notice that the ticks are a bit difficult to see. We would prefer larger point marks to emphasize the difference in growth between China and India and the others. An anchored recommendation handles this smoothly, modifying only the mark type. A cold recommendation (b), however, adds binning and a count aggregate to avoid overplotting. Anchored recommendations integrate minute adjustment with automated design.

Towards a specific visualization

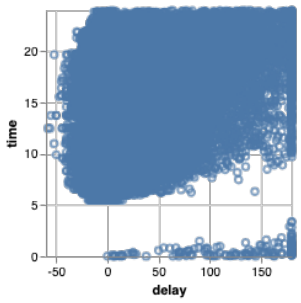
In some cases of exploratory analysis, authors may already have a visualization in mind when interacting with data. In such cases, automated recommendation systems can assist in reducing tedium and removing the syntactical burden of authoring. However, stateless recommendations (e.g., cold recommendations from Draco) may involve a frustrating series of modifications to achieve a particular result. This is especially evident when the intended visualization is similar to the existing one, but contains a few design decisions that might be considered less optimal.

In Figure 7, attempting to move towards the bubble chart (bottom) with cold recommendations results in Draco ignoring consistency (swapping channel assignments [a]) and optimizing effectiveness (binning [a, b] and switching to ticks [c] to avoid occlusion). The examples posed by Figure 3 and Figure 6 exhibit similar issues when attempting to course-correct cold recommendations. Anchored recommendations, by incorporating similarity, allow for more controlled authoring.

To Be Extra Sure...

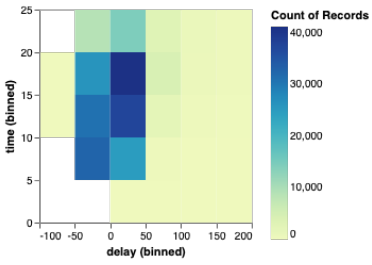
Assume we are analyzing a dataset of flights, and are interested in the relationship between flight length and delay. In Dziban, we can start with a query over two fields:

```
time_by_delay = Chart(flights).field('time', 'delay')
time_by_delay
```



We immediately see that the scatter plot is too dense; perhaps a heatmap would be better suited for this dataset [17]. As we know we are refining a query, rather than asking a new question, we specify an anchor:

```
tbd_rect = time_by_delay.anchor().mark('rect')
# or time_by_delay.mark('rect')
```



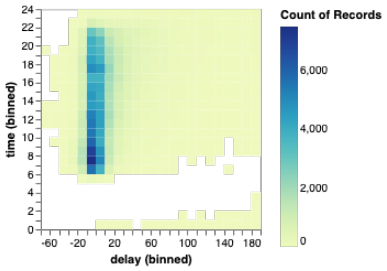
As it turns out, anchoring was not necessary, as a cold recommendation results in the same visualization. However, we note that anchoring does no harm here. A common pattern with Dziban can be to anchor when in doubt, switching back to cold recommendations if it becomes clear that the intended visualization goal need not be constrained by prior context.

Comparing Dziban to other visualization authoring tools, we note that a similar visualization in Altair would require manual specification of binning directives for both fields and the addition of the count aggregate to the color channel:

```
alt.Chart(flights).mark_rect().encode(
    x=alt.X('time', bin=True),
    y=alt.Y('delay', bin=True),
    color='count()'
)
```

Dziban performs these automatically. Moreover, by iterating upon, but not replacing, the original scatter plot, we can use it for context to identify patterns where the heatmap hides them (such as the rift between 0 and 5 hours). As in Altair, we can manually tweak minor parameters to correct this issue.

```
tbd.field('time', 'delay', maxbins=25)
```



Here, we take advantage of `field`'s multi-argument functionality to modify `maxbins` for both "time" and "delay" at once.

BENCHMARKING ANCHORED RECOMMENDATIONS

We now present a quantitative assessment of anchored recommendations in Dziban. We wish to better understand the impact of anchored recommendations relative to cold recommendations. By Dziban's design, more similar visualizations come at the cost of lower effectiveness, but to what extent? To answer this question, we measure anchored recommendations' effectiveness and perceptual similarity scores relative to a baseline of cold recommendations provided by Draco.

Benchmark Design

The goal of this benchmark is to compare anchored recommendations and cold recommendations along two axes: similarity and effectiveness. To determine whether anchored recommendations provide a favorable or unfavorable tradeoff, we require a common metric. Score cannot be used, as Draco and GraphScape weights were tuned independently and are thus incomparable. Nor should we use a method similar to the normalization Dziban uses to reconcile Draco and GraphScape weights: normalization occurs within each query and thus values are incomparable across multiple queries as score distributions differ. Instead, we use the *rank* of a recommendation along each axis. With both systems being ranking recommenders, this provides a meaningful comparison between Draco and GraphScape and across multiple queries.

To cover a relatively comprehensive space, we programmatically generate a set of "priors" and "edits." A "prior" represents a recommendation query that results in a chart. An "edit" represents a modification to that query. We generate priors to cover a combinatorial space of data fields, types, and

Property	Values
field	*quantitative (q), nominal (n)*
transform	*raw, mean, bin*

Table 1. Properties & possible values for "prior" chart generation.

Edit	Values
add field	*q, n, bin(q), mean(q)*
change mark	*point, bar, line, area, tick, rect*
transform	*mean, bin*

Table 2. Edits & possible values for query modification.

transformations, described in Table 1. We run our benchmark over three datasets: IMDB movies[1] (3,201 rows), Cars[2] (406 rows), and a subset of the Chicago Crimes[3] dataset (100,000 rows) for their data type coverage and real-world relevance. We generate edits to cover a reasonable space of query and visualization refinements. These include changing marks and adding fields or data transformations, as shown in Table 2.

For each prior, we apply every edit in two ways. First, we apply the edit and request a cold recommendation; next, an anchored recommendation with the "prior" as the anchor. We introduce a few constraints to prevent illegal edits, such as aggregating an already binned field.

Next, we obtain the relative Draco and GraphScape ranks of each recommendation. The Draco rank of the cold recommendation is 0, as Draco is the only optimizing function. The Draco rank of the anchored recommendation is obtained by solving for the top N recommendations of the corresponding cold query and searching those recommendations for a match. We obtain the GraphScape rank for both the cold and anchored recommendations by anchoring on the prior and solving for the top N recommendations under the GraphScape optimization function, searching for a match. For this experiment, we use $k = 200$, the same value seen in regular usage of Dziban, to obtain recommendations. We use $N = 1{,}000$ to search for matches. With $N \gg k$, we can obtain the rank of visualizations that would otherwise fall outside of the k threshold. For example, cold recommendations may routinely fall outside of the top 200 in similarity (GraphScape rank) to the prior.

Benchmark Results

We report benchmark results in Table 3. Of 447 valid prior and edit pairs across the three datasets (149 for each), 230 resulted in identical recommendations between cold and anchored queries, and 217 resulted in differing recommendations. Of these differing pairs, 127 had cold recommendations with a GraphScape rank outside the top 1,000. As designed, we find that none had anchored recommendations with Draco rank outside the top 200 (as discussed earlier, we defer to the top k Draco charts if we cannot find effective GraphScape charts). The mean GraphScape rank of anchored recommendations is

[1] https://vega.github.io/vega-datasets/data/movies.json
[2] https://vega.github.io/vega-datasets/data/cars.json
[3] https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-present/ijzp-q8t2

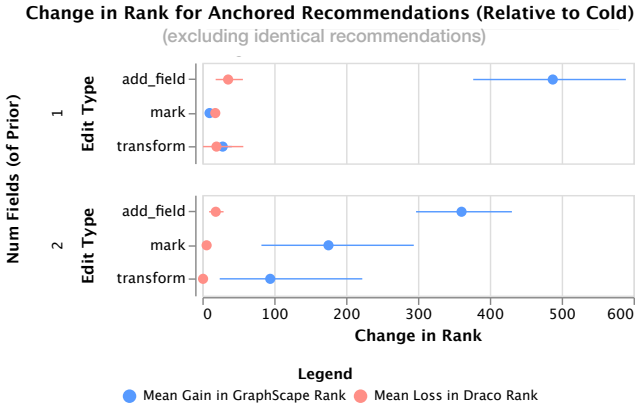


Figure 8. Gain in GraphScape rank (blue) versus loss in Draco rank (red) for anchored recommendations from cold recommendations, along with a 95% confidence interval. This figure excludes identical recommendations for cold and anchored variants and queries that resulted in an undefined (outside top 1000) Cold GraphScape rank. All three datasets are represented.

significantly lower than the GraphScape rank of cold recommendations (aggregate mean of 14.08 vs. 106.07—an order of magnitude difference), while the Draco ranks are within a few values (aggregate mean of 5.42 vs. 0). The numbers vary slightly by dataset—this can be attributed to both variations in characteristics of their data and fields—but a trend is consistent across all: we observe a large gain in GraphScape rank at the expense of a relatively small decrease in Draco rank. We see an exception in one-dimensional mark modifications, where the mean gain in GraphScape rank is lower than its loss in Draco rank, but the difference is less than one rank index (average GraphScape gain of 0.48 vs. average Draco loss of 0.88). In all other situations, anchored recommendations provide a favorable tradeoff between GraphScape and Draco rank, particularly when adding a field (likely due to preservation of channel assignments, which otherwise incurs a high cost in GraphScape).

Figure 8 visualizes anchored recommendations' loss in Draco rank compared to their gain in GraphScape rank for all 217 differing recommendations that possess ranks *within* the top 1,000. A big factor of the large variance in GraphScape rank gain is the distribution of cold recommendation GraphScape ranks, which are dependent on the prior and edit performed. For example, some may incur high cost channel reassignments, while others may only incur lower transform costs.

DISCUSSION

We now reflect on Dziban and discuss areas for future work.

Takeaways from Use Cases and Benchmark

Anchored recommendations enable greater control in chart authoring by suggesting charts that are effective, but also similar to a previously constructed chart. Our design ensures that anchored recommendations never fall out of some top k (200, for this benchmark) Draco visualizations. Our benchmark study shows that, in addition, anchored recommendations provide a considerable benefit in visualization similarity, as ranked by GraphScape, across a variety of priors and edits. As Figure 8 shows, this benefit comes at a comparatively minimal

Fields of Prior	Edit	Count	# Cold w/ GraphScape Rank Outside Top 1,000			Draco Rank				GraphScape Rank					
						Cold	Avg. Anchored			Avg. Cold			Avg. Anchored		
1	all	40	7	12	12	0*	13.2	11.4	4.6	151.1	137.8	140.1	20.6	9.2	35.7
	mark	20	6	7	6	0*	1.3	0.0	1.3	.9	.2	.9	.1	.2	.1
	add field	16	1	5	6	0*	27.9	28.9	5.3	328.7	346.7	386.7	43.7	21.3	99.5
	transform	4	0	0	0	0*	.3	.3	14.3	10.8	10.8	10.8	5.3	5.3	.5
2	all	109	27	29	40	0*	2.8	3.5	4.9	85.9	87.3	103.6	10.1	9.9	13.8
	mark	56	12	13	18	0*	1.5	1.1	.4	30.8	28.9	35.1	.5	.7	.4
	add field	40	15	15	20	0*	6.4	9.1	16.1	221.1	194.8	282.7	30.2	28.2	44.3
	transform	13	0	1	2	0*	0.4	0.3	.3	12.5	72.8	14.3	4.1	4.3	4.5

Table 3. Results of anchored recommendation benchmark for the movies, cars, and crimes datasets. Calculations include queries that resulted in identical recommendations between cold and anchored variants. When calculating means, queries that resulted in an undefined cold GraphScape rank (i.e., outside the top 1,000) are ignored. * A cold recommendation is always the same as the Draco recommendation.

reduction in visualization effectiveness, as ranked by Draco. The effectiveness of the anchored recommendations in the use cases presented provide credence to these conclusions: the decrease in Draco rank does not result in poor visualizations.

The use cases we described also demonstrate that this favorable tradeoff can improve a user's exploratory analysis process. In particular, Figures 3, 6, and 7 show that similarity, with only a slight loss of assumed effectiveness, can provide reasonable visualizations when the instability of cold recommendation sequences might otherwise throw an exploration awry.

Future work should further validate this approach through human-subject evaluations. By comparing Dziban to existing recommendation and full-specification authoring tools, such studies could demonstrate the usability of Dziban as an authoring interface and its efficacy in practice.

Improving Dziban

Dziban's API does not currently support the entire Vega-Lite [18] design space. This can be improved by further implementation, but a key hurdle is expanding the expressiveness of Dziban's supporting recommendation model, the Draco [14] knowledge base. Its lack of support for multi-view and layered composition is one large hole that GraphScape [11] also shares. Moreover, Dziban's recommendations are far from perfect, primarily stemming from flaws in the Draco knowledge base. The "top" Draco recommendation is not always, in our opinion, the most effective one, and a set of visualizations may have differing scores when their effectiveness remains nearly identical. We hope to improve both Draco's constraint system and the GraphScape model.

A large challenge in developing Dziban was implementing a multi-objective optimization function used to reconcile the Draco and GraphScape models. We attempted a variety of linear weighting and normalization routines before we landed at our current implementation. We believe our approach strikes a nice balance between effectiveness and consistency, but future work could explore the tuning of this function to accommodate different use cases. For example, one could lean more heavily towards GraphScape weights if minute visualization refinement, rather than query refinement, is the objective. Better yet, we could provide users the ability to control the function as they please, or adapt it to their usage patterns.

Applications of Dziban and Anchored Recommendations

We also hope to explore alternative designs for Dziban's output interface. Currently, Dziban renders a single recommendation as output for a Jupyter notebook cell. Rendering multiple visualizations—either with variation in design, or variation in data (as with Voyager [27, 28])—could facilitate exploration and increase user agency. This could take the form of an interactive carousel, tabs of visualizations, or a simple stepper that allows users to page through recommended visualizations. We are particularly excited about the potential for these interfaces to improve Dziban's recommendation model. By learning from user input (if permission is granted), Dziban might adapt to individual domains, tasks, and design preferences.

We are also eager to explore Dziban's use outside of programming environments. For example, Dziban could be used in place of Voyager's [28] "locking" functionality to provide more flexibility in exploration. View similarity and consistency have roles outside exploratory analysis as well. Dziban's multi-objective optimization can be an asset in tools used for narrative authoring (e.g., DIVE [10]), where similar views are often placed in sequence to tell a story. Where GraphScape could only recommend sequence design or chart modifications, its inclusion in Dziban can provide tools the opportunity to suggest a breadth of narratively compatible visualizations.

CONCLUSION

We presented Dziban, a visualization API that supports both ambiguous partial specification visualizations and a novel *anchoring* mechanism for conveying desired context. Dziban takes advantage of automated visualization design to provide a concise specification interface, and encourages small modifications to queries to facilitate common patterns in exploratory visual analysis. These attributes balance Dziban's automated design system and provide users with increased agency as they create and refine visualizations.

ACHNOWLEDGEMENTS

We would like to thank Rastislav Bodik and Kanit "Ham" Wongsuphasawat for their helpful guidance. We also thank the UW Interactive Data Lab, colleagues at the University of Washington, and anonymous reviewers for their feedback. This work was supported by NSF award IIS-1907399.

REFERENCES

[1] Michelle Q. Wang Baldonado, Allison Woodruff, and Allan Kuchinsky. 2000. Guidelines for Using Multiple Views in Information Visualization. In *Proceedings of the working conference on Advanced visual interfaces, AVI 2000, Palermo, Italy, May 23-26, 2000*. 110–119. DOI:http://dx.doi.org/10.1145/345513.345271

[2] Leilani Battle and Jeffrey Heer. 2019. Characterizing Exploratory Visual Analysis: A Literature Review and Evaluation of Analytic Provenance in Tableau. *Computer Graphics Forum (Proc. EuroVis)* (2019). DOI: http://dx.doi.org/10.1111/cgf.13678

[3] Jacques Bertin. 1983. *The Semiology of Graphics*.

[4] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. 2011. Answer set programming at a glance. *Commun. ACM* 54, 12 (2011), 92–103. DOI: http://dx.doi.org/10.1145/2043174.2043195

[5] William S Cleveland and Robert McGill. 1984. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American statistical association* 79, 387 (1984), 531–554.

[6] Camilla Forsell and Jimmy Johansson. 2010. An heuristic set for evaluation in information visualization. In *Proceedings of the International Conference on Advanced Visual Interfaces, AVI 2010, Roma, Italy, May 26-28, 2010*. 199–206. DOI: http://dx.doi.org/10.1145/1842993.1843029

[7] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2014. Clingo = ASP + Control: Preliminary Report. *ArXiv* abs/1405.3694 (2014).

[8] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. 2011. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.* 24, 2 (April 2011), 107–124. http://dl.acm.org/citation.cfm?id=1971622.1971623

[9] Jeffrey Heer. 2019. Agency plus automation: Designing artificial intelligence into interactive systems. *Proceedings of the National Academy of Sciences* 116, 6 (Feb. 2019), 1844–1850. DOI: http://dx.doi.org/10.1073/pnas.1807184115

[10] Kevin Zeng Hu, Diana Orghian, and César A. Hidalgo. 2018. DIVE: A Mixed-Initiative System Supporting Integrated Data Exploration Workflows. In *HILDA@SIGMOD*.

[11] Younghoon Kim, Kanit Wongsuphasawat, Jessica Hullman, and Jeffrey Heer. 2017. GraphScape: A Model for Automated Reasoning About Visualization Similarity and Sequencing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 2628–2638. DOI:http://dx.doi.org/10.1145/3025453.3025866

[12] Jock Mackinlay. 1986. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics* 5, 2 (1986), 110–141. DOI: http://dx.doi.org/10.1145/22949.22950

[13] Jock D. Mackinlay, Pat Hanrahan, and Chris Stolte. 2007. Show Me: Automatic Presentation for Visual Analysis. *IEEE Trans. Vis. Comput. Graph.* 13, 6 (2007), 1137–1144. DOI: http://dx.doi.org/10.1109/TVCG.2007.70594

[14] Dominik Moritz, Chenglong Wang, Gregory Nelson, Halden Lin, Adam M. Smith, Bill Howe, and Jeffrey Heer. 2019. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2019). DOI: http://dx.doi.org/10.1109/TVCG.2018.2865240

[15] Zening Qu and Jessica Hullman. 2016. Evaluating Visualization Sets: Trade-offs Between Local Effectiveness and Global Consistency. In *Proceedings of the Sixth Workshop on Beyond Time and Errors on Novel Evaluation Methods for Visualization, BELIV 2016, Baltimore, MD, USA, October 24, 2016*. 44–52. DOI: http://dx.doi.org/10.1145/2993901.2993910

[16] Zening Qu and Jessica Hullman. 2018. Keeping Multiple Views Consistent: Constraints, Validations, and Exceptions in Visualization Authoring. *IEEE Trans. Vis. Comput. Graph.* 24, 1 (2018), 468–477. DOI: http://dx.doi.org/10.1109/TVCG.2017.2744198

[17] A. Sarikaya, M. Gleicher, and D. A. Szafir. 2018. Design Factors for Summary Visualization in Visual Analytics. *Computer Graphics Forum* 37, 3 (June 2018), 145–156. DOI:http://dx.doi.org/10.1111/cgf.13408

[18] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Trans. Vis. Comput. Graph.* 23, 1 (2017), 341–350. DOI: http://dx.doi.org/10.1109/TVCG.2016.2599030

[19] Tarique Siddiqui, Albert Kim, John Lee, Karrie Karahalios, and Aditya Parameswaran. 2016. Effortless Data Exploration with Zenvisage: An Expressive and Interactive Visual Analytics System. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 457–468. DOI: http://dx.doi.org/10.14778/3025111.3025126

[20] Melanie Tory and Vidya Setlur. 2019. Do What I Mean , Not What I Say ! Design Considerations for Supporting Intent and Context in Analytical Conversation. *IEEE Trans. Visualization & Comp. Graphics (Proc. VAST)* (2019). https://research.tableau.com/paper/intent-VAST

[21] Jacob VanderPlas, Brian Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. 2018. Altair: Interactive Statistical Visualizations for Python. *Journal of Open Source Software* (dec 2018). DOI:http://dx.doi.org/10.21105/joss.01057

[22] Manasi Vartak, Samuel Madden, Aditya G. Parameswaran, and Neoklis Polyzotis. 2014. SEEDB:

Automatically Generating Query Visualizations. *PVLDB* 7 (2014), 1581–1584. DOI: http://dx.doi.org/10.14778/2733004.2733035

[23] Vega-Lite API 2019. (2019). https://github.com/vega/vega-lite-api Accessed: 2019-09-19.

[24] Hadley Wickham. 2016. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. https://ggplot2.tidyverse.org

[25] Leland Wilkinson. 2005. *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Berlin, Heidelberg.

[26] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2016a. Towards a general-purpose query language for visualization recommendation. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics - HILDA '16*. 1–6. DOI: http://dx.doi.org/10.1145/2939502.2939506

[27] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2016b. Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations. *IEEE Transactions on Visualization and Computer Graphics* 22, 1 (Jan. 2016), 649–658. DOI: http://dx.doi.org/10.1109/tvcg.2015.2467191

[28] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. 2017. Voyager 2. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems - CHI '17*. ACM Press. DOI:http://dx.doi.org/10.1145/3025453.3025768