Interactive Systems for Scalable Visualization and Analysis

Dominik Moritz

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Jeffrey Heer, Chair

Bill Howe, Chair

Danyel Fisher

Program Authorized to Offer Degree:

Computer Science & Engineering

University of Washington

Abstract

Interactive Systems for Scalable Visualization and Analysis

Dominik Moritz

Co-Chairs of the Supervisory Committee:

Jeffrey Heer

Computer Science & Engineering, University of Washington

Bill Howe

School of Information, University of Washington

While computers can help us manage data, human judgment and domain expertise is what turns it into understanding. Meeting the challenges of increasingly large and complex data requires methods that richly integrate the capabilities of both people and machines. In response to these challenges, this thesis contributes new languages and models for visualization design that power interactive systems for scalable data analysis.

In these languages, users can be imprecise about low-level design decisions as the system leverages this ambiguity to optimize the visual design and necessary computation. **Vega-Lite** is a high-level declarative language for rapidly creating interactive visualizations, while also providing a convenient yet powerful representation for tools that generate visualizations. Vega-Lite uses smart defaults to fill in low-level details to create effective designs. The declarative design facilitates optimization of the required data processing. **Draco** is a model of visualization design that extends Vega-

Lite with shareable design guidelines, formal reasoning over the design space, and visualization recommendation. We show how we can use Draco to construct increasingly sophisticated automated visualization design and recommendation systems, including systems based on weights learned directly from the results of graphical perception experiments.

We take a user-centric perspective on systems for scalable exploratory analysis. Considering both the backend and frontend concerns, we present **Falcon**, an interactive crossfilter application where users can interact with billions of records without latencies that negatively affect their exploration. To scale beyond billions of records, we present **Pangloss**, a visual analysis system that uses approximate query processing but provides eventual guarantees using **Optimistic Visualization**. In this concept, we treat approximate query processing as a user experience problem to address users' primary concern: trust in their exploration results. Falcon and Pangloss contribute techniques for scalable interaction and exploration of large data volumes by making principled trade-offs among people's latency tolerance, precomputation, and the level of approximation.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Getting into and actually finishing a PhD in computer science requires amazing people to support you and a lot of happy little accidents. I would like to acknowledge those who were there along the journey and helped me chart new territory.

I feel extremely privileged to have worked as part of an amazing community of people during the last six years. First and foremost, I am immensely grateful to my two inspiring advisors Bill Howe and Jeffrey Heer. Bill is a tireless champion of looking for the big problems and a true visionary. He kept my fire for databases alive that brought me to UW for my one-year exchange. Jeff is an inspiration, who embodies commitment to high-quality research and high-impact tools. He is both kind and smart and has always led by example. Bill and Jeff are my fearless champions; no page in this dissertation could exist without either of them.

I am grateful to my internship mentors. In particular, Danyel Fisher, my mentor for two internships at Microsoft Research, thesis committee member, fierce advocate for tasteful research, and enemy of bullshit. My internships would not have been half as much fun without Steven Drucker, Bolin Ding, Chi Wang, Arnd Christian König, and the VIBE and DMX groups. I also had the privilege of learning at Google from Sudip Roy, Natasha Noy, Alkis Polyzotis, Xiao Yu, Sitaram Iyer, Chris Olston, Alon Halevy, Shishi Chen, and Ben Atkin. Adrien Treuille, Thiago Teixeira and the rest of the Streamlit team showed me the inner workings of their exciting startup. These experiences honed by software development and research skills and helped me stay motivated in grad school.

I was lucky to have worked closely with brilliant colleagues and mentors. Dan Halperin led me towards becoming a researcher when I started at UW and had no idea what I was doing. I would not have a thesis without Kanit “Ham” Wongsuphasawat, who convinced me to do research in

visualization and co-authored many tools. I am grateful to have worked closely with Shrainik Jain, Arvind Satyanarayan, Shumo Chu, Chenglong Wang, Greg Nelson, Michael Correll, Leilani Battle, and Matthew Conlen. Other collaborators include Daniel Epstein, Zening Qu, Jennifer Rogers, Daniela Huppenkothen, Anushka Anand, and Jock Mackinlay. I am grateful for the support from the Interactive Data Lab, the Database Group, the Human-Computer-Interaction Group, and the Programming Languages Group. Thank you to the whole CSE family. In particular, Lindsay Michimoto who encouraged me to apply to CSE and Elise deGoede Dorough who makes CSE an awesome place for graduate students.

Thank you to the UW undergraduates, masters students, and Google Summer of Code students who withstood my mentorship. They keep the Vega ecosystem alive. The open source community, in particular Jake VanderPlas, continues to amaze me with their engagements. There are too many people to thank them individually. As a non-representative sample, I want to mention Saul Shanabrook, Brian Granger, Brian Hulette, Jim Vallandingham, Yannick Assogba, K. Adam White, Irene Ros, Yuri Astrakhan, and Mike Bostock. Falcon would not exist without the support from the OmniSci team, especially from Venkat Krishnamurthy, Todd Mostak, and Randy Zwitch.

I would also like to acknowledge VoiceOver for its contribution to this acknowledgment section. Without it, I would not have been able to recover this section from a MacBook whose screen was stuck at 0% brightness. I dedicate this paragraph to computers, the cause of—and solution to—all of life's problems.

Finally, I am grateful for my family in Germany and my friends who became my Seattle family on the other side of the planet. Above all, I want to thank Lilian and my parents for their endless support, love, and wisdom.

Thank you.

1 Introduction

Visual representations of abstract data are often critical to making and understanding new insights. They allow viewers to see the magnitude of data, draw comparisons, and reveal patterns or trends. Visualizations can improve communication of results (*explanatory visualization*) and help analysts understand and discover patterns in data (*exploratory visualization*) [208]. This is possible because visual representations can leverage the powerful capabilities of the human visual system.

The introduction of user interfaces and graphics software has enhanced our ability to analyze data interactively and look at it from different angles to reveal patterns [159, 228]. Computers can process data faster, more accurately and more reliably than any human. With vast amounts of data being generated, collected, and stored, data visualization with computers has become an essential tool for data scientists in businesses, government agencies, and science.

However, existing programming tools used to create visualizations are typically designed for manual authoring without computational support for following established practices. This lack of integration leaves good design a responsibility of the human designer. To address this problem, we need to design new tools where people and machines can meaningfully participate in the visualization process.

Computers have unique and powerful capabilities for helping people interact with data. However, they also have limited speed at which they can deliver requested information. The computer hardware we use improves slower than the amount of data increases that analysts want to store

and visualize [215]. One example is sky surveys in modern astronomy. Since 2000, the Sloan Digital Sky Survey (SDSS) [1] has collected information on hundreds of millions of stars and galaxies. In early 2018, the European Space Agency released a survey of 1.7 billion sources from the GAIA space telescope [22]. The LSST, a ground-telescope which is planned to go online in 2019, will collect 32 trillion observations of 40 billion objects [99]. These orders of magnitude larger datasets challenge the tools data analysts use today—and by extension, the people who build these tools.

Modern data warehouses often include tables with billions or more records. Most visual analysis tools are not designed to work at this scale, let alone support real-time interaction [109]. As the amounts of data that we wish to analyze continues to grow rapidly, our tools rarely keep up with this development. With large data volumes and demanding latency requirements, data processing systems for interactive visualization run against hardware limitations. Meeting the challenges of perceptual and interactive scalability is therefore not only a matter of engineering more powerful systems but requires a deep understanding of the capabilities and limitations of both the people and the machine.

1.1 Thesis Statement

In this thesis, we hypothesize that high-level visualization languages designed for both human authoring and programmatic generation facilitate systematic exploration of the design space, reuse across computing environments, and automatic optimization. Based on these representations, we can create formal models of design to build shared and extensible knowledge bases of design practices. These models facilitate smart design assistants. Combining the strengths of people and machines, and co-designing the data processing systems and their user experience, enables interactive visualizations of billion-record datasets.

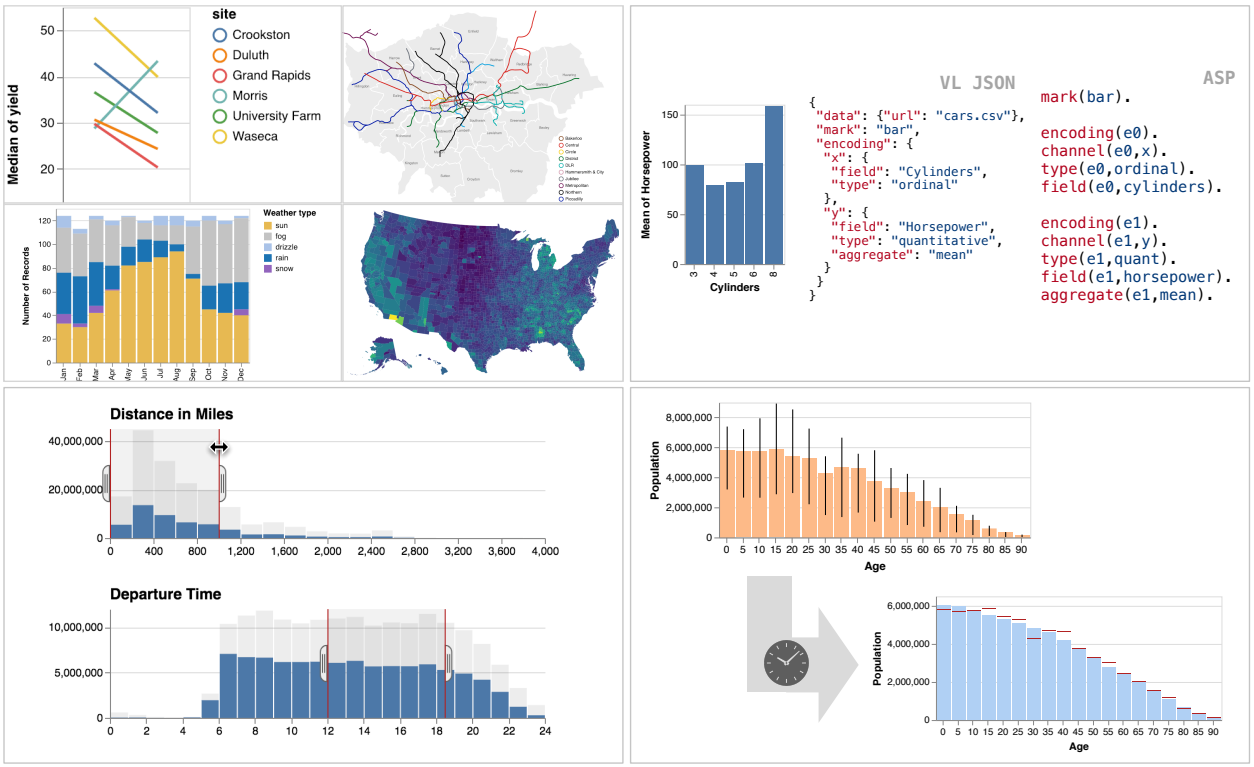

Figure 1.1: Overview of the contributions of this thesis (from top-left to bottom-right): Vega-Lite, Draco, Falcon, and Optimistic Visualization.

1.2 Thesis Contributions

To support the hypothesis, this thesis contributes new languages and models for visualization design and interactive systems for scalable visual analysis. It makes contributions in four categories illustrated in Figure 1.1.

The design of a grammar for interactive multi-view graphics for both people and machines. Visualization grammars are a popular way of specifying charts. They describe a combinatorial space of possible graphics as a composition of a few building blocks. Because they are declarative, designers can focus on design questions and defer execution concerns to a runtime. Grammars can raise the level of abstraction by omitting low-level details that are filled in by smart defaults.

These high-level grammars enable rapid authoring of charts making them especially attractive for data exploration. Most visualizations today are authored through end-user applications but many of the existing languages are designed to be written by people. Moreover, existing grammars provide only limited support for interactivity. To address these limitations, we present **Vega-Lite**, a declarative visualization grammar (Figure 1.1 top-left, chapter 3) and accompanying compiler. Vega-Lite expresses visualizations in a declarative format that can easily be manipulated with computers. These specifications facilitate systematic enumeration of the design space, reuse across different platforms and devices, and automatic optimization of the execution and presentation. Vega-Lite also introduces a view algebra for combining basic plots into multi-view displays, and a new selection abstraction to declaratively specify interaction techniques.

A formal model to specify visualizations and a knowledge base of visualization design.
Vega-Lite is designed for statistical graphics. It enables a rich (yet constrained) design space with a grammar fit for human and machine specification. However, the choice of data transformations and visual encodings require design expertise. Mistakes in these choices can be costly; people may overlook important features in their data or derive false insights. Good designs can prevent these mistakes and make designs more appealing and easier to read. Vega-Lite's defaults can help with good designs. However, Vega-Lite scopes ambiguity to low-level details of scales, axes, legends, and data transformations. This scoping leaves high-level design decisions to the designer. We lack systems that leverage ambiguity about high-level encoding decisions to guide people towards good designs. In response, we present **Draco**, a formal model to specify visualizations and a knowledge base of visualization design as a set of constraints and associated weights (Figure 1.1 top-right, chapter 4). Draco makes design decisions based on the constraints that are specified by experts or learned from data. An end-user evaluates Draco's recommendations and thus incrementally refines their visual encoding. Draco's constraints and associated weights enable formal reasoning and reuse across different tools. They facilitate the development of visualization recommendation systems that assist users in improving their designs.

An interactive system that balances interactive latency and resolution sensitivity for scalable linked visualizations.

Analysts often wish to work with increasingly large datasets. Many of today's visualization systems only work with in-memory data or resort to querying scalable database systems. While the latter support massive datasets, query response times often exceed what is considered interactive. It is crucial to maintain interactivity when exploring data to enable data analysis at "rates resonant with the pace of human thought" [83, 91]. Traditionally, visualization system design has often taken a modular approach where the visualization pipeline is divided into separate and independent components. We discuss the design space and ultimately find a lot of effort has gone into optimizing the data processing systems but there has been little consideration of the corresponding user interfaces. While this separation of concerns has been one of the factors that contributed to the success of (in particular relational) databases, this approach overlooks important problems but also potential for optimization. In this thesis we take a holistic approach to system design and contribute **Falcon** (Figure 1.1 bottom-left, chapter 5). Falcon is a web-based system that supports interactions across linked multi-view visualizations of large datasets. Falcon uses smart prefetching and indexing to support analytics over billions of records without expensive precomputation. In particular, we show specialized algorithms and data structures that support real-time brushing and linking. Falcon builds on the insight from prior work that while delays cause analysts to lose their thought process, some operations are more latency sensitive than others [122]. We leverage this insight and prioritize latency sensitive interactions.

A framework for reliable exploration of approximate results.

As data sizes keep growing rapidly, even scanning a large dataset may take minutes and inhibit interactive exploration. Luckily, analysts can often make the same insights on a sample of the data if the system computes approximations [3, 62]. The error of an approximation depends only on the size of the sample and not the size of the data it was drawn from; additional data has diminishing returns. There are several well-known challenges with approximations. The most critical of these is trust: approximate values can be—by their nature—incorrect. In an

exploratory visualization, an analyst might see dozens of visualizations that are accurate most of the time. With enough visualizations they are almost guaranteed to encounter a visualization where the errors are outside the predicted bounds. In this thesis, we present **Pangloss**, a visual analysis system that uses approximate results but provides eventual guarantees using **optimistic visualization** (Figure 1.1 bottom-right, chapter 6). Optimistic visualization produces approximate results quickly, and computes precise results in the background. The analyst can make observations on the approximation, and later check them against the precise results. It has the benefits of both approximation and computing results offline: analysts can work at interactive speeds and rely on their findings.

1.3 Perceptual and Interactive Scalability

Many datasets today are too large for the tools we use. For example, today's screens have millions of pixels, but datasets have billions or more records; not every record can be rendered. Even if we could show every point, the visual representations overwhelm viewers. This example illustrates the two main challenges that analysts face. First, a visual representation that shows every record as its own mark quickly overwhelms our visual system; we call this problem *perceptual scalability*. The solution is to reduce the data in a way that preserves visual cues to relevant patterns. Second, for very large data the reduction itself can overwhelm the data processing system; we call this the problem of *interactive scalability*.

The systems we propose in this thesis address different issues that a data analysts may encounter when working with increasingly large data. Figure 1.2 gives an overview of how this thesis addresses perceptual and interactive scalability issues. Vega-Lite and Draco are useful even for small data and Vega-Lite has already become a popular tool in the JavaScript and Python data science communities.

Even though we have only implemented prototypes, they can address perceptual and interactive scalability. In subsection 4.6.4, we outline how Draco can be extended to use ambiguities in specifications and recommend scalable alternatives to common chart types. In subsection 3.5.3,

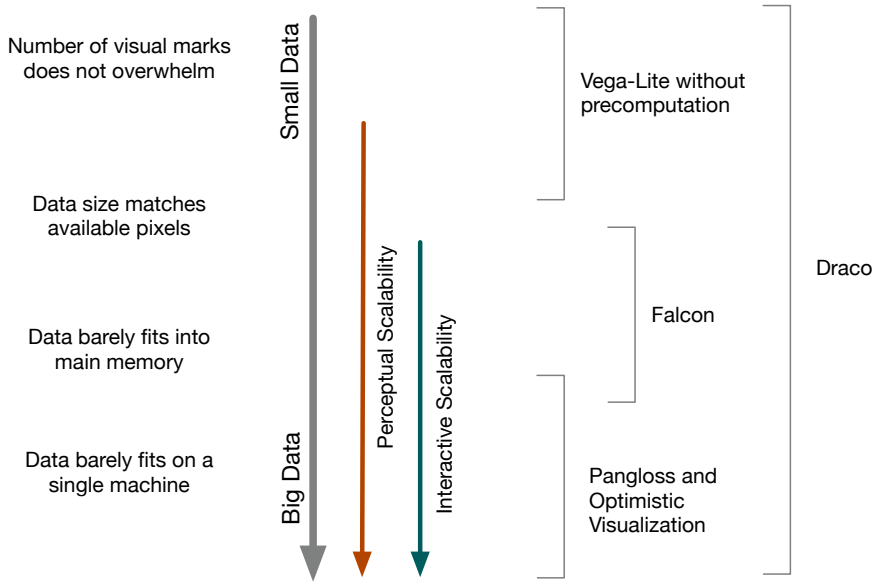

Figure 1.2: Overview at which data sizes Vega-Lite, Draco, Falcon, and Pangloss address the issues of perceptual and interactive scalability. Draco handles small data and lays the groundwork for supporting big data.

we discuss how our compiler already leverages the high-level declarative design of Vega-Lite to automatically optimize the execution and remove redundant or unnecessary computation. In subsection 3.7.2, we discuss a prototype of a runtime system that moves expensive computation to a scalable backend system. However, this prototype sends a query for every interaction incurring latencies incompatible with some interactions [122].

We provide a solution for a common interaction pattern in Falcon. With Falcon running in a modern browser, analysts can brush and link between one-dimensional and two-dimensional histograms of millions or tens of millions of records. Falcon can move heavy computation to a data management system and scale up to billions of records. Eventually, these data management systems may use approximation to save resources or handle even larger datasets. Optimistic visualization then addresses the problem that approximate results are not reliable.

1.4 Prior Publications and Authorship

Although I am the principal author of the research in this dissertation, much of the work was done collaboration with my advisors Jeffrey Heer, Bill Howe, my mentor at Microsoft Research Danyel Fisher, and my colleagues at the Interactive Data Lab and the Database Group at the University of Washington. Vega-Lite was published at IEEE VIS 2016 [177] as joint work with Arvind Satyanarayan who contributed selections as interactive primitives and Kanit "Ham" Wongsuphasawat who I worked with to design and implement visual encodings. Optimistic visualization and Pangloss were published at ACM CHI 2017, and HILDA at ACM SIGMOD 2017 [133, 134] after an internship at Microsoft Research. During the internship, Danyel Fisher and I closely collaborated with Bolin Ding and Chi Wang from the Data Management, Exploration and Mining (DMX) group. Draco was published at IEEE VIS 2018 [137] as the result of a collaboration with Chenglong Wang from the Programming Languages and Software Engineering group at UW. Greg Nelson and I developed an earlier prototype as part of a constraint programming class with Alan Borning. Halden Lin helped develop user interfaces and collect data for the system. Draco is heavily influenced by my earlier work with Kanit Wongsuphasawat including CompassQL [223], and Voyager [224, 225]. In this thesis, I use the first person plural to reflect my collaborators' contributions.

2 Background and Related Work

This thesis builds on prior work in data visualization, human computer interaction, data management, and programming languages. Here, we review relevant work framing the thesis. Related work specific to a particular chapter is introduced before each chapter.

2.1 Information Visualization

In the 18th and 19th century William Playfair [162], Florence Nightingale, John Snow, and Charles Minard pioneered the use of quantitative graphs that can be understood and enable viewers to grasp the relevant information quickly; they often used visualization to make something more accessible to the public. In the second half of the 20th century Jacques Bertin [15], a French cartographer, provided a theoretic foundation of information visualization. He methodically examined graphical representations and developed categorizations of different data types (e.g., numerical, categorical, and temporal) and encodings (e.g., retinal variables such as color, position, size, and shape). Most of Bertin's design principles were based on his own judgments but were later confirmed and extended with rigorous experiments. Cleveland and McGill [41] conducted experiments on the effectiveness of different visual encodings for conveying different data types. From these experiments we can derive that position and length are quite effective encodings of quantitative data while area and color are less effective. These experiments have been replicated [88] and extended [112] since then. Mackinlay formalized these results in APT [127], an automated visualization design tool. Edward Tufte's books on visualization design [206] prescribed many influential design guidelines.

The guidelines include the use of small multiples (series of small charts using the same scales and axes), maximizing the ratio of data-ink to non-data-ink, and eliminating unnecessary chart elements, which he calls chart junk. Analysts who follow the best practices of good visual design create visualization that are easier to understand. The tools they use should guide them towards following these best practices.

2.2 Interactive Data Visualization

Even though Bertin's work focused on maps and charts printed on white paper and seen from a typical reading distance, a lot of the principles he discovered still apply today with computers. Moreover, he used interaction with data—but without graphical user interfaces on computers—as part of the analysis. For example, in his discussion of visual permutation matrices [14], he explains how reorganizing the data and looking at in different configurations can reveal patterns that are otherwise hidden. Interactivity is a crucial component of effective visualization as it supports the construction of knowledge [7, 124, 159]. It is woven into many areas of information visualization today. Card et al. define visualization itself as "interactive, visual representations of abstract data to amplify cognition" [25].

Visualization interfaces typically use dynamic queries [5, 183] that update continuously as the user adjusts sliders or selects items in the UI. These UI elements visually represent components of a query so that users interact with a query through pointing, not typing. Modern user interface applications such as Tableau—which grew out of the Polaris research project [198]—support data analysis through direct manipulations of data and charts. These applications are grounded in work in the HCI and Visualization community. Ben Shneiderman is one of the pioneers of data visualization and proposed a taxonomy of data types to be visualized and tasks that visualizations should support [184]. In the same paper he suggests a simple guideline for interactive visualization interfaces: start with an overview of all data, then zoom and filter to relevant items, and finally

view details of individual items. Card et al. situated Shneiderman's principles within the larger process of sense making [25].

Interactivity is critical for visual data analysis. Therefore, our programming tools should have support for interactivity as a first-class design consideration. With Vega-Lite, we take this consideration to heart and present a grammar of graphics tightly integrated with a novel grammar of interaction.

2.3 Exploratory Data Analysis

Visualization has the goal of deeper understanding of data [25]. In Exploratory Data Analysis (EDA) [208]—as coined by the statistician John Tukey—visualization plays an important role in helping users get familiar with data, clarify goals, and devise a strategy to achieve these goals. Data should be used to suggest hypotheses to test. Tukey's view was that too little emphasis in statistics was placed on parts of the analysis process that are not hypothesis testing. By exposing analysts to their data, they may become aware of unanticipated features of the data [207, 209].

Conflating hypothesis generation and hypothesis testing and running them on the same data leads to bias and spurious insights. During exploration the analyst should clarify the goals of their analysis and check assumptions (e.g., about data quality and distributions [110]) that are used to test hypotheses later.

Exploratory analysis is by its nature open-ended and the exact exploration path is not prescribed. This open-endedness implies that data systems for exploratory analysis need to be flexible enough to support many analysis methods, questions, and answers and different orders in which these analysis stages are executed. Since many questions in this exploration space might be one-off questions that can come up at any stage of the analysis, queries are hard to anticipate. This scenario where a system is queried without prior configuration is called *cold-start* analytics; as opposed to *warm-start* analytics where a system can run costly precomputation of likely queries.

2.4 Visualization Specification

One way to describe a visualization is through a grammar, a set of atoms and rules. A good grammar of visualizations will help us understand how complex charts are composed and show parallels between seemingly unrelated visualizations [42]. A grammar may guide us on what correct visualizations are, but there will still be many grammatically correct but nonsensical visualizations. This is similar to grammar in natural languages such as English. A good grammar is necessary but not sufficient to form a good sentence.

One of the most influential formalisms for specifying statistical graphics is the *Grammar of Graphics* by Leland Wilkinson [222]. The Grammar of Graphics uses combinatorial building blocks to rapidly construct graphics rather than picking from a limited set of charts in a chart typology [91]. The basic building blocks of this grammar are primitives to specify the data and transformations, visual encodings defined as mappings between data fields and channels (e.g., position, size, and color), and scales and guides (i.e., axes and legends). The popular ggplot library [219] implements a variant of this grammar in the R statistical computing language. A language can be useful both for specifying as well as reasoning over visualizations. For example, we developed Vega-Lite originally as a language to automatically recommend visual designs in Voyager [224]. Stolte et al. developed *VizQL* [84] as the formalism underlying Polaris [198].

These languages focus on statistical graphics and favor concise specifications over expressiveness. Low-level details such as the choice of a color palette or the scale type (e.g., linear or logarithmic) may be omitted and filled in by smart defaults. Lower-level libraries such as *Protovis* [17], *D3* [18], and *Vega* [178] enable an expressive space of explanatory, custom designs. Bostock and Heer show that the advantages of declarative design for exploratory graphics also apply to explanatory graphics [89]; separating specification from execution facilitate runtime optimization, retargeting to different platforms, and an iterative development process.

With Vega-Lite, we present a new language that builds on this prior work. In contrast to prior work, Vega-Lite expresses visualizations in JSON (JavaScript Object Notation) making them easier to manipulate with computers. Vega-Lite also extends the ideas of previous high-level grammars with composition and interactions. Since Vega-Lite is a grammar, it allows for valid specifications but nonsensical visualizations. To guide people towards good designs within the space of possible specifications, we have developed the Draco visualization model and recommendation system.

2.5 Visualizing Large Datasets

Visualization systems for big data must address two major challenges: *perceptual scalability*—how to encode data in a way that it does not overwhelm the viewer—and *interactive scalability*—respond to interactions with without long delays [122]. In a truly scalable visualization perceptual and interactive scalability should be limited by the chosen resolution of the visualized data, not the number of records [123].

Given the resolution of conventional displays (~1–3 million pixels), visualizing every data point can lead to overplotting. If every data record is rendered as a mark, even modest amounts of data can complicate people's ability to interpret the data (Figure 2.1).

Often a different encoding (e.g., changing from a positional to a color encoding) can reduce overplotting and the amount of screen estate required. Visualization researchers have explored parts of this design space [132, 176, 204] but we are still missing comprehensive design guidelines for scalable visualizations.

As an alternative way to achieve perceptual scalability, we can reduce the amount of data that is rendered. Many data reduction techniques exist [218]: filtering, sampling, aggregation, and modeling.

Sampling techniques select a subset of data, to which standard visualization techniques can be applied (Figure 2.1 (b)). The selected subset, however, may still be too large to visualize effectively

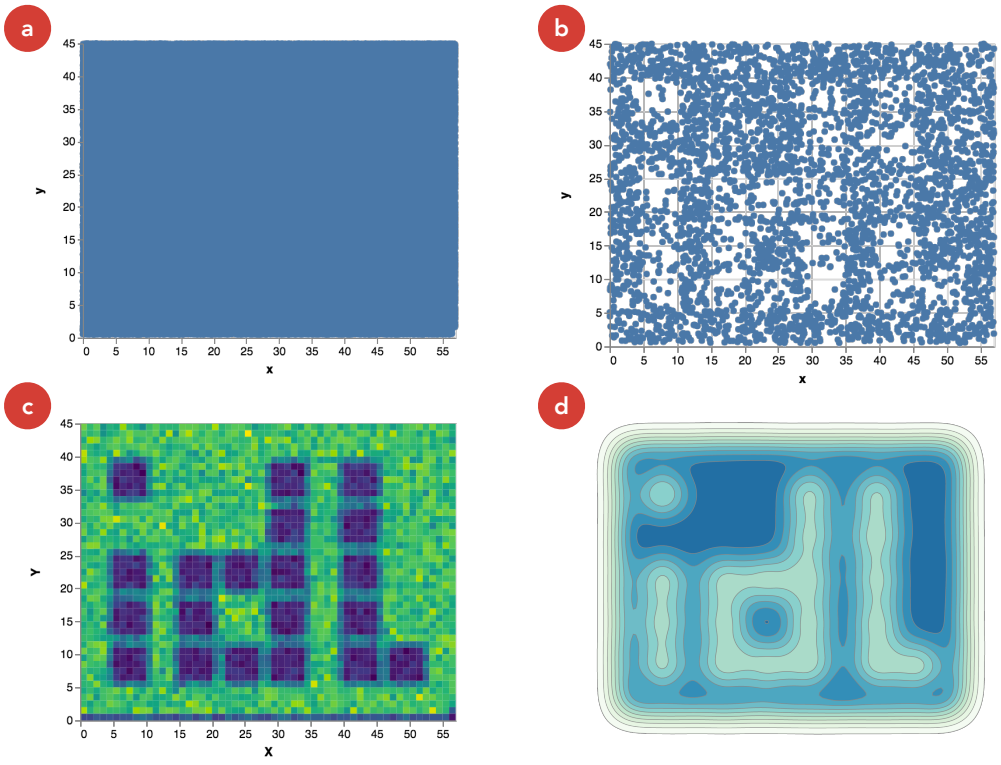

Figure 2.1: The same 200k points plotted four different ways: as a scatterplot (a), sampled (b), binned aggregation (c), and contour plot (d).

and may omit elements of interest. In simple random sampling, every data point has the same probability of being selected. The resulting sample may not be representative and can miss important structures or outliers. For example, the absence of a point at a particular location does not mean that there is no data at this point in the full data. Other sampling methods such as stratified sampling or custom sampling schemes [107, 155] speed up data processing but generally aim to show the same plot type as a plot of the full data.

Binned aggregation summarizes data by dividing the domain of variables into discrete units, and then summarizing the points that fall within each bin [123, 218] (Figure 2.1 (c)). A simple count of records forms a density estimate. Depending on the task, other aggregate functions (sum, average, min, max, etc.) can be applied. For a numeric variable, we can define bins as adjacent intervals over a continuous range. For categorical variables, we can simply treat each value as a bin. Aggregation can also be defined at multiple scales over a hierarchy [54], with nested, potentially non-uniform, bins. For example, temporal values can be aggregated by day, week, month, quarter, year, and so on. In terms of visualization, histograms and heatmaps are exemplary 1D and 2D binned plots.

Binned aggregation is one of the most flexible approaches to perceptually scalable visualization, as it can convey both global patterns (e.g., densities) and local features (e.g., outliers), while enabling multiple levels of resolution via the choice of bin size. Statisticians have proposed various heuristics to select bin sizes for a numeric range (e.g., Sturges' formula [201] and Scott's reference rule [180]). These heuristics can vary significantly and their applicability to big data is unclear. When visualizing large numbers of records, systems may treat bin count as an adjustable parameter, bounded by the screen pixels allocated to a plot and available resources. At the limit, we can map one bin to one pixel and include as many bins as memory constraints allow.

Alpha blending (reduced opacity or transparency) is often used instead of aggregation to combat over-plotting [103]. Alpha-blending, however, is effectively an aggregation in image space rather than data space. It requires drawing every mark and does not let us customize the transfer function between density and visual encoding [28].

Another reduction strategy is to describe data in terms of parametric **statistical models** (Figure 2.1 (d)). For example, we might fit a model and visualize the resulting parameters or theoretical density. For scatter plots we might show trend lines and error envelopes for regression models or contour plots to indicate a probability density. Examples for time series data include moving averages and auto-regressive models. To create a model, we need to know exactly what we want to show and often make certain assumptions about the data. Since the goal of visualization is often to "learn about the unknowns" and discover aspects of the data that are not known a priori, creating a model is often not an acceptable option.

2.6 Scalable Data Analysis Systems

Because people are limited in the number of data points they can effectively perceive, we need to reduce the number of data points being rendered. For interactive visualization "at the speed of thought" the primary bottleneck is then data processing. In a perfectly scalable system the latency between an interaction and the corresponding response does not depend on the size of the data [123]. In conventional client-server architectures, this latency is typically dominated by the time to send a query and receive the results and the query time in the data management system.

There is a design space of systems that combine different approaches to reduce query times: distributed computation, precomputation, indexing, sampling and approximation.

2.6.1 Online Analytical Processing

Online analytical processing, also known as "OLAP" [34], is a well-studied approach to enabling fast analytical queries over large, multidimensional datasets. The focus of OLAP is on supporting aggregate operations, such as computing the average, maximum, or total results of a given measure (e.g., total sales) across multiple data attributes (e.g., across US regions and fiscal quarters). OLAP is used primarily in business intelligence. In OLAP, four primary operations are supported for data analysis: *rollup* (i.e., aggregation to create a coarser summary), *drill-down* (i.e., unrolling

of aggregate results to produce a more detailed summary), *slice-and-dice* (i.e., database filters and projections), and *pivots* along the desired data dimensions [34]. Commonly, the computation of OLAP queries is distributed across a cluster of machines that evaluate parts of the query on partitions of the data (e.g., in large-scale shared-nothing systems [200]).

OLAP engines are designed to process large compute jobs in distributed clusters of machines. While these systems handle massive datasets, query response times often exceed what is considered interactive. To respond faster to user queries, systems use indexes, prefetching, and approximation.

2.6.2 Data Cubes

The data cube [79] is a fully materialized multi-dimensional table containing all aggregate results for any aggregation operation that can be performed on a given dataset (i.e., any rollup or drill-down operation). In a data cube, we can access aggregate results directly; instead of computing them from the base data. However, the size of the data cube grows exponentially in the number of dimensions. Even though these cubes can be computed in parallel [104, 139, 140], high materialization costs make data cubes impractical for cold-start analytics.

Recent research has proposed specialized low latency systems to explore massive datasets such as large time series [31]. *Nanocubes* [119] and *imMens* [123] are systems that store data in multi-dimensional data cubes at multiple levels of resolution to perform accelerated query processing. Nanocubes builds on the ideas of Dwarf cubes, a compact data structure for sparse cubes [188]. While Nanocubes requires a round trip for each interaction, imMens decomposes the cubes into tiles that can be loaded into the browser. Through this decomposition and preloading imMens achieves interactive scalability. However, it requires costly precomputation (many hours, no cold start), limits interaction to a few dimensions, and only supports interaction at a bin resolution.

2.6.3 Data Tiles and Prefetching

An alternative to precomputed indexes and data cubes is to reduce latency with caching, preloading, and compression [12, 52]. Preloading is most effective when the system fetches data that is needed as early as possible. Cetintemel et al. built on the assumption that interactions follow common patterns and propose to use a model of user interactions to load the most useful data [30]. Battle et al. implemented this idea in *ForeCache*, a system for browsing (zooming and panning) data tiles, for example of satellite imagery. ForeCache speculatively loads data tiles that the user is most likely going to request in the near future [11]. However, ForeCache is limited to browsing interactions. We build on the ideas of data cubes and prefetching to build Falcon; a system for real-time brushing and linking for billions of records.

2.6.4 Approximate Query Processing

An orthogonal approach to distributing work across multiple machines or indexing schemes is using only a representative subset—a sample—of the data and computing an approximation. These samples can be used to extrapolate estimated final values and the degree of certainty of the estimate. The error in an approximation is often proportional to $\mathcal{O}(1/\sqrt{n})$ where n is the number of tuples in the sample. We can make two observations. First, to reduce the error from 2% to 0.2% one typically needs 100x more data. Second, the error only depends on the number of tuples in the sample and not the size of the original data. For big data, we can often have a good approximation with only a fraction of the original data.

The idea behind approximate query processing (AQP) is to run aggregation queries on a sample and estimate the true value of the aggregation. BlinkDB [3] implements this idea but requires costly precomputation of samples to provide guarantees.

Rather than forcing users to settle on a fixed size sample, Hellerstein et al. [93] proposed online aggregation: using an incrementally growing set of samples. The analyst can get a response quickly, if imprecisely; the system converges on more precise values. Online aggregation has been adopted

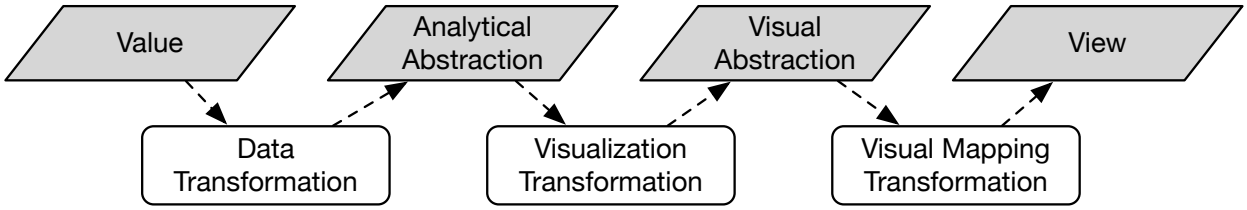

Figure 2.2: The Information Visualization Data State Reference Model from Chi et al. [38], which can be used to model many visualization techniques and applications.

by the visualization community as a "progressive analytics" approach. Fisher et al. [62] showed that analysts find approximate results sufficiently robust before the complete query finishes. Adopting a progressive computation model requires support for streaming in the whole data processing pipeline and thus almost a complete rewrite of existing applications. With optimistic visualization in Pangloss, we can use existing blocking implementations for the precise results.

2.6.5 Data Visualization as Dataflows

One way to modularize an interactive application is the *Information Visualization Reference Model* [25]. The model decomposes the visualization process into data acquisition and storage, visual encoding of data, and rendering and interaction. Equivalent to the information visualization reference model, Chi et al. define the *Data State Reference Model* [39] (illustrated in Figure 2.2) to describe the set of transformation stages which raw data goes through until visualized. Transformed data is ultimately presented in a view after a visual mapping transform. Chi showed that this taxonomy is generic enough to describe many common visualization techniques [38].

We can use this taxonomy to describe how a visualization application uses a transformation pipeline as a dataflow graph of operators such as `filter`, `map`, `reduce`, `join`, and `aggregate`. A dataflow graph is almost equivalent to a query execution plan in databases. After the transformation part of the pipeline, the data must be mapped to visual attributes such as `x`, `y`, `color`, or `shape` using a scale (e.g., a mapping from the data domain to a color palette) [222]. In addition to the above

operators, a visualization application could use caching "operators". For example, an interactive map application that generates static tiles inserts a cache after the visual abstraction (tiles are pregenerated).

The pipeline can be partitioned among different computers according to available resources [135]. For example, the data may be residing on a database server while the user is interacting with a visualization application on a thin client such as a web browser. In these scenarios, developing an interactive visualization application means optimizing the whole dataflow from bytes on disk to pixels on screen. The client–server architecture is popular on the web and in analytics applications. One of the most widely used systems for visually exploring data is Tableau, the commercial version of Polaris [198].

2.6.6 Data Management Systems for Visualization

The Tableau visualization system connects to relational databases and generates queries from a visualization specification. However, unsatisfied with the performance of this online approach, Tableau created the Tableau Data Engine [216], a specialized data analytic engine tightly coupled with the desktop software. The authors emphasize the need for tighter coupling of the data processing and visualization systems. In traditional systems, the visualization pipeline is divided into separate and independent components. A lot of effort has gone into optimizing the data processing systems but there has been little consideration of the corresponding user interfaces. In this thesis, we show that deep integration of a visualization frontend with the data processing system improves the user experience by reducing response times.

Wu et al. [227] describe their vision of a database system optimized for visualization applications. They propose a *Data Visualization Management System* (DVMS) that reduces query times by considering visualization constraints for query optimization. Even though no such system exists yet, such a system will need a machine readable specification of the appearance and behavior of visualizations. The languages proposed in this thesis can express these specifications.

3 Vega-Lite: A Grammar of Interactive Multi-View Graphics

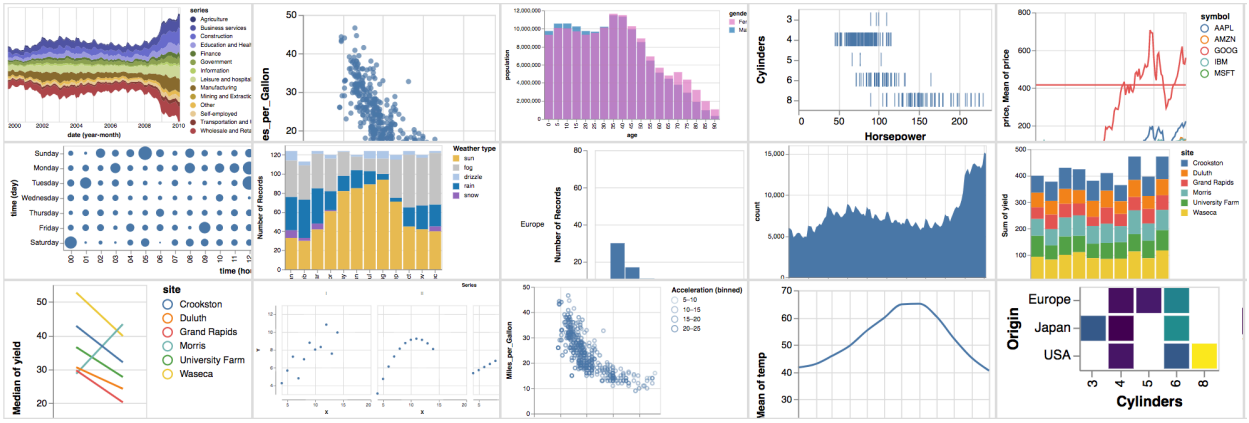

Figure 3.1: Example visualizations created with Vega-Lite.

Most visualizations are authored through end-user applications such as spreadsheets or business intelligence tools. Many of these tools lack consideration of perceptual principles or fall short of fully supporting an expressive range of graphics. To raise the abstraction level of visualization, this thesis contributes Vega-Lite as a foundation for writing programs that generate visualizations, for example in the Voyager visualization recommendation browser [224]. Vega-Lite is a declarative high-level format for representing and reasoning about interactive, multi-view visualizations.

By deferring execution concerns to the runtime, designers can focus on design questions rather than implementation details. The declarative specification facilitates systematic enumeration of the design space, retargeting to different platforms, reuse, and automatic optimization of the execution.

Vega-Lite is designed for statistical graphics and—compared to the lower-level Vega [178] that it compiles to—trades off general expressivity for orders of magnitude shorter specifications. A chart is specified as a set of encodings that map data fields to properties (e.g., color or size) of graphical marks (e.g., points or bars). By combining these basic building blocks, users can create an expressive range of graphics (Figure 3.1). To keep specifications concise, users can omit low-level details such as axes and scales from their specifications. The gap between the high-level abstractions and the low-level execution leads to ambiguity. The Vega-Lite compiler resolves this ambiguity with carefully designed rules. Vega-Lite uses a declarative model for visual encoding, providing a balance of expressive power and usable, domain-specific constructs. This approach provides an abstraction where people can rapidly create visualizations in the midst of an analysis session and where the runtime system can automatically optimize how data is processed.

In contrast to prior declarative visualization languages, Vega-Lite introduces a view algebra for combining basic plots into more complex multi-view displays, and a new selection abstraction for declarative specification of interaction techniques. In Vega-Lite, a selection is an abstraction that defines input event processing, points of interest, and a predicate function for inclusion testing. Selections parameterize visual encodings by serving as input data, defining scale extents, or by driving conditional logic. The Vega-Lite compiler automatically synthesizes requisite data flow and event handling logic, which users can override for further customization. In contrast to existing reactive specifications, Vega-Lite selections decompose an interaction design into concise, enumerable semantic units.

We first published Vega-Lite at IEEE VIS 2015 as part of the Voyager system (co-authored with Kanit Wongsuphasawat, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer) [224] and extended

it with selection to support interactions at IEEE VIS 2016 (co-authored with Arvind Satyanarayan, Kanit Wongsuphasawat, and Jeffrey Heer) [177]. Vega-Lite is an open source system—including examples, tutorials, documentation, and an online editor—available at vega.github.io/vega-lite.

3.1 Introduction

Grammars of graphics span a gamut of expressivity. Low-level grammars such as Protovis [17], D3 [18], and Vega [178] are useful for *explanatory* data visualization or as a basis for customized analysis tools, as their primitives offer fine-grained control. However, for *exploratory* visualization, higher-level grammars such as ggplot2 [217], and grammar-based systems such as Tableau (née Polaris [199]), are typically preferred as they favor conciseness over expressiveness. Analysts rapidly author partial specifications of visualizations; the grammar applies default values to resolve ambiguities and synthesizes low-level details to produce visualizations.

High-level languages can also enable search and inference over the space of visualizations. For example, Wongsuphasawat et al. [224] introduced Vega-Lite to power the Voyager visualization browser. By providing a smaller surface area than the lower-level Vega language, Vega-Lite makes systematic enumeration and ranking of data transformations and visual encodings more tractable. The first version of Vega-Lite introduced with Voyager only supports static, single-view visualizations.

However, existing high-level languages provide limited support for interactivity. An analyst can, at most, enable a predefined set of common techniques (linked selections, panning & zooming, etc.) or parameterize their visualization with dynamic query widgets [182]. For custom, direct-manipulation interaction they must instead turn to imperative event handling callbacks. Recognizing that callbacks can be error-prone to author, and require complex static analysis to reason about, Satyanarayan et al. [179] formulated declarative interaction primitives for Vega. While these additions facilitate programmatic generation and retargeting of interactive visualizations, they

remain low-level. Verbose specification impedes rapid authoring and hinders systematic exploration of alternative designs.

As part of this thesis, we present Vega-Lite, a high-level declarative language for interactive multi-view graphics. We designed Vega-Lite with three design goals: concise specifications, support for common designs, and support for interaction. Short specifications facilitate rapid authoring in the midst of an analysis session. While concise, Vega-Lite's API supports different designs that can facilitate different tasks [176] of common analysis scenarios. Interaction is critical for effective exploration [91, 159].

Towards these design goals, we describe how Vega-Lite enables concise, high-level specification of *static* and *interactive* data visualizations.

We first contribute a grammar to describe static multi-view visualizations. A single view is specified as a set of encodings that map data fields to properties (e.g., color or size) of graphical marks (e.g., points or bars). To support expressive interaction methods, Vega-Lite contributes an algebra to compose single-view Vega-Lite specifications into multi-view displays using *layer*, *concatenate*, *facet* and *repeat* operators. Vega-Lite's compiler infers how input data should be reused across constituent views, and whether scale domains should be unioned or remain independent.

Second, we contribute a high-level interaction grammar. With Vega-Lite, an interaction design is composed of *selections*: visual elements or data points that are chosen when input events occur. Selections parameterize visual encodings by serving as input data, defining scale extents, and providing *predicate* functions for testing or filtering items. For example, a rectangular "brush" is a common interaction technique for data visualization. In Vega-Lite, a brush is defined as a selection that holds two data points that correspond to its extents (e.g., captured when the mouse button is pressed and as it is dragged, respectively). Its predicate can be used to highlight visual elements that fall within the brushed region, and to materialize a dataset as input to other encodings. The selection can also serve as the scale domain for a secondary view, thereby constructing an overview + detail interaction.

For added expressivity, Vega-Lite provides a series of operators to *transform* a selection. Transforms can be triggered by input events as well, and manipulate selection points or predicate functions. For example, a *toggle* transform adds or removes a point from the selection, while a *project* transform modifies the predicate to define inclusion over specified data fields.

The Vega-Lite compiler synthesizes a lower-level Vega specification [178] with the requisite data flow, and default event handling logic that a user can override. Through a range of examples, we demonstrate that Vega-Lite brings the advantages of high-level specification to interactive visualization. Common methods, including linked selection, panning, and zooming, as well as custom techniques (drawn from an established taxonomy [228]) can be concisely described. Moreover, selections, transformations, and their application to visual encodings decompose interaction into a parametric design space. We show how each of these parameters can be systematically varied to generate alternate interaction techniques for a given set of visual encodings. Such enumeration can be useful to explore alternative designs, and can aid higher-level reasoning about interaction—for example, recommending suitable interaction techniques as part of a design tool.

3.2 Related Work

Vega-Lite builds on prior work on grammars of graphics, visualization systems, and techniques for interactive selection and querying.

3.2.1 Grammar-Based Visual Encoding

Since the initial publication of Wilkinson's *The Grammar of Graphics* [222] in 1999, formal grammars for statistical graphics have grown increasingly popular as a way to succinctly specify visualizations. In this grammar, a visualization is described as a mapping of fields to visual properties of graphical marks. Wilkinson's work was quickly followed by the Stanford Polaris system [199], later commercialized as Tableau. Hadley Wickham's popular ggplot2 [217] and ggvis [74] packages implement variants of Wilkinson's model in the R statistical language. These tools eschew chart

templates, which offer limited means of customization, in favor of combinatorial building blocks. Abstracting data models, graphical marks, visual encoding channels, scales and guides (i.e., axes and legends) yields a more expressive design space, and allows analysts to rapidly construct graphics for exploratory analysis [91]. Concise specification is achieved in part through ambiguity: users may omit details such as scale transforms (e.g., linear or log) or color palettes, which are then filled in using a rule-based system of smart defaults. More expressive lower-level (and thus more verbose) grammars, including those of Protovis [17], D3 [18], and Vega [178], have been widely used for creating explanatory and highly-customized graphics.

The design of Vega-Lite is heavily influenced by these works. Drawing from Wilkinson's grammar and Polaris/Tableau, Vega-Lite similarly represents basic plots using a set of encoding definitions that map data attributes to visual channels such as position, color, shape, and size, and may include common data transformations such as binning, aggregation, sorting, and filtering. Drawing from Vega, Vega-Lite uses a portable JSON syntax that permits generation from a variety of programming languages. Vega-Lite specifications are compiled to full Vega specifications, hence the expressive gamut of Vega-Lite is a strict subset of that of Vega. As we will later demonstrate, Vega-Lite sacrifices some expressiveness for dramatic gains in the conciseness and clarity of specification.

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

3.2.2 Specifying Interactions in Visualization Systems

Despite the central role of interaction in effective data visualization [91, 159], little work has been done to develop a grammar for specifying interaction techniques. Wilkinson's grammar includes no notion of interaction. Neither does VizQL, the language underlying Tableau [84]. In Tableau, interactions are handled by a separate layer, implemented as an event model. Early systems like GGobi [202] support common techniques as well, and provide imperative APIs for custom methods. However, such APIs make easy tasks needlessly complex, burdening developers with learning low-level execution details. More recent systems, including Protovis, D3, and VisDock [40], offer a typology of common techniques that can be applied to a visualization. Such top-down approaches, however, limit customization and composition. For example, D3's interactors encapsulate event processing, making it difficult to combine them if their events conflict (e.g., if dragging triggers brushing *and* panning).

The prior work perhaps most closely related to Vega-Lite is the Reactive Vega language [179]. Reactive Vega draws on Functional Reactive Programming techniques to formulate composable, declarative interaction primitives for data visualization. Reactive Vega models input events as continuous data streams. To succinctly define event streams of interest, Vega employs an *event selector* syntax, which Vega-Lite also uses for customized event logic. Event streams, in turn, drive dynamic variables called *signals*. Signals parameterize the remainder of the visualization specification, endowing it with reactive semantics. When a new event fires, it propagates to dependent signals; visual encodings that use them are automatically re-evaluated and re-rendered. This reactive approach is not only capable of expressing a diverse set of interactions [179], it is performant as well [178], with interactive performance at least twice as fast as the equivalent D3 program. Moreover, declarative specifications are easier to analyze, optimize, and retarget to different platforms than imperative programs.

However, Vega's reactive specifications are low-level and verbose. As a user, you have to specify event triggers and update expressions. Specifying common techniques can be time-consuming,

requiring tens of lines of JSON, and it is difficult to know how to adapt techniques in pursuit of alternative designs. Vega has low viscosity in the *Cognitive Dimensions of Notation* [80]. In contrast, Vega-Lite is a higher-level specification language, with primitives that decompose interaction design into a parametric space. Common methods require typically 1-2 lines of code, and design variations can be explored by systematically enumerating defined properties. Nevertheless, Reactive Vega provides a performant runtime and an "assembly language" to which Vega-Lite specifications are compiled.

3.2.3 Interactive Selection and Querying

Selection, often in the form of users clicking or lassoing visual items of interest, is a fundamental operation in user interfaces and has been well-studied in the context of data visualization. For example, in Snap-Together Visualization [146], multiple views are coordinated via "primary-" and "foreign-key actions," which propagate selected data tuples from one view to the others. Wilhelm [221] describes the need for such "indirect object manipulation" methods as an axiom of interactive data displays. Chen's compound brushing [36] provides a visual dataflow language for specifying a rich space of transformations of brush selections. More recently, Brunel [23] provides a special `#selection` data field that is dynamically populated with the elements a user interacts with, and can be used to link multiple views or filter input data. Similarly, RStudio's Shiny [182], an imperative web application layer, provides `brushedPoints` and `nearestPoints` functions which can be used throughout an R script to operate on selected elements.

Other systems have studied formally representing selections as data queries [221]. For example, brushing interactions in VQE [49] generate *extensional* queries that enumerate all items of interest; a form-based interface enables specification of *intensional* (declarative) queries. Individual point and brush selections in DEVise [125], known as *visual queries*, map to a declarative structure and are used to link together multiple views. With VIQING [148], rectangular "rubber band" selections are modeled as range extents, and views can be dropped on top of each other to join their underlying

datasets. Heer et al. [87] demonstrate that by modeling a selection as a declarative query, interactive "query relaxation" can successively capture more items of interest.

Vega-Lite builds on this work by richly integrating an interactive *selection* abstraction with the primitives of visual encoding grammars. Vega-Lite selections are populated with one or more points of interest, in response to user interaction. Extensible *predicate* functions map selections to declarative queries, and allow a minimal set of "backing" points to represent the full space of selected points. Additional operators can transform a selection's predicate or backing points (e.g., offsetting them to translate a brush selection or perform panning). Selections then parameterize visual encodings by serving as input data, defining scale extents, or using predicates to test or filter items. The result is an enumerable, combinatorial design space of *interactive* statistical graphics, with concise specification of not only linking interactions, but panning, zooming, and custom techniques as well.

3.3 The Vega-Lite Grammar of Graphics

Vega-Lite combines a grammar of graphics with a novel grammar of interaction. In this section, we describe Vega-Lite's basic visual encoding constructs and an algebra for view composition. In Voyager [224], we first introduced the simplest Vega-Lite specification—here referred to as a *unit* specification—that defines a single Cartesian plot with a specific mark type to encode data (e.g., bars, lines, plotting symbols). Given multiple unit plots, Vega-Lite introduces *layer*, *concat*, *facet*, and *repeat* operators to provide an algebra for constructing *composite* views. This algebra can express layered plots, trellis plots, and arbitrary multiple view displays. Each operator is responsible for combining or aligning underlying scales and axes as needed.

3.3.1 Unit Specification

A unit specification describes a single Cartesian plot, with a backing *data* set, a given *mark-type*, and a set of one or more *encoding* definitions for visual *channels* such as position (*x*, *y*), *color*, *size*, etc.. Formally, a unit view is a four-tuple:

unit := (data, transform, 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 *transformations*, 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*). Vega-Lite version 3 introduces composite marks (e.g., *boxplot*, *errorband*), which are compiled to layered marks. Composite marks allow for even more concise specifications of common designs. 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 additionally be transformed using *functions* such as binning, aggregation (sum, average, etc.), 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 [199, 224].

Unit specifications can express 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 syntax, as shown in Figure 3.2.

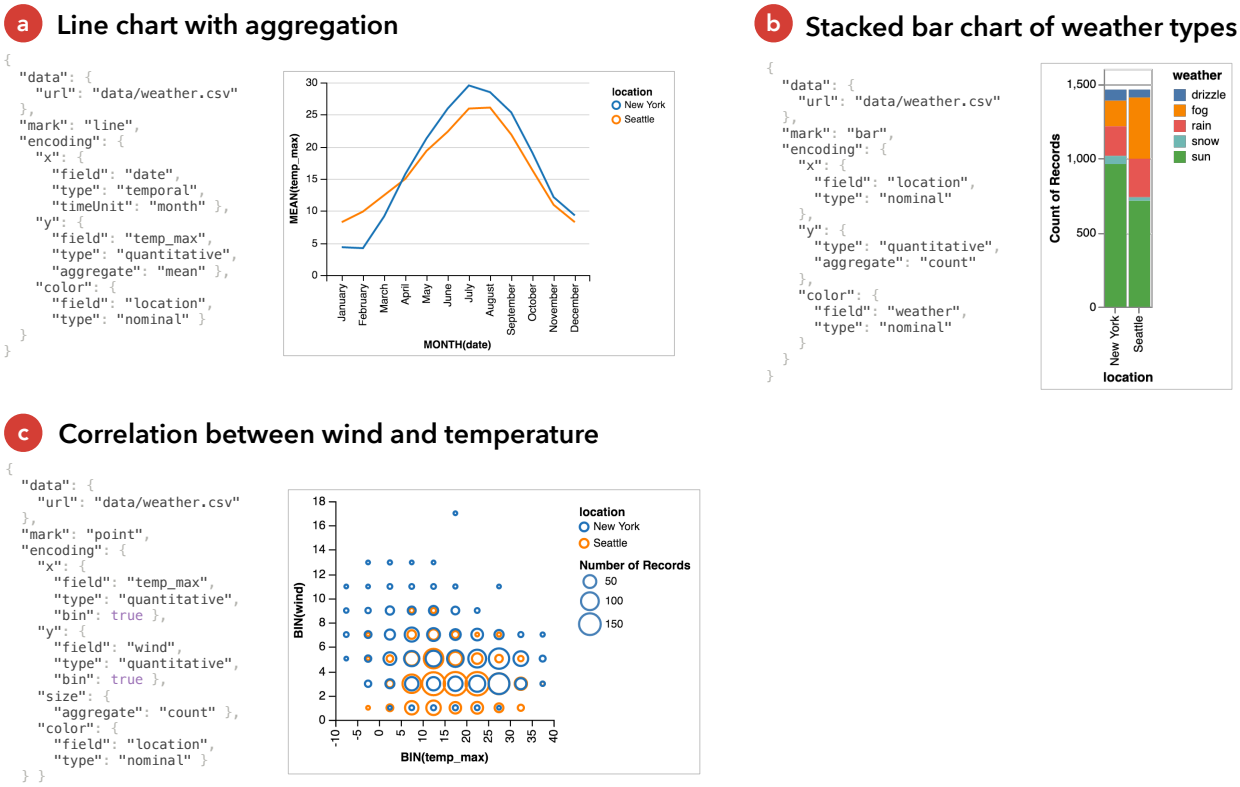

Figure 3.2: Vega-Lite *unit* specifications visualizing weather data. These examples demonstrate varied mark types and data transformations.

3.3.2 View Composition Algebra

Given multiple *unit* specifications, *composite* views can be created using a set of composition operators. Here we describe the set of supported operators. We use the term *view* to refer to any Vega-Lite specification, whether it is a *unit* or *composite* specification.

3.3.2.1 Layer

The *layer* operator accepts multiple *unit* specifications to produce a view in which subsequent charts are plotted on top of each other. For example, a layered view could consist of one layer showing a histogram of a full data set, and another overlaying a histogram of a filtered subset (Figure 3.13). The signature of the operator is:

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

To create a layered view, 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. We believe this is a useful default for producing coherent and comparable layers. However, Vega-Lite cannot enforce that an 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.

To override the default behavior, users can specify strategies to *resolve* scales and guides using tuples of the form *(scale|axis|legend, channel, resolution)*, where *resolution* is one of *independent* or *shared*. Independent scales and guides for each layer produce a dual-axis view, as shown in the layered plots in Figure 3.3.

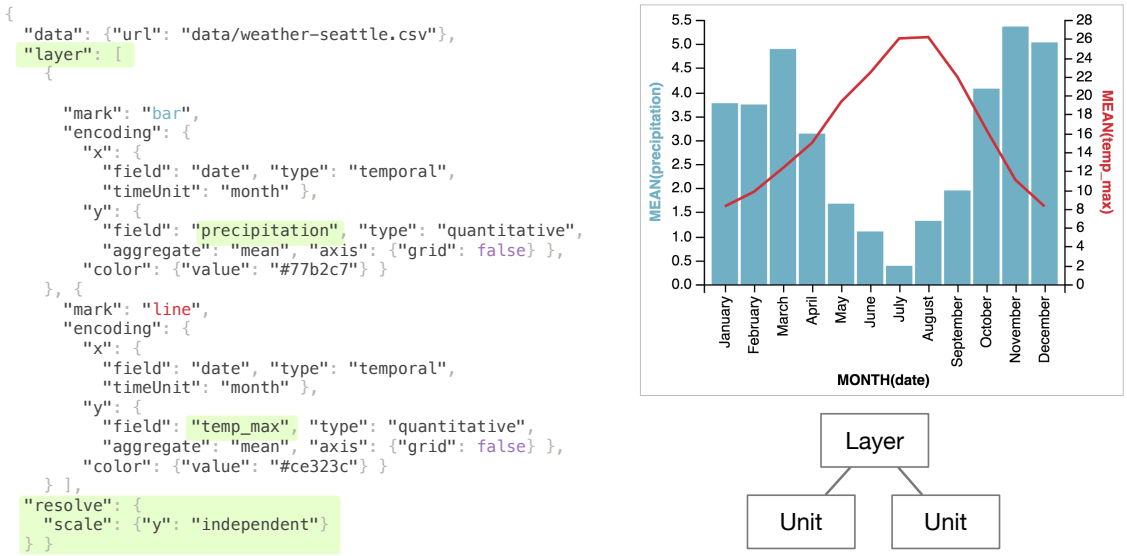

Figure 3.3: A dual axis chart that *layers* lines for temperature on top of bars for precipitation; each layer uses an independent y-scale.

3.3.2.2 Concatenation

To place views side-by-side, Vega-Lite provides operators for horizontal and vertical concatenation and wrapping. The signatures for these operators are:

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

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

$$concat([view_1, view_2, ...], columns, resolve)$$

If aligned spatial channels have matching data fields (e.g., the *y* channels in a *hconcat* use the same field), a shared scale and axis are used. Axis composition facilitates comparison across views and optimizes the underlying implementation. Figure 3.4 concatenates the line chart from Figure 3.2(a) with a dot plot, using independent scales.

```
{
  "vconcat": [
    { ... },
    {
      "data": {
        "url": "data/weather.csv",
      },
      "transform": {
        "filter": "datum.precipitation > 0"
      },
      "mark": "point",
      "encoding": {
        "y": {"field": "location","type": "nominal"},
        "x": {
          "type": "quantitative",
          "aggregate": "count"
        },
        "color": {
          "field": "date",
          "type": "temporal",
          "timeUnit": "year"
        }
      }
    }
  ]
}
```

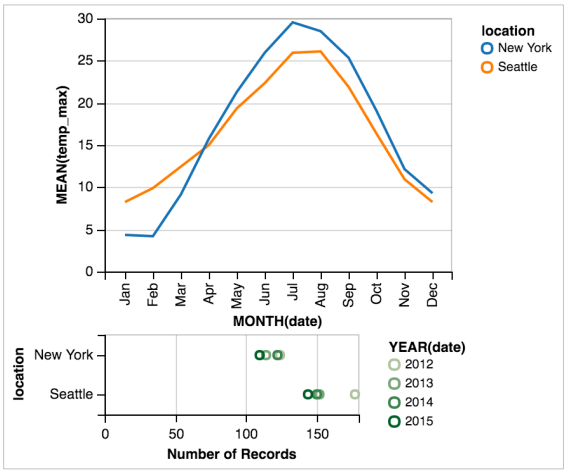

Figure 3.4: The temperature line chart from Figure 3.2(a) *concatenated* with rainy day counts in New York and Seattle; scales and guides for each plot are independent.

3.3.2.3 Facet

While concatenation allows composition of arbitrary views, one often wants to set up multiple views in a parameterized fashion. The *facet* operator produces a trellis plot [13] by subsetting the data by the distinct values of a field. The signature of the facet operator is:

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

The *channel* indicates if sub-plots should be laid out vertically (*row*) or horizontally (*column*). The given *data* source is partitioned using distinct values of the *field*. The *view* specification provides a template for the sub-plots, inheriting the backing *data* for each partition from the operator. The *scale* and *axis* parameters specify how sub-plots are positioned and labeled. Figure 3.5 demonstrates faceting into columns.

For concision, each sub-plot's unit view can omit the *data* property and implicitly adopt the partitioned data.

To facilitate comparison, scales and guides for quantitative fields are shared by default. This ensures that each facet visualizes the same data domain. Users can override the default behavior via the *resolve* component.

```
{
  "data": {
    "url": "data/weather.csv",
  },
  "facet": {
    "column": {
      "field": "location",
      "type": "nominal"
    }
  },
  "spec": {
    "mark": "line",
    "encoding": {
      "x": { ... },
      "y": { ... },
      "color": { ... }
    }
  }
}
```

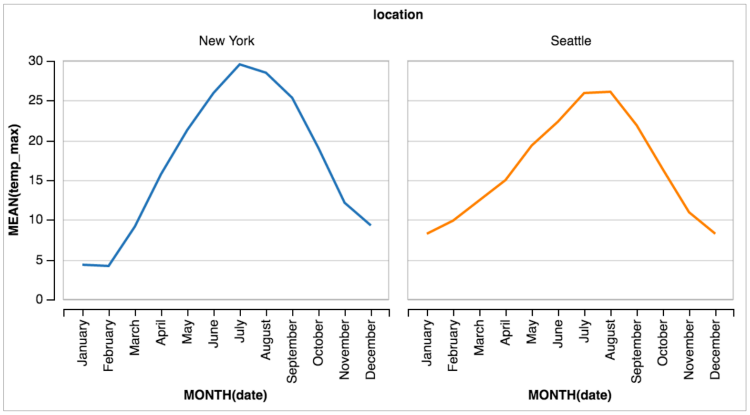

Figure 3.5: Weather data *faceted* by location; the y-axis is shared, and the underlying scale domains unioned, to enable easier comparison.

3.3.2.4 Repeat

The *repeat* operator generates multiple plots, but unlike *facet* allows full replication of a data set in each cell. For example, *repeat* can be used to create a scatterplot matrix (SPLOM), where each cell shows a different 2D projection of the same data table. The signature is:

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

Like *facet*, the *channel* parameter indicates if plots should divide by *row* or *column*. Rather than partition data according to a field, this operator generates one plot for each entry in a list of *values*. Encodings within the repeated *view* specification can refer to this provided *value* to parameterize the plot[1]. By default, scales and axes are independent, but legends are shared when data fields coincide. Like *facet*, the *scale* and *axis* components allow users to override defaults for how sub-plots

[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

are positioned and labeled, while *resolve* controls resolution of scales and guides within the plots themselves.

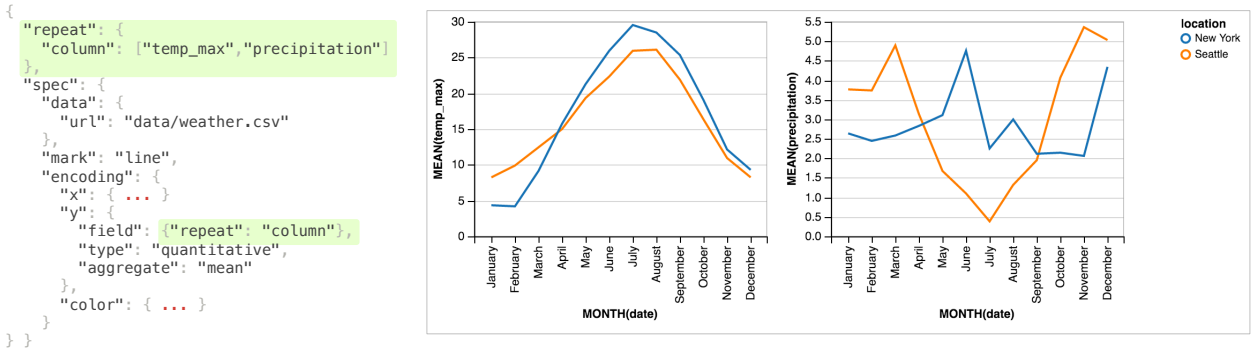

Figure 3.6: *Repetition* of different measures across columns; the *y* channel references the *column* template parameter to vary the encoding.

3.3.3 Nested Views

Composition operators can be combined to create more complex nested views or dashboards, with the output of one operator serving as input to a subsequent operator. 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.

---

including rewrite rules that apply to the inner view specification). However, we believe the current syntax to be more usable and concise than these alternatives.

3.4 The Vega-Lite Grammar of Interaction

To support specification of interaction techniques, Vega-Lite extends the definition of unit specifications to also include a set of *selections*. Selections identify the set of points a user is interested in manipulating. We formally define a selection 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.

A single selection is backed by a single 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 [179]), and can be invoked as such. For example, it can be referenced by name within a filter expression, or its values used directly for encoding channels. Multi selections, on the other hand, are backed by datasets into which points 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. Figure 3.7 illustrates how points are highlighted in a scatterplot using single and multi selections.

Intervals are similar to multi selections. They are backed by datasets, but their predicates determine whether an argument falls within the minimum and maximum extent defined by the backing points. Thus, they express continuous selections. The compiler automatically adds a rectangle mark, as shown in Figure 3.8(a), 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.

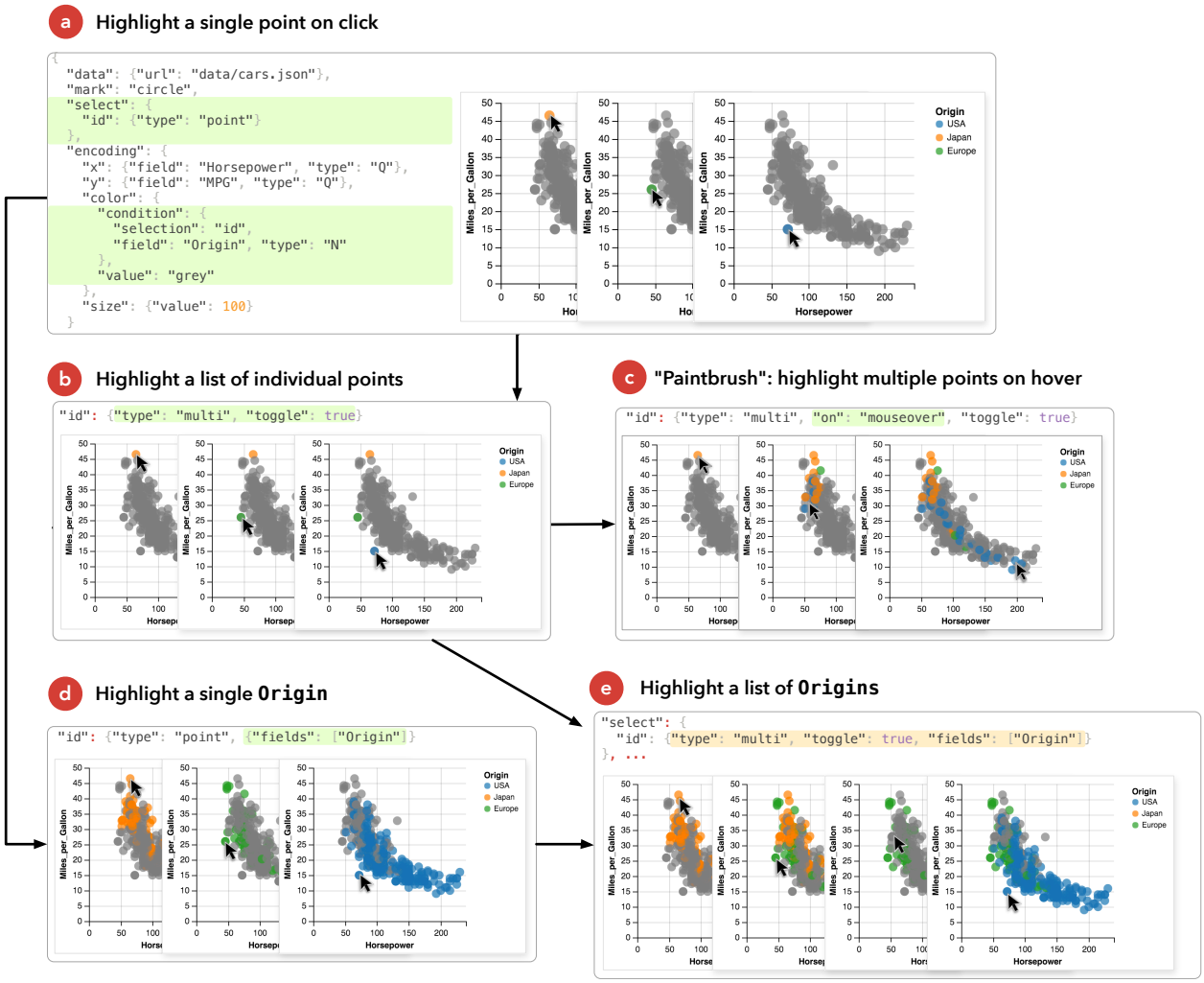

Figure 3.7: (a) Adding a *single* point selection to parameterize the fill color of a scatterplot's circle mark. (b) Switching to a *multi* selection, with the *toggle* transform automatically added (`true` enables default shift-click event handling). (c) Specifying a custom event trigger: the first point is selected on `mouseover` and subsequent points when the shift key is pressed (customizable via the *toggle* transform). (d) Using the *project* transform with a single selection to highlight all points with a matching `Origin`, and (e) combining it with a multi selection to select multiple `Origins`.

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

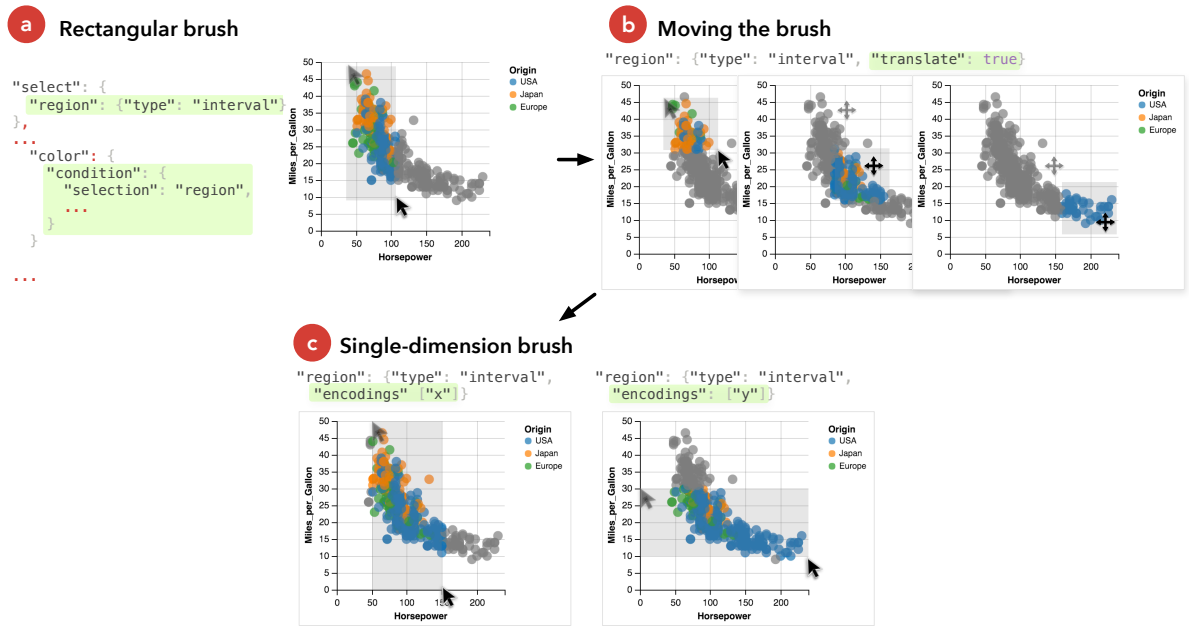

Figure 3.8: (a) Adding a rectangular brush, as an *interval* selection, which can be (b) moved with the *translate* transform (automatically instantiated by the compiler) or (c) restricted to a single dimension with the *project* transform.

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 can be customized. By default, we use mouse events. A user can specify alternate events (e.g., touch events on mobile) using Vega's event selector syntax [179]. For example, Figure 3.7(c) demonstrates how `mouseover` events are used to populate a multi selection. With the event selector syntax, multiple events are specified using a comma (e.g., `mousedown, mouseup` adds items to the selection when either event occurs). A sequence of events is denoted with the right-combinator. For example, `[mousedown, mouseup] > mousemove` selects all `mousemove` events that occur between a `mousedown` and a `mouseup` (otherwise known as "drag" events). Events can also be filtered using square brackets (e.g., `mousemove [event.pageY > 5]` for events at the top of the page) and throttled using braces (e.g., `mousemove{100ms}` populates a selection at most every 100 milliseconds).

Finally, selections can be *initialized* with specific backing points (we defer discussion of *transforms* and *resolve* to subsequent sections). Vega-Lite provides a built-in mechanism to initialize multi and interval selections using the scales of the unit specification they are defined in. Doing so populates the selection with the given scales' domain or range, as appropriate for the selection, and parameterizes the scales to use the selection instead. By default, this occurs for the scales of the `x` and `y` channels, but alternate scales can be specified by the user. This step allows scale extents to be interactively manipulated, yet remain automatically initialized by the input data.

3.4.1 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. We identify the following transforms as a minimal set to support both common and custom interaction techniques:

3.4.1.1 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`). Figure 3.7(d, e) 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 (Figure 3.8(c)) or to determine which scales initialize a selection.

3.4.1.2 Toggle

toggle(event)

The toggle transform is automatically instantiated for uninitialized multi selections. When the *event* occurs, the corresponding point is added or removed from a multi selection's backing dataset. By default, the toggle *event* corresponds to the selection's event but with the shift key pressed. For example, in Figure 3.7(b), additional points are added to the multi selection on shift-click (where `click` is the default event for multi selections). The selection in Figure 3.7(c), 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.

3.4.1.3 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), an optional *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.

3.4.1.4 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. An optional *factor* should be specified, if it cannot be determined from the events (e.g., when the arrow keys are pressed).

3.4.1.5 Bind

bind(widget|scales)

The bind transform establishes a two-way binding between the selection and input elements or scales. One input element per projection is generated and can be used to manipulate the selection; any direct manipulation interactions (e.g., clicking on the visualization) will similarly update the input element. If multiple projections are specified, customized bindings can be specified by mapping the projected field/encoding to a binding definition.

With interval selections, the *bind* property can be set to the value *scales* to enable a two-way binding between the selection and the scales used within the same view. This binding first populates the interval selection with the scale domains, and then uses the selection to drive the scale domains. As a result, the view now functions like an interval selection and can be panned and zoomed as shown in Figure 3.9.

3.4.1.6 Nearest

nearest()

The nearest transform computes a Voronoi decomposition, and augments the selection's event processing, such that the data value or visual element nearest the selection's triggering *event* is selected (approximating a Bubble Cursor [81]). Currently, the centroid of each mark instance is

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

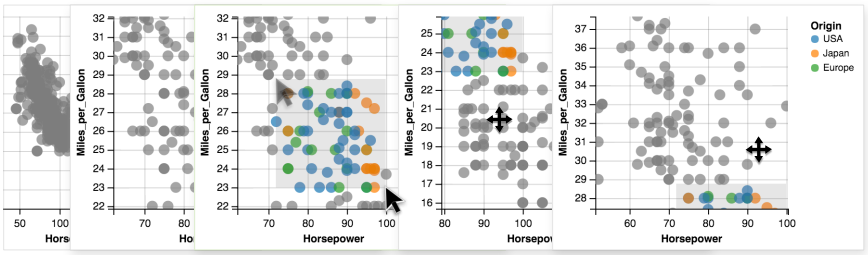

Figure 3.9: Panning and zooming the scatterplot is achieved by binding an interval selection to the scales, and then applying *translate* and *zoom*. Alternate events are specified to prevent collision with the brushing interaction, previously defined in Figure 3.8.

used to calculate the Voronoi diagram, but we plan to extend this operator to account for boundary points as well (e.g., rectangle vertices).

3.4.2 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 an `if-then-else` chain of logic within an encoding channel definition. Each data tuple participating in the encoding is evaluated against selection predicates in turn, and visual properties are set corresponding to the first branch that evaluates to `true`. For example, as shown in Figure 3.7, the fill color of the scatterplot circles is determined by a data field if they fall within the `id` selection, or set to gray 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

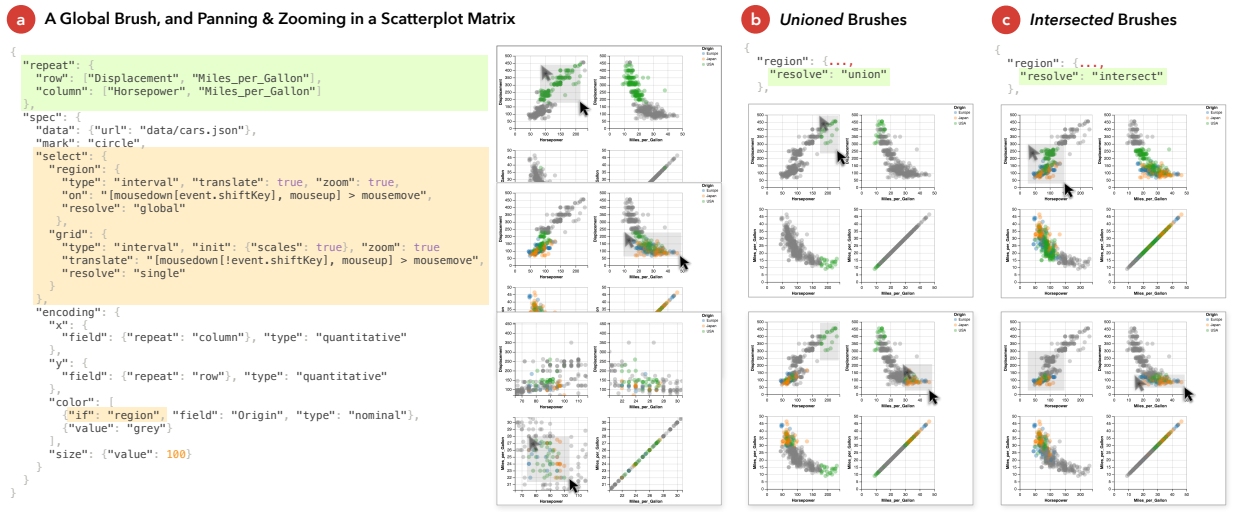

Figure 3.10: (a) By adding a *repeat* operator, we compose the encoding and interactions from Figure 3.9 into a scatterplot matrix. Users can brush, pan, and zoom within each cell, and the others update in response. By default, composite selections are resolved to a *single* global selection: brushing in a cell replaces previous brushes. However, the resolution scheme can be set to (b) *union*, such that points highlight if they fall in any brush; and (c) *intersect*, such that points highlight only when they are within all brushes.

(or visual elements) of the unit specification it is defined in. However, selections can also be materialized against arbitrary datasets; a *map* transform supports rewriting the predicate function in case of 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 channel's scale. Doing so constructs interactions that manipulate viewports, including panning & zooming (Figure 3.9) and overview + detail (Figure 3.11).

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

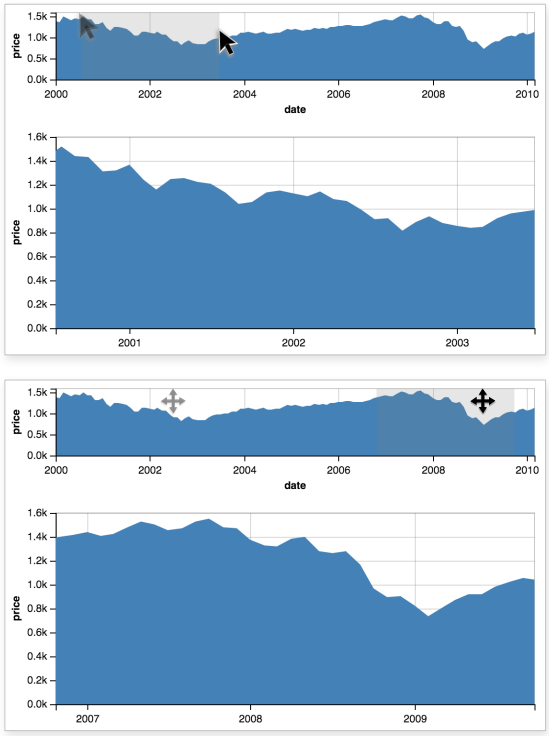

Figure 3.11: An overview+detail visualization is constructed by concatenating two unit specifications, with a selection in the first one parameterizing the x scale domain in the second.

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 compute expression, or to determine a visual encoding channel without the overhead of an `if-then-else` chain. For example, the position of the rule in Figure 3.12 is set to the `date` value of the `indexPt` selection.

3.4.3 Disambiguating Composite Selections

Selections are defined within unit specifications, providing a default context. For example, a selection's events are registered on the unit's mark instances, and materializing a selection applies

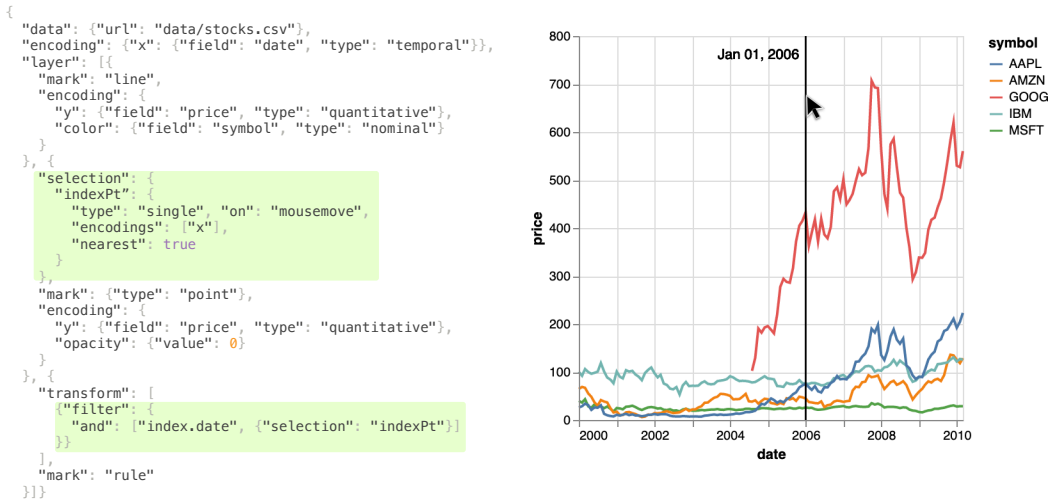

Figure 3.12: A line chart uses a point selection to draw a vertical rule nearest the mouse cursor.

its predicate against the unit's input data by default. When units are composed, however, selection definitions and applications become ambiguous.

Consider Figure 3.10(a), 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? This ambiguity recurs when selections serve as input data or scale extents, and when selections share the same name across a layered or concatenated views.

Several strategies exist for resolving this ambiguity. By default, a *global* selection is created across all views. With our SPLOM example, this setting causes a single 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 (Figure 3.10).

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.

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 Figure 3.10(b, c).

3.5 The Vega-Lite Compiler

The Vega-Lite compiler ingests a JSON specification and outputs a lower-level Reactive Vega specification (also expressed as JSON). There are two main challenges when compiling Vega-Lite to Vega. First, there is no one-to-one correspondence between components of the Vega-Lite and Vega specifications. For instance, the compiler must 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. Second, 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 generates the output Vega specification in four phases: *parse* ingests and disambiguates the Vega-Lite specification; *build* creates the necessary internal representations to map between Vega-Lite and Vega primitives; *merge* optimizes this representation to remove redundancies; and finally, *assemble* compiles this representation into a Vega specification.

3.5.1 Parse

In the first step, the compiler parses a Vega-Lite specification to disambiguate it. It does so primarily by applying rules crafted to produce perceptually effective visualizations. For example, if the color channel is mapped to a 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.

In chapter 4, we will discuss how Draco uses logic programming to assign default values when they are not specified by the user. Draco can assign any property of a specification while Vega-Lite's defaults focus on the most common attributes (i.e., axes, legends, mark properties etc.).

3.5.2 Build

Next, the compiler builds an internal representation of this unambiguous specification, consisting of a tree of *models*. Each model represents a unit or composite view produced by the algebraic operators described in section 3.3, and stores a series of *components*. Components are data structures that loosely correspond to Vega primitives (such as data sources, scales, and marks) and provide a mapping to Vega-Lite primitives. Thus, they allow the compiler to bridge the gulf between the two levels of abstraction. For example, a data component 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. The Vega-Lite compiler uses selection components to encapsulate the information needed to parse selections and construct the required scales, transforms, and signals.

3.5.3 Merge and Optimize

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 predicate function defined in a selection component, while the *nearest* transform augments a mark component with

a Voronoi diagram. This phase also constructs a layout component to calculate suitable spatial dimensions for views. This component emits Vega data sources and transforms to calculate a bottom-up layout at runtime.

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 serializes them using 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. This step also unions scale domains and resolves selection components.

During the merge step, the Vega-Lite compiler also performs optimizations of the dataflow graph created during parsing. The optimizer removes redundant transforms, merges identical dataflows, and reorders dataflow nodes such that the resulting dataflow is more efficient. While moving facetet operators closer towards the source nodes of the dataflow, it creates a copy of the traversed nodes for scale domains that are shared between faceted views.

3.5.4 Assemble

The final phase assembles the requisite Vega specification. Selection components, in particular, 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.

3.6 Example 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 choose examples that cover Yi et al.'s [228] taxonomy of interaction methods. 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 [151, 178, 179]. Vega provides a higher-level visualization specification language on top of the popular D3 [18] library. Where applicable, we also show how construction of our examples can be systematically varied to explore alternate points in the design space.

3.6.1 Select with Clicking and Brushing

Figure 3.7(a) 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 by `if-then-else` logic.

Modifying a single property, *type*, as in Figure 3.7(b), 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 project on *fields* (Figure 3.7(d)) such that marking a single point of interest highlights

all other points that share particular data values—a *connect*-type interaction. Such changes to the specification are not mutually exclusive, and can be composed as shown in Figure 3.7(e).

By using the *interval* type, users can mark items of interest within a continuous region. As shown in Figure 3.8(a), the compiler automatically adds a rectangle mark to depict the selection, and instantiates *translate* to allow it to be repositioned (Figure 3.8(b)). In this context, *project* restricts the interval to a single dimension (Figure 3.8(c)).

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 Figure 3.7(c). 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.

3.6.2 Explore & Encode with Zooming and Panning

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 initialize it using *scales* (Figure 3.9). 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 Figure 3.7(c)) but also necessary. We override the default events that trigger the two interactions using Vega's event selector syntax [179]. As

Figure 3.9 shows, we specify that brushing only occurs when the user drags with the shift key pressed.

By enabling this interaction through composable primitives (rather than a single, specific "pan and zoom" operator [18]), 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 data values considered high and low density. If the selections were defined over the visual *range*, users could instead shift the colors used in a sequential color scale.

3.6.3 Connect with Brushing and Linking

We can wrap our previous example, from Figure 3.9, in a *repeat* operator to construct a scatterplot matrix (SPLOM) as shown in Figure 3.10. With no further modifications, all our previous interactions now work within each cell of the SPLOM and are synchronized across the others. For example, dragging pans not only the particular cell the user is in, but related cells along shared axes. Similarly, dragging with the shift key pressed produces a brush in the current cell, and highlights points across all cells that fall within it.

As its name suggests, the repeat operator creates one instance of the child specification for the given parameters. By default, to provide a consistent experience when moving from a unit to a composite specification, Vega-Lite creates a *point* instance of the selection that is populated and shared between all repeated instances (Figure 3.10(a)). With the *resolve* property, users can specify alternate disambiguation methods including unioning the brushes, or intersecting them (Figure 3.10(b, c) respectively).

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. Mimicking 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.

3.6.4 Abstract & Elaborate with Overview+Detail

Thus far, selections have parameterized scale extents through the initialization step. Previous examples have demonstrated how visualized data can be abstracted/elaborated via zooming. In Figure 3.11, we show 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 top (overview) chart displays only the brushed items at a higher resolution in the larger (detail) chart at the bottom.

3.6.5 Reconfigure with the Index Chart

Figure 3.12, uses a point selection to interactively normalize stock price time series data as the user moves their mouse across the chart. We apply the *nearest* transform, which calculates a Voronoi tessellation to accelerate the selection. By projecting the `date` field, the point selection represents both a single data value as well a set of values that share the selected `date`. Thus, we can reference the point selection directly, to position the black vertical rule, and also materialize it as part of the *lookup* data transform.

3.6.6 Filter with Crossfilter

As selections provide a predicate function, it is trivial to use them to filter a dataset. Figure 3.13, for example, presents a specification to enable filtering across three distinct binned histograms. It uses a *repeat* operator with a single-dimensional interval selection over the bins. The *filter* data transform materializes the selection of the backing datasets as a new dataset used in a separate layer. Only data values that fall within the selection are displayed. As the user brushes in one histogram, bars highlight to visualize the proportion of the overall distribution that falls within the brushed region. 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.

3.7 Discussion

Vega-Lite is not only the first high-level visualization language to offer a multi-view grammar of interactive graphics but also a system that thousands of people use worldwide to create visualizations. Here, we discuss some of these users, how Vega-Lite can facilitate scalable visualization, and its limitations.

3.7.1 Broad Adoption of Vega-Lite

Vega-Lite is used for teaching in a book for practitioners [61], and in classes at Stanford, Carnegie Mellon, Northwestern, the University of Maryland, the University of Washington, and others. Researchers at various universities such as the University of Cambridge, University of Washington, University of Pennsylvania, and Stanford use Vega-Lite as a tool (often via the Altair [211] bindings). Vega-Lite is also used by newspapers such as the Los Angeles Times [6], research labs such as CERN [161], and various companies including Apple, Microsoft Research, and Google.

Figure 3.13: Layered cross filtering interaction of binned histograms by (a) repeating a unit specification with a *point* selection that is materialized to serve as the input data for the second layer. When a user hovers over a bar in one histogram, bars in the others highlight to depict the distributions of the selected bin. By varying the selection *type*, users can (b) select multiple bins on shift-click or (c) brush a continuous interval.

Vega-Lite's declarative format also enables sharing across applications and platforms. Both Vega and Vega-Lite are included as the official plotting formats in the JupyterLab [166] data science environment. In addition, many third-party bindings have been created for programming environments including Python [211], Elm [48], R [126, 212], Scala [214], Julia [213], and Clojure [214]. The feedback, especially from the Python community, has been positive. The developers of Altair, a popular wrapper of Vega-Lite in Python, called Vega-lite and Vega

> perhaps the best existing candidate for a principled *lingua franca* of visualization.
> – The Altair team on their website

Another widely shared review of Python visualization libraries commented that

> it is this type of 1:1:1 mapping between thinking, code, and visualization that is my favourite thing about Altair [and the underlying Vega-Lite].
> – Dan Saber [172]

Vega-Lite has also enabled many other research projects beyond this thesis. Our lab used Vega-Lite to build recommendation systems [224, 225], develop an automatic model to reason about visualization similarity and sequencing [113], and to build a model to reverse engineer visualizations [164]. Researchers at Stanford [56], Georgia Tech, and Princeton use Vega-Lite to build natural language interfaces for data visualization and analysis. Vega-Lite is also popular in literate programming environments [226]. The Vega-Lite formalism is used as an implementation-independent language e.g., for augmented and virtual reality [185].

3.7.2 Vega-Lite for Scalable Visualization

Vega-Lite specifications are compiled to Vega specifications and then parsed by the Vega runtime to generate both static images and interactive web-based views. The Vega runtime performs some transformations of the data, including filtering, binning, and aggregation, limiting the datasets to the constraints of browsers. For large datasets, serialization and transfer from a backend such as a

Jupyter Kernel and is another scalability bottleneck. We aim to run expensive computation should in scalable backend systems such as databases and only transfer the results to the browser.

Declarative Vega-Lite and Vega specifications describe how data should be encoded visually and how interactive visualizations should behave. The specifications leave execution concerns about how the data is processed to the runtime engine. Our declarative design can enable a runtime system that automatically splits computation across a scalable backend and a browser [150]. To run such an interactive visualization, one must implement a custom Vega transform that can run asynchronous queries in backend systems (such as OmniSci [170]) and inject the query results in the Vega dataflow. If the queries are parameterized by our interaction primitives (i.e., selections in Vega-Lite and signals in Vega [179]), then any interactive chart can be made scalable. A major limitation of the naive approach of pushing all computation to a backend system are query and data transfer latencies that can have negative effects on the user experience. We will discuss these issues in chapter 5.

The main takeaway here is that the declarative design of Vega-Lite and Vega gives runtime systems the flexibility to make visualizations scalable that otherwise cannot run entirely in the browser. We have, however, not yet completed an implementation of such a runtime system.

3.7.3 Limitations

The examples demonstrate that Vega-Lite specifications are more concise than those of the lower-level Vega language, and yet are sufficiently expressive to cover a static and interactive visualization taxonomy. Moreover, we have shown how primitives can be systematically enumerated to facilitate exploration of alternative designs. Nonetheless, we identify three 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 an interactive Vega-Lite chart, 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 entire specification within a Vega-Lite *interpreter*.

A similar class of limitations stem from Vega-Lite's focus on rapid authoring and high-level abstractions. The current version of Vega-Lite does not expose the reactive dataflow defined by signals [178]. Some of these barriers will be resolved in future versions of Vega-Lite. For example, Vega-Lite 4 supports custom formatting expressions for axis, legend, and header labels. However, e.g., defining custom signals or accessing signals in transforms is not yet supported.

Similar to limitations for arbitrary interactions, our implementation of static graphics is limited by the combinatorial space of our building blocks, i.e., the mark types and encodings we have implemented. For example, the current version of Vega-Lite does not support pie charts since we have not yet added arc marks. The Vega-Lite community asks us to implement primitives for animations, which we hope to do once Vega supports animations. Especially users with journalism backgrounds commonly request support for lightweight annotations of charts. Other commonly requested features revolve around integration with other JavaScript frameworks. As a user of Vega-Lite, you are dependent on the developers to add support for basic building blocks whereas as a user of D3, you have the power of JavaScript to implement new feature yourself. We share this limitation with other declarative languages, which typically have a lower expressive ceiling than imperative languages.

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 for interactions, such as querying time-series data via relaxed selections [96], cannot be expressed by default. Fortunately, our formulation of selections,

which decouple backing points from selected points via a predicate function, provide a useful abstraction for extending our base semantics with new, custom transforms. For example, the aforementioned technique could be encapsulated in a *relax* transform applicable to multi selections.

While our selection abstraction supports *interactive* linking of marks, our view algebra does not yet provide means of *visually* linking marks across views (e.g., as in the Domino system [78]). 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.

Especially novice users of Vega-Lite often struggle to efficiently debug declarative specifications. The runtime hides execution details that connect input events, program state, the specification, and visual outputs. The Vega-Lite online editor [151] helps users with visual debugging tools [95] and warns users when they commit common errors, but more work is needed to provide an equivalent debugging experience to imperative languages.

Lastly, we designed Vega-Lite for authoring by machines (e.g., in Voyager [224] or Altair [211]) and to be understandable by people. With Vega-Lite, we can describe visualizations. Our formalism is, however, not well suited for reasoning about transformations and equivalences. Reasoning about equivalences as in relational algebra enables sophisticated optimizations in databases [101]. Nonetheless, Vega-Lite can be useful to achieve formal reasoning over databases and visualizations in the future [33]. Any formalism should aim to have at least the expressiveness of Vega-Lite.

3.8 Conclusion

To our knowledge, Vega-Lite is the first high-level visualization language to offer a multi-view grammar of graphics tightly integrated with a grammar of interaction. By offering this grammar, Vega-Lite facilitates rapid exploration of design variations. We hope that it enables analysts to

produce and modify interactive graphics with the same ease with which they currently construct static plots.

Vega-Lite facilitates effective visualization design for interactive statistical graphics with a concise grammar using a rule-based system of smart defaults. With Vega-Lite the runtime systems can make low-level encoding decisions and optimize evaluation. The formalism already serves as the basis of the CompassQL recommendation engine [223] in Voyager [224, 225]. In the next chapter we show that be used for sophisticated reasoning including recommending scalable alternatives to common chart types. The low-level Vega [177] formalism is already used in the high-performance GPU database OmniSciDB [170]; Vega-Lite omits low-level data processing details and has more potential for optimization. We do not believe that Vega-Lite's formalism is mathematically rigorous enough to facilitate cross domain optimizations, but it offers a starting point for a formal algebra that models both data processing and visualization [33].

We end this chapter with a quote the author of the Grammar of Graphics who said about Vega-Lite:

> Having gone through a lot of these GG-inspired systems, I believe yours is the most authentic implementation. I'm using it every day. Thanks for all the great work you've done. – Leland Wilkinson [222]

4 Draco: Formalizing Visualization Design Knowledge as Constraints

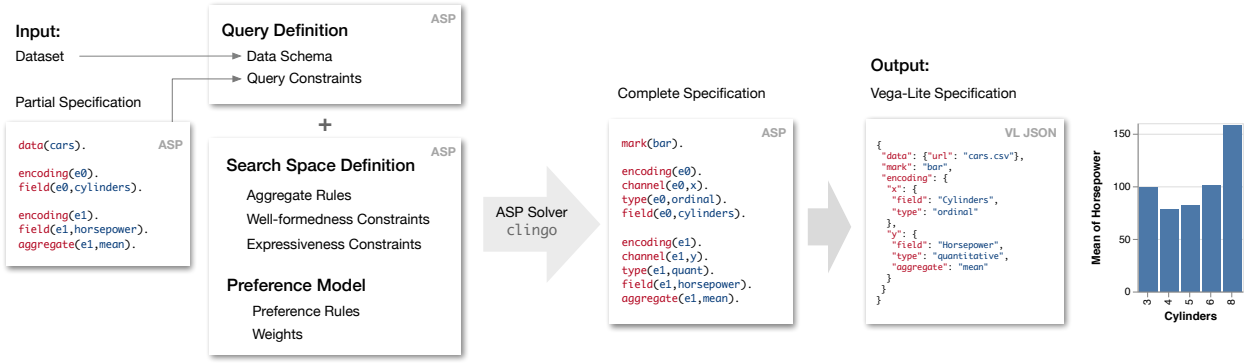

Figure 4.1: Draco compiles a user query to a set of rules and combines them with the existing knowledge base (search space definition + preferences) into an Answer Set Program (ASP). It then calls the Clingo solver to obtain an optimal solution and translates it to Vega-Lite.

To create effective visualizations, designers must consider the data domain and perceptual principles for design. Existing work has attempted to formalize this design knowledge as logical rules. Even though these rules are informed by perceptual research, they are engineered by hand, often only cover a few chart types, and are not reused or automatically validated. Moreover, the primary approaches used to apply these rules are now decades old, based on static models and greedy

optimization methods. Rather than building idiosyncratic representations of design knowledge for individual systems, in this chapter, we seek to make formal models of design knowledge a shared resource that can be extended, tested, and provide the base for future research.

We realize this vision in Draco, a formal model that represents visualizations as sets of logical facts (i.e., specifying choices of dataset, mark type, and visual encoding channels) and expresses best practices and trade-offs among design guidelines as a collection of hard and soft constraints over these facts (e.g., one might prefer bars to start at zero). Draco is implemented as a constraint-based system based on Answer Set Programming (ASP). Draco's visualization description language is based on the Vega-Lite (chapter 3) grammar and extends it to express characteristics about the data (e.g., cardinality, skew) and task (e.g., summary, value). The constraints express preferences validated in perceptual experiments and general visualization design best practices. We demonstrate how to construct increasingly sophisticated automated visualization design systems, including systems based on weights learned directly from the results of graphical perception experiments.

As a visualization tool, Draco automates the tedious and repetitive parts of authoring visualizations. It can automatically synthesize effective designs from partial specifications as queries over the space of visualizations (Figure 4.1). We model the input query as additional constraints and use Draco to systematically enumerate the visualizations that do not violate the hard constraints and find the most preferred visualizations according to the soft constraints. We formalize the problem of finding appropriate encodings as finding optimal completions of partial inputs, which provides well-defined semantics. A constraint solver with efficient domain-independent search algorithms replaces the otherwise necessary custom enumeration and scoring logic.

While Draco can synthesize visualizations in automated design tools, its applications go far beyond. Using constraints, we can take theoretical design knowledge and express it in a concrete, extensible, and testable form. Draco provides a platform for systematic discussions about visualization design. The model formally describes trade-offs among design guidelines. Researchers can now experiment with different trade-offs to improve the science of visualization. They can systematically sample and

enumerate the design space and concretely compare design models. Tool builders can use evolving knowledge bases and benefit from efficient search algorithms provided by modern constraint solvers. Using the implementation-independent language of constraints to model design knowledge could accelerate the transfer of research into practical tools.

We published Draco at IEEE VIS 2018 (co-authored with Chenglong Wang, Greg L. Nelson, Halden Lin, Adam M. Smith, Bill Howe, and Jeffrey Heer) [137]. Draco is available as open source at uwdata.github.io/draco.

4.1 Introduction

Visualization designers benefit from familiarity with both the data domain under consideration and principles of effective visual encoding. Although designers can learn these principles from books, research papers, and experience, they do not always follow these principles in practice [20, 153]. Automated design tools [127, 224] are designed to help address this problem: they use formally-encoded design guidelines to promote effective visualizations. However, our design knowledge is incomplete and continually evolving. In order to incorporate new experimental results or compare different theories of effective design, we need to elaborate and refine these bodies of formal design knowledge.

Visualization researchers regularly publish empirical study results of how people decode and interpret visualizations (e.g., [85, 112, 167, 203]). However, new results often make their way into practical tools slowly: even though our knowledge is evolving, we lack a shared medium for representing and acting upon this knowledge. For example, existing automated design systems [127, 128, 130, 223] do not explicitly reuse the knowledge bases implemented in previous systems. Rather than building idiosyncratic representations of design knowledge for individual systems, we seek to make formal models of design knowledge a shared resource for the visualization community.

We present Draco, a formal model that represents visualizations as sets of logical facts and represents design guidelines as a collection of hard and soft constraints over these facts. Draco can systematically enumerate the visualizations that do not violate the hard constraints and find the most preferred visualizations according to the soft constraints. We first formulate a simple yet powerful visualization description language based on the Vega-Lite grammar [177] and then extend this language to express dataset and task characteristics. To represent design knowledge, we contribute a set of extensible constraints that can encode expressiveness criteria [127], preference rules validated in perception experiments, and general visualization design best practices.

We view the constraints in Draco as the starting point of an evolving knowledge base of design considerations for researchers and tool designers to extend and use. Hard constraints must be satisfied (e.g., shape encodings cannot express quantitative values), whereas soft constraints express a preference (e.g., temporal values should use the x-axis by default). By changing the weights associated with soft constraints, we can trade off the relative importance of these preferences. However, updating these weights presents a challenge, as local changes may have unexpected global effects. To update preferences in a principled way, we also contribute a method to automatically configure weights from experimental data. By formulating this process as a *learning to rank* [121] problem, we can begin to integrate knowledge scattered across various research papers into a single system.

We implement Draco using Answer Set Programming, a domain-independent constraint programming language. We formalize the problem of finding appropriate encodings as the problem of finding optimal answer sets [70], which provides well-defined semantics and can be solved with efficient domain-independent algorithms.

We first evaluate Draco by using it to re-implement the APT [127] and CompassQL [223] automated design tools, demonstrating Draco's expressiveness and improved performance. We then show how Draco can go beyond these systems by adding new constraints concerning data and a user's primary task. Instead of manually specifying weights, we learn them from two independent graphical

perception studies [112, 173]. We compare the learned visualization model to a hand-tuned model, demonstrating improved automated design suggestions. Finally, we describe how to extend Draco to recommend scalable alternatives to common chart types that become perceptually overwhelming with large data volumes.

Encoding design knowledge as constraints has many advantages from both practical and academic perspectives [189]. Tool builders can use evolving knowledge bases of best practices instead of (re)implementing ad-hoc rules, and can benefit from the efficient search algorithms provided by state-of-the-art constraint solvers. Most importantly, an independent knowledge base may allow anyone to formulate and disseminate design preferences as a small set of independent constraints and/or weight updates. Accordingly, we believe Draco can accelerate the transfer of research knowledge into practical tools. Researchers can also use Draco to systematically sample, enumerate, and reason about the design space of possible visualizations, or to concretely compare different design models.

4.2 Related Work

Draco builds on prior work on automated visualization design systems, visualization specification languages, and constraint programming.

4.2.1 Automated Visualization Design

To recommend a visualization, automated design systems enumerate visual encodings that satisfy both user-defined constraints (such as which fields to visualize) and design constraints (Figure 4.2). They then rank candidate visualizations by a utility function. These systems may return the top encodings or perform subsequent clustering to avoid redundancy [223].

Mackinlay's APT system [127] automatically designs graphical representations of relational data using *expressiveness* and *effectiveness* criteria to prune and rank encoding choices. A visualization is considered *expressive* if it conveys all the facts in the data, and only the facts in the data. A

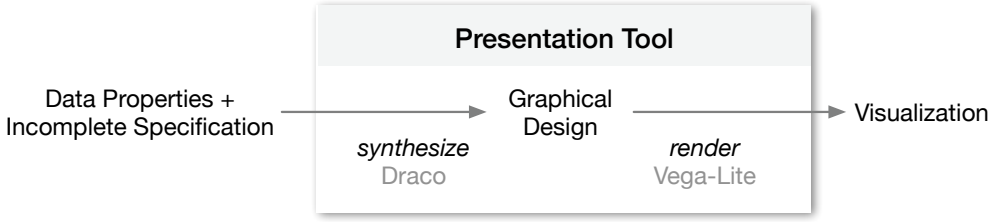

Figure 4.2: A model of automated visualization design tools, inspired by APT. One component synthesizes a design from data and incomplete specifications, and the other renders the design. Draco produces Vega-Lite specifications as output.

visualization is considered most *effective* when the information it conveys is more readily perceived than with other visualizations. APT codifies these design criteria. To ensure that charts are expressive, APT prunes visualizations that violate expressiveness (e.g., using length to encode categorical data). To optimize for effectiveness, APT uses a ranking of the accuracy of visual channels broken down by the data type (quantitative, nominal, or ordinal) and an importance ranking of the fields the designer wishes to analyze. APT assigns the most effective (remaining) encoding channel to the fields in order of decreasing importance. APT demonstrates that visualizations can be automatically designed by machines. With APT, Mackinlay developed a formal model for analyzing visualizations.

The original APT system consists of roughly 200 rules in a declarative logic programming language. The logic programming approach has many advantages over procedural approaches. First, it is flexible and can be adapted with additional constraints. Second, global optimization can find satisfiable solutions that procedural approaches may fail to identify. Draco takes the logic programming approach from APT but extends it in a few ways. The search strategy in APT is depth-first search with simple backtracking, which is inefficient for large design spaces. Draco uses a modern constraint solver and a standardized representation language. APT is built on a graphical language that is no longer used, whereas Draco synthesizes Vega-Lite, a more complete graphical language.

Lastly, Draco can deal with multiple (possibly competing) criteria, which was beyond the scope of APT.

The SAGE project [130] extends APT by considering additional data properties such as cardinality, uniqueness, and functional dependencies. We adopt this idea in Draco. Casner's BOZ [29] additionally models the low-level perceptual tasks of reading and comparing values. In addition to value tasks, Draco's model includes summary tasks involving aggregate properties of visual ensembles [204]. ShowMe [128] uses heuristic rules to suggest encodings from groups of charts, including trellis plots. Draco similarly models faceting of data into trellis plots.

While traditional systems rely on carefully designed rules and defaults, more recent systems like Voyager's CompassQL [223–225] use hand-tuned scores to specify fine-grained criteria such as space efficiency and legibility based on encoding and data properties. Draco is most similar to CompassQL and its weighted preferences. However, CompassQL is implemented in imperative JavaScript. Moreover, like all prior systems presented in this section, it uses similar heuristics and ad-hoc rules. In contrast, Draco's learning approach offers a "programmatic" way to turn experimental results into preference rankings.

4.2.2 Effective Visualization Design

To rank candidate visualizations, automated design tools use models of visual encoding effectiveness. These models encode the insights of Bertin [15], Cleveland & McGill [41], and others that encoding effectiveness varies depending on the visualized data type and related perceptual tasks. For example, APT uses a ranking of encoding channels by data type informed by human-subject studies of visual decoding performance. Other studies (e.g., [88]) confirm and extend such rankings.

However, most work on effectiveness focuses on the performance of reading or comparing individual marks in a visualization. Recent work investigates the effects of reading ensembles of visual elements [204]: for example, how users read aggregates [75], distributions, trends, or correlation [85]. Experimental results from Kim et al. [112] and Saket et al. [173] analyze how

effectiveness varies by task. These results also show that effectiveness varies with respect to data characteristics, such as the cardinality or entropy of data fields. Draco's declarative design can combine classical work on effectiveness using strict preference rules with recent work that considers data and task characteristics in a single system. If activity-oriented [138] or low-level [8] task taxonomies were expressed as constraints, they could be used in automated design systems. To demonstrate this conceptually, we use a simple task classification into *value tasks* for reading and comparing values and *summary tasks* for comparing ensembles [112].

4.2.3 Visualization Specification

Automatic visualization tools synthesize graphical designs, which are abstract descriptions of visualizations (Figure 4.2). For example, the underlying language for APT describes graphical techniques (e.g., color variation and position on axis) to encode information, whereas ShowMe [128] synthesizes encodings using VizQL [84]. Following CompassQL [223], Draco uses a logical representation of the Vega-Lite grammar [177], which we introduced in chapter 3.

In Draco, we focus on single views and faceted views (using the row and column encoding channels). Previous work has focused on similarly restricted design spaces [113, 163, 223, 224]. This subset of Vega-Lite is capable of expressing a variety of plots of both raw and aggregate data, including bar charts, histograms, dot plots, scatter plots, line graphs, and area graphs.

Even though Draco builds on Vega-Lite, extensions of Draco are not restricted to what is expressible in Vega-Lite. We could, for example, add a new mark type or encoding channel to Draco and write constraints over these attributes. Even though we could not render synthesized visualizations with Vega-Lite, we could reason about their design in Draco. With Draco, we can prototype new language features.

4.2.4 Constraint-Based Knowledge Representation

Constraint programming is a declarative programming paradigm with wide applications in scheduling [64], graphical design [16], and natural language specification [165]. A constraint program is a set of constraints defining relations among several unknowns (i.e., variables) that must be or should be satisfied by its solutions. Constraints thus restrict the possible values that variables can take, representing partial information about the variables of interest [10]. Solutions are computed by constraint solvers via inference and search over the constrained space [100]. In visualization, the design recommendation process (i.e., generating, testing [171], subsequent ranking) can be modeled as a constrained combinatorial optimization problem.

The declarative nature of constraint programming allows users to focus on modeling high-level knowledge, while delegating low-level algorithmic details to off-the-shelf constraint solvers. In contrast, imperative implementations of recommendation systems are often hard to implement efficiently and maintain. For example, CompassQL wastes resources in enumerating and evaluating infeasible solutions. Changes in the specification may require a complete overhaul of generators with an imperative implementation [191]. When building on a constraint programming language of sufficient expressiveness (such as beyond-NP reach of ASP technology), complex tests can be folded into the generation part without inventing new algorithms [190].

In Draco, we model visualization knowledge using Answer Set Programming (ASP) [21, 118], a declarative constraint logic programming language that seeks to balance expressivity, efficiency, and ease of use. ASP has been deployed in contexts such as decision support systems [144], product configuration [194], and educational game design [190]. Draco uses Clingo [69, 71], a state-of-the-art answer set solver. Clingo leverages conflict reasoning [72] and heuristics [68] to direct the search process and efficiently solve problems with up to millions of variables. The Clingo guide [67] provides a comprehensive resource of the ASP language features that Draco uses (§3), advice for how to model problems (§1 and §6), and documentation of the constraint solver (§7).

4.2.5 Learning Preferences

Both CompassQL and Draco use weights of visualization features to trade-off competing effectiveness criteria. Although the weights can be defined by visualization experts, tuning these weights involves ad-hoc choices that are difficult to maintain and extend, especially when the visualization model is complex.

Instead, the preference model might be learned from data. For example, VizDeck [111] learns a linear scoring function that uses data statistics and chart type to predict users' up/down votes on presented charts. However, VizDeck's model features only capture direct correlations between data statistics and chart type, and its learning algorithm is limited to a small set of predefined visualizations.

To improve expressivity and extensibility, Draco leverages domain knowledge from experts (in the form of soft constraints) as visualization features. Draco's preference model forms a Markov logic network (MLN) [169], where the weights corresponding to soft constraints are learnable parameters reflecting preference levels. Draco can capture rich attribute relations using few expert-defined rules due to the expressive and compact nature of MLNs [165, 187].

To train a recommendation model, a common practice is to use pairs of incomplete specifications and their corresponding optimal completion [186]. However, for visualization, such training datasets are not generally available and are hard to obtain because there typically does not exist a single optimal completion. Instead, with Draco we take a *learning to rank* [121] approach, where the preference model is learned from ordered pairs of visualizations (i.e., complete specifications). A dataset of ranked pairs can either be harvested from experimental data based on human-subject performance measures or solicited from experts. To learn a preference model, we use RankSVM [94] over the structural features defined by Draco's soft constraints.

4.3 Background: Answer Set Programming

The building blocks of ASP programs are *atoms*, *literals*, and *rules*. Atoms are elementary propositions that may be true or false. Literals are atoms A or their negation *not* A. Rules are expressions of the form A `:-` $L_1,\ldots,L_n$. where each L_i is a literal. The atom A of a rule (also called its head) is derivable (true) if all literals in the body (right of `:-`) $L_1,\ldots,L_n$ are true. A positive literal is true if it has a derivation, and a negative literal *not* B is true if the atom B does not have a derivation. For example, the rule `light_on :- power_on, not broken.` informally states that the light is on if we can derive that the power is on and there is no reason to say that the lamp is broken [21].

Rules can be bodiless or headless. A bodiless rule, such as `power_on :-.`, simply asserts the fact that its head is true. A fact can also be stated using only its head, e.g., `power_on.`. Headless rules of the form `:-` $L_0,\ldots,L_n$ are *integrity constraints* that derive false from their body. Thus, satisfying the body $L_0,\ldots,L_n$ results in a contradiction.

An ASP program consists of a set of rules. Note that `not` in an ASP program means "not derivable". For example, given the two rules described above, `power_on` can be derived from a fact (our bodiless rule), while `broken` cannot. Consequently, we can derive `light_on`. Such derivations are formally defined as *stable models*[73] or *answer sets*: sets of atoms that are consistent with the constraints, justified by a derivation, and minimal with respect to unknown facts. Answer Set Programming has a constructive flavor: negative literals need only be true, whereas positive ones must also be provable.

On top of the stable model formalism, the language of ASP [66] introduces powerful modeling constructs. *Aggregate* rules of the form l $\{A_0,\ldots,A_n\}$ k are read as: at least l and at most k atoms in the set $\{A_0,\ldots,A_n\}$ are true. Aggregates can appear in the head or body of a rule and aggregate rules can be used to define a design space. We can eliminate answer sets with integrity constraints (headless rules) to restrict the design space. *Soft constraints* are headless rules with an associated weight and are written as `:~` $L_0,\ldots,L_n$. $[w]$. Unlike hard constraints, soft constraints may be

violated by a solution, but each violation of a soft constraint imposes a penalty (or cost) equal to its weight w. Soft constraints can express preferences in the search. In a program with soft constraints, an answer set is optimal if it minimizes the sum of weighted costs of all violated soft constraints. Although one can express richer forms of optimization in ASP (such as Pareto optimality for combining preferences without first establishing a fixed trade-off between them) [70], the weighting scheme for soft constraints is sufficient for the linear preference models we will learn from data.

4.4 Modeling Visualization Design in Draco

In this section, we first describe how we model visualizations as sets of facts. We then explain the design space of our model and how we can query the model with constraints. Modern constraint solvers efficiently search for *optimal* visualization specifications within a defined space. Imperative systems, which use an exhaustive generate-and-test method, couple knowledge representation and the algorithm for finding effective designs. In Draco, solutions to the base problem of finding optimal designs should be found via automated search, whereas solutions to the higher level problem of what preferences should be used and how competing preferences are resolved should be determined by designers (via refinement of the design space and preference definition).

The term "optimal" here does not refer to the definitively best or "correct" visual design, as this would make two arrogant presumptions. First, the system would have to fully understand the user's intentions and their ability to read visualizations—an unlikely proposition. Visualization is always an abstraction where choices are made about what to emphasize. Second, visual analysis is an iterative process, involving any number of visualizations, not just a singular view. By "optimal", we refer to an optimal specification according to a set of formally-defined preferences: the system can find no other visualization that would be scored as preferred to this one. A user-facing application can show more than just an optimal visualization, and a user may select between

multiple recommendations or refine their query until they have the right design(s) for their task [223, 224].

4.4.1 Mapping Visualization Specifications to Logical Facts

A Vega-Lite specification describes a single Cartesian plot with a backing dataset, a given mark type, and a set of one or more encoding definitions for the visual channels. Figure 4.3 shows a Vega-Lite specification for a bar chart. Vega-Lite expects a relational table of records with named fields. The mark type specifies the geometric objects used to visually encode data records. Possible values include bar, point, area, line, and tick. The encodings determine how data fields map to visual properties of the marks. An encoding uses a visual channel such as spatial position (x, y), color, size, shape, or text. A detail channel can be used to add group-by fields in aggregate plots. An encoding includes the data field to visualize and its given data type (nominal, ordinal, quantitative, temporal). The data can also be transformed via binning, aggregation (sum, average, etc.), and sorting. An encoding may specify scales that define how the data domain maps to the visual range or guides that visualize scales (axis and legend). Examples include whether a scale domain should include zero or whether the scale is linear or logarithmic. If omitted, the Vega-Lite compiler will infer defaults based on the channel and data type.

We represent the Vega-Lite specification, user task, and data schema as a set of atoms. To enable reasoning over atoms, we encode them as *predicates* (i.e., relations or functions). A predicate defines what value is assigned to an attribute of a visualization specification.

To set the **mark** type, we use a predicate `mark/1`.[1] For example, `mark(bar).` defines that the visualization should use the bar mark (i.e., assign the value `bar` to the attribute `mark`).

To define an **encoding**, we use `encoding/1` to establish that it exists. For example, `encoding(e).` declares the encoding `e`. We then use binary predicates to define properties of the encoding.

[1] Per Prolog traditions, predicates are identified by their symbolic name and the number of arguments they take (signified with `/n`).

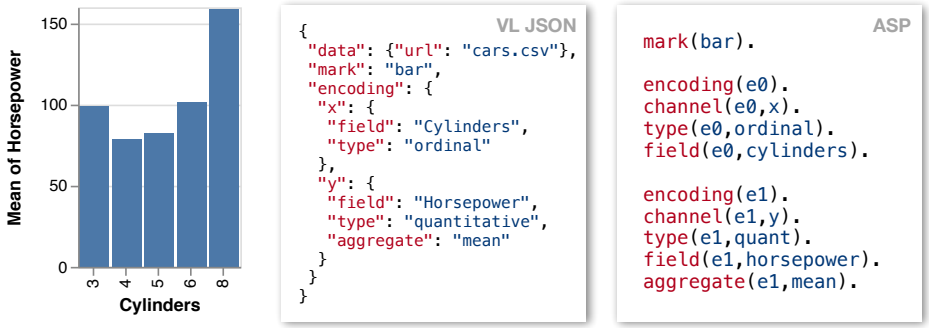

Figure 4.3: An example of a bar chart, its specification (in Vega-Lite JSON), and its equivalent specification using Draco constraints (in ASP). The specification defines the marktype and encodings, which includes a specification of the fields, data type, and data transformations.

`channel(e,x).` defines that the encoding `e` uses the `x` encoding channel. The field being visualized is defined with `field/2`. In addition, we use `aggregate/2` to define an aggregation function, `bin/2` to discretize continuous data, `stack/1` to define whether marks should be stacked, and `zero/1` and `log/1` to customize scales. Compared to Vega-Lite's JSON syntax, we un-nest scale properties to simplify the logical encoding.

The **data schema** is defined as the size of the data `num_rows/1` (e.g., `num_rows(42).`) and facts about data fields. We use `fieldtype/2` to specify the data type (string, number, date, ...) and `cardinality/2` to define how many distinct entries there are, `entropy/2` to define the entropy, and `extent/3` to set the minimum and maximum values. We see this set of data attributes as a starting point; future extensions of Draco can define other features relevant to automated design.

We currently model a user's **primary task** as a single function `task/1`. Following Kim et al. [112], we distinguish between *value* and *summary* tasks. Since tasks regard specific fields (e.g., "What is the maximum acceleration across cars?"), fields can be marked as relevant to the task with `interesting/2`.

We designed this logical visualization language to be extensible and enable reasoning. For example, we could have defined predicates for each channel such as `field_x`, `aggregate_x`. This design would automatically ensure that each channel is only used once; however, it would limit the expressiveness of the language (e.g., detail can in fact be used multiple times) and would make it more difficult to define general constraints over attributes that are not tied to a specific channel. One way of extending Draco is to define new attributes as predicates. For example, to add a data property that measures data skew, we can add `kurtosis/2` where the first argument is the field and the second is the kurtosis measure.

4.4.2 Representing Design Knowledge as Constraints

The goal of a visualization model is to distinguish desirable visualizations from undesirable ones. In Draco, our visualization model consists of two parts: the space of all visualizations considered valid, and an evaluation function over the space to measure preferences. Figure 4.4 illustrates the design space. We describe a visualization model in Draco as a declarative answer set program.

4.4.2.1 Design Space Definition

The design space of a visualization model is defined by a set of constraints. A visualization is considered valid only if all constraints are satisfied. Following best practices in logic program design [117], we define the space of possible visualization specifications with two sets of rules: (1) a set of aggregate rules that specifies the domains of the attributes defined in the previous section (mark, encoding, ...) and (2) a set of integrity constraints that defines how different attributes can interact with each other.

We use aggregate rules to define which values can be assigned to a visualization attribute. For example, the rule `1 { mark(bar); mark(line); mark(area); mark(point) } 1.` restricts choices of mark to one of bar, line, area or point. Similarly, we can use the constraint `0 { aggregate(E,count), aggregate(E,mean), aggregate(E,median), aggregate(E,sum) } 1 :- encoding(E).` to declare that each encoding may have up to one aggregate of count, mean, median,

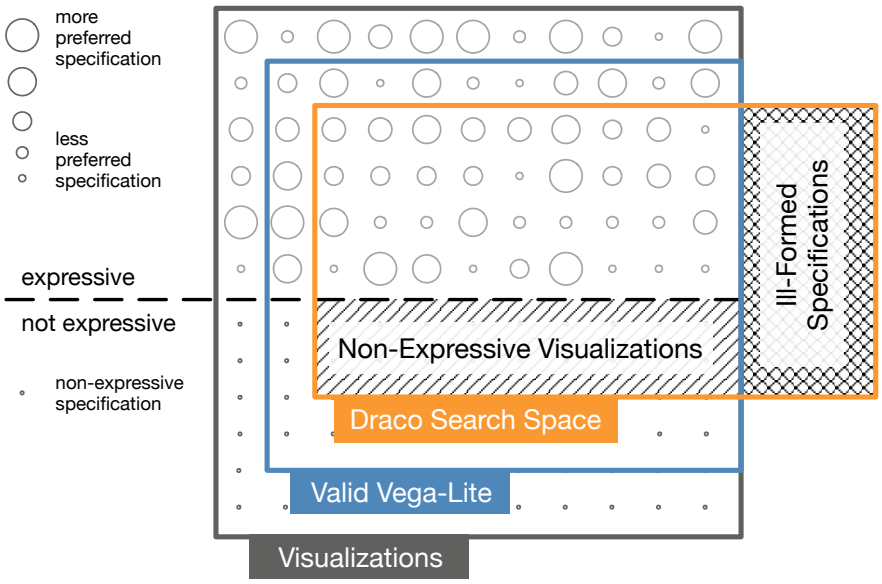

Figure 4.4: Illustration of the design space in Draco. The set of valid Vega-Lite specifications is a subset of all possible visualizations, and Draco's design space overlaps with that subset. Given a preference model, expressive visualizations are ranked according to their preference scores in the model. Draco eliminates ill-formed or non-expressive specifications using hard constraints and encodes preferences using soft constraints.

or sum. Note that `E` is capitalized, which identifies it as a variable. To extend the domain of an attribute (e.g., support tick marks), we would add facts to existing aggregate rules (e.g., by adding `mark(tick)`). The complete list of aggregate rules, and all other constraints in this section, are included as supplemental material.

The aggregate rules declare the domain of attributes, but not the validity of interactions of different attribute values. To capture such interactions, we introduce an additional set of constraints. Using ASP notation, we write these constraints as a headless rule (integrity constraint). `:- X.` is read as "it cannot be the case that X". Integrity constraints can be used to encode expressiveness and restrictions to the attributes of a visualization specification, for example, to match the implementation of Vega-Lite.

First, we use constraints to rule out specifications that do not specify a valid visualization (i.e., that are ill-formed or ungrammatical). We call these constraints *well-formedness* constraints. For example, the constraint `:- channel(_,shape), not mark(point).` is an integrity constraint stating that it cannot be the case that there exists a shape encoding unless the mark that is used to encode data is "point", as other mark types such as area, line, bar, or text cannot encode a shape. Another example is `:- log(E), zero(E).`, which ensures that we do not synthesize a log scale that requires zero in its domain. We also assert that visualizations must use a text mark when the text channel is used (and vice versa) and that only discrete data can be mapped to facet (row and column) channels. Well-formedness depends on the syntax and semantics of the graphical language. We can use constraints to generate only visualization specifications that are supported by a concrete visualization model such as Vega-Lite. For example, Vega-Lite only implements 8 shape types; we can use the integrity constraint `:- channel(E,shape), cardinality(E,C), C > 8.` to model this restriction. When defining the design space, conflicting constraints must be avoided, as they result in an empty space that cannot be satisfied by *any* visualization.

Second, we use constraints to eliminate *non-expressive* visualizations that do not convey all and only the facts in the data. For example, `:- mark(bar), channel(E,y), continuous(E), not`

`zero(E).` ensures that the model will not consider vertical bar charts that do not use zero as a baseline. (We actually implement this as a more general rule `:- mark(bar), channel(E,(x;y)), continuous(E), not zero(E).`, which also covers horizontal bar charts. Here the semicolon denotes an expansion into disjunctions, implying one constraint for each channel type.) Another expressiveness constraint is `:- channel(E,size), type(E,nominal).`, which ensures that size is not used to encode nominal data, as size implies an order. We also assert that the size channel is only permitted for compatible marks, that zero baselines are used for area and bar charts, and that bar charts with a color channel encoding use stacking so that bars do not occlude each other.

4.4.2.2 Preference Over the Design Space

We now define an evaluation function over the visualization design space to encode preferences. The (linear) evaluation function maps a valid visualization specification into an integer representing its preference level. This function defines a total ordering over the design space, as illustrated in Figure 4.4. Instead of defining the evaluation function as a procedure, we use a set of soft constraints to implicitly define it. The weight of a soft constraint reflects its strength: the higher the weight (penalty), the higher cost that violating the constraint imposes on the cost of an answer set.

Soft constraints are similar to integrity constraints, but start with `:~` instead of `:-` and include a weight declared in square brackets. They can be read as "prefer not to . . . ". As an example, the soft constraint `:~ continuous(E), not zero(E). [5]` states that the model prefers to include zero for continuous fields and that violating the rule increases the cost of the visualization by 5. A soft constraint is appropriate: though omitting a zero baseline for ratio data can mislead [153], it is still sometimes reasonable to do. Note that a visualization may violate a soft constraint multiple times. For example, given a visualization with two encodings, the soft constraint above may be violated twice if two of its encodings use continuous fields but omit zero.

In order to extend Draco to support new visualization types or data properties, a visualization expert can add soft constraints to capture intended preferences. For example, in order to extend a

visualization model to handle the relation between the new data property kurtosis (as discussed in the previous section) and using a log scale, we add the soft constraint `:~ kurtosis(F,K), field(E,F), log(E), K > 42. [w].`

The set of soft constraints defines a cost model for visualizations in the design space that we can use to evaluate preferences. The cost of a visualization is the sum of the costs of all soft constraint violations multiplied by their count of violations. Concretely, given a set of soft constraints $S = \{(p_1, w_1), \ldots, (p_m, w_m)\}$, the cost of a visualization specification v is calculated as follows, where n_{p_i} is the number of violations of the soft constraint p_i by v:

$$Cost(v) = \sum_{i=1\ldots k} w_i \cdot n_{p_i}(v)$$

Given a visualization v, the vector $\mathbf{x} = [n_{p_1}(v), \ldots, n_{p_k}(v)]$ fully determines the cost of v in the given visualization model, and we refer to $\mathbf{x}$ as the *feature vector* of v. Using the feature vector x, the cost of v can be represented as $Cost(v) = \mathbf{x}^T\mathbf{w}$, where $\mathbf{w} = [w_1, \ldots, w_k]$ is the vector consisting of all soft constraint weights. Note that setting the weight w for this new rule requires the expert to know the existing constraints and carefully trade-off among competing preferences. In section 4.5, we instead present a method to learn w from data.

4.4.3 Preference Models as Markov Logic Networks

Draco's preference model forms a Markov logic network (MLN), a graphical model that integrates logic with statistical reasoning to handle uncertainty robustly [169]. This interpretation as an MLN offers theoretical insight into the expressiveness of our model. The soft constraints are structural features of visualizations that capture hidden relations among visualization attributes, and their weights are learnable parameters reflecting their importance. These weights reflect the difference in log probability between a visualization satisfying the constraint and one that does not. The joint distribution modeled by a MLN is:

$$P(v) = \frac{e^{-Cost(v)}}{\sum_{u \in \mathcal{V}} e^{-Cost(u)}} = \frac{e^{-\sum_{i=1}^{k} w_i n_{p_i}(v)}}{\sum_{u \in \mathcal{V}} e^{-\sum_{i=1}^{k} w_i n_{p_i}(u)}}$$

The probability $P(v)$ of a visualization in the distribution is its exponentiated cost normalized by the exponentiated costs of all visualizations in the design space $\mathcal{V}$, using a softmax function. Note that the partition function $Z = \sum_{u \in \mathcal{V}} e^{-\sum_{i=1}^{k} w_i n_{p_i}(u)}$ is a fixed term in a given visualization model, and the difference of costs of two visualizations v_1, v_2 results in the log difference of their probability in the model.

The problem of finding the optimal completion of a partial specification is the same as performing maximum a posteriori (MAP) inference in the probability model [186]. Given a partial specification Y, its optimal completion X maximizes the posterior probability of $P(x \mid y = Y)$ (i.e., $X = \max_x P(x \mid y = Y)$). The ASP solver solves this inference problem efficiently by minimizing the overall cost of the generated visualization; it is not necessary to compute the partition function.

Since a MLN is a linear model over structural features rather than linear model directly over attributes, it has the advantage of capturing structural relations between attributes that cannot be captured otherwise. For example, a channel ranking that is independent of the data type would have to prefer color and size independent of the data type. Moreover, as opposed to explicitly representing correlations between every attribute and all other attributes, a MLN is more compact and interpretable.

4.4.4 Completing Specifications by Solving Constraints

Given a visualization model, a user can query said model for optimal completions of a partially specified visualization. Figure 4.6 illustrates the search process, while Figure 4.5 summarizes our implementation of it.

4.4.4.1 Partial Specification

A user's query is a partial specification that describes their incomplete intent for a desired visualization. Our partial specification language allows the user to specify unknown visualization attributes by leaving them blank. Draco also supports converting CompassQL queries and Vega-Lite

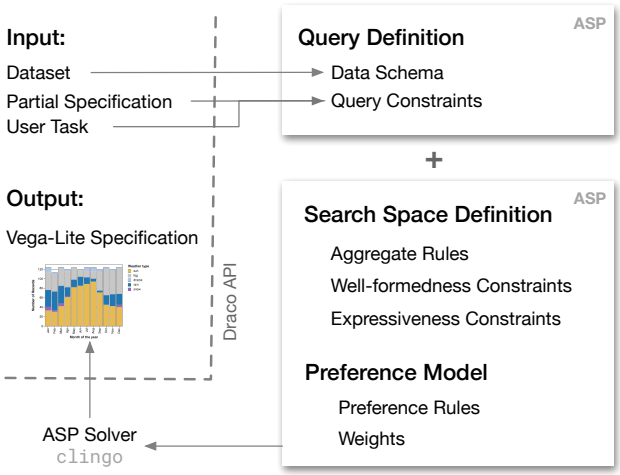

Figure 4.5: Our implementation of the optimal encoding search process using constraints. Draco compiles a user query (including the dataset, the partial specification, and the task) into a set of rules and combines them with the existing knowledge base to form an ASP program. Draco then calls Clingo to solve the program to obtain the optimal answer set. Finally, Draco translates the answer set into a Vega-Lite specification.

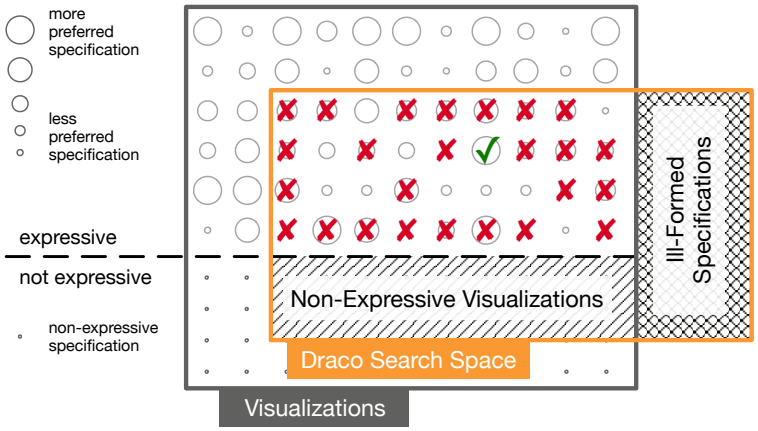

Figure 4.6: To find the optimal completion of a partial specification, Draco compiles a user query into a set of constraints that removes all candidates (✗) that do not match the query from the design space illustrated in Figure 4.4. Draco selects the optimal visualization (✓) within the remaining space.

```
{
  "data": { "url": "cars.csv" },
  "encoding": [ {"channel": "x", "bin": true,
                 "field": "horsepower"} ]
}
```

Figure 4.7: An example query over the cars dataset, in the form of a partial (incomplete) Vega-Lite specification.

specifications into queries. In addition to a partial specification, a query can specify the schema of the dataset and the user's primary task.

As an example, Figure 4.7 shows a query over the classic *cars* dataset [4]. In this query, the user specifies that "I want a visualization that shows binned horsepower along the x-axis". Using this query, Draco must then determine completions of the mark and other attributes of the specified encoding, as well as whether other encodings are necessary. Draco then searches to find the mark for the chart, completes the specified encoding, and potentially adds additional encodings (including which channels to use, whether to use aggregation, etc.).

4.4.4.2 Queries as Constraints over the Design Space

To answer a query, Draco first compiles the query into a set of facts and constraints that defines a subspace of visualizations and then searches over the subspace for the lowest-cost specifications (Figure 4.6). Concretely, the subspace is defined by (1) a set of facts describing the dataset specified in the query (subsection 4.4.1) and (2) a set of constraints that restrict the available choices for visualization attributes.

For example, the query in Figure 4.7 is compiled into a set of facts and constraints. Unless the data schema is provided explicitly, Draco infers a schema (including fields, their types, and data properties) from the provided dataset ("cars.csv") and uses it to generate facts that describe the dataset's size and fields:

```
num_rows(407).
fieldtype(name,string).
cardinality(name,311).
fieldtype(miles_per_gallon,number).
cardinality(miles_per_gallon,130).
...
```

Based on the partial specification, Draco then generates a fact declaring an encoding `e1` and associated constraints. These constraints restrict the design space to specifications with an encoding `e1` that uses the `x` encoding channel for binned values from the horsepower `field`:

```
encoding(e1).
:- not channel(e1,x).
:- not field(e1,horsepower).
:- not bin(e1,_).
```

To find optimal specifications within the subspace, Draco sends data constraints, query constraints, and constraints from the knowledge base to the Clingo solver (Figure 4.5). For example, Draco suggests the following optimal completion of the query above, which adds a new encoding `e2` on the `y`-axis for a `count` aggregate.

```
encoding(e2).
channel(e2,y).
aggregate(e2,count).
```

Finally, Draco converts the optimal solutions to Vega-Lite specifications and returns them to the user.

4.5 Learning Preference Models

Although it is possible for model designers to tune preference weights for small models, tuning weights for complex models is challenging: it requires the visualization expert to reason globally about competing conditions among different preferences. In this section, we describe a learning

algorithm that allows the model to learn soft constraint weights $\mathbf{w} = [w_1, ..., w_k]$ from ranked pairs of visualizations.

We learn weights using a RankSVM (Rank Support Vector Machine) model [94] trained on labeled visualization pairs. Given a visualization pair (v_1, v_2), the cost model should determine whether v_1 is preferred to v_2 based on $\text{sign}(Cost(v_1) - Cost(v_2))$. This model can be learned from a dataset where each entry $(v_1, v_2; y)$ is a visualization pair associated with a label y indicating if v_1 is preferred to v_2 ($y = -1$) or vice versa ($y = 1$). Given a visualization model with a set of soft constraints $S = \{p_1, \ldots, p_k\}$, we show how we train the cost model (i.e., training weights $\mathbf{w} = [w_1, ..., w_k]$ for S) using a dataset $\mathscr{D} = \{(v_{11}, v_{12}; y_1), \ldots, (v_{n1}, v_{n2}; y_n)\}$.

As shown in subsubsection 4.4.2.2, the cost of a visualization v is determined by its feature vector $\mathbf{x} = [n_{p_1}(v), \ldots, n_{p_k}(v)]$. Accordingly, we first run Clingo on the complete specifications and count how often each soft constraint is violated to vectorize all visualizations in the dataset $\mathscr{D}$ and obtain their vector representation: $\mathscr{D}' = \{(\mathbf{x}_{11}, \mathbf{x}_{12}; y_1), \ldots, (\mathbf{x}_{n1}, \mathbf{x}_{n2}; y_n)\}$.

The cost model is a linear model over soft constraint weights. Given a pair (v_1, v_2) with feature vectors $\mathbf{x}_1, \mathbf{x}_2$, its class is determined by the sign of the following function:

$$f(v_1, v_2) = Cost(v_1) - Cost(v_2) = \mathbf{w}^T(\mathbf{x}_1 - \mathbf{x}_2)$$

Using the RankSVM algorithm to train weights $\mathbf{w}$, we perform linear regression (with L_2 regularization) over the dataset $\mathscr{D}'$ by minimizing the hinge loss. The loss function L is defined as follows, and it is minimized by the solution $\mathbf{w}^*$.

$$L = \frac{1}{n} \sum_{i=1}^{k} \max\left(0, 1 - y_i \mathbf{w}^T(\mathbf{x}_{i1} - \mathbf{x}_{i2})\right) + \lambda \|w\|_2$$

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} L$$

As the cost model is a linear model over inputs $(\mathbf{x}_{i1} - \mathbf{x}_{i2})$, the weights $\mathbf{w}^*$ can be efficiently found using an off-the-shelf linear optimizer. By minimizing the loss function L, we obtain a cost

model with weights $\mathbf{w}^*$ that is most consistent with the rankings of visualization pairs in the dataset. The order of v_1, v_2 in a visualization pair from the training data does not matter, as the classification problem is symmetric with respect to the origin ($-y_i \mathbf{w}^T(\mathbf{x}_{i1} - \mathbf{x}_{i2}) = y_i \mathbf{w}^T(\mathbf{x}_{i2} - \mathbf{x}_{i1})$ in the loss function). Thus, a pair $(v_1, v_2; y)$ is equivalent to $(v_2, v_1; -y)$ in the training set, and we can standardize all pairs in the form $(v_1, v_2; -1)$ (such that v_1 is preferred over v_2) without worrying about an imbalance between classes. For our initial experiments, we set the regularization parameter λ to 0.1.

By integrating the learned weights $\mathbf{w}^*$, the visualization model becomes a knowledge base for visualization recommendation that integrates both expert knowledge and empirical data.

4.6 Demonstration of Draco

We present three applications of Draco to demonstrate its expressivity, extensibility, and usability. First, we implement APT's preference rules via a set of strict preference constraints (Draco-APT); this shows Draco can express a classic yet useful automated design system. Next, we reimplement CompassQL by adding soft constraints with weights hand-tuned by experts to match the semantics of CompassQL (Draco-CQL). Finally, we introduce additional effectiveness criteria learned from data from two different studies (Draco-Learn); this shows how Draco can partially automate combining effectiveness results from different research studies.

4.6.1 Reimplementing APT: Draco-APT

Draco-APT provides a re-implementation of APT's channel preferences as a set of soft constraints. APT uses a principle of importance ordering: each field is assigned to the most effective channel (for the corresponding data type) in order of decreasing user-specified importance.

Draco-APT starts with the set of well-formedness and expressiveness constraints from subsection 4.4.2. We add a set of soft constraints to express channel preferences. Each preference constraint is of the form `:~ type(E,T), channel(E,C), priority(E,P). [w@P,E]`, which states

that for any encoding E, using channel C for a field of type T incurs a cost of w at priority level P equivalent to the priority of the field. To determine the optimal solution, the solver first satisfies all hard constraints followed by soft constraints, ordered by priority level.

```
:~ type(E,quant), channel(E,x), priority(E,P). [1@P,E]
:~ type(E,quant), channel(E,y), priority(E,P). [1@P,E]
:~ type(E,quant), channel(E,size), priority(E,P). [2@P,E]
:~ type(E,quant), channel(E,color), priority(E,P). [3@P,E]

:~ type(E,ordinal), channel(E,x), priority(E,P). [1@P,E]
:~ type(E,ordinal), channel(E,y), priority(E,P). [1@P,E]
:~ type(E,ordinal), channel(E,color), priority(E,P). [2@P,E]
:~ type(E,ordinal), channel(E,size), priority(E,P). [3@P,E]

:~ type(E,nominal), channel(E,x), priority(E,P). [1@P,E]
:~ type(E,nominal), channel(E,y), priority(E,P). [1@P,E]
:~ type(E,nominal), channel(E,color), priority(E,P). [2@P,E]
:~ type(E,nominal), channel(E,shape), priority(E,P). [3@P,E]
:~ type(E,nominal), channel(E,size), priority(E,P). [4@P,E]
```

Using Draco-APT, we can find optimal completions of partial specifications using APT's effectiveness criteria. For example, given a query with four fields with decreasing priority—two quantitative fields (encoded as e_q1 and e_q2), one nominal field (e_n), and one ordinal field (e_o)—Draco-APT synthesizes the following two optimal results.

```
1 channel(e_q1,y) channel(e_q2,x) channel(e_n,color) channel(e_o,size)
2 channel(e_q1,x) channel(e_q2,y) channel(e_n,color) channel(e_o,size)
```

4.6.2 Reimplementing CompassQL: Draco-CQL

We now show that Draco is expressive enough to re-implement CompassQL [223], a state-of-the-art automated visualization design system that includes additional forms of effectiveness knowledge. We compare the original CompassQL implementation with our new implementation, showing Draco-CQL is more concise, extensible, and provides superior performance when searching for optimal visualizations.

CompassQL uses a generate-and-test approach [171]: for a given partial specification, CompassQL generates all matching full specifications. The changes made to the partial specification may

include both data query parameters (e.g., fields, aggregation) and encoding parameters (e.g., channels). It then uses data query, encoding, and expressiveness constraints similar to those described in subsection 4.4.2 to eliminate invalid encodings. The system then assigns each valid candidate an effectiveness score. The scoring function incorporates preferences for type-channel interactions (e.g., it is preferred to encode nominal fields using x or y before using row or column) and mark-type interactions (e.g., point marks are preferred over tick marks for quantitative by quantitative plots). These effectiveness scores are then used to rank and recommend visualizations. Critically, this approach allows CompassQL to trade off among competing preferences.

We identified two places for improvement in the process taken by CompassQL. First, CompassQL generates and tests all possible candidate visualizations, which leads to an ineffective exhaustive search. By expressing the hard constraints as integrity constraints, we can pass this process off to a modern constraint solver. Moreover, ASP allows for concise representations. For example, the constraint that invalidates encodings that reuse channels that should only be used once, requires 14 lines of JavaScript code in CompassQL but can be expressed in Draco as `:- single_channel(C), 2 { channel(_,C) }.`, stating that we prefer not to use a single channel 2 or more times.

Second, CompassQL's scoring function can be expressed naturally in Draco with soft constraints. For example, CompassQL penalizes aggregation when grouping by a continuous field, implemented in 12 lines of code Figure 4.8. Draco's implementation is more concise and readable:

```
:~ aggregate(_,_), continuous(E), not aggregate(E,_). [3].
```

Reimplementing CompassQL in Draco requires authoring soft constraints that express CompassQL's imperative rankings and design preferences. These constraints include channel, mark type, and aggregation function rankings as well as soft constraints to prefer compact layouts (e.g., fewer encodings) or promote best practices such as placing time on the horizontal axis. For example, channel preferences for nominal fields can be expressed as follows, where lower weights (penalties) indicate higher preference:

```
:~ channel(E,y), type(E,nominal). [0]
:~ channel(E,x), type(E,nominal). [1]
```

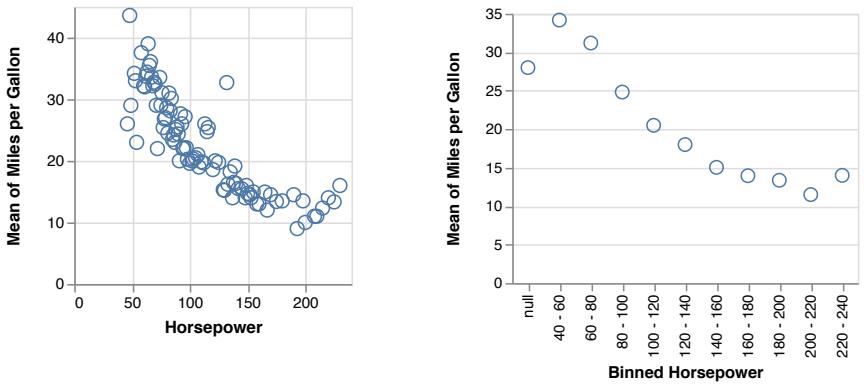

Figure 4.8: A comparison of generated aggregate charts to demonstrate design choices. The chart to the left aggregates over a continuous grouping field, which is unlikely to be effective. The more sensible chart to the right instead groups according to a discretized (binned) version of the same field. Both CompassQL and Draco prefer the right design.

```
:~ channel(E,row), type(E,nominal). [6]
:~ channel(E,column), type(E,nominal). [7]
:~ channel(E,color), type(E,nominal). [7]
:~ channel(E,shape), type(E,nominal). [8]
:~ channel(E,text), type(E,nominal). [9]
:~ channel(E,detail), type(E,nominal). [20]
```

Using a full set of these constraints, Draco-CQL synthesizes identical optimal specifications as CompassQL for all 17 partial specifications included in CompassQL's test suite. It does so while reducing specification complexity. Draco-CQL is implemented as 70 hard and 110 soft constraints. In contrast, CompassQL is implemented in 4,324 lines of imperative code, with 1,134 of those lines devoted to hard constraints and 786 devoted to scoring logic.

Draco-CQL also exhibits better performance, especially for highly unconstrained problems. Figure 4.9 shows the results of a benchmark study comparing CompassQL and Draco-CQL across varied numbers of input dataset fields and output encoding channels. All measurements were taken on CentOS Linux 7 with an Intel Xeon CPU E5-2690 v3 with 2.60GHz; CompassQL used Node v9.9.0. Other than a constant startup overhead, Draco exhibits superior scalability. On a real dataset with

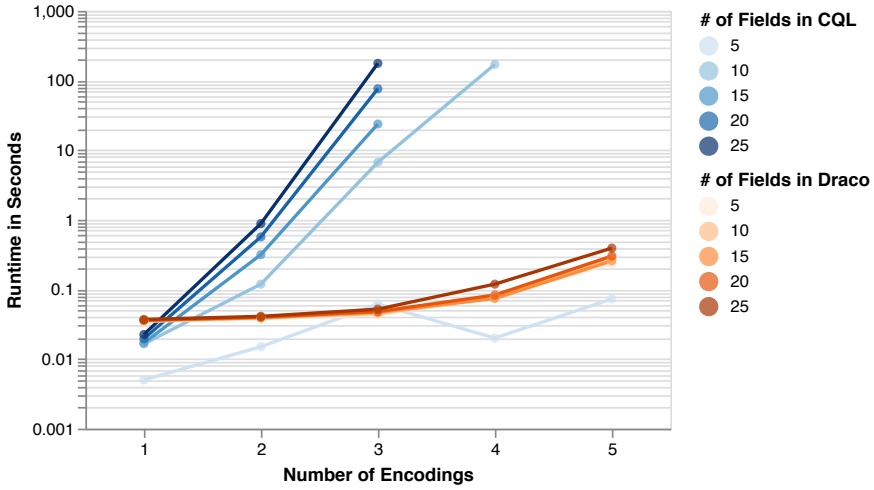

Figure 4.9: Median runtime for CompassQL (blue) and Draco (orange) across different numbers of data fields and encodings. CompassQL performance rapidly degrades with additional encodings and runs out of heap memory (set to 4GB) for most queries with four or more encodings because it exhaustively searches all combinations of fields in the schema.

25 fields and a query with 5 encodings, Draco returns an answer in less than half a second. In contrast, CompassQL's exhaustive search runs out of heap memory after a few minutes.

4.6.3 Learning Preferences from Experiments: Draco-Learn

When developing automated visualization design systems, developers may hand-tune weights until the system synthesizes the desired specifications across test cases. This process is time-consuming and error-prone. Instead, Draco can automatically learn parameters from data. Draco's preference model allows competing preferences and, via learning to rank, can learn weights for soft constraints from ranked pairs of visualizations. The same generalizability and validity issues that all empirical studies have also apply to Draco's empirically learned weights. Draco's flexible learning system allows us to harvest training pairs from data originating from different experimental studies, even those carried out under different conditions.

To demonstrate this advantage, we harvested ranked pairs from two recent experiments on effectiveness. Kim et al. [112] measured subject performance across task types and data distributions. They compared performance across 12 scatterplot encoding specifications of trivariate data involving 1 categorical and 2 quantitative fields, encoded with x and y channels along with the color, size, or row channel - in total, 185,000 responses from 1,920 participants. The visualizations tested by Kim et al. are only a fraction of the design space that Draco supports; thus, we are not able to learn the weights of a system that can compete with CompassQL from this data alone. Saket et al. [173] conducted a similar experiment with 180 participants to evaluate task performance across visualization types. Their study is limited to encodings with one quantitative y-encoding with a mean aggregate, and an x-encoding with nominal, ordinal, or quantitative data.

4.6.3.1 Harvesting Training Data and Learning Weights

For both studies, the data contains the visualization type, data properties, task, and whether the user correctly completed the task. To create ranked pairs, we first group the response data by data schema and task. Within each group, we group again by visualization and create every possible pair. The difference in task performance between the visualizations in each pair may or may not be significant. We use Fisher's test to check whether the accuracy scores are significantly different between the two visualizations. We keep only pairs where the p-value is lower than a threshold (in our case 0.01). We consider both accuracy and timing results and include a pair if either exhibits a significant difference. Ranked pairs of visualizations could be harvested from other studies in a similar fashion. Our harvesting results in about 1,100 pairs from Kim et al. and 10 pairs from Saket et al. We get few pairs for Saket et al. because for each data and task combination, only three visualizations are compared (Vega-Lite supports bar, line, and scatter) and few exhibit significant differences.

We then apply the learning approach described in section 4.5. First, we transform every visualization that appears in the dataset into a feature vector of soft constraint violation counts in Draco. We start with Draco-CQL's constraints and add soft constraints for the preferences described in Kim et al.'s

paper. Specifically, we added rules to capture task-channel and task-marktype interactions, along with a handful of rules for the most important interactions from the discussion of the paper (see supplemental material for a full listing). We implemented these rules in a few hours. These preference rules can be also included in the CompassQL implementation from the previous section, as we can simply initialize the weights for new constraints to zero. With the new rules, we train a classifier on the difference between the two feature vectors for each pair of ranked visualizations using RankSVM. We trained an off-the-shelf SVM from scikit-learn [156] on the two datasets derived from the studies by Kim et al. and Saket et al.

4.6.3.2 Applying the Learned Model

We first evaluate our trained model directly on ranked pairs by measuring the percentage of pairs that are correctly ranked based on their costs. We train our model on a training set of 55% of the full data, validate on 15% of the data, and assess generalization of the final model with 30% test data [86]. The trained model achieves 93% accuracy on the test set, whereas Draco-CQL with hand-tuned weights achieves 65% accuracy—only slightly better than chance. Our model achieves perfect training accuracy on the dataset from Saket et al. even if we include all data harvested from Kim et al. Our model was able to learn from different datasets without degrading performance in either of them. The model correctly labels 96% of the validation dataset when we increase the p-value threshold for harvesting from 0.01 to 0.1. We also found that the test accuracy only starts to significantly fall behind the validation accuracy when we train on less than 250 pairs. This observation indicates that there is redundancy in the data and that our model generalizes. These results support our model and feature selection choices.

In practice, finding the optimal encoding given a dataset, task, and partial specification matters more than accuracy across all ranked pairs of visualizations. For example, correctly ordering the second and third best visualizations may matter less than getting the optimal encoding right. We built Draco-Learn using only the trained weights for the soft constraints in our preference model. First, we restrict the design space to only those encodings used in Kim et al. and Saket et al.'s

studies (as described at the beginning of this section) by adding about 10 additional constraints each (included in supplemental material). Adding these constraints adapts Draco-Learn to synthesize only specifications that are in a restricted design space. We then query Draco-Learn to synthesize specifications for all combinations of data properties (cardinality and entropy) and tasks (value and summary). In all conditions (48 for Kim et al., 8 for Saket et al.), Draco-Learn returns a top-performing encoding for the harvested data.

Draco-Learn outperforms Draco-CQL within the restricted design space covered by the experimental data. Figure 4.10 shows the optimal visualizations synthesized by Draco with default weights (Draco-CQL) and with learned weights (Draco-Learn) for a specification with three encodings across value and summary tasks. In Draco-CQL, the weights for all features related to task are zero. Consequently, Draco-CQL synthesizes the same specification regardless of task: a scatterplot with the primary variable (Q1) on y and category (N) on color. Draco-Learn synthesizes different charts depending on the task. To compare individual values, the scatterplot works well and reduces overplotting. However, to summarize Q1 relative to N, spatially grouping values by category (N) better facilitates perception of distributional properties such as min, max, or average [112].

4.6.4 Recommending Perceptually Scalable Designs

Data-intensive analysis routinely produces perceptually overwhelming visualizations (e.g., a scatterplot with a billion points). Here, we outline how an extended Draco recommendation model can address these perceptual scalability issues and suggest scalable alternatives (e.g., a heatmap instead of a scatterplot [175] or density visualizations of many series instead of line charts [132]). Even though we outline how Draco can be extended, more work is needed to define a comprehensive set of guidelines and derived rules for various chart and data types.

Since Draco finds the complete design that extends a partial input with the least cost, we have to develop a model—consisting of constraints and weights—that penalizes perceptually overwhelming designs. These designs are characterized by overplotting, too many marks, and excessively large

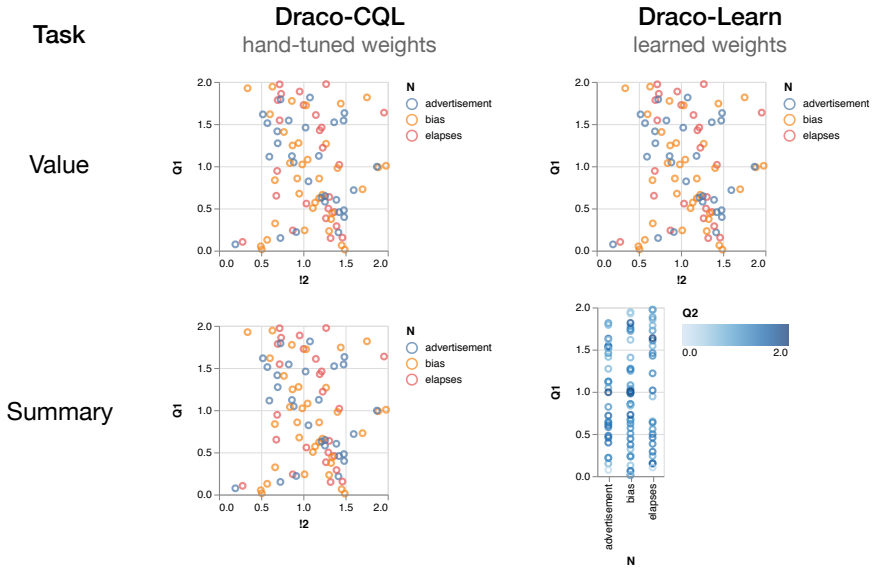

Figure 4.10: The optimal visualization synthesized by Draco with hand-tuned weights (left) and Draco with learned weights (right) for two queries with varying tasks. Draco with default weights cannot distinguish by task as the weights for all soft constraints related to task are set to zero.

dimensions (e.g., many bars or facets). We can write rules to derive whether an encoding has these properties based on properties of the fields (e.g., the cardinality or skew) and what visual channels the fields are mapped to.

For example, we can write a rule that penalizes overplotting in a scatterplot. Overplotting typically increases with the number of marks. We can use this relationship and penalize scatterplots with many marks `:~ mark(point), not aggregate(EX,_), channel(EX,x), not aggregate(EY,_), channel(EY,y), num_rows(R), R > 10000 [X].`. Here, we use a fixed threshold, which may be too brittle. In our current design we cannot write a single rule where the penalty increases with the number of rows. We can, however, write multiple rules (each with their own weight) for different thresholds. The current Draco model already uses this approach for constraints on the number of bins in a chart. Another limitation of this rule is that it only takes the cardinality into account. Skew also increases the likelihood of overplotting. To take this into account, we could penalize scatterplots with skewed data (along x and y) even with fewer rows `:~ mark(point), not aggregate(EX,_), channel(EX,x), not aggregate(EY,_), channel(EY,y), num_rows(R), R > 5000, field(EX,F;EY,F), kurtosis(F,K), K >= 3.`.

To choose designs based on local density as suggested by Sarikaya et al. [175], we would need to expand the data model from subsection 4.4.1 with facts over multiple fields. For instance, to know whether circles in a scatterplot overlap, we need to know the joint distribution of the fields mapped to x and y.

We can write rules similar to those for scatterplots but for other overwhelming designs such as bar charts with hundreds of bars or designs with high cardinality fields mapped to a nominal color channel or facet.

The final critical step is to adjust the weights such that Draco favors scalable designs over perceptually overwhelming ones. If there are no competing rules (as is likely with only the first rule from above), we can set the weights to any sufficiently large value or even make constraints hard. If

there are competing rules, weights need to be adjusted. More work is needed to support designers in finding weights to trade off among many competing rules.

4.7 Discussion and Future Work

We presented Draco, a formal model for visualization knowledge representation, and demonstrated its use for automated visualization design. Draco models visualization design knowledge using constraints and associated weights; this separation of knowledge representation from search procedures enables easier development and maintenance. The Draco-APT and Draco-CQL examples demonstrate how Draco can support increasingly sophisticated visualization design tools with less development effort and better performance than prior approaches. The Draco-Learn example demonstrates that Draco can combine data from different studies to learn weights for a state-of-the-art visualization design tool, further accelerating modeling efforts.

We now discuss how Draco's constraint system can enable new usage scenarios, such as design space enumeration, visualization model comparison, and design debugging. We go on to describe how future work might address current limitations of Draco's implementation.

4.7.1 Draco from a Software Engineering Perspective

Draco's use of constraint programming enables easier development and maintenance of automated visualization design tools. Due to implementation complexity, prior approaches often have to compromise the implementation of effectiveness criteria. Although Draco does not completely solve this problem, it shifts the problem to the more tractable and well-defined problem of defining weights to trade-off competing preferences. We show that these weights can be effectively learned from data even when the dataset is assembled from different sources.

Using constraints also decouples knowledge representation from the code that applies that knowledge. Although this approach aims to benefit visualization tool developers, it may also benefit end-users as it makes knowledge bases easier to contribute to. Visualization researchers can

disseminate their results as constraints to make them more easily accessible by visualization designers; in a declarative system, designers might use more nuanced models that would otherwise be too complex to maintain. We contend that software engineering and developer productivity should be given more attention in the visualization community. Human designers should focus on the design of the visualization design space and preferences rather than the design of search algorithms that are already available in domain-independent constraint solvers.

Draco's knowledge base can be adapted or extended to fit specific needs. Each component of Draco can be easily modified: the definition of the design space, the preferences within the valid solution space, their weights, and how the visualization model is queried. For example, in subsubsection 4.6.3.2, we limited the design space to scatter plots with three encodings (two quantitative, one nominal). The same expressiveness and preference constraints could be used in a tool that targets full Vega-Lite specifications or in a tool that targets only specific visualizations, such as vertical bar charts. Similarly, Draco can be extended with richer descriptions of input data (subsection 4.4.1) that can then be referenced by new soft constraints (subsubsection 4.4.2.2). A researcher who wants to extend the Draco knowledge base with new design rules could distribute their rules as an independent set of constraints or updates to the weights.

We hope that Draco's current set of constraints can serve as the starting point of an evolving knowledge base that can be extended by researchers and practitioners. For example, Draco could be extended to include richer task taxonomies [8]. One challenge for visualization design tools is that the "task" is typically inaccessible (e.g., within a user's mind). Natural language interfaces may be better suited for communicating user intent [65, 181, 195], which could then be expressed as Draco constraints. Draco's visualization model supports synthesis of marks, encodings, and simple transformations (binning and aggregation). We plan to extend the model to transformations such as filtering and sorting, and incorporate Vega-Lite's interaction primitives [177] (chapter 3).

We are excited to explore how our visualization model can be extended to support chart composition, for instance into layered views or dashboards. Applying design guidelines to multiple charts

separately can lead to locally effective, yet globally inconsistent views [167]. For example, different fields might confusingly be encoded with the same color scheme across charts. With the right set of weighted constraints, Draco could trade-off among the effectiveness of single views and global consistency within a multi-view display [167, 224].

In our demonstration of Draco-Learn, we modeled a restricted subspace of visualizations that mirrors the limits of the available experimental data. We hope to encourage more researchers to make data from effectiveness studies available, such that their results may be used by Draco or related systems. Future work might provide tools to help researchers convert their results into constraints or ranked pair datasets. We plan to collect more comprehensive data by systematically generating visualization pairs and having human subjects evaluate them. In addition to independent studies, we might leverage Draco's design space to guide data collection in an active learning process.

With sufficient data, it may even be possible to go beyond learning weights and attempt to learn preference rules themselves [174]. The AI community uses inductive logic programming methods to infer logic programs from databases of positive and negative examples [168]. To learn from noisy data (common in the visualization domain!), we could combine inductive logic programming with statistical models such as Markov logic networks [55, 114]. For example, Law et al.'s ILASP (Inductive Learning of Answer Set Programs) [116] is a logic-based learning system that can learn preferences in answer set programs. To understand differences in preferences represented by two or more distinct data sources, we can use multi-objective (Pareto) optimization in ASP to enumerate designs that map the trade-off frontier.

Because the effectiveness of a visualization can depend on low-level features not captured in a high-level specification (for example, over-plotting), we can imagine applying a *re-ranking* strategy in which Draco enumerates a number of top-scoring candidate designs (ranked by high-level features) that are then re-ranked by another learned classifier operating on low-level features that may be impractical to model directly in ASP. The sub-symbolic models learned by such classifiers could constitute another valuable form of visualization design knowledge to represent and share.

4.7.2 Beyond Automated Visualization Design

Up to this point, we have positioned Draco as a tool for synthesizing optimal visualization designs from partial specifications. However, Draco could be used in a variety of other contexts. In the following, we discuss four directions that Draco could be extended.

First, Draco can be used as a general "visualization spell checker" to validate and auto-correct designs independently, or within a broader system for people to "learn by doing". Currently, Draco can use expressiveness and effectiveness constraints to report errors for designs that violate design guidelines. However, given a visualization, we could extend Draco to additionally automatically correct the visualization, removing the most severe violations and suggesting alternative valid designs to users. The problem of finding the minimal set of constraints that need to be removed for the remaining constraints to be satisfiable is known as the *unsatisfiable cores problem* [47]; related techniques could be applied to visualization design constraints. Draco might also explain those violations and why they matter, to teach students or visualization designers about best practices, help them spot (intentionally or unintentionally) misleading visualizations, critique visualizations, and perhaps contribute new visualization knowledge or explanations.

Second, Draco can facilitate exploration of the visualization design space. Besides surfacing violations of design guidelines, Draco can rank visualizations by their costs. Designers might use this function of Draco to choose among different alternative designs. Draco could also be used to cluster designs based on their violations (using the same feature vectors used in our learning to rank approach). An exciting avenue for future research is to use Draco's design space definition to systematically generate visualizations to build a corpus of visualizations and interactions. Creating such a corpus is as simple as running Clingo on the Draco design space definition without preferences, which enumerates all valid answer sets. Testing generated designs with human subjects will allow us to understand the costs and benefits of different encodings and interactions. Although the current design space in Draco is limited, as noted above we plan to extend the model further, including interaction primitives such as Vega-Lite selections [177].

Third, Draco can be used as a tool for researchers to compare the implications of different effectiveness studies. Concretely, if a researcher finds a new design guideline, they could add it to Draco as a constraint and assess whether it conflicts with, or is subsumed by, existing design guidelines. Based on comparison results, researchers could share their design results as constraints to improve the common knowledge base of visualization design tools.

Lastly, an important future extension is tooling to support developers, researchers, and designers. In addition to collecting more data to learn preference weights, we hope to provide tools to browse the visualization design space and knowledge base rules, as well as tools to understand violations and fine-tune trade-offs among competing design guidelines. With the right tooling and fine-tuned visualization models, Draco's declarative approach to automated visualization design could bring us one step closer towards building assistive interfaces for effective design that canvas a much broader swath of the visualization design space. Such interfaces should allow visualization designers to consider a greater variety of approaches, while also focusing on the creative aspects of visualization design.

4.7.3 Limitations of the Model

Besides the limitations of our implementation and tooling, discussed in the previous sections, some of our design decisions make it difficult to express some constraints you might want to express efficiently.

Draco models knowledge about visualization design as weighted constraints. A violated constraint incurs a cost for each violation. To express constraints whose severity increases with some measure (e.g., higher cost for more marks in subsection 4.6.4), we need to define thresholds and write one constraint and weight per threshold. The ASP formalism would allow for weights that are scaled with some measure, but our learning method is not designed to learn the scales weights.

The constraint solver minimizes the weighted sum of violations. Therefore, Draco can only express linear combinations of the constraint violations. non-linear relationships of the raw features (e.g.,

properties of fields, mark type) can only be expressed and learned with constrains over multiple of these raw features. Some preferences are inexpressible with pure updates to the weights and require new constraints. We discuss these issues in a follow-up paper [174], but more work is needed to support people who design visualization models.

5 Falcon: Brushing and Linking Billions of Records

Figure 5.1: Falcon visualizing binned aggregates for 180 million flights [205] in a web browser. The brushes select short afternoon flights with no more than a 10 minute arrival delay. The views update instantly when the user draws, moves, or resizes a brush.

As the scale at which analysts need to work has outpaced the tools they use, we are challenged to create new tools that do not overwhelm people or their computational resources. Vega-Lite (chapter 3) and Draco (chapter 4) can help address the first issue. With Vega-Lite, we can describe interactive visualizations for large and complex datasets. With Draco, we can describe design

guidelines for visualizations that guide analysts towards visual encodings that show important patterns and outliers regardless of the scale of the data. The challenge for the visual analysis system is to manage the amount of data and computation while remaining responsive.

The Vega-Lite runtime can leverage that specifications and execution are separate to partition the dataflow and push expensive computations into a scalable backend system [135]. While this approach supports static visualizations well, current data processing tools are often insufficient for interactive visual analysis.

To support interactive analysis at scale, this chapter contributes prefetching and indexing methods for low-latency interaction across linked 1D and 2D views. We implemented these ideas in Falcon, a web-based visualization system. Falcon models and optimizes a user's session with client-side state rather than treating every query as an independent request. In a session, Falcon leverages that users typically only interact with a single view at a time. When the user interacts with a view, Falcon creates an index that supports interactions with that view. Falcon can calculate the index from an aggregate query in a database system. As the user moves the brush in the view, Falcon computes the necessary data for all other views in constant time. When the user interacts with a different view, Falcon reindexes the data, which incurs a short delay. We accept this trade-off since users are more sensitive to latencies in brushing interactions than delays after switching views [122]. Falcon further trades-off initial accuracy of the visualizations for faster view switching. By applying these principled trade-offs, Falcon sustains real-time interactivity at 50fps for pixel-level brushing and linking interactions among multiple visualizations of billion-record datasets. We show constant brushing performance regardless of data size on datasets ranging from millions of records in the browser to billions when connected to a backing database system.

Falcon advances the state-of-the-art in two ways. First, we exploit that not all interactions are equally latency sensitive to develop user-centered indexing strategies for a scalable interactive system. Second, unlike previous systems that required custom data structures, Falcon can use existing database systems to help create its index.

We published Falcon at CHI 2019 (co-authored with Bill Howe and Jeffrey Heer) [136]. We make Falcon available as open source software with supporting documentation and demos at www.github.com/uwdata/falcon.

5.1 Introduction

To support effective exploration, interactive visualization systems must provide rapid response times for latency-sensitive operations. Further, delays between user actions and corresponding updates may break the perceived correspondence between action and response, reducing the user's engagement with the system and leading to fewer observations made [62, 122, 230]. As the scale and heterogeneity of available data continue to increase, greater emphasis is being placed on efficient exploration. However, large datasets incur delays that negatively affect user's exploration. Poor support for interactive exploration can skew analyst attention toward "convenient" and familiar datasets, causing selection biases. This work seeks to reduce the friction of using new data by enabling *cold-start analytics*: exploration without time-consuming precomputation.

Traditionally, the different stages of the data processing pipeline—query processing, data transfer, and rendering—have been optimized as independent modules. For example, many efforts to reduce latency have centered on query processing, paying scant attention to the corresponding interface design. Recent GPU databases [170] can achieve query times of seconds or hundreds of milliseconds over billions of records; however, interactivity is still difficult to achieve due to factors outside the scope of database optimizations, including network latency and sub-optimal client-side application design. Even short query times accumulate when a UI generates thousands of queries. And network-induced delays remain unpredictable, especially in low-connectivity or mobile networks.

In Falcon, we take a holistic approach to system design by optimizing the interface and query systems together. We prioritize the allocation of compute resources to interactions for which users

are more latency-sensitive, in particular *brushing and linking*. For example, in Figure 5.1 a user can resize the brush in the arrival time view, which immediately updates all other views.

To eliminate latency for brushing interactions, we contribute prefetching and indexing techniques. Rather than treating every query as an independent request, we model and optimize a user's session with client-side state. In a session, we leverage that users typically only interact with a single view at a time—the *active view*. When the user moves the brush at pixel resolution, the aggregated data for all other views—the *passive views*—are computed in constant time using Falcon's indexes. Brushing interactions are decoupled from computations over the raw data; the interactive resolution of the brushes is decoupled from the bin resolution. As a result, both index size and interactive latency depend only on bin size and available pixel resolution and are independent of the raw data. Brushing in Falcon therefore meets our definition of perfect scalability from section 2.5.

When the active view changes, Falcon reindexes the data to support interactions with the new active view. For datasets of up to millions of records, the client can perform the necessary aggregations. For larger datasets, aggregation can be offloaded to a backing database system. Switching the active view in Falcon incurs processing delays. Such switches usually occur with a shift in a user's attentional focus, a less latency-sensitive action [26]. To limit view switching times, Falcon initially lowers the resolution of the index data so brush boundaries "snap-to" units larger than individual pixels. Analogous to progressive rendering or query processing (e.g., *online aggregation* [35, 93]), this reduced interaction resolution can then progressively improve.

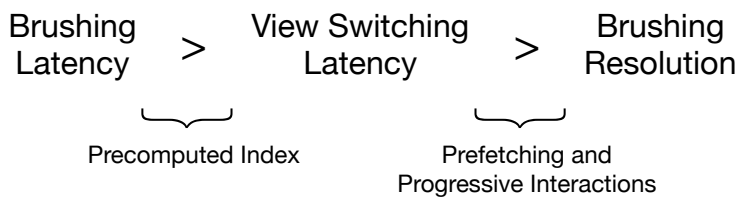

Figure 5.2: The Falcon system uses indexes to optimize brushing latencies and progressively improves interactive resolution to reduce switching times.

In summary, Falcon prioritizes brushing latency over view switching delay, and it prioritizes view switching delay over initial interactive resolution (Figure 5.2). Our prototype implements methods to support coordinated brushing and filtering for cross-filter and aggregation applications (i.e., ensembles of visualizations of 1D and 2D bin counts). The system avoids expensive precomputation by prefetching only the data necessary for interactions with a single active view, enabling *cold-start* analytics. In Falcon, charts update in response to brush changes at 50 fps. We demonstrate that this performance is invariant on data sets ranging from thousands to billions of records. Because the system progressively improves interactive resolution, we use interpolation to approximate higher-resolution interactions.

5.2 Background and Related Work

Falcon is a dynamic query UI [183] where users interact with visual representations of the components of a query. In particular, Falcon is a visualization system for interactive brushing and linking across coordinated views of *binned aggregates*. Binned aggregates summarize data by dividing the domain of variables into discrete units (bins), and then by aggregating the data records in each bin [123, 218]. For example, *histograms* are visualizations of *bin counts*. Each graphical mark in a visualization of binned aggregates summarizes a large subset of records in the original dataset. Falcon uses binned aggregates because they convey both global patterns (e.g., densities) and local features (e.g., outliers) and enable multiple levels of resolution via the choice of bin size. To focus on relevant subsets of the data, analysts select ranges of data in one view using an interactive brush, which then updates all linked views. Commercial data visual analysis systems such as Tableau [192], PowerBI [129], and Immerse [98] use coordinated views with visual querying.

Falcon extends prior work on scalable interactive analysis systems, incorporating findings from experimental studies of the effect of latency on exploratory visual analysis.

5.2.1 Latency in Interactive Analysis

Informed by prior accounts of latency in psychology and HCI [9, 25, 26, 106], Liu and Heer [122] conducted controlled experiments (later replicated by Zgraggen et al. [230]) to understand how latency affects user behavior in exploratory visual analysis. Comparing different operations under two latency conditions, they found that additional delays of 500ms over the low-latency condition (20ms for both brush and link and select, 100ms for pan, and 1s for zoom) negatively impact user behavior. They also found that initial exposure to delays negatively affects subsequent performance even when the delays were removed in later sessions.

In recent years, system designers have reduced latency by optimizing the different stages of the visualization pipeline: data management, scenegraph construction, and rendering [122]. Their efforts have largely addressed each stage separately. Many system designers have adopted 500ms as a uniform latency threshold goal (e.g., [12, 44, 53, 154]); however, experimental results indicate that some operations (e.g., zooming) are less sensitive to delays than others. Liu and Heer [122] found that panning, brushing, and range selection are the most latency-sensitive of the studied operations.

5.2.2 Scalable Visual Analysis Systems

Interactive analysis systems with coordinated views run in a client application. The data being analyzed can either be loaded into this client or offloaded to a dedicated server. Table 5.1 compares the characteristics of different visual analysis systems for coordinated brushing and linking. For a dataset small enough to be loaded client-side, visual analytics tools support real-time interactivity using local indexes. Square's Crossfilter [43] uses bitmap indexes to support brushing and linking of hundreds of thousands of records entirely in the browser. Liu et al.'s imMens [123] uses precomputed summaries to enable real-time interactions in the browser, but their interactions are limited to the binning resolution and a single interactive brush.

For greater scalability, many systems adopt a client-server architecture. In this approach, all components of the information visualization reference model [25] are not necessarily on the same machine. Instead, the server stores the data, processes incoming queries, and sends reduced aggregates to the client. In the client-server model, changes to the client-side UI state require a request to the server; this incurs a delay between the user action and the corresponding update, a delay dominated by the network round-trip and query execution times. Since network bandwidth and latency are often beyond the control of the tool designer, optimizations aim to increase query performance. We discuss these techniques in the next subsection.

Falcon supports both client-only and client-server setups. For data sizes up to tens of millions of records, Falcon can load the full dataset in the browser and index it there. For larger datasets, costly computations can be offloaded to a scalable server-side database system.

Feature \ System	Square Crossfilter	imMens	Nanocubes	OmniSci Immerse	Falcon
Approach	Client-side Index	Dense Data Tiles	Sparse Cube	Live Queries	View-Specific Tiles
Architecture	Client	Client (Server)	Client-Server	Client-Server	Client (Server)
Demonstrated data size	10^5	10^{12}	10^{12}	10^{12}	10^{12}
Cold-start	Yes	No	No	Yes	Yes
Interactive resolution	Pixels	Bins	Pixels	Pixels	Pixels
Multiple brushes	Yes	No	Yes	No	Yes
2D binning	No	Yes	Yes	Yes	Yes
Zooming	No	Yes*	Yes	Yes	Yes
View switching cost	No	No	No	No	Yes

Table 5.1: Comparison of different approaches to scalable linked views. Not shown: PowerBI [129] and Tableau Public [193] use a similar approach to Immerse; Hashedcubes [152] builds on Nanocubes [119] and shares similar properties.

* supported for predefined zoom levels

5.2.3 Scalable Data Processing for Visualization

Three main approaches can speed up query evaluation: *parallel evaluation*, *indexing*, and *approximation*.

5.2.3.1 Parallel evaluation.

To reduce query latency in large-scale online analytical processing (OLAP) systems [34], we can distribute work across multiple machines. However, the additional communication overhead typically exceeds the query times necessary for interactive data exploration.

5.2.3.2 Indexing.

Indexes and data cubes [79] significantly speed up query evaluation by precomputing aggregates along some dimensions. Specialized hierarchical data structures for visualization, like Nanocubes [119], are compact indexes of spatiotemporal data that can fit in the main memory of a single machine. The Nanocube structure leverages sparsity to more efficiently build a specialized tree index that consists of quadtrees (for spatial dimensions) or flat trees (for categorical attributes). The trees organize and aggregate data records for each dimension, which are then combined into a larger index. Hashedcubes [152] further improves this design with a more compact index. Profiler [110] also builds in-memory data cubes for query processing. The size of the data cube depends on the resolution and number of dimensions, not on the data size. Thus, data cube approaches enable scalable data processing on a single machine and support low latency responses to aggregation queries over billions of records. However, they can impose lengthy index building times, e.g., Nanocubes takes up to 6 hours to build an index for a dataset with 210M objects [119].

Liu et al.'s imMens [123] uses a dense data cube structure with precomputed aggregations. The size of the full data cube is $\prod_i b_i$, where b_i is the bin count for dimension i; it is polynomial in the bin count and exponential in the number of dimensions. To overcome the exponential growth with more dimensions, Liu et al. decompose the full cube into a set of overlapping 3- and

4-dimensional projections, or "data tiles." This approach enables real-time brushing and linking but limits interactions to a single brush. Moreover, since the resolution of the imMens data cube is the resolution of the visible bins, brushes snap to these bins. For large datasets, the tiles must be precomputed since they are costly to calculate.

Falcon makes a critically different trade-off: it decomposes a data cube so that it supports interactions with a single active view only. The size of its full index is linear in the number of views, which avoids a combinatorial explosion. An index is loaded when the user interacts with a particular view. Falcon supports multiple brushes by conditioning it on the brushes in the passive views. Further, each view can be filtered by all brushes except the brush in the view itself instead of only the union of all filters. Querying and transferring the smaller index for a single view is less costly than doing so with a full data cube (e.g., imMens) that supports interactions with all views. The smaller index can be computed and loaded on demand. We can also increase its resolution to support brushing at pixel resolution rather than snapping brushes to visible bins. Falcon supports cold-start exploration of new datasets and is more flexible about zoom levels (as both imMens and Falcon require a new index when the user zooms).

The Dice system [102] explored a similar approach but focused on exploration of the data cube rather than interactive visualization.

5.2.3.3 Approximation.

Approximate query processing systems [35] estimate result values and their uncertainty using a data sample. SampleAction [62], Vizdom [44], and Pangloss [134] demonstrate that progressively refined approximate results are often sufficient for exploratory analysis. However, these systems do not support interactive brushing and linking. Although the current version of Falcon does not use progressively growing samples to approximate aggregates, we apply an analog of these techniques in the interaction domain: we initially load an index that supports interactions at a granularity larger than single pixels, then we progressively refine the granularity to one pixel.

5.2.4 Prefetching

To mitigate query and network latency, a system can try to predict queries that the UI will likely issue, then precompute and cache results [51]. Chan et al. [32] show this approach for time series data. Battle et al. developed a series of systems [11, 12] that prefetch data tiles for a panning and zooming interface. Falcon combines ideas, such as projected data cubes in imMens [123], with prefetching methods. It prefetches an index that supports all interactions with the current view, and is conditioned on the brushes in all other views.

5.3 The Falcon Interface Design

We now describe the Falcon interface, how users can interact with its charts, and how it prefetches data to rapidly update charts. Here, we highlight how Falcon works. The following section provides a more detailed discussion of its implementation.

Falcon provides a dashboard of views that visualize the number of records, grouped by zero, one, or two binned dimensions. Figure 5.1 shows Falcon loaded with a U.S. flight delays dataset [205]. Using application constructor parameters, developers can configure the dataset, configure the chart layout, and customize the chart design.

5.3.1 Charts for Zero-, One-, and Two-Dimensional Data

Falcon's views show aggregates grouped by zero, one, or two binned dimensions. We implement all visualizations in Vega [178]. For zero-dimensional data, a view simply shows the record count. Developers can choose among a vertical or horizontal bar chart (e.g., Figure 5.1, top left) or a text view. Blue bars show the number of records that match all filters, while gray bars show unfiltered counts for comparison.

For views that are grouped by a single binned dimension, Falcon uses bar charts (e.g., Figure 5.1, top center). Again, Falcon shows the total unfiltered counts as gray bars. Unfiltered counts provide

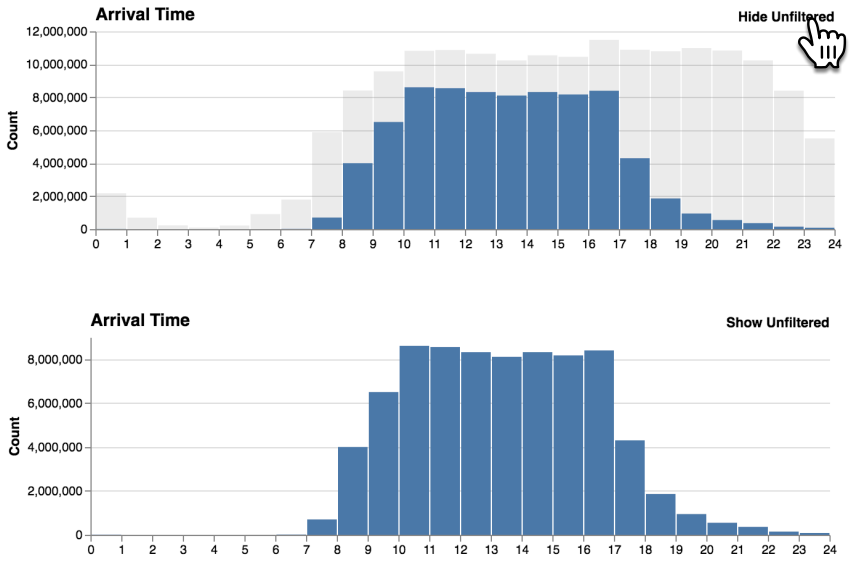

Figure 5.3: The top histogram plots filtered counts (blue bars) relative to the unfiltered data (gray bars). A user can toggle to show the filtered distribution only (bottom).

context and keep the domain of the Y-scale constant as a user filters data. To see the filtered distribution only, users can toggle the gray bars (Figure 5.3).

For bivariate views, we must use additional visual channels, such as size (e.g., Figure 5.1, right) or color. Size (e.g., circular area) is known to be more perceptually effective for numerical comparison [41], though it requires significantly more pixels compared to color encoding. Most importantly, by encoding counts as the size of circles, Falcon can show unfiltered counts as gray circles behind blue circles, establishing a consistent visual language across all three chart types. Nonetheless, developers can switch to color encoding [123, 134], but they can no longer see unfiltered counts.

Applying consistent binning schemes over 1D and 2D views ensures compatibility of linked selections between plots. Falcon uses Vega's [178] binning algorithm to compute bin width and offset from the scale range. The algorithm may extend the scale range to find "nice" bin thresholds (e.g., using only multiples of 5 and 10).

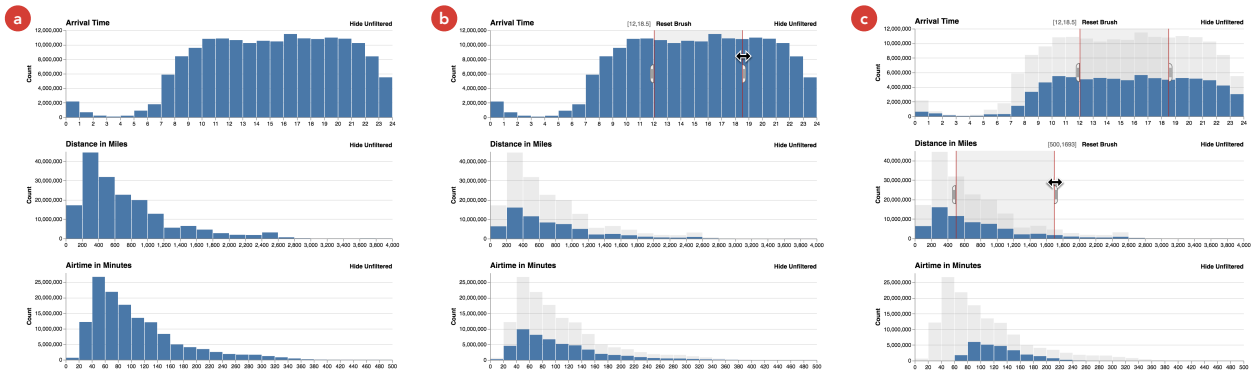

Figure 5.4: View switching in Falcon. (a) When the user initially loads Falcon, it shows unfiltered histograms. (b) The user can draw a brush in the histogram of the arrival time (active view), and all other passive views will be updated. (c) After a view switch the distance histogram is active, and the user can draw a brush there.

5.3.2 Brushing in the Active View

Upon initialization, Falcon shows unfiltered counts (Figure 5.4, a). A user can then filter the data—for example, to show only flights that arrive in the afternoon—by drawing a brush in any view (Figure 5.4, b). We call the view, with which the user interacts, the *active view*. Falcon aims to show the counts of the selected subset and update the data for all other views—*the passive views*—as the user changes the brush. The active view does not change.

Because re-aggregating counts from the raw data for every brush movement can be too costly, Falcon uses an index, which is a compact summary that contains the details needed to update passive views for any possible brush in the active view. Falcon decouples rapid brush updates (using any of the actions in Figure 5.5) from costly computations over the full dataset. To limit its size, the index contains binned aggregates for passive views at their bin resolution and supports brushes in the active view at pixel resolution. Falcon achieves a much smaller index than one that supports interactions with all views [119, 123] by focusing on a single active view.

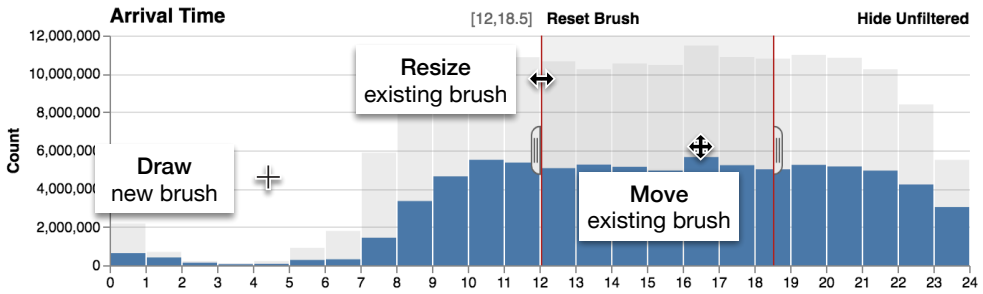

Figure 5.5: Brush interactions in Falcon. Users can draw a new brush or move and resize an existing one.

5.3.3 Switching Active Views

For a user to draw a brush in a different view, Falcon must switch the active view, requiring it to make a new index. For example, to change from brushing over arrival time to distance (Figure 5.4, c), the index for the arrival view must be replaced with an index that supports brushing in the distance view. The new index must be conditioned on any existing brushes (here, for arrival time), which means the counts in the new index must be filtered according to the selected ranges. The only exception is for the arrival time view itself, since the bin counts should not be filtered by its own brush. This rule generalizes to any number of brushes: the index for each passive view must always be conditioned on the brushes in all other passive views.

5.3.4 Zooming the Active View

When a user zooms in a binned chart, the visualized range changes. Since scale changes require no new data, Falcon can immediately give visual feedback and zoom the chart. When the zoom interaction ends, Falcon computes a new bin width and offset. If the computed parameters differ from the current ones, Falcon computes updated bin counts for the active view as well as a new index. The latter is needed to support interactions at the new resolution. Recomputing the

index may impose delay. However, as discussed earlier, research has shown that zooming is less latency-sensitive than brushing [122].

5.3.5 Prefetching

Instead of waiting for the first interaction with a new active view to create a new index, Falcon can prefetch indexes before a user starts brushing. Figure 5.6 shows example timings for the brushing interactions shown in Figure 5.4. After modifying the arrival time brush, an analyst might move the cursor to hover over the distance view when preparing to draw a new brush. In this case, Falcon would not yet perform a view switch (i.e., change the index), but it would prefetch the index. When the analyst starts brushing (around second 20), Falcon switches to the prefetched index. Hovering over a chart is only one signal that we could use to predict what chart the user will interact with next. Techniques from previous work on prefetching [11] could also be used, but we found that mouse hover is a strong indicator of user attention [37]. In addition to prefetching on mouse hover, Falcon can use long idle times between interactions to precompute additional indexes.

5.4 Falcon System Implementation

We now discuss how Falcon implements the interactions just described in section 5.3.

5.4.1 An Index of Data Tiles

A Falcon index contains data needed to render passive views for every possible brush in the active view. The data for binned aggregate views with zero, one, or two grouping dimensions can be stored in a zero-, one-, or two-dimensional array; this projection of the data cube [79] is called a *cube slice*. For example, the histogram in Figure 5.5 needs an array with 24 entries for the flights for each hour of the day.

The user can draw brushes in one-dimensional histograms or two-dimensional visualization of bin counts. For a 1D histogram, there are in theory an infinite number of possible brush configurations.

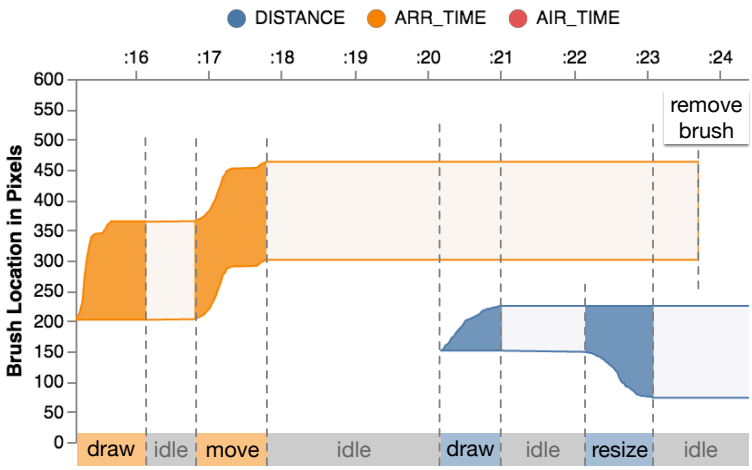

Figure 5.6: Visualization of the timing for the brushing interactions in Figure 5.4. The user first draws and then moves a brush in the arrival time histogram before drawing and resizing a brush in the distance view. Finally, the user deletes the arrival time brush. Between interactions, the app is idle waiting for user inputs.

However, in a pixel display a histogram that is p pixels wide has only p^2 distinct brushes, with a brush start and end at two pixel locations. Storing p^2 cube slices for each passive view remains prohibitively large. Falcon therefore encodes these p^2 slices as p *cumulative* slices and stores these cumulative counts in a single multidimensional array, which (following imMens) we call a *data tile*.

Figure 5.7 shows a data tile with airtime as the active view and arrival delay as the passive view. Since each column stores the sum of all counts from the start, a specific cube slice is the difference between the cumulative slices for the start and end of the brush. For a fixed number of bins, this difference is computed in constant time ($\mathcal{O}(1)$).

Computing a sum (e.g., of counts) as the difference of cumulative sums is often used in computer graphics and is known as *summed area tables* [45] or *integral images*. Summed area tables generalize to many dimensions; in Falcon, we use the same approach for brushing in 2D views. Here, a cube

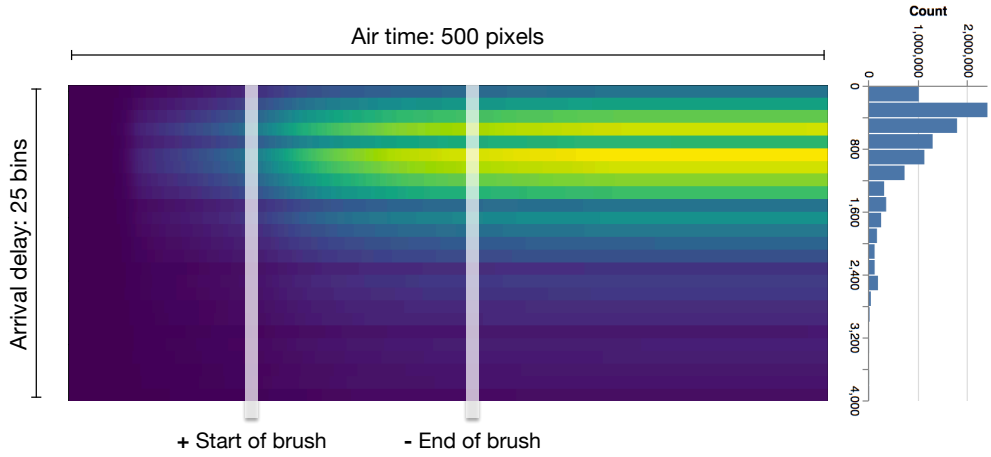

Figure 5.7: A visualization of a data tile with departure time as the active view and distance as the passive view. A lighter color indicates larger cumulative counts. A histogram for the passive view conditioned on a brush can be computed as the difference between the cumulative bin counts at the end of the brush and the start of the brush.

slice for a passive view is computed from the four corners of the brush in the active view. A data tile stores the cumulative sums along the dimensions of the active view.

In addition to a data tile for each passive view, the index must also store the cube slice for the case where there is no brush. This is necessary because a brush that spans the full extent of the active view does not contain all records in the raw data when a user has zoomed in on a view.

In the general case, a data tile is an array whose dimensionality is the sum of the dimensionalities of its corresponding active and passive views. In Figure 5.7, the active and passive dimensions are each grouped by a single dimension, so the data tile has two dimensions. This concept generalizes: to support brushing in a 2D active view and filter a 2D passive view, we need a four-dimensional array.

The size of a data tile is the product of the bin counts for each dimension of both the active and passive views. The number of data tile bins corresponding to the active view depends on the active

view's pixel screen size. The ratio of active view pixels to corresponding data tile bins determines the interactive resolution, which is at most 1 pixel per bin.

5.4.2 Computing Data Tiles

We implemented two query systems to compute data tiles for Falcon. The first is a query engine that runs in the user's browser alongside the Falcon UI. This engine supports queries over tens of millions of records (as Apache Arrow files [63]), above which latency becomes unacceptably large. The second system generates SQL queries for a database server. The scalability of this approach depends on the database system.

Both engines use a similar approach to compute data tiles. For each passive view, they aggregate the records that are not filtered out by any brush in other passive views. From these counts, the engines build the cumulative data tile.

5.4.2.1 In-Browser Engine

The engine in the browser computes a data tile by first creating an empty multidimensional array of the binned dimensions. In a single pass over the filtered data, it then counts how many records fall into each cell of the array. In a final step, it computes the cumulative sums along the dimensions of the active view. While the engine iterates over the records, it counts how many records match the filters in the other passive views but fall outside the extent of the active view; it uses these values to determine unfiltered counts. Since the size of the data tile is independent of the size of the data, the running time of the last step is bounded only by the number of bins in the dimensions of the active and passive views. We implemented the engine in JavaScript; thus, it is single-threaded and blocking.

5.4.2.2 Engine for Database Server

The database engine issues queries that perform binning and aggregation in a scalable database system, such as OmniSciDB [170] (OmniSci was formerly known as MapD). Falcon generates

```sql
SELECT
    CASE
        WHEN airtime BETWEEN 0 AND 500
        THEN floor((airtime - 0) / 1)
        ELSE -1 END AS binned_airtime
    , count(*) AS cnt
    , floor((arrdelay - -20) / 5) AS binned_delay
FROM flights
WHERE arrdelay BETWEEN -20 AND 60
GROUP BY binned_airtime, binned_delay
```

Figure 5.8: The SQL query to compute the counts for Figure 5.7 and a special bin (-1) for the unfiltered counts. The cumulative counts are computed on the aggregated data.

aggregate queries that filter by the brushes in the other passive views and group by the bins in the dimensions of the active and passive views (Figure 5.8). The same query counts the records that fall outside the extent of the active view. Because the queries for each passive view use different filters and group-by clauses, they cannot be naturally expressed as a single query. Query results are received client-side and written into a multidimensional array. Falcon computes the cumulative counts in the client, since some databases (including OmniSciDB) do not support window aggregates, which are necessary to compute cumulative counts efficiently. Queries execute asynchronously without blocking the UI.

5.4.3 Progressive Interaction

Switching the active dimension and zooming are not as latency-sensitive as brushing. Nonetheless, delays may be frustrating to users. To address this issue, we propose *progressive interaction*, an analog of progressive refinement of approximate aggregate queries [93] or progressive loading of images [19] and data [76], but in the interaction space. It works as follows: initially, Falcon loads small data tiles, where the bin count of the dimensions of the active view is lower than the pixel count. The user can interact with the active view, but the brush will snap to the closest available data tile bin boundaries. Falcon then loads data tiles for interactions at the pixel resolution in the background. Our current prototype implements progressive interactions in two steps: (1)

Falcon loads data at the resolution of the visible histogram bins, and (2) Falcon loads the full pixel resolution.

5.4.4 Interpolation

When brushes snap to the closest bins, users cannot set brushes at the pixel resolution. Continuous brushes have two advantages. First, users can set brushes between bin boundaries. Second, histograms in the passive views change smoothly as users brush. When Falcon starts with low-resolution data in progressive interaction, it approximates brushing at pixel resolution using interpolation. We interpolate between bin counts for the passive view at the bin boundaries closest to the current brush ends. Though interpolation errors are usually small and resolved as soon as high-resolution data arrives, interpolation remains an approximation with unknown error bounds. To make users aware of this, we decrease the opacity of passive views while users interact with an active view with low-resolution data tiles. Interpolated bin counts for smooth brushing at pixel resolution let users place brushes at a precise location; this helps users place multiple brushes without waiting for a full resolution version to load. Falcon uses linear interpolation for 1D brushes and bilinear interpolation for 2D brushes.

5.5 Benchmark Evaluations

We now present a benchmark evaluation of Falcon's brushing performance and the cost of indexing datasets of different sizes. Falcon reduces latency by progressively computing indexes with increasing resolution. We measure the time to compute an initial low-resolution index and the errors of interpolating pixel-level interactions from this data. We then discuss our results and their implications.

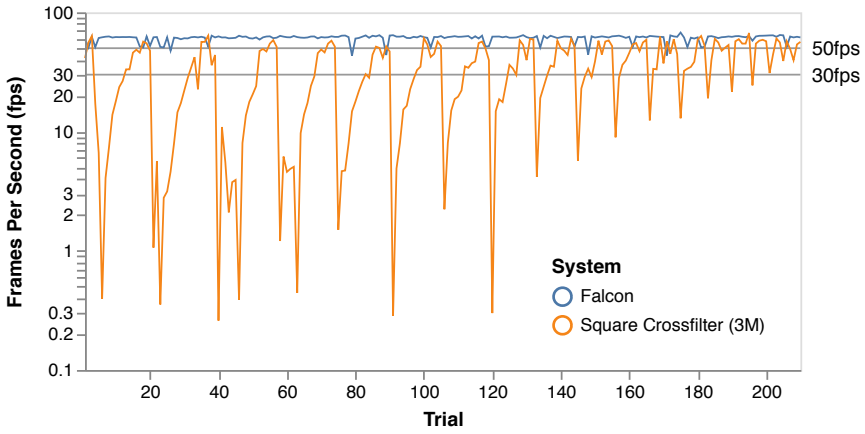

Figure 5.9: Latency between brush interactions with one chart and updates to 5 passive views, averaged across 5 trials. We compare Falcon to Square's Crossfilter with 3 million records. Falcon's performance is constant, close to the browser's maximum frame rate of 60fps regardless of the full dataset size. Crossfilter reacts slowly when many records are added to or removed from the brush.

5.5.1 Brushing Performance

Figure 5.9 compares Falcon's brushing performance to Square's Crossfilter. For this benchmark, we programmatically update the brush by iteratively changing its start and end. We run experiments on a 13" 2014 MacBook Pro using the Chrome 70 browser, with Falcon performing all indexing on the client. Using a chart configuration of 6 histograms for flight delays [205], Falcon consistently updates the 5 passive views at more than 50 frames per second. Due to its incremental processing of original data records, Crossfilter's query update times spike when the brush moves over parts of the view where many records are either added to or removed from the filter. In addition, Crossfilter needs ~10s to parse the CSV file and ~30s to initialize internal data structures for 3M records. Falcon works on binary data [63]—there is no parsing or initialization cost—and for up to 10M records requires less than one second to switch views.

5.5.2 View Indexing Cost

Before a user switches views, Falcon prefetches data tiles for all passive views. Table 5.2 shows the mean, median, and 95th percentile of the time it takes to prefetch data for all passive views for different configurations and data sizes. We measure indexing times for high resolution (500 pixels for 1D and 200 × 200 pixels for 2D) and bin resolution (25 and 25 × 25 bins). We use three datasets: *flights*, *weather*, and *GAIA* (Figure 5.11). The *flights* dataset [205] contains information about flight time, length, distance, and delays for all 180M commercial flights in the U.S. since 1987. The *weather* dataset [143] contains NOAA weather statistics for different locations in the U.S. *GAIA* [22] is a sky survey from the European Space Agency with records for more than a billion celestial objects.

We find that indexing time unsurprisingly increases with data size. In the browser, view switching times stay below 5s even for datasets with 10 million records. Computing a low-resolution index does not reduce indexing time in the browser but can reduce average time by up to 6× with a backing database server (here, OmniSciDB). The time to load the first data tile in an index from OmniSciDB is up to 24× faster than the time to finish loading all data tiles in an index.

Dataset	Engine	Size	Views	Indexing at Pixel Resolution						Indexing at Bin Resolution					
				Mean		Median		P_{95}		Mean		Median		P_{95}	
Weather	Browser	1M	$1 \times 0D$, $6 \times 1D$	0.34		0.33		0.38		0.33		0.33		0.38	
Weather	Browser	3M	$1 \times 0D$, $6 \times 1D$	1.0		1.0		1.1		1.0		1.0		1.1	
Weather	Browser	10M	$1 \times 0D$, $6 \times 1D$	1.2		1.2		1.3		1.2		1.2		1.2	
Flights	Browser	1M	$1 \times 0D$, $4 \times 1D$, $1 \times 2D$	0.29		0.31		0.52		0.30		0.28		0.38	
Flights	Browser	3M	$1 \times 0D$, $4 \times 1D$, $1 \times 2D$	0.92		0.88		1.2		0.95		0.90		1.1	
Flights	Browser	10M	$1 \times 0D$, $4 \times 1D$, $1 \times 2D$	3.0		2.8		3.9		2.8		2.6		3.4	
Flights	OmniSciDB	7M	$1 \times 0D$, $4 \times 1D$, $1 \times 2D$	0.15	0.15	0.13	0.11	0.30	0.13	0.13	0.11	0.13	0.11	0.15	0.13
Flights	OmniSciDB	180M	$1 \times 0D$, $4 \times 1D$, $1 \times 2D$	2.1	0.34	1.7	0.33	3.9	0.47	1.2	0.36	1.3	0.34	1.5	0.46
GAIA	OmniSciDB	1.2B	$1 \times 0D$, $3 \times 1D$, $2 \times 2D$	33.8	1.4	6.5	1.3	94	2.6	5.8	1.0	5.3	0.93	9.5	1.5

Table 5.2: Mean, median, and 95th percentile time in seconds to compute data tiles for all views for different dataset sizes across 5 runs and for pixel resolutions (500 for 1D and 200 × 200 for 2D) and bin resolutions (25 and 25 × 25 bins). Times for OmniSciDB include network roundtrip on a university network when accessing a cloud-based instance. Gray colors show the time until the data tile for the first view is computed (applies only to non-blocking requests).

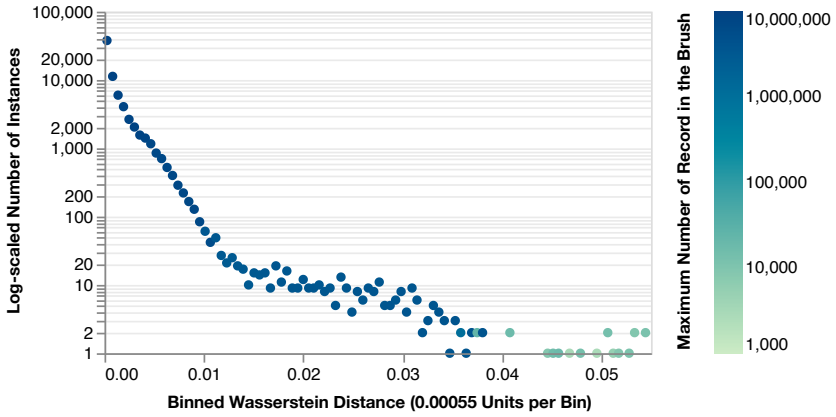

Figure 5.10: Wasserstein distance between the true and interpolated bin counts for various brushes in the flight dataset. Most instances have small distances. Some instances with high selectivity (few tuples remain) have distances over 0.04.

5.5.3 Approximation Error of Interpolated Brushes

To support brushing at pixel resolution even with low-resolution data tiles, users can enable interpolation. In this experiment, we measure interpolation error using the Wasserstein metric (i.e., earth mover's distance) between interpolated and true bin counts. The metric is 1 when the complete mass of a distribution must be moved from one end to the other. We compare interpolated bin counts (from data tiles with 25 bins in the active dimension) in the flight dataset to true bin counts for various systematically enumerated brushes. As Figure 5.10 shows, most of the cases show small errors ($<< 0.01$). The error is largest (> 0.04) for highly selective filters ($< 0.2\%$) since bin counts with few records are more susceptible to noise, and the Wasserstein metric compares two distributions.

5.5.4 Discussion

Our benchmarks show that Falcon delivers constant brushing performance regardless of how many records match the filter defined by a brush. Its update rates are comparable to those of imMens

and outperform those for Square's Crossfilter. Unlike imMens, Falcon supports higher interactive resolutions and multiple brushes. This demonstrates that even though our prefetching methods were designed for client-server applications with massive datasets, precomputing a fixed-size index also benefits client-only applications.

In addition to low latency, constant and predictable performance are important. When a system behaves inconsistently, users may adapt by only performing those interactions that are fast [122]. For the Square Crossfilter application, users might begin to explore only histograms showing few changes. Falcon decouples brushing actions from the full dataset, and all computations have constant complexity with respect to data and brush sizes. Future systems might consider not just the average or worst performance but also the degree to which performance varies for different interactions.

5.5.4.1 In-Browser Engine Performance.

Indexing times for small datasets ($\leq$ 1M records) in Table 5.2 are at most a few hundred milliseconds, about the time needed to hover over a view and begin to brush, so users may not even notice a delay. For datasets up to 10M records, indexing times in our browser engine are at most a few seconds, as shown in Table 5.2. After these few seconds, users can brush without delays, a trade-off not available in previous systems [98, 119, 123].

Since computing a low-resolution index in the browser is not significantly faster than computing a high-resolution one, we do not use progressive interaction here. Although a high-resolution index has more bins, the vast majority of time is spent iterating over the full dataset. The index has the same size regardless of the full dataset's size.

5.5.4.2 Database Engine Performance.

For datasets that are too large to be loaded and processed in the browser, Falcon can issue queries to a database system. Our prototype uses OmniSciDB, one of the fastest analytics database systems available [120]. Our benchmark evaluation shows that the time to compute an index for the dataset

of all commercial flights in the U.S. (180M records) never exceeds a few seconds. However, for the GAIA dataset, computing a high-resolution index can take more than a minute.

A significant amount of time is spent receiving and deserializing database query results. Some of this overhead results from limitations in the OmniSci API. For instance, aggregated counts must be sent as a relational table instead of as a dense multidimensional array. Nevertheless, while the index is loading, users can already be drawing a brush in the active view.

To further improve the user experience, Falcon leverages two observations from the benchmark. First, loading the initial data tiles takes about a second. Thus, Falcon's UI can update individual passive views as soon as their data tiles have been loaded and provide visual feedback. Second, low-resolution indexes load much faster using the database-backed engine. Our progressive interaction method leverages these faster query times to update passive views faster—improving average initial load times by 5×, from ~30s to ~6s—and then progressively improves them as high-resolution indexes are loaded. Our experience shows that view switching times are reasonable, but more careful assessment of how these delays affect people's behavior remains as future work.

Falcon's interpolation of high-resolution interactions from low-resolution data tiles enables progressive interactions without snapping brushes to the low resolution. Our measurements in Figure 5.10 show that the interpolation error is negligible in most cases. The largest errors occur when brush filters are highly selective. Since Falcon visualized bin counts by default on the scale of unfiltered data (Figure 5.3), the filtered bin counts and any visual differences in the chart are small. Moreover, when filters are highly selective and visualizations of aggregates are based on few records, the visual gestalt of the chart is susceptible to noise in the data. In general, analysts should make judgments about distributions based only on large samples.

5.6 Limitations and Future Work

Falcon's index is significantly smaller than that in previous approaches (e.g., [123]), allowing it to be computed on the fly from a single scan by the backing database under appropriate latency assumptions. To support datasets that are too large to be scanned within a given latency threshold, Falcon uses existing databases to compute the necessary aggregates; it can take advantage of approximation and sampling techniques from the database literature. However, our prototype does not yet apply these approximation techniques.

For binned aggregate plots, the data necessary to render visualizations depends only on pixel resolution and not data size. This common assumption enables visualizations whose visual complexity is invariant to the size of the full dataset. Thus, Falcon does not support non-aggregated views where each record is rendered as a separate mark. Future work might involve separate marks for outliers. Our prototype system implements aggregate visualizations of the counts with zero-, one-, and two-dimensional grouping by bins. We have not yet implemented grouping by categorical dimensions, but we plan to add them.

Falcon assumes that a user interacts with a single view at a time. On a desktop computer with a mouse, this assumption is trivially met. However, on touch-enabled devices, a user could use both hands to modify multiple brushes simultaneously. We believe that this scenario is rare. We conducted informal user observations with an iPad, and none of the participants attempted to use simultaneous brushes. Nonetheless, a future version of Falcon could support simultaneous brushes by combining the dimensions of multiple active views, though at the cost of much larger data tiles.

While Falcon prioritizes brushing and linking as the most latency-sensitive interactions [122], future systems should use techniques presented here to prioritize zooming or other interactions. By prefetching data at different zoom levels, a system could support continuous zooming and re-binning. Battle et al. [12] demonstrate prefetching techniques for panning interactions and show that prediction models can help prioritize which data to prefetch. This was not necessary in

Falcon since the computation of data for one brush or for all brushes both require one pass over the data. However, future systems could use prediction models and prefetch in multiple interaction vectors: e.g., linked brushing, linked selection, zooming, and panning.

The Falcon system does not take advantage of concurrent queries: the in-browser engine is written in JavaScript and thus single threaded, while OmniSciDB executes queries sequentially. Future system iterations could speculatively precompute indexes for interactions with a non-active view. The cost of such aggressive prefetching could be offset by caching results in a middleware layer, possibly even for other users. The middleware could also leverage structure in the data tiles to compress them. Neighboring cells in a data tile often have similar values (see Figure 5.7, subsection 5.5.3), which is similar to images or videos. The large body of work on perception-aware image and video compression could be applied to compressing data tiles between the server and client. Compression could significantly reduce the time needed to transfer a Falcon index from server to client.

To support constant latency for brushing interactions, we limit Falcon to summable aggregate functions (e.g., sum and count). Our prototype only implements count. Some aggregate functions are algebraic, meaning they can be constructed as a combination of summable functions; this includes the mean (sum, count) and variance (sum, count, sum of squares). Distributive functions (e.g., min and max) can be computed by iterating over matching bins, which results in a linear (or, with extra data structures, logarithmic) lookup time with respect to brush size. Future iterations of Falcon could implement these aggregate functions.

5.7 Conclusion

In this chapter, we contribute the idea of prioritizing brushing latency over view switching latency, as suggested by prior work on the impact of latency on analysts' behavior. We also show that it is possible to lower the initial resolution of interactions to improve view switching times. We implement these methods in Falcon, our prototype system. Falcon supports brushing and linking

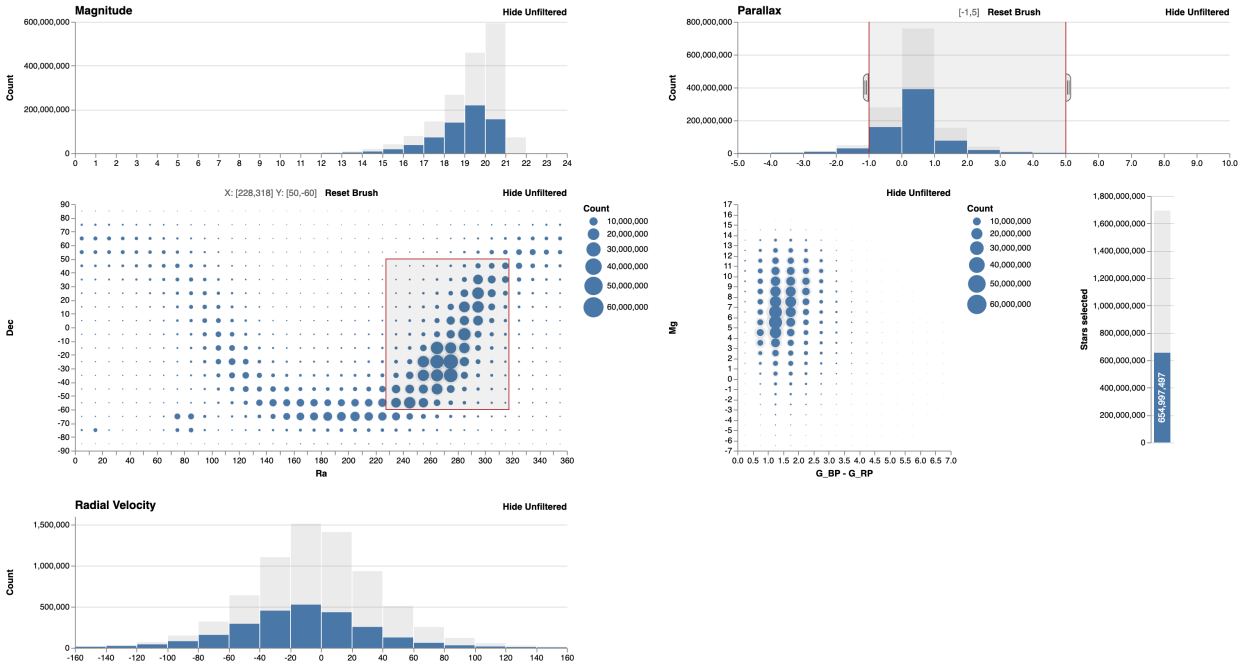

Figure 5.11: The GAIA [22] dataset loaded into Falcon. The GAIA spacecraft measures the positions and distances of about 1.7 billion objects, consisting of stars, planets, comets, asteroids, and quasars.

across views over datasets of tens of millions of records in the browser and billions of records when connected to a backing database system (Figure 5.11), without the need for costly precomputations or significant limitations to supported interactions.

6 Optimistic Visualizations of Approximate Queries

Prefetching and indexing techniques, like the ones implemented in Falcon (chapter 5), work best when queries take a few seconds. However, for petabyte-scale datasets, even scanning a large dataset may take minutes and inhibit interactive exploration. In this situation, users are given a choice: they can wait for the system to complete long-running queries or rely on an approximation based on a sample of the data. While attractive, approximate values can be—by their nature—incorrect. In exploratory visualization, an analyst might see dozens of visualizations; they are almost guaranteed to encounter a visualization where the errors are outside the predicted bounds. In this chapter, we propose to address this issue with *optimistic visualization*. In an optimistic visualization system, an analyst begins by constructing a fast, approximate query. Computation of the precise results is placed in the background, allowing analysts to continue their exploration without watching for updates. When the query is complete, the system invites the analyst to verify their observations. Pangloss implements these ideas. We discuss design issues raised by optimistic visualization systems. A laboratory study and three case studies at Microsoft show that this method can meet analysts' needs for both speed and precise results. Optimistic visualization gives people confidence in working with approximate results and paves the way towards broader adoption of approximate methods in exploratory analysts.

We published optimistic visualization and Pangloss at ACM CHI 2017 (co-authored with Danyel Fisher, Bolin Ding, and Chi Wang) [134]. We discuss design implications for data management systems in our HILDA 2017 paper (co-authored with Danyel Fisher) [133].

6.1 Introduction

Data analysts want to be able to derive insights from increasingly large datasets. Exploratory visualization, however, runs into an obstacle where the scale of the data is sufficiently large that a screen cannot render each point, and where database queries would take a long time to return. By sampling the dataset, though, we can create a visualization with approximate values in interactive time. This is known as *Approximate Query Processing* (AQP).

There are several well-known challenges with approximations. The most critical of these is trust: approximate values can be, by their nature, possibly incorrect. In an exploratory visualization, an analyst might see dozens of visualizations that are accurate 95% of the time. Can an analyst trust an approximation with a business-critical decision?

In this chapter, rather than addressing the problems with AQP from an algorithmic or systems perspective, we formulate them as user experience problems. What user experience would enable analysts to gain the benefits of approximate queries, while still being able to trust the results?

We propose an approach which we call *optimistic visualization*. Optimistic visualization produces approximate results quickly, and computes precise results in the background. The analyst can make observations on the approximation, and later check them against the precise results.

We call this approach "optimistic" because the analyst expects the approximation to be very close to the precise value; in those rare cases when there is a significant difference between the approximate and precise results, the analyst can decide which parts of the exploration have to be redone. Optimism provides a way to detect and recover from errors, and so increases confidence in

working with samples. The technique can be combined with other approximation techniques such as confidence intervals and online aggregation.

We present *Pangloss*, an optimistic visualization tool based on AQP. With it, analysts can rapidly explore very large multidimensional datasets by grouping, aggregating, and filtering. Working in a sample-based system affects the user experience of visual data exploration. We describe the design decisions that went into the system. We validated our decisions by running a user study with five participants exploring a 170 million row dataset about flight delays; we then deployed our prototype system to three data scientists using their own data.

6.2 Background and Related Work

The concept of optimistic visualization builds on past research on data exploration, AQP, and uncertainty visualization. We first discuss the importance of rapid iteration in exploratory data analysis, and then the ways in which it changes when working with very large datasets. Last, we focus specifically on sample-based queries and visualization.

6.2.1 Exploratory Visualization

Exploratory data analysis (EDA), a term coined by Tukey [208], is broadly understood as a process of examining multi-dimensional data by looking at the distributions and correlations of fields. As Card et al. [27] note, this process prizes iteration and speed of exploration; an analyst might look at dozens or hundreds of graphs as they get to know the data.

The process of moving through these dimensions is iterative. An analyst begins with a broad question, and creates views that address some part of it. This view can inform a more-specific question, leading them to create another view to address that question [160, 196, 224]. These increasingly-specific questions require analysts to change representations, to filter the data by zooming or filtering views, and to choose new fields to explore. Some of these views will contain interesting insights; others will be dead ends with less value. When the analyst has sufficiently

addressed the broad question and follow-up questions, they often continue with a new broad question and chains of specific follow-up questions.

Visualization tools—whether point-and-click, such as Tableau (an extension of Polaris [198]) and PowerBI, or programming, like Matplotlib—support this process with tools that allow users to rapidly specify and refine their visualizations.

Each step in this process involves generating *observations* of the data. Optimistic visualization helps analysts confirm (or challenge) their observations. An observation is a single fact about the data; it is the unit of knowledge that allows an analyst to move on to the next step of their analysis [229]. For example, when examining a dataset of flight data, an observation might be "Delta is the airline with the most flights in the dataset." It is a more modest unit than the insights that the research community has focused on as the outcome of the analysis process. An insight can bring in external context and the results of some queries; an example might be "the biggest airlines have trouble with congestion near the holidays, while smaller airlines do not" [145, 229].

The visualization system must be fast enough to enable iteration. Liu and Heer show that analysts lose effectiveness when a result takes more than 500 ms to return [122]; Nielsen argues that when a computer operation takes more than a second, users lose their flow of thought [142].

6.2.2 Big Data Visualization

These requirements for responsiveness become urgent when dealing with large datasets. As Fisher [60] and Godfrey et al. [77] outline, when dataset sizes exceed even a few million records, analysts run into two fundamental issues: visual scalability and data processing scalability.

It is impractical to display every element of a large dataset: the number of records may far exceed the available pixels. For example, drawing raw data in a scatterplot without aggregation leads to overplotting—drawing many points in the same place—and visual clutter. The data can be grouped by a dimension, however, and a single aggregate measure computed for each group. The simplest

such aggregate visualization is a bar chart, in which each bar represents the aggregated value of a group. Elmqvist [54] outlines visualizations that work with aggregate data; Wickham proposes the bin-summarize-smooth framework [218] as a general strategy for visualizing big data.

Data retrieval and processing are the other major bottleneck. Handling very large datasets can be comparatively slow. Analysts sometimes resort to an offline process: formulating and submitting a query, waiting for a result, and then formulating a follow-up question. This is not only frustrating but requires analysts to carefully design their queries to be worth the wait and the resources.

There are three major technologies to achieve more-responsive queries. Data cubes precompute and store partially-aggregated results; at query time, the system can assemble these partial answers quickly [79, 119, 123]. Unfortunately, these cubes require a designer to select the fields to optimize. Second, the query can be spread across many computers, which assemble an answer [24, 157]. In these distributed *Online Analytical Processing* (OLAP) systems [34], though, network latencies can last into the seconds. The third major approach is to sample the dataset.

6.2.3 Approximate Query Processing

Optimistic visualization is based on the technique of Approximate Query Processing (AQP). In AQP, the tool uses a representative subset, or sample, of the data; the goal is to look at less data more quickly. Tools can estimate the true value of an aggregation function based on that sample. As a simple example, we can approximate the sum of a set of values by computing the sum of 10% of the values and then estimating the true sum to be ten times the aggregate value of the sample. This value is an estimate, and carries some uncertainty, which can be expressed as error bounds. Those bounds widen with the variance of the data, and narrow with the square root of the size of the sample.

Some tools create a sample of the data before the user begins their analysis. In these systems, the precision of the approximation greatly diminishes as the analyst filters away more records. For example, every record in a census can help the approximation for “average age”, but far fewer will

be helpful for "average age of unemployed men living in Aspen, Colorado". Choosing a sample that is large enough to lead to statistically meaningful results while maintaining interactive response times is important.

Sampling can be integrated directly into databases [147]; other systems build on different sampling and estimation methods [3, 50, 108]. These systems pick a sample and compute a result along with estimated error bounds; the analyst may choose either a maximum amount of time that the query runs, or desired error bounds. Interactive systems tend to use time bounds to get a best-effort approximation within that time bound.

6.2.4 Progressive Visualization with Online Aggregation

Rather than forcing the user to settle for a fixed-size sample, or wait for the system to reach a fixed level of precision, Online Aggregation (OLA) picks ever-growing samples and displays results to the user; when the analyst determines the visualization has tight-enough bounds, they can end the process. OLA computes aggregations and confidence intervals and returns them to the user. Hellerstein et al. [93] first suggested the idea; it has been adopted by the visualization community as a "progressive analytics" approach [57, 62, 158, 197, 210].

Optimistic visualization can be seen as an asynchronous form of progressive sampling; it places the updates in the background, allowing the analyst to continue their analysis without watching for updates. The CONTROL project [92] noted that progressiveness adds costs; e.g., "ripple joins" [82] require multiple passes over the data, and so may take longer to reach a precise result. Pangloss can use existing, highly-optimized database systems for the precise results.

6.2.5 Visualizing Approximations

There are many techniques for communicating approximate query results [149]. Commonly used methods are confidence intervals [141], visualizing distributions [220], or visualizing possible

instances of the underlying statistical model [97]. Researchers are less certain of ways to visualize uncertainty on some visualization types such as heatmaps.

Even with the help of visualization, users struggle to correctly interpret uncertainty and can draw incorrect conclusions [46, 105]. In a visual data exploration where an analyst creates tens or hundreds of visualizations, the "rare" cases when the true values are outside the estimated bounds become likely. Worse, many confidence estimates are inaccurate: Agarwal et al. examined logs of 70,000 approximate queries from Facebook and found a large fraction had error estimates that were too wide or too narrow [2].

Optimistic visualization ensures that precise results will eventually be available so that the analyst can discover places where estimates were unavailable, hard to understand, or far from the true value.

6.3 Optimistic Data Visualization

We address the challenges of sample-based visualization with *optimistic visualization*. In an optimistic visualization system, an analyst begins by constructing a fast, approximate query and seeing results instantly. The analyst may choose to *remember* that query, in which case it will run in the background. While the query is running, the analyst continues their exploration. The system allows the analyst to know when the query is complete; at which point the analyst can verify their observations. The system shows the error between the approximate and precise views.

In most cases, the final result validates their earlier work. Should a past approximation turn out to be inaccurate, however, the analyst must then re-evaluate how much exploration must be redone. There are many edge cases for approximation: it is difficult to choose good confidence intervals on some functions, such as percentile measures; some datasets have high enough variances that confidence intervals are extremely wide; and some approximations turn out to be incorrect.

Even in these riskier scenarios, optimistic visualization allows the analyst to feel certain that their final results will confirm whether their assumptions were justified. Without optimism the analyst can only rely on the *uncertainty*—the estimated error of the approximation—to make a decision. The uncertainty should be a good predictor of the *approximation error*; the true error of the approximation. However, only the precise results can confirm this; analysts have to know when the approximation error was significantly larger than the uncertainty.

In progressive visualization systems [62, 158, 210] analysts watch progressive updates as more data arrives. During the waiting period, though, values continuously change [58]. Analysts can be distracted by these *dancing bars*; they can also incorrectly assess trends as the confidence intervals converge. Progressive systems require accurate communication of uncertainties so analysts can decide when to stop. However, some forms of uncertainty are difficult to visualize, and true errors can be outside the estimated bounds. Optimistic visualization defers the confirmation and moves the computation of the precise result into the background; analyst can continue their exploration sooner.

An optimistic visualization is most effective when the complete query takes long enough to get in the way of interactivity, but shorter than an analysis session—it is highly effective when a query takes several minutes to return.

6.4 The Pangloss System

We implemented optimistic visualization in the Pangloss system. Pangloss is a web-based UI that queries Sample+Seek [50], an AQP system. Because Sample+Seek has some unique features, we discuss it in some detail before presenting our interface.

6.4.1 Sample+Seek for Approximate Query Processing

Sample+Seek [50] is designed to be highly responsive for aggregate queries on a single table. It incrementally loads more records into the sample until either the uncertainty bound is lower than

a predefined threshold or until a timeout. Instead of uniformly sampling records—as many other AQP systems do—Sample+Seek uses *measure-biased* sampling, a method that biases sampling according to the aggregation measure. Measure-biased sampling has a tremendous advantage over uniform sampling: fewer samples are necessary for the same accuracy in a visualization. This sampling method has been developed to optimize the *distribution uncertainty*. It is a metric of uncertainty across all groups in the result, and is defined as the expected distance (e.g., sum of distances, Euclidean, etc.) between the normalized distributions of the approximate answer and the precise one. For example, the Euclidean distance between the normalized distribution answer $x = \langle 0.39, 0.61 \rangle$ and the approximation $\hat{x} = \langle 0.40, 0.60 \rangle$ is $||x - \hat{x}||_2 = \sqrt{0.01^2 + 0.01^2} = 0.014$.

More general, the distribution uncertainty with Euclidean distance is [50]:

$$\sqrt{\sum_{\text{group } i} \left(\frac{\text{group } i\text{'s value}}{\text{total group value}} - \frac{\text{estimated group } i\text{'s value}}{\text{total estimated group value}} \right)^2}$$

Distribution uncertainty is different from familiar per-group confidence intervals [62, 92, 97]. Rather than seeing each group as having its own confidence interval, the distribution uncertainty is a total amount by which the whole visualization is likely to be imprecise. Distribution uncertainty recognizes that uncertainties are not independent. As Figure 6.1 illustrates, one group might be off by a lot, or many groups might be off by just a little.

Besides the distribution uncertainty, we also compute a confidence interval for each group. However, the sum of these per-group uncertainties are worst-case estimates, and can be significantly higher than the overall distribution uncertainty. When we visualize confidence intervals, the possible range of uncertainty they imply is far greater than the distribution uncertainty (Figure 6.1). For the analyst, knowing the overall distribution uncertainty means that they do not have to expect the worst case for all groups.

Sample+Seek supports aggregate measures such as count, sum, and average. Queries can have multiple group-by dimensions, of either categorical values or binned numerical values. Queries can also filter the data, based on Boolean predicates. When predicates are selective, AQP systems need

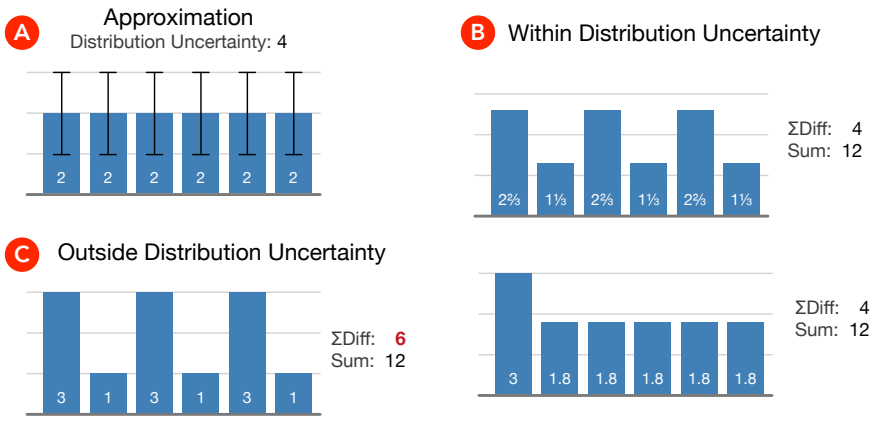

Figure 6.1: The uncertainty implied by the confidence intervals from (A) is larger than the distribution uncertainty of 4 (here for illustration defined as the sum of absolute, unnormalized differences). (B) are possible instances that stay within both the confidence intervals and the distribution uncertainty. The values for all groups in (C) are within the confidence intervals but the distribution is off by 6. The sum of the values is always 12.

to look at many records before they find records that match the filters. Sample+Seek maintains indices that help rapidly identify matching records. Each query uses a different sample because we limit the time per query.

In the algorithm a sample is always a strict subset of all rows. Even if we scan all rows, we can only get within a fixed factor of the precise answer. This prevents us from providing users with progressively improving results.

In the original Sample+Seek [50], data samples were kept in memory; Pangloss uses a modified version that supports larger datasets and reduces the memory footprint by keeping the dataset as a randomly shuffled file on disk. Our version of Sample+Seek can respond within 100 ms with acceptable levels of approximation error.

6.4.2 Designing the Pangloss UI

Sample-based visualizations require a different user experience than more traditional visualization systems. In this section, we discuss the Pangloss user interface, highlighting design decisions that accommodate the unusual aspects of AQP and optimistic visualization. The interface is implemented as a web application, using the React framework and D3 [18].

The interface, shown in Figure 6.2, uses interaction paradigms well-known in visualization tools such as Tableau and PowerBI. On the left is a searchable schema (A) with fields that can be dropped to the chart specification (B). Below the chart specification form are fields for filtering (C) and showing the current zoom predicates (D). The largest area of the screen is taken up by the view (E) and its uncertainty (F). Above the view is a textbox for observations and a button to *remember* the current view (G), which computes the precise result for this view. Remembered views are listed in the history on the right; they are drawn as orange while they are still loading, and turn blue when precise result is available (H).

6.4.2.1 Approximate Visualizations

Pangloss supports two core visualization types: bar charts and heatmaps. The bar chart shows aggregated measures grouped by values; it can also be used as a histogram by binning numeric fields. The heatmap, a generalization of density plot, allows users to see the interaction between two dimensions; aggregate values are encoded using a color scale. Each dimension of the heatmap can be binned. Other visualizations can be implemented in this system; in theory, any aggregation-oriented visualization [54] can be accommodated.

Because many queries have long tails, Pangloss limits the number of bars or cells that can be shown to a top k and shows a warning if groups are hidden (Figure 6.3-A). When dealing with samples, the values further down the tail are based on fewer samples, and so are likely to be very uncertain; we chose not to add tools to scroll over to the tail of the distribution. Of course, an analyst can filter the highest values, working their way down the distribution.

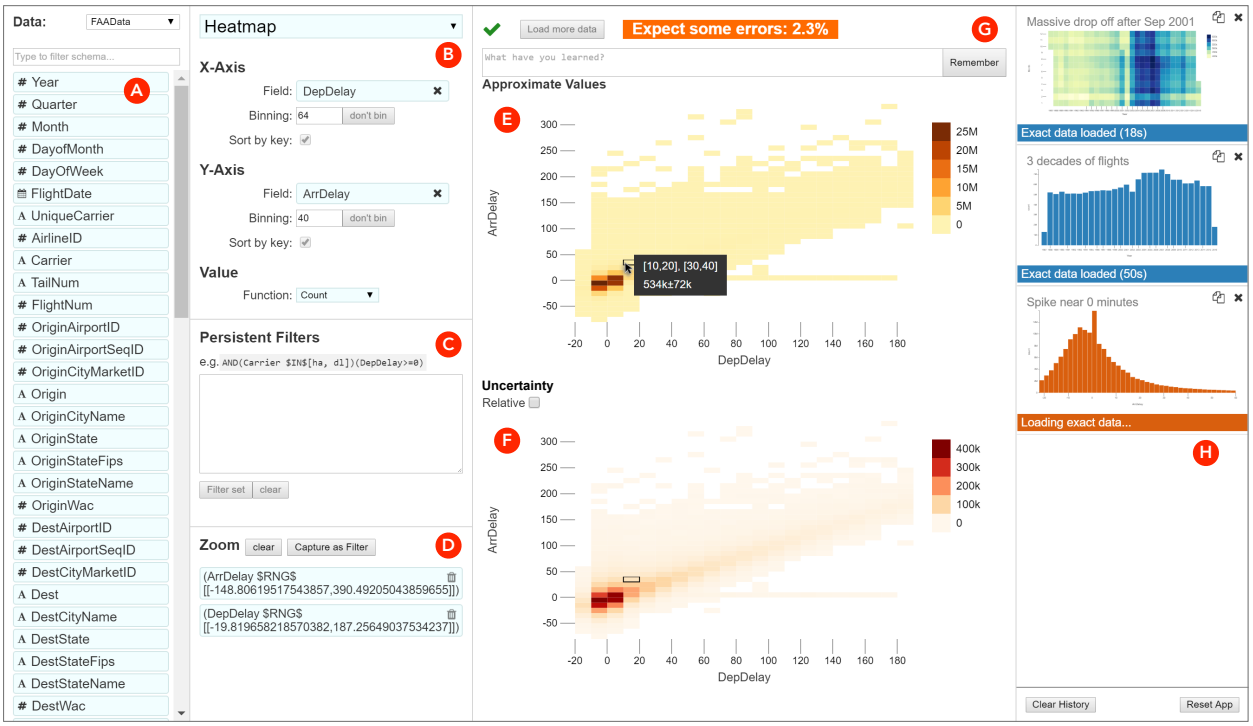

Figure 6.2: The Pangloss UI, exploring a flight delay dataset, with a list of fields (A), chart specification forms (B), a textfield for filters (C), zoom specification (D), approximate visualization (E), visualization of uncertainty (F), field for annotations and "remember" button (G), and a list of views in the history (H). Two precise results are ready, while a third is loading.

The distribution uncertainty of the approximation is displayed above the main view (Figure 6.3-B). The system computes confidence intervals for each group; however, these per-group intervals are worst-case estimates (Figure 6.1). For bar charts, Pangloss draws the confidence intervals directly on the bar chart. As there is no standard way to show confidence intervals for heatmaps, Pangloss instead displays a second parallel heatmap that shows uncertainty (Figure 6.8, right). In all visualizations, tooltips show the group name or bin bounds, the approximate value, and the uncertainty (Figure 6.4).

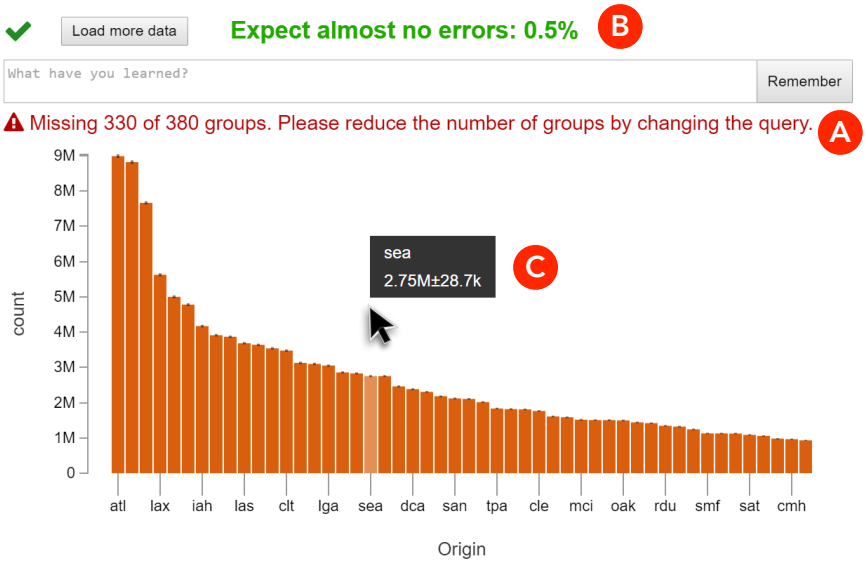

Figure 6.3: A bar chart for a result with a long tail. The view warns that it only shows the top groups (A). Above the chart is the distribution uncertainty (B). Tooltips in bar charts are shown for the area above the bar (C).

Figure 6.4: Left, tooltips for approximations show the group, the value, and the associated uncertainty. Right, tooltips for precise results show how much the estimate was off.

6.4.2.2 Zooming with Samples

Pangloss supports zooming and filtering, like most visualization systems. In in-memory visualization tools, zooming and focusing only change the domain of the dimensions and measures; the group categories stay constant. With samples, every zoom focus interaction changes the predi-

cate, forcing a new query to run on the AQP system. Consequently, the aggregate value and the uncertainty can change.

As we noted above, the groups themselves can change as the user adds filters: collecting a new sample might mean that numerical ranges might expand, and new groups might appear. The semantics of filtering, then, call for a design decision. An analyst cannot know whether there are more groups to be seen until they filter. If the analyst filters ten categories down to three, do we interpret that as a *negative* filter, removing seven, or a *positive* one, limiting to just those three? The difference is that "removing seven" might discover more groups (Figure 6.5).

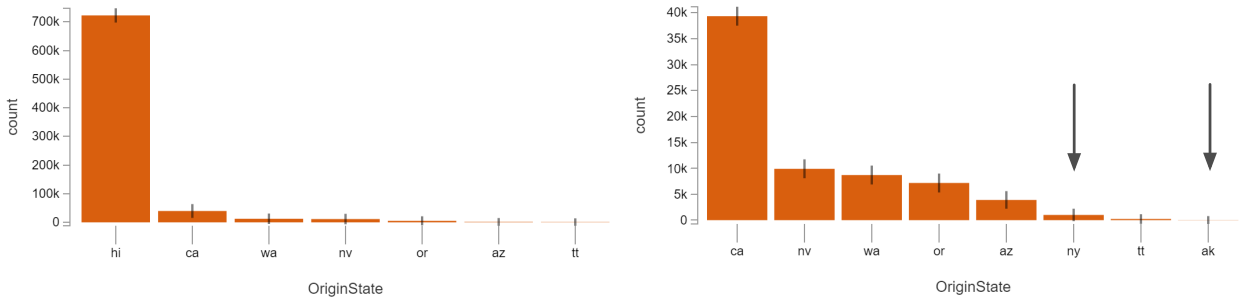

Figure 6.5: Left, an approximate histogram of origin states for Hawaiian Airlines flights. Right, if we filter out Hawaii, a new query runs on the AQP system and the approximation also shows that New York and Alaska (arrows) are also origin states. Because the new predicate is more selective, the uncertainty decreases for all groups.

To ensure that analysts never lose information, Pangloss treats categorical filters as negative by default; analysts can explicitly select positive filters if they need. Similarly, with numerical data, the domain can change (Figure 6.6); we add an inequality constraint (as opposed to a range predicate) when the user brushes all the way to the end of an axis.

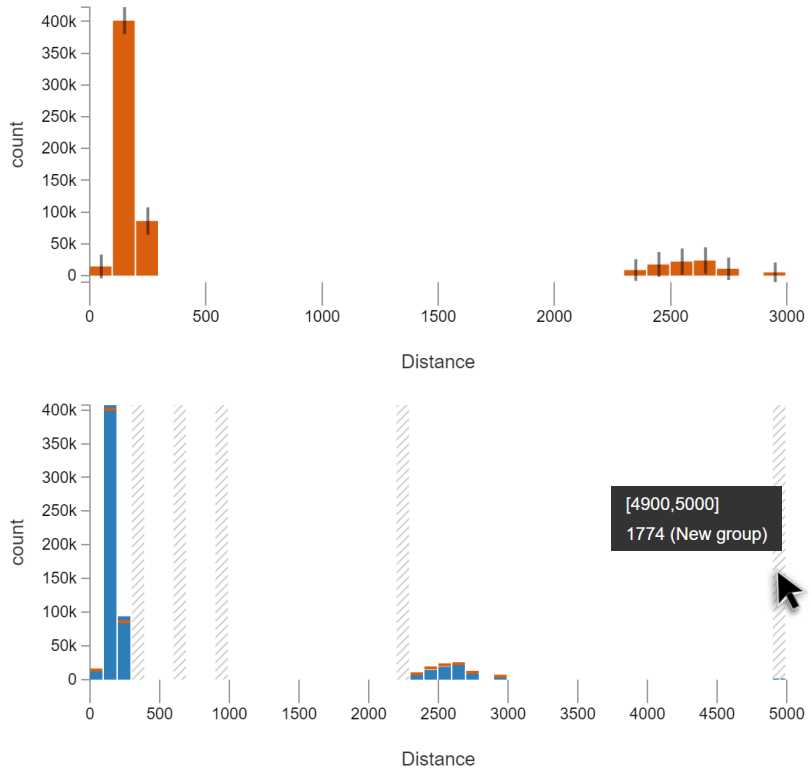

Figure 6.6: Approximate histogram at the top shows that Hawaiian Airlines has short range (inter state) and long range (island to mainland) flights. The precise histogram below shows additional flights around 5000 miles. The range has changed but bins are still aligned.

6.4.2.3 Functions and Transformations with Samples

In most visualization systems, the user can carry out transformations and functions on the data. For example, it is easy to add a logarithmic transformation over a visualization, or to compute average by dividing sum by count. Distribution uncertainty, however, is a global measure across a view. As such, the distribution uncertainty for a value will be very different from the log of that same measure value; they will be based on different numbers of samples. Just as zooms must be computed as separate computed queries, so too must be transforms and functions.

Many existing visualization systems and techniques build on assumptions that are not true when samples are used to approximate results. With samples, we cannot assume that the result of an aggregation query does not miss groups. Consequently, if we calculate the average of a measure as the ratio of the sum and the count, we can only do so for the common groups in the two samples. We cannot, however, compute the combined distribution uncertainty.

6.4.3 Remembering Views and History

The heart of optimistic visualization is the process of selecting a view and re-running its query to get a precise result. In Pangloss, we call this *remembering* the view. We wish to support analysts being able to look back at a past view, and verify the observation they made with it.

One important design decision is which views should be remembered. We considered verifying every past view that Pangloss produces, modeled after Graphical Histories [90]. There are several disadvantages to keeping this complete history. First, we expect users to review the precise views; it would be overwhelming to review every view. Second, precise queries are computationally expensive; issuing hundreds of them can overwhelm back-end data systems. As such, we want to encourage users to remember only views that are relevant for observations.

In Pangloss, we decided to make this an explicit process. The "Remember" button (Figure 6.2-G) stores the view in the history and runs the precise query in the background. We would like to encourage analysts to track the observations; we support it by allowing them to add a small textual annotation describing the remembered view.

The entries in the history (Figure 6.2-H) change when a precise result is available: we render approximate views in orange shades, and precise views in blue. Views in the history are immutable, but users can revise their annotations to note new information. In addition, users can make a mutable (and approximate) copy of the view if they wish to modify it.

Queries for precise answers are sent to a commercial SQLServer database. On current hardware precise queries return within 30 seconds to a few minutes for datasets of around 50 GB to 1 TB.

6.4.4 Visualizing Approximation Error

Once the precise query has completed, an analyst must be able to verify whether their observation during the exploration was justified. Several types of changes might occur between the approximate and the precise views. First, the values of some groups can change; if the chart is sorted, this also means that bars might be in a new order. Second, some groups that were not in the chart before might have been added. Last, binned data might change its range.

We wish to support the analyst in comparing the approximate and precise views.

What should happen when a bar is added, or the relative order of sorted bars changes? There are advantages to both maintaining *stability*, by keeping the layout of the approximate visualization with revised values, or *precision* by showing the precise view. We settled on the latter, because our major goal is encouraging users to interpret the view they see. The final visualization emphasizes the precise values.

Visualizing the difference and the true values in the same chart poses challenges similar to uncertainty visualizations; we use similar methods. Pangloss superimposes the approximate value on the bars as orange lines, as in Figure 6.7. This makes changes in order very visible, as the orange lines no longer decrease monotonically. When a new group appears, we highlight it with a gray striped background (Figure 6.9, right). It is worth noting that a histogram's range might change between sample and final. Therefore, in the precise query, we fix the bin width and the offset so that the precise histogram always aligns with the approximate one (Figure 6.6).

For heatmaps, one chart shows the true value, while the approximation error is shown in a separate chart (Figure 6.8). The analyst can toggle between seeing absolute error, which shows the difference between the estimate and the actual value, and relative error, measured as a percentage. We found

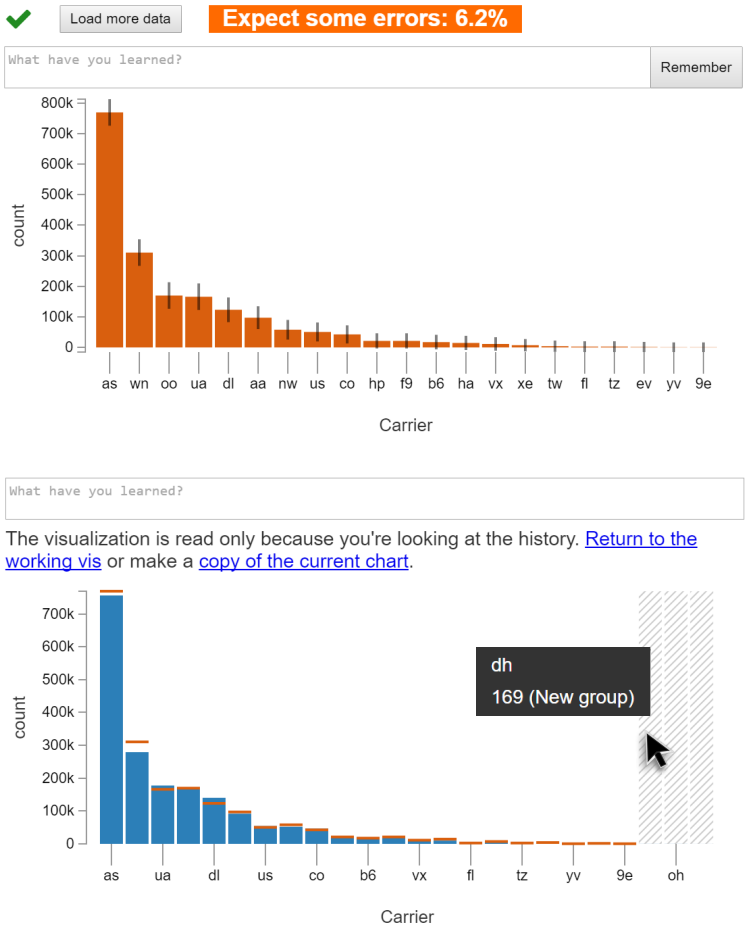

Figure 6.7: Above, approximate bar chart for count of flights out of Washington state, grouped by airline, with per-group confidence intervals and distribution uncertainty. Below, the precise result for the same chart shows the values as blue bars, the approximate result as orange lines, and highlights airlines that were missing from the approximation (e.g., Independence Air (dh) with 169 flights).

that each is useful for different circumstances: relative error is good when cells have similar values; on the other hand, a small amount of error can still be thousands of percentage points from a small cell value. Consistent with how new bars are highlighted, any new cells have a diagonally striped pattern (Figure 6.9, left).

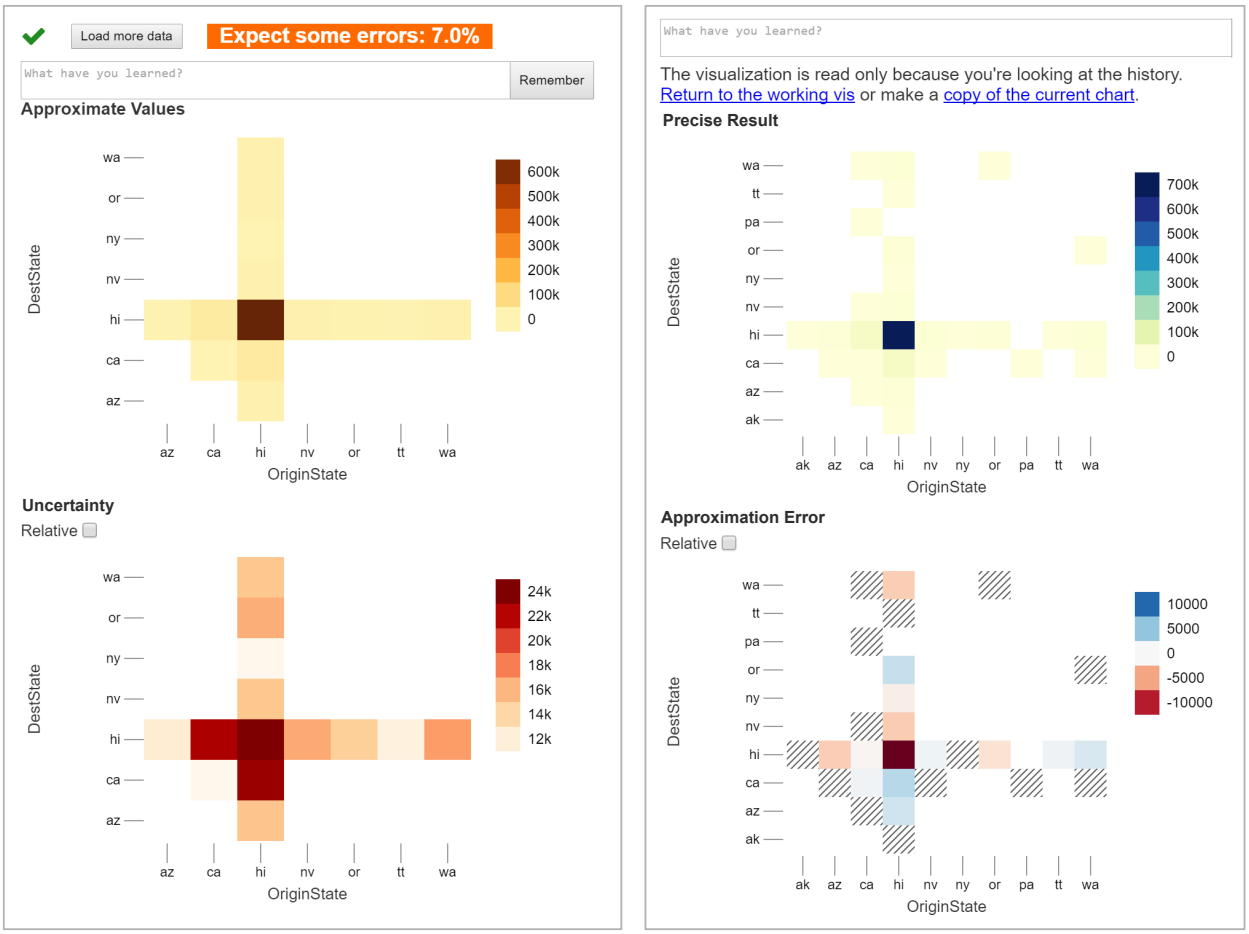

Figure 6.8: Approximate (left) and precise (right) heatmaps, examining origin and destination states for Hawaiian Airlines. (Left) The approximate view shows the estimated count above and the uncertainty below. (Right) The precise view shows the count above and the approximation error below. Both lower images toggle to show relative or absolute differences.

6.5 User Studies

There are several motivating concepts behind Pangloss that we wished to validate. First, are analysts comfortable with incomplete or inaccurate results, and are they willing to use them to explore approximate data? Progressive visualization systems [62, 158, 210] allow analysts to linger

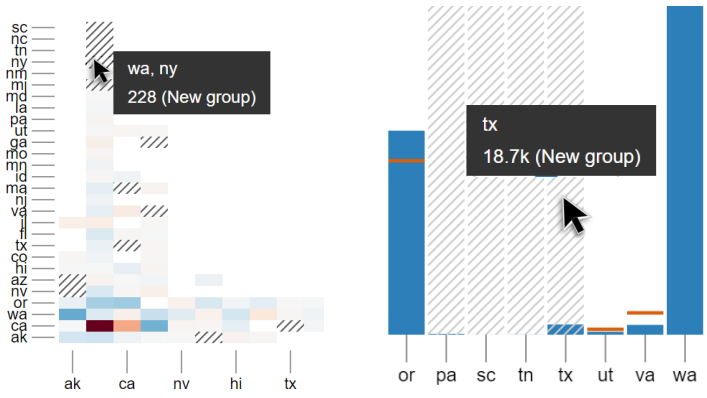

Figure 6.9: To draw attention to new groups, the corresponding cells in heatmaps (left) and bars (right) are highlighted with a stripe pattern.

at one view until it reaches a level of accuracy; Pangloss does not. Second, optimistic visualization expects users to proceed with their exploration even without precise results; we wanted to know whether they would do so. Last, we wanted to know how users would interact with precise results, and whether checking those results would interfere with their flow.

We believed these questions would be best addressed by collecting rich stories of users interacting with the system; we modeled our user study after Fisher et al. [62]. We carried out two studies. We first wished to establish that Pangloss works as a data analytics toolkit that can enable users to come up with usable insights. We chose a single dataset and recruited data analysts from within Microsoft. We encouraged them to explore the data, providing them with guiding questions. The shared dataset allows us to factor out feedback specific to the data.

We wanted to also validate that this system works for real-world situations. We solicited data scientists from Microsoft, and asked them to share a current dataset with us; we loaded it into Pangloss. We then invited them to explore the system and interviewed them about their experience.

6.5.1 Flight Delay Study

Our first study used the "BTS Flight Delays dataset" [205]. The full dataset is 70 GB, and contains records on about 170 million commercial domestic flights from the last three decades within the United States. Each record represents one flight, with its arrival and departure time and location, as well any delays and reroutes in flight. There are 109 fields in the raw data; for the user study, we removed some sparser fields, to result in 78 fields. Pangloss can maintain the 100 ms response time for histograms and bar charts at a 1%–2% uncertainty level; heatmaps had higher uncertainty of 2%–5%. Views with averages and highly selective predicates have much larger errors. In contrast, a full SQL query on the dataset runs in one minute.

We recruited five data scientists for the study. All participants were associated with a product or consulting team, including IT Support, software development, and post-deployment monitoring. Each of them was familiar with creating visualizations in R, PowerBI, Tableau, or Excel. We call them P1 through P5 below.

Three of the sessions were carried out in person; two others, with analysts located further away, were carried out by video-sharing session. At Microsoft, it is not unusual to meet via video-sharing; all users were familiar with the technology. Sessions lasted an hour. We started sessions with a tutorial, which guided subjects through a series of training questions to help them learn the system. We then invited them to explore the data using the tool; we gave some introductory questions but invited them to pursue questions that caught their curiosity for half an hour. At the end of the time, if the users had not reviewed precise results, we encouraged them to do so. Near the end of the hour, we asked our subjects to discuss advantages and trade-offs of Pangloss.

We encouraged users to think aloud; all sessions were voice- and screen-recorded. Remote users used the tool in their web browser, so that they did not suffer from video-sharing lag.

6.5.1.1 Flight Delay Study Results

During the study, most users were resistant, at first, to explicitly recording their observations: we needed to remind almost all users to record observations they were making. After the first few, however, it became more habitual for them to record their observations over time. Only one user refused to use the "remember" function.

We had feared that some users would be unwilling to explore their data, knowing that their initial results were inaccurate; perhaps they would wait to ensure their earlier investigation was valid. We saw this only once; P2, mid-analysis, paused to wait for his remembered view to complete. A moment later, impatient, he resumed his flow, and continued to "remember" things. By the end of the study, four of our users were regularly "remembering" visualizations. Users "remembered" 4-7 views during their half hour.

Some users found opportunities to check in on their history during the study. P4 said, "I was thinking what to do next—and I saw that it had loaded, so I went back and checked it . . . [the passive update is] very nice for not interrupting your workflow."

Interacting with Big Data: All users had dealt with slow queries in big data systems, and appreciated the speed of Pangloss. As P4 said, "[with a competitor] I was willing to wait 70-80 seconds. It wasn't ideally interactive, but it meant I was looking at all the data." All the users commented on the responsiveness of the system.

Our users were also familiar with sampling; for example, P4 had used sampling for other projects: "We can't look at all the sensor data at once—it wouldn't work. So we sample."

Waiting for Precise Results: We wondered whether analysts would find the precise version of the visualization useful: after all, they had already seen the approximate version. P1 said, "From my perspective, [uncertainty] almost passed me by. Nothing that I saw after-the-fact fundamentally changed any conclusions."

Despite that, this made Pangloss feel safer to him: P1 said he checked whether the precise results "are fundamentally different in a way that warrants my attention—if the top 5 are different, that's important." P5 used precise results to feel more confident in the approximations: "[The precise view] is an enabler. [Sometimes] you have a doubt whether sampling has made it better or worse, but seeing something right away at first glimpse is really great."

Our participants most often decided to remember very uncertain results: the large distribution uncertainty and wide confidence intervals cued them to "remember" their findings and request more precise results. In contrast, they chose to trust more certain results.

The data scientists also talked about the importance of presenting precise data to their teams. While they might be able to use sample data with approximation, P3 said, "I know some information gets lost when I use samples . . . If I want to give my boss specific numbers, I don't want to use samples." P2 said, "full and complete data always makes the most sense."

We wondered whether going back to check the results of queries would disrupt their workflow. P4 felt that the color change to cue the arrival of precise results was not disruptive, and "the ability to keep working . . . and know that you will get a complete visualization is very handy."

Limitations: Users did bump into the limitations of Pangloss, which shows specific visualizations of aggregate data. P5 said "When I'm using R, it's like I'm on a mountaintop, I can go anywhere I want; when I'm using your system, there is a path that I need to follow." Similarly, P3 wanted to see lower-level data: "you want to go down to the sample level to see which samples are causing this pattern."

6.5.2 Case Studies

Our second study emphasized real-world use of datasets. We recruited data analysts from an internal list of data scientists. We looked specifically for users who had datasets over 10 gigabytes with structured tabular data, and selected three candidates. We ingested their data into Sample+Seek,

and then scheduled meetings with the groups: we met with David in person, while we met remotely with Madhu and Faraz. [1]

We followed a similar protocol to the Flight Delay study: each session started off with a short demonstration and tutorial. When subjects made observations out loud, we encouraged them to "remember" the views. At the end of the experiment, if they had not reviewed some of the remembered values, we encouraged them to click through those observations, and to evaluate whether their observations had changed. These case study sessions ran between an hour and a half and two hours.

6.5.2.1 Case Study 1: David and Software Crashes

David works on the telemetry team for a family of software products. His team is responsible for helping developers identify which features are causing problems for their users. They do so by examining telemetry across multiple builds of their software, both beta and released, categorizing the circumstances under which software fails.

David's dataset consists of activities that users are carrying out in these products with error information. Their data collection gathers 100 TB/day of raw timestamped event data. This is too much for their system (or for Pangloss) to analyze; instead, his team pre-aggregates this data into summaries, which average around 200 MB/day. These summaries consist of activities, hierarchically categorized by product and feature; broken down by minute of the day; it stores the number of times that users attempted to use the feature, and how many of them succeeded.

His team's visualization technology cannot view more than a gigabyte of data, representing a week of data. This can miss out on deployment problems that emerge over longer ranges of time, and makes it hard to compare between builds of the software. Pangloss was able to load three months' data.

---

[1] All names are anonymized.

David is used to working with slow queries: as he began to navigate his data, he excitedly said that "instantaneous visualizations are great!" He freely jumped between different slices of the data, admiring "the power of being able to pivot so quickly." David applied multiple filters to the data, limiting it by date, code branch, and application.

At first, he did not see the utility in remembering his queries: "going back and looking at the more precise data would be valuable if I were drawing any real conclusions—if I was going to send an email to somebody. But in a lot of these cases, if I didn't see anything too exciting in the approximate number, I'd be OK with that." We insisted he remember his first few queries to get precise results; by partway through the session, he did so on his own. When we went back through his results at the end, he reflected "Now that I've been sitting here for an hour, after I go back, it makes a lot of sense [to have these annotations], but as I was doing it, I was thinking, 'I want to move on, I want to move on."

Like some users in the flight delay study, he began to think about the value of precise data: "A lot of what we do gets used in ship rooms and standups; ship decisions are made on those numbers. Those meetings cost thousands of dollars a minute. You need super-precise data for them."

During the study, David ran into a surprising result: one day had an aberrantly low value for a particular data series. He went off wanting to delve into it more: "I'm going to go over this with my team and send them some screenshots. I want to find out what happened on 8/8 with this stream."

David had been limited by the amount of data they collected; Pangloss allowed him and his team to broaden their view.

6.5.2.2 Case Study 2: Madhu and Search Terms

Madhu works on the advertising platform for a search engine. His team is responsible for trying to predict trends in searches and keywords. They analyze usage data from the search engine, looking for terms and concepts that are gaining in popularity. Madhu wanted to look at trends

and changes in the popularity of these topics across different countries. His dataset consisted of topics, with categories, countries, and timestamps. As in David's case, the dataset he gave us was pre-aggregated: each of these categories was then labeled with the number of impressions—that is, the number of people who searched for terms that matched this category—and the number of people who clicked through on those searches. Madhu gave us 994 million rows of data, covering eight months of usage.

Madhu was excited that Pangloss could let him get to know the shape of his full dataset. In the past, he had not felt like he could explore his data: queries took too long, and so he would focus only on specific questions that he needed to answer.

Madhu searched carefully for patterns: he wanted to find ways that the data changed in regular and systematic ways. He spent most of his time in the heatmap, looking at a dozen or more keywords at a time. He did manage to find trends in the data, including a weekly pattern in one keyword, and another that spiked over a month. He found these results useful enough that he later asked whether he could send us a new, less aggregated dataset, and asked for a follow-up appointment with his team, in the hope that more of his colleagues had an opportunity to understand how the data they worked with operated.

6.5.2.3 Case Study 3: Faraz and Social Computing

Faraz is a data scientist who works with a Twitter "firehose" feed. One of his projects is to assess the credibility of Twitter users. His team expects that Twitter users can be distinguished by their followers lists, and by the keywords they use. His dataset looks at Twitter users, their hashtags, and the people who they follow; his statistical algorithms also label users who are likely to be spammers.

Faraz looked at the top k charts of the most commonly-used tags, finding terms like "brexit" and "rio2016" were popular; he contrasted this list to keywords tweeted by persons who were labeled as likely to be spammers. Looking at the precise view, he asked to look further down, at tail queries.

Faraz encountered some unintuitive aspects of dealing with sample-based analysis: for example, when seeing the top ten keywords, his first impulse was to "remember" just the top keywords. This would not have the desired effect: the full query might discover new words, but his filtered list would not show them.

Faraz became accustomed to seeing very high uncertainty levels for his high cardinality data. He began to use the approximate query as a draft, less concerned about the answer it showed and more concerned about whether it showed that he had the right query. "I'm doing exploratory data analysis and I don't need full accuracy." When he accidentally formed a view with a bad filter, he quickly noticed the approximate view was incorrect and adjusted the query; after he felt sure he had the right query, he remembered the view.

Pangloss allowed Faraz to explore complex aspects of his data rapidly, and come back to check his results later.

6.5.3 Discussion of User Studies

We were gratified that users were able to use Pangloss to explore their large datasets, and take away actionable and novel conclusions. Users were willing to trust the approximation, and to generate precise results afterward.

Our user studies taught us more about how users see approximate data. Our users see precision broadly: they want both rapid interaction for exploratory phase, and to present precise results to decision makers. Precision is not just a way to verify the approximation, but is an end in itself. In small-data systems, the same tool can fulfill both these roles; here, Pangloss separated those goals.

The process of recording observations during exploratory visualization was unintuitive to all our users. Most of our users were grateful to have been forced to record their observations, and later found it useful to reconstruct their path. This suggests that Pangloss does not have the right balance of encouraging users to record their observations. There is a broad spectrum of possible

approaches–from notebook interfaces that require explicit queries, to systems that automatically recommend visualizations [224]; this design space would reward further exploration.

Users wanted more features from Pangloss: several wanted to be able to see the underlying data: although big data demands aggregations, analysts wanted to see individual records to spot-check their results, and to get a sense of what sat in a bucket. Other users asked for transformations, aggregations, and ways to project the data that Pangloss does not currently support.

6.6 Conclusion

We note some implications of optimistic visualization that emerge from the user study and our design work.

The concept of optimistic visualization can help users adopt approximate and progressive systems. It is comparatively easy to implement, but the benefits for the users are large. For example, user David used optimism to build confidence in the approximate results after seeing the results of precise queries. Some of our users needed precise data. Under progressive visualization, that means waiting until computation finishes. In Pangloss, they could run it in the background. Future work should combine progressive and optimistic, and explore the design space to understand what best benefits users: a system might improve the results for remembered views in the background, allowing the user to check on progress and how the approximation has changed.

Existing visualization tools and techniques make assumptions that do not hold for approximate results. In a sampling environment, more selective predicates can have surprising effects. Additional groups may appear, and both aggregate values and uncertainty levels might change. The same can happen in the transition from approximate to precise results. New visualization systems must be able to handle shifting axes and changing color scales. We will need to develop a vocabulary of visual cues to highlight order changes, new groups (Figure 6.9), and other qualitative changes.

There are many opportunities to refine the user experience of optimistic visualization for data exploration to address the issues raised during the user studies. One question is how to identify whether a precise view is meaningfully different from the approximation. In Pangloss every precise result is treated the same; it could be valuable to highlight views with significant differences. Whether a change is significant depends on the observation: the observation that $A > B$ is different from observing that $A > 50$; the precise results might invalidate one but not the other.

One interesting question is in deciding what to remember; our users found that the major challenge when using the system. In Pangloss, users must select views explicitly. Pangloss supports an exploratory process; there are cues in the sets of visualizations that the analyst creates to decide which views are worth remembering. With better provenance tracking, we might be able to better decide which observations should be remembered; we might also group visualizations in the history by their broad tasks.

This thesis contributes the concept of optimistic visualization. We have shown a first implementation of optimism; and discussed ways in which the exploratory data analysis process is different under approximate data. We have shown ways to visualize the difference between the approximate and precise view. Last, we have presented the results of eight users working with an optimistic system, and showed that it can help meet their needs for both speed and precise results.

7 Conclusion

This thesis investigates the design of new languages and models for visualization as well as interactive systems for scalable visual analysis. These ideas are implemented in tools for data analysis and communication that richly integrate the strengths of both people and machines. In this section, we review these contributions, their limitations, and look into possible future directions of scalable visual analysis systems.

7.1 Review of Thesis Contributions and Impact

In this thesis, we proposed that high-level visualization languages for both human authoring and programmatic generation facilitate systematic exploration of the design space, reuse across computing environments, and automatic optimization. To support this hypothesis, we contribute Vega-Lite as a language for users to rapidly create statistical graphics; and developers to build applications with. Vega-Lite's focus on programmatic generation enables systematic exploration of the design space of visualization for example in Voyager and Draco. Vega-Lite can be used on servers and in the browser, written by hand or generated by end-user applications, and there now exist wrappers in various programming languages (including JavaScript, Python, and R). Vega-Lite specifications can omit low-level details so that the runtime system can optimize the execution; the runtime can optimize both the encoding decisions (e.g., the choice of color palette) and data processing (e.g., remove redundant computation).

We also hypothesized that formal models of design built on these representations enable shared and extensible knowledge bases. Draco extends Vega-Lite with shareable design guidelines, formal reasoning over the design space, and visualization recommendation. In Draco, users can omit any part of the specification and hand over decisions to the recommendation engine; which needs to implement many design rules. With software engineering and developer productivity in mind we describe these rules as constraints; with evaluation handled by high-performance constraint solvers. Vague design guidelines that were scattered across books and research papers are concrete and actionable in Draco.

Lastly, we hypothesized that combining the strengths of people and machines, and co-designing the data processing systems and their user experience, enables interactive visualizations of billion-record datasets. Falcon supports brushing and linking across multiple charts of billion-record datasets. It solves the problem of latency between the server and client by prefetching the data that is needed for the interaction with the current view; at the cost of some latency for interacting with a different view. This trade-off is informed by research on the perception of latency and in particular the fact that brushing interactions are latency sensitive while switching views is not. We take a similar user-experience perspective on approximate query processing and propose optimistic visualization. The core idea is that we can trade off accuracy for response time but only temporarily; without optimistic visualization users may lose trust, which hinders adoption.

As part of this thesis, we developed four systems; some with more direct impact than others. Our papers were well-received at conferences. The open source library has become a popular tool in the Python and JavaScript data science communities. Draco has also caught interest from industry. Falcon's potential has been recognized in industry despite it being a prototype. Pangloss was well-received at academic conferences and workshops but the adoption of approximate query processing systems is still in its infancy; optimistic visualization could be more relevant in the future.

7.2 Limitations of Perceptual and Interactive Scalability

The ultimate goal to address perceptual scalability is a set of design guidelines for visualizing large data and smart design assistants that help users apply these guidelines in their exploration tools. While more work is needed to define a comprehensive set of guidelines, we believe that Vega-Lite and Draco are well-equipped to formally express such guidelines and develop tools based on them.

The ultimate goal of interactive scalability are tools that instantly respond to user interactions. These ideal tools behave the same regardless of the size of the data. Falcon achieves this vision for brushing and linking across charts of billion-records datasets. Optimistic visualization in Pangloss enables exploration of even larger datasets through approximation. Falcon and Pangloss, however, are bespoke prototypes for scalable visual analysis. They only support a narrow range of visual and interactive designs. Vega-Lite and Draco support a rich (yet constrained) range of graphics. In the future, we hope to see systems are both expressive and scalable. There are already prototype applications that run expensive portions of a Vega-Lite dataflow on a powerful server [115, 131]. Our hope is that these efforts lead to systems that achieve interactive scalability for common visualizations expressed in Vega-Lite.

7.3 Limitations of the Systems

As discussed in subsection 3.7.3, Vega-Lite is designed for rapid authoring of statistical graphics and the model as well as the implementation have limited expressiveness. Even though we are adding new features in every new version, Vega-Lite's design space is constrained to common chart designs for analysis. Using Vega-Lite for non-cartesian plots or bespoke designs is tedious at best.

Similarly, Draco's model is limited to express linear models over constraints, hindering the development of comprehensive models of visualization design section 4.7. Our implementation is currently limited to single views. Building on the design of Vega-Lite, Draco shares its limitation to

cartesian designs. We hope to alleviate some of these limitations in future versions of the software but our focus on common statistical graphics will restrict the scope for the foreseeable future.

Falcon, as a prototype system, demonstrates that fast linked interactions over large data are possible. As discussed in section 5.6, our implementation as well as the model have some limitations. Our current implementation is limited to count aggregates; Falcon only supports binned aggregate visualizations with up to two spatial dimensions; visualizations with categorical dimensions such as bar charts are planned for future versions. To make Falcon usable for larger datasets, future versions may use approximate aggregates or more aggressive prefetching to reduce view switching latencies.

In section 6.6, we describe a new design space for applications building on the ideas of optimistic visualization. Our implementation in Pangloss is only one point in this design space. Pangloss also showed new challenges that arise with the use of optimistic visualization: When should users be notified? How we avoid users only remembering the approximate results and ignoring the precise results? Future work in this space must answer these questions to design more effective visual exploration systems.

7.4 Future Directions

Here we outline high-level directions that build on Vega-Lite, Draco, Falcon, and Pangloss towards a future of scalable visualization systems with tighter human-machine integration.

7.4.1 Design Goals for Scalable Visual Analysis Systems

Systems should automatically ensure that interactive visual analysis interfaces are appropriate for the amount and distribution of the data. Fulfilling this vision requires innovations in different areas of computer science: Visualization, Data Management, Human Computer Interaction, and Programming Languages. These traditionally separate concerns interact in complex ways. For example, an expressive visualization algebra that a database can optimize is considered a grand

challenge of scalable visualization. To unify the various components (e.g., automated reasoning, full-stack optimization, and making approximation accessible) in a single system, we must explore the design space and make principled trade-offs informed by the complementary strengths and weaknesses of computers and people. Future research in this area may contribute systems that use automated reasoning over domain-specific representations of data analysis to inform how to efficiently run data science pipelines and enhance our ability to analyze and communicate data.

7.4.2 Full-Stack Optimization for Interactive Data Systems

Traditionally, the different stages of a data processing pipeline have been optimized as independent components, with optimizations focused on either the user interface or the data processing system. However, many optimizations—such as prefetching and indexes per view in Falcon—are born out of a holistic consideration of front-end and back-end concerns together.

Future data science systems should integrate various optimization strategies such as indexing, prefetching, perceptually motivated cost models, and algorithms that optimally place data and computation on the client or the server [135]. By leveraging ambiguity and reasoning about declarative specifications of the data transformations and visual encodings such as Vega-Lite, systems could apply the optimizations automatically.

For example, new systems can implement appropriate approximations (e.g., stop computation once some perceptual tolerance is reached) and prefetching methods (e.g., to speculatively query for data a user is likely to request soon). Falcon already implements a simple prediction model. We developed the prediction model in Falcon by hand and hard coded it into the system but in the future we might be able to learn a model that can adapt to the user's exploration. Declarative specifications of the visualization and possible interactions such as Vega-Lite will be crucial in this endeavor.

An essential aspect of this research is to integrate various optimizations into one system so that we can systematically evaluate design trade-offs. In subsection 4.6.4, we discussed how Draco

can choose appropriate sampling strategies, data transformations, and visual encodings based on the analyst's goals. Future systems building on these ideas should suggest scalable interactive visualizations that are compatible with data processing techniques such as precomputation in Falcon. The analyst can then interact with their data without having to worry about idiosyncrasies of the querying system.

7.4.3 Automated Reasoning over Visualization Knowledge

While Draco currently supports automated reasoning for individual static charts, a large swath of the visualization design space remains uncharted. Future research may extend Draco to interactive multi-view charts [59, 167] and integrate richer task models. Another exciting avenue for future research is to use Draco as an intermediate formalism for natural language interfaces for data exploration. To manage Draco's expansion in scope and complexity, we will need interactive systems that enable quick adaptation of the knowledge base to an organization's needs. By integrating Draco into existing analysis tools like Altair [211], one can collect user actions to continuously improve Draco's suggestions.

As we increase automation and provide computational guidance, we also need to preserve a balance between automation and autonomy of the analyst. Only then can we get the benefits of scale and not lose the benefits of human expertise and intuition. People should stay in control and maintain agency. As one aspect of preserving agency, systems should always explain their recommendations. They should let people override any default decisions. For example, a design recommender should warn designers against misleading plots (like a linter or spell checker) and propose alternatives with explanations to educate novices and improve visualization literacy. Another aspect of preserving agency is to ensure that tools are not causing undue specification friction when people know what they want. While we believe that Vega-Lite has low friction, evaluating how much friction a specification language causes is an understudied future research area.

Draco draws a new frontier of research. Studying how people use visualization models opens opportunities to make models more personal and adapt to specific domains. Draco's formal model of visualization design facilitates structured exploration of the design space of visualizations. Studies of human perception can not only inform the visualization model in Draco. We may also go the other way and using Draco's reasoning power to identify holes in our knowledge of expressive design. By formalizing the results of perceptual studies, we may assess whether they are in conflict with or subsumed by existing knowledge. Based on these results, active learning techniques could inform experiments that close gaps in our knowledge.

7.5 Concluding Remarks

As data scientists face increasingly large and complex datasets, they need new means to explore and analyze this data as though it was small and clear. Towards these goals, we present languages (Vega-Lite) and models (Draco) for visualization design that power interactive systems for scalable data analysis (Falcon and Pangloss). Going forward, I hope and expect that these systems and the ideas they spark lead to a new generation of tools. These tools will make data analysis more efficient and enjoyable as intelligent assistants automate the tedious parts of analysis. These assistants build on evolving models of best practices. The next generation of visualization tools will also make analysis results more dependable as they automatically optimize data processing and check specifications for various pitfalls.

Bibliography

[1] Jennifer K Adelman-McCarthy, Marcel A Agüeros, Sahar S Allam, Carlos Allende Prieto, Kurt S J Anderson, Scott F Anderson, James Annis, Neta A Bahcall, C A L Bailer-Jones, Ivan K Baldry, et al. "The sixth data release of the sloan digital sky survey". In: *The Astrophysical Journal Supplement Series* (2008) (page 2).

[2] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. "Knowing when You'Re Wrong: Building Fast and Reliable Approximate Query Processing Systems". In: *Proceedings of the International Conference on Management of Data*. SIGMOD '14. ACM, 2014. ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2593667 (page 137).

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. "BlinkDB: queries with bounded errors and bounded response times on very large data". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013 (pages 5, 18, 136).

[4] David W Aha, Dennis F Kibler, and Marc K Albert. *Instance-based prediction of real-valued attributes*. 1989 (page 82).

[5] Christopher Ahlberg and Ben Shneiderman. "Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '94. Boston, Massachusetts, USA: ACM, 1994. ISBN: 0-89791-650-6. DOI: 10.1145/191666.191775 (page 10).

[6] Emily Alpert and Ben Welsh. *L.A. is slammed with record costs for legal payouts*. http://www.latimes.com/local/lanow/la-me-ln-city-payouts-20180627-story.html. June 2018 (page 54).

[7] S. Alspaugh, N. Zokaei, A. Liu, C. Jin, and M. A. Hearst. "Futzing and Moseying: Interviews with Professional Data Analysts on Exploration Practices". In: *IEEE Transactions on Visualization and Computer Graphics* (Jan. 2019). ISSN: 1077-2626. DOI: 10.1109/TVCG.2018.2865040 (page 10).

[8] Robert A. Amar, James Eagan, and John T. Stasko. "Low-Level Components of Analytic Activity in Information Visualization". In: *IEEE Symposium on Information Visualization (InfoVis 2005), 23-25 October 2005, Minneapolis, MN, USA*. 2005. DOI: 10.1109/INFOVIS.2005.24 (pages 68, 96).

[9] Dana H Ballard, Mary M Hayhoe, Polly K Pook, and Rajesh PN Rao. "Deictic codes for the embodiment of cognition". In: *Behavioral and Brain Sciences* (1997) (page 106).

[10] Roman Barták. "Constraint programming: What is behind". In: *Proceedings of CPDC99* (1999) (page 69).

[11] Leilani Battle, Remco Chang, and Michael Stonebraker. "Dynamic Prefetching of Data Tiles for Interactive Visualization". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882919 (pages 18, 111, 115).

[12] Leilani Battle, Michael Stonebraker, and Remco Chang. "Dynamic reduction of query result sets for interactive visualization". In: *Proceedings - 2013 IEEE International Conference on Big Data, Big Data 2013*. IEEE. 2013. ISBN: 9781479912926. DOI: 10.1109/BigData.2013.6691708 (pages 18, 106, 111, 127).

[13] Richard A Becker, William S Cleveland, and Ming-Jen Shyu. "The visual design and control of trellis display". In: *Journal of computational and Graphical Statistics* (1996) (page 34).

[14] Jacques Bertin. *Graphics and graphic information processing*. Walter de Gruyter, 1982 (page 10).

[15] Jacques Bertin. *Semiology of graphics: diagrams, networks, maps*. University of Wisconsin press, 1983 (pages 9, 67).

[16] Alan Borning. "ThingLab - An Object-Oriented System for Building Simulations Using Constraints". In: *Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, USA, August 22-25, 1977*. 1977 (page 69).

[17] Michael Bostock and Jeffrey Heer. "Protovis: A Graphical Toolkit for Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* (Nov. 2009). ISSN: 1077-2626 (pages 12, 23, 26).

[18] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. "D^3: data-driven documents". In: *Visualization and Computer Graphics, IEEE Transactions on* (2011) (pages 12, 23, 26, 50, 52, 141).

[19] Thomas Boutell. *Png (portable network graphics) specification version 1.0*. Tech. rep. 1997 (page 119).

[20] J. Boy, R. A. Rensink, E. Bertini, and J. D. Fekete. "A Principled Way of Assessing Visualization Literacy". In: *IEEE Transactions on Visualization and Computer Graphics* (Dec. 2014). ISSN: 1077-2626. DOI: 10.1109/TVCG.2014.2346984 (page 63).

[21] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. "Answer Set Programming at a Glance". In: *Commun. ACM* (Dec. 2011). ISSN: 0001-0782. DOI: 10.1145/2043174.2043195 (pages 69, 71).

[22] A G A Brown, A Vallenari, T Prusti, J H J de Bruijne, C Babusiaux, C A L Bailer-Jones, Gaia Collaboration, et al. "Gaia Data Release 2. Summary of the contents and survey properties". In: *arXiv preprint arXiv:1804.09365* (2018) (pages 2, 122, 129).

[23] *Brunel Visualization*. https://developer.ibm.com/open/brunel-visualization/. June 2016 (page 28).

[24] Mihai Budiu, Rebecca Isaacs, Derek Murray, Gordon Plotkin, Paul Barham, Samer Al-Kiswany, Yazan Boshmaf, Qingzhou Luo, and Alexandr Andoni. "Interacting with large distributed datasets using Sketch". In: *Eurographics Symposium on Parallel Graphics and Visualization*. University of Wisconsin-Madison. 2015. ISBN: 978-3-03868-006-2. DOI: 10.2312/PGV.20161180 (page 135).

[25] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, eds. *Readings in Information Visualization: Using Vision to Think*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN: 1-55860-533-9 (pages 10, 11, 19, 106, 107).

[26] Stuart K. Card, Allen Newell, and Thomas P. Moran. *The Psychology of Human-Computer Interaction*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1983. ISBN: 0898592437 (pages 104, 106).

[27] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. "The Information Visualizer, an Information Workspace". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '91. New Orleans, Louisiana, USA: ACM, 1991. ISBN: 0-89791-383-3. DOI: 10.1145/108844.108874 (page 133).

[28] D. B. Carr, R. J. Littlefield, W. L. Nicholson, and J. S. Littlefield. "Scatterplot Matrix Techniques for Large N". In: *Journal of the American Statistical Association* (1987). ISSN: 0162-1459 (page 15).

[29] Stephen M. Casner. "Task-analytic approach to the automated design of graphic presentations". In: *ACM Transactions on Graphics* (1991). ISSN: 07300301. DOI: 10.1145/108360.108361 (page 67).

[30] Ugur Cetintemel, Mitch Cherniack, Justin DeBrabant, Yanlei Diao, Kyriaki Dimitriadou, Alexander Kalinin, Olga Papaemmanouil, and Stanley B Zdonik. "Query Steering for Interactive Data Exploration." In: *CIDR*. 2013 (page 18).

[31] Sye Min Chan, Ling Xiao, John Gerth, and Pat Hanrahan. "Maintaining interactivity while exploring massive time series". In: *VAST'08 - IEEE Symposium on Visual Analytics Science and Technology, Proceedings*. IEEE. Oct. 2008. ISBN: 9781424429356. DOI: 10.1109/VAST.2008.4677357 (page 17).

[32] Sye-Min Chan, Ling Xiao, J. Gerth, and P. Hanrahan. "Maintaining interactivity while exploring massive time series". In: *IEEE Symposium on Visual Analytics Science and Technology*. Oct. 2008. DOI: 10.1109/VAST.2008.4677357 (page 111).

[33] Remco Chang, Jean-Daniel Fekete, Juliana Freire, and Carlos E Scheidegger. "Connecting Visualization and Data Management Research (Dagstuhl Seminar 17461)". In: *Dagstuhl*

Reports (2018). Ed. by Remco Chang, Jean-Daniel Fekete, Juliana Freire, and Carlos E Scheidegger. ISSN: 2192-5283. DOI: 10.4230/DagRep.7.11.46 (pages 59, 60).

[34] Surajit Chaudhuri and Umeshwar Dayal. "An Overview of Data Warehousing and OLAP Technology". In: SIGMOD '97 (1997). DOI: 10.1145/248603.248616 (pages 16, 17, 109, 135).

[35] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. "Approximate Query Processing: No Silver Bullet". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3056097 (pages 104, 110).

[36] Hong Chen. "Compound brushing [dynamic data visualization]". In: *Information Visualization, 2003. INFOVIS 2003. IEEE Symposium on*. IEEE. 2003 (page 28).

[37] Mon Chu Chen, John R. Anderson, and Myeong Ho Sohn. "What Can a Mouse Cursor Tell Us More?: Correlation of Eye/Mouse Movements on Web Browsing". In: *CHI '01 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '01. Seattle, Washington: ACM, 2001. ISBN: 1-58113-340-5. DOI: 10.1145/634067.634234 (page 115).

[38] Ed H. Chi. "A Taxonomy of Visualization Techniques Using the Data State Reference Model". In: *Proceedings of the IEEE Symposium on Information Visualization 2000*. INFOVIS '00. Washington, DC, USA: IEEE Computer Society, 2000. ISBN: 0-7695-0804-9 (page 19).

[39] Ed Huai-hsin Chi and John Riedl. "An Operator Interaction Framework for Visualization Systems". In: *Proceedings of the 1998 IEEE Symposium on Information Visualization*. INFOVIS '98. North Carolina: IEEE Computer Society, 1998. ISBN: 0-8186-9093-3 (page 19).

[40] Jungu Choi, Deok Gun Park, Yuet Ling Wong, Eli Fisher, and Niklas Elmqvist. "VisDock: A Toolkit for Cross-Cutting Interactions in Visualization". In: *Visualization and Computer Graphics, IEEE Transactions on* (2015) (page 27).

[41] William S. Cleveland and Robert McGill. "Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods". In: *Journal of the American Statistical Association* (Sept. 1984). DOI: 10.1080/01621459.1984.10478080 (pages 9, 67, 112).

[42] D. R. Cox. “Some Remarks on the Role in Statistics of Graphical Methods”. In: *Applied Statistics* (1978). DOI: 10.2307/2346220 (page 12).

[43] *Crossfilter: Fast Multidimensional Filtering for Coordinated Views*. https://square.github.io/crossfilter// (page 106).

[44] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. “Vizdom: Interactive Analytics Through Pen and Touch”. In: *Proc. VLDB Endow.* (Aug. 2015). ISSN: 2150-8097. DOI: 10.14778/2824032.2824127 (pages 106, 110).

[45] Franklin C. Crow. “Summed-area Tables for Texture Mapping”. In: *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’84. New York, NY, USA: ACM, 1984. ISBN: 0-89791-138-5. DOI: 10.1145/800031.808600 (page 116).

[46] Geoff Cumming and Sue Finch. “Inference by eye: confidence intervals and how to read pictures of data.” In: (2005). DOI: 10.1037/0003-066X.60.2.170 (page 137).

[47] Gennady Davydov, Inna Davydova, and Hans Kleine Büning. “An efficient algorithm for the minimal unsatisfiability problem for a subclass of CNF”. In: *Annals of Mathematics and Artificial Intelligence* (Nov. 1998). ISSN: 1573-7470. DOI: 10.1023/A:1018924526592 (page 98).

[48] *Declarative visualization for Elm*. https://github.com/gicentre/elm-vega. Jan. 2018 (page 56).

[49] Mark Derthick, John Kolojejchick, and Steven F Roth. “An interactive visual query environment for exploring data”. In: *Proceedings of the 10th annual ACM symposium on User interface software and technology*. ACM. 1997 (page 28).

[50] Bolin Ding, Silu Huang, Surajit Chaudhuri, Kaushik Chakrabarti, and Chi Wang. “Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee”. In: *Proceedings of the International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: ACM, 2016. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2915249 (pages 136, 138–140).

[51] Punit R. Doshi, Elke A. Rundensteiner, and Matthew O. Ward. “Prefetching for Visual Data Exploration”. In: *Proceedings of the Eighth International Conference on Database Systems*

for Advanced Applications. DASFAA '03. Washington, DC, USA: IEEE Computer Society, 2003. ISBN: 0-7695-1895 (page 111).

[52] Punit R Doshi, Elke A Rundensteiner, and Matthew O Ward. "Prefetching for visual data exploration". In: *Dasfaa 2003*. IEEE. 2003 (page 18).

[53] Philipp Eichmann, Carsten Binnig, Tim Kraska, and Emanuel Zgraggen. "IDEBench: A Benchmark for Interactive Data Exploration". In: 2018 (page 106).

[54] Niklas Elmqvist and Jean Daniel Fekete. "Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines". In: *IEEE Transactions on Visualization and Computer Graphics*. INFOVIS '10 (2010). ISSN: 1077-2626. DOI: 10.1109/TVCG.2009.84 (pages 15, 135, 141).

[55] Richard Evans and Edward Grefenstette. "Learning Explanatory Rules from Noisy Data". In: *Journal of Artificial Intelligence Research* (2018) (page 97).

[56] Ethan Fast. *A Conversational Agent for Data Science*. https://hackernoon.com/a-conversational-agent-for-data-science-4ae300cdc220. 2016 (page 56).

[57] Jean-Daniel Fekete and Romain Primet. "Progressive Analytics: A Computation Paradigm for Exploratory Data Analysis". In: (2016) (page 136).

[58] Nivan Ferreira, Danyel Fisher, and Arnd Christian König. "Sample-oriented task-driven visualizations: allowing users to make better, more confident decisions". In: *Proceedings of the Conference on Human Factors in Computing Systems*. CHI '14. ACM, 2014. DOI: 10.1145/2556288.2557131 (page 138).

[59] Stephen Few. *Information Dashboard Design: The Effective Visual Communication of Data*. O'Reilly Media, Inc., 2006. ISBN: 0596100167 (page 165).

[60] Danyel Fisher. "Big Data Exploration Requires Collaboration Between Visualization and Data Infrastructures". In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. HILDA '16. San Francisco, California: ACM, 2016. ISBN: 978-1-4503-4207-0. DOI: 10.1145/2939502.2939518 (page 134).

[61] Danyel Fisher and Miriah Meyer. *Making Data Visual: A practical guide to using visualization for insight*. O'Reilly, 2017. ISBN: 978-1-491-92846-2 (page 54).

[62] Danyel Fisher, Igor Popov, Steven Drucker, and m.c. schraefel m.c. "Trust Me, I'm Partially Right: Incremental Visualization Lets Analysts Explore Large Datasets Faster". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '12. Austin, Texas, USA: ACM, 2012. ISBN: 978-1-4503-1015-4. DOI: 10.1145/2207676.2208294 (pages 5, 19, 103, 110, 136, 138, 139, 149, 150).

[63] Apache Software Foundation. *Arrow*. https://arrow.apache.org/. 2017 (pages 118, 121).

[64] Mark S. Fox. "Constraint-Directed Search: A Case Study of Job-Shop Scheduling". PhD thesis. Pittsburgh, PA: Carnegie Mellon University, Dec. 1983 (page 69).

[65] Tong Gao, Mira Dontcheva, Eytan Adar, Zhicheng Liu, and Karrie G. Karahalios. "DataTone: Managing Ambiguity in Natural Language Interfaces for Data Visualization". In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology*. UIST '15. Charlotte, NC, USA: ACM, 2015. ISBN: 978-1-4503-3779-3. DOI: 10.1145/2807442.2807478 (page 96).

[66] Martin Gebser, Amelia Harrison, Roland Kaminski, Vladimir Lifschitz, and Torsten Schaub. "Abstract gringo". In: *TPLP* (2015) (page 71).

[67] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, and Sven Thiele. "Potassco user guide". In: *Institute for Informatics, University of Potsdam, second edition edition* (2015). https://github.com/potassco/guide/releases/ (page 69).

[68] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. "Answer Set Solving in Practice". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* (2012). ISSN: 1939-4608. DOI: 10.2200/S00457ED1V01Y201211AIM019 (page 69).

[69] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. "Clingo = ASP + Control: Preliminary Report". In: *CoRR* (2014) (page 69).

[70] Martin Gebser, Roland Kaminski, and Torsten Schaub. "Complex Optimization in Answer Set Programming". In: (July 2011) (pages 64, 72).

[71] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. "Potassco: The Potsdam Answer Set Solving Collection". In: *AI Commun.* (Apr. 2011). ISSN: 0921-7126 (page 69).

[72] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. "Conflict-Driven Answer Set Solving". In: *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*. 2007 (page 69).

[73] Michael Gelfond and Vladimir Lifschitz. "The Stable Model Semantics For Logic Programming". In: MIT Press, 1988 (page 71).

[74] *ggvis 0.4 overview*. https://ggvis.rstudio.com. June 2016 (page 25).

[75] Michael Gleicher, Michael Correll, Christine Nothelfer, and Steven Franconeri. "Perception of average value in multiclass scatterplots". In: *IEEE Transactions on Visualization and Computer Graphics* (2013). ISSN: 10772626. DOI: 10.1109/TVCG.2013.183 (page 67).

[76] Michael Glueck, Azam Khan, and Daniel J. Wigdor. "Dive in!: Enabling Progressive Loading for Real-time Navigation of Data Visualizations". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '14. Toronto, Ontario, Canada: ACM, 2014. ISBN: 978-1-4503-2473-1. DOI: 10.1145/2556288.2557195 (page 119).

[77] P. Godfrey, J. Gryz, and P. Lasek. "Interactive Visualization of Large Data Sets". In: *IEEE Transactions on Knowledge and Data Engineering*. 2016. DOI: 10.1109/TKDE.2016.2557324 (page 134).

[78] Samuel Gratzl, Nils Gehlenborg, Alexander Lex, Hanspeter Pfister, and Marc Streit. "Domino: Extracting, Comparing, and Manipulating Subsets across Multiple Tabular Datasets". In: *IEEE Transactions on Visualization and Computer Graphics* (2014) (page 59).

[79] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total". In: *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, USA*. 1996. DOI: 10.1109/ICDE.1996.492099 (pages 17, 109, 115, 135).

[80] T.R.G. Green and M. Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework". In: *Journal of Visual Languages & Computing* (June 1996). DOI: 10.1006/jvlc.1996.0009 (page 28).

[81] 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*. ACM, 2005 (page 42).

[82] Peter J. Haas and Joseph M. Hellerstein. "Ripple Joins for Online Aggregation". In: *Proceedings of the International Conference on Management of Data*. SIGMOD '99. ACM, 1999. ISBN: 1-58113-084-8. DOI: 10.1145/304182.304208 (page 136).

[83] Pat Hanrahan. "Analytic Database Technologies for a New Kind of User: The Data Enthusiast". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. New York, NY, USA: ACM, 2012. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213902 (page 5).

[84] Pat Hanrahan. "VizQL: A Language for Query, Analysis and Visualization". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142560 (pages 12, 27, 68).

[85] Lane Harrison, Fumeng Yang, Steven Franconeri, and Remco Chang. "Ranking visualizations of Correlation Using Weber's law". In: *IEEE Transactions on Visualization and Computer Graphics* (2014). ISSN: 10772626. DOI: 10.1109/TVCG.2014.2346979 (pages 63, 67).

[86] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition*. Springer series in statistics. Springer, 2009. ISBN: 9780387848570 (page 91).

[87] 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*. ACM. 2008 (page 29).

[88] Jeffrey Heer and Michael Bostock. "Crowdsourcing Graphical Perception: Using Mechanical Turk to Assess Visualization Design". In: *Proceedings of the SIGCHI Conference on Human*

Factors in Computing Systems. CHI '10. Atlanta, Georgia, USA: ACM, 2010. ISBN: 978-1-60558-929-9. DOI: 10.1145/1753326.1753357 (pages 9, 67).

[89] Jeffrey Heer and Michael Bostock. "Declarative language design for interactive visualization". In: *IEEE Transactions on Visualization and Computer Graphics* (2010). ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.144 (page 12).

[90] Jeffrey Heer, Jock Mackinlay, Chris Stolte, and Maneesh Agrawala. "Graphical Histories for Visualization: Supporting Analysis, Communication, and Evaluation". In: *IEEE Transactions on Visualization and Computer Graphics*. INFOVIS '08. 2008. DOI: 10.1109/TVCG.2008.137 (page 146).

[91] Jeffrey Heer and Ben Shneiderman. "Interactive dynamics for visual analysis". In: *Communications of the ACM* (Apr. 2012). ISSN: 0001-0782. DOI: 10.1145/2133806.2133821 (pages 5, 12, 24, 26, 27).

[92] Joseph M. Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, Tali Roth, and Peter J. Haas. "Interactive Data Analysis: The Control Project". In: (1999). ISSN: 0018-9162. DOI: 10.1109/2.781635 (pages 136, 139).

[93] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. "Online Aggregation". In: *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. SIGMOD '97. Tucson, Arizona, USA: ACM, 1997. ISBN: 0-89791-911-4. DOI: 10.1145/253260.253291 (pages 18, 104, 119, 136).

[94] R. Herbrich, T. Graepel, and K. Obermayer. "Support vector learning for ordinal regression". In: *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*. 1999. DOI: 10.1049/cp:19991091 (pages 70, 84).

[95] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. "Visual Debugging Techniques for Reactive Data Visualization". In: *Computer Graphics Forum (Proc. EuroVis)* (2016). DOI: 10.1111/cgf.12903 (page 59).

[96] 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*. ACM. 2009 (page 58).

[97] Jessica Hullman, Paul Resnick, and Eytan Adar. "Hypothetical Outcome Plots Outperform Error Bars and Violin Plots for Inferences about Reliability of Variable Ordering". In: PloS one. Public Library of Science, 2015. DOI: 10.1371/JOURNAL.PONE.0142444 (pages 137, 139).

[98] *Interactive Visual Analytics for Big Data*. https://www.mapd.com/platform/immerse/ (pages 105, 125).

[99] Z. Ivezic, J. A. Tyson, R. Allsman, J. Andrew, and R. Angel. "LSST: From Science Drivers to Reference Design and Anticipated Data Products". In: *Evolution* (2008) (page 2).

[100] Joxan Jaffar and Michael J Maher. "Constraint logic programming: A survey". In: *The journal of logic programming* (1994) (page 69).

[101] Matthias Jarke and Jurgen Koch. "Query Optimization in Database Systems". In: *ACM Comput. Surv.* (June 1984). ISSN: 0360-0300. DOI: 10.1145/356924.356928 (page 59).

[102] Prasanth Jayachandran, Karthik Tunga, Niranjan Kamat, and Arnab Nandi. "Combining User Interaction, Speculative Query Execution and Sampling in the DICE System". In: *Proc. VLDB Endow.* (Aug. 2014). ISSN: 2150-8097. DOI: 10.14778/2733004.2733064 (page 110).

[103] D. F. Jerding and J. T. Stasko. "The Information Mural: a technique for displaying and navigating large information spaces". In: *IEEE Transactions on Visualization and Computer Graphics* (July 1998). ISSN: 1077-2626. DOI: 10.1109/2945.722299 (page 15).

[104] Ruoming Jin, Karthikeyan Vaidyanathan, Ge Yang, and Gagan Agrawal. "Communication and memory optimal parallel data cube construction". In: *IEEE Transactions on Parallel and Distributed Systems* (2005) (page 17).

[105] Susan Joslyn and Jared LeClerc. "Decisions with uncertainty: the glass half full". In: (2013). DOI: 10.1177/0963721413481473 (page 137).

[106] Ricardo Jota, Albert Ng, Paul Dietz, and Daniel Wigdor. "How Fast is Fast Enough?: A Study of the Effects of Latency in Direct-touch Pointing Tasks". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '13. Paris, France: ACM, 2013. ISBN: 978-1-4503-1899-0. DOI: 10.1145/2470654.2481317 (page 106).

[107] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. "M4: A Visualization-oriented Time Series Data Aggregation". In: *Proc. VLDB Endow.* (June 2014). ISSN: 2150-8097 (page 15).

[108] Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. "Distributed and interactive cube exploration". In: *International Conference on Data Engineering*. IEEE. 2014. DOI: 10.1109/ICDE.2014.6816674 (page 136).

[109] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. "Enterprise data analysis and visualization: An interview study". In: *IEEE Transactions on Visualization and Computer Graphics* (2012). ISSN: 1077-2626. DOI: 10.1109/TVCG.2012.219 (page 2).

[110] Sean Kandel, Ravi Parikh, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. "Profiler: Integrated Statistical Analysis and Visualization for Data Quality Assessment". In: *Proceedings of the International Working Conference on Advanced Visual Interfaces*. AVI '12. Capri Island, Italy: ACM, 2012. ISBN: 978-1-4503-1287-5. DOI: 10.1145/2254556.2254659 (pages 11, 109).

[111] Alicia Key, Bill Howe, Daniel Perry, and Cecilia R. Aragon. "VizDeck: self-organizing dashboards for visual analytics". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. 2012. DOI: 10.1145/2213836.2213931 (page 70).

[112] Younghoon Kim and Jeffrey Heer. "Assessing Effects of Task and Data Distribution on the Effectiveness of Visual Encodings". In: *Computer Graphics Forum (Proc. EuroVis)* (2018). DOI: 10.1111/cgf.13409 (pages 9, 63, 65, 67, 68, 74, 90, 92).

[113] Younghoon Kim, Kanit Wongsuphasawat, Jessica Hullman, and Jeffrey Heer. "GraphScape: A Model for Automated Reasoning about Visualization Similarity and Sequencing". In: *Proc. of ACM CHI 2017*. 2017. ISBN: 9781450346559. DOI: 10.1145/3025453.3025866 (pages 56, 68).

[114] Stanley Kok and Pedro Domingos. "Learning the Structure of Markov Logic Networks". In: *Proceedings of the 22Nd International Conference on Machine Learning*. ICML '05. Bonn, Germany: ACM, 2005. ISBN: 1-59593-180-5. DOI: 10.1145/1102351.1102407 (page 97).

[115] Venkat Krishnamurthy. *To Jupyter and Beyond: Interactive Data Science at Scale with OmniSci*. https://www.omnisci.com/blog/to-jupyter-and-beyond-data-science-with-omnisci. Sept. 2019 (page 162).

[116] Mark Law, Alessandra Russo, and Krysia Broda. *The ILASP system for learning Answer Set Programs*. https://www.doc.ic.ac.uk/~ml1909/ILASP. 2015 (page 97).

[117] Vladimir Lifschitz. "Answer set programming and plan generation". In: *Artificial Intelligence* (June 2002). DOI: 10.1016/s0004-3702(02)00186-8 (page 75).

[118] Vladimir Lifschitz. "What is Answer Set Programming?" In: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*. AAAI'08. Chicago, Illinois: AAAI Press, 2008. ISBN: 978-1-57735-368-3 (page 69).

[119] Lauro Lins, James T. Klosowski, and Carlos Scheidegger. "Nanocubes for real-time exploration of spatiotemporal datasets". In: *IEEE Transactions on Visualization and Computer Graphics* (2013). ISSN: 1077-2626. DOI: 10.1109/TVCG.2013.179 (pages 17, 108, 109, 113, 125, 135).

[120] Mark Litwintschik. *Summary of the 1.1 Billion Taxi Rides Benchmarks*. 2017 (page 125).

[121] Tie-Yan Liu. "Learning to Rank for Information Retrieval". In: *Foundations and Trends in Information Retrieval* (2009). ISSN: 1554-0669. DOI: 10.1561/1500000016 (pages 64, 70).

[122] Zhicheng Liu and Jeffrey Heer. "The effects of interactive latency on exploratory visual analysis". In: *IEEE Transactions on Visualization and Computer Graphics* (2014). ISSN: 1077-2626. DOI: 10.1109/TVCG.2014.2346452 (pages 5, 7, 13, 102, 103, 106, 115, 125, 127, 134).

[123] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. "ImMens: Real-time visual querying of big data". In: *Computer Graphics Forum* (June 2013). ISSN: 1467-8659. DOI: 10.1111/cgf.12129 (pages 13, 15–17, 105, 106, 109, 111–113, 125, 127, 135).

[124] Zhicheng Liu and John T Stasko. "Mental models, visual reasoning and interaction in information visualization: A top-down perspective". In: *IEEE Trans. Visualization & Comp. Graphics* (2010) (page 10).

[125] Miron Livny, Raghu Ramakrishnan, Kevin Beyer, Guangshun Chen, Donko Donjerkovic, Shilpa Lawande, Jussi Myllymaki, and Kent Wenger. "DEVise: integrated querying and visual exploration of large datasets". In: *ACM SIGMOD Record* (1997) (page 28).

[126] Ian Lyttle. *R interface to Altair*. https://vegawidget.github.io/altair/. Mar. 2018 (page 56).

[127] Jock Mackinlay. "Automating the design of graphical presentations of relational information". In: *ACM Transactions on Graphics* (1986). ISSN: 0730-0301. DOI: 10.1145/22949.22950 (pages 9, 63–65).

[128] Jock D. Mackinlay, Pat Hanrahan, and Chris Stolte. "Show Me: Automatic presentation for visual analysis". In: *IEEE Transactions on Visualization and Computer Graphics* (2007). ISSN: 10772626. DOI: 10.1109/TVCG.2007.70594 (pages 63, 67, 68).

[129] Microsoft. *PowerBI*. https://powerbi.microsoft.com/ (pages 105, 108).

[130] Vibhu O. Mittal, Giuseppe Carenini, Johanna D. Moore, and Steven Roth. "Describing complex charts in natural language: A caption generation system". In: *Computational Linguistics* (1998). ISSN: 0891-2017 (pages 63, 67).

[131] Dominik Moritz. *A demo of scaling Vega to billions of records*. https://github.com/vega/scalable-vega. Feb. 2019 (page 162).

[132] Dominik Moritz and Danyel Fisher. *Visualizing a Million Time Series with the Density Line Chart*. 2018 (pages 13, 92).

[133] Dominik Moritz and Danyel Fisher. "What Users Don't Expect about Exploratory Data Analysis on Approximate Query Processing Systems". In: *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics - HILDA'17*. 2017. ISBN: 9781450350297. DOI: 10.1145/3077257.3077258 (pages 8, 132).

[134] Dominik Moritz, Danyel Fisher, Bolin Ding, and Chi Wang. "Trust, but verify: Optimistic visualizations of approximate queries for exploring big data". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems - CHI '17* (2017) (pages 8, 110, 112, 132).

[135] Dominik Moritz, Jeffrey Heer, and Bill Howe. "Dynamic Client-Server Optimization for Scalable Interactive Visualization on the Web". In: *Workshop on Data Systems for Interactive Analysis (DSIA '15)*. 2015 (pages 20, 102, 164).

[136] Dominik Moritz, Bill Howe, and Jeffrey Heer. "Falcon: Balancing Interactive Latency and Resolution Sensitivity for Scalable Linked Visualizations". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*. ACM Press, 2019. DOI: 10.1145/3290605 (page 103).

[137] Dominik Moritz, Chenglong Wang, Greg L. Nelson, Halden Lin, Adam M. Smith, Bill Howe, and Jeffrey Heer. "Formalizing Visualization Design Knowledge as Constraints : Actionable and Extensible Models in Draco". In: *IEEE Vis (Proc. InfoVis)*. 2019. DOI: 10.1109/tvcg.2018.2865240 (pages 8, 63).

[138] Tamara Munzner. *Visualization analysis and design*. CRC press, 2014 (page 68).

[139] Arnab Nandi, Cong Yu, Philip Bohannon, and Raghu Ramakrishnan. "Data cube materialization and mining over mapreduce". In: *IEEE transactions on knowledge and data engineering* (2012) (page 17).

[140] Arnab Nandi, Cong Yu, Philip Bohannon, and Raghu Ramakrishnan. "Distributed Cube Materialization on Holistic Measures". In: *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*. ICDE '11. Washington, DC, USA: IEEE Computer Society, 2011. ISBN: 978-1-4244-8959-6 (page 17).

[141] Jerzy Neyman. "Outline of a theory of statistical estimation based on the classical theory of probability". In: (1937). DOI: 10.1098/RSTA.1937.0005 (page 136).

[142] Jakob Nielsen. "Response times: The 3 important limits". In: (1993) (page 134).

[143] NOAA. *Climate Data Online*. https://ncdc.noaa.gov/cdo-web/datasets. Accessed: 2018-09-12 (page 122).

[144] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. "An A-Prolog Decision Support System for the Space Shuttle". In: *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*. PADL '01. London, UK, UK: Springer-Verlag, 2001. ISBN: 3-540-41768-0 (page 69).

[145] Chris North. "Toward Measuring Visualization Insight". In: (2006). ISSN: 0272-1716. DOI: 10.1109/MCG.2006.70 (page 134).

[146] Chris North and Ben Shneiderman. "Snap-together visualization: a user interface for coordinating visualizations via relational schemata". In: *Proceedings of the working conference on Advanced visual interfaces*. ACM. 2000 (page 28).

[147] Frank Olken and Doron Rotem. "Simple Random Sampling from Relational Databases". In: *Proceedings of the 12th International Conference on Very Large Data Bases*. VLDB '86. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986. ISBN: 0-934613-18-4 (page 136).

[148] C Olsten, Michael Stonebraker, Alexander Aiken, and Joseph M Hellerstein. "VIQING: Visual interactive querying". In: *Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*. IEEE. 1998 (page 28).

[149] Chris Olston and Jock D. Mackinlay. "Visualizing Data with Bounded Uncertainty". In: *Proceedings of the IEEE Symposium on Information Visualization*. INFOVIS '02. 2002. ISBN: 0-7695-1751-X (page 136).

[150] *Omnisci integrations for JupyterLab*. https://github.com/Quansight/jupyterlab-omnisci. June 2019 (page 57).

[151] *Online Vega Editor*. https://vega.github.io/editor/. Accessed: 2019-08-23. 2016 (pages 50, 59).

[152] Cicero A. L. Pahins, Sean A. Stephens, Carlos Scheidegger, and Joao L. D. Comba. "Hashedcubes: Simple, Low Memory, Real-Time Visual Exploration of Big Data". In: *IEEE Transactions on Visualization and Computer Graphics* (Jan. 2017). ISSN: 1077-2626. DOI: 10.1109/TVCG.2016.2598624 (pages 108, 109).

[153] Anshul Vikram Pandey, Katharina Rall, Margaret L. Satterthwaite, Oded Nov, and Enrico Bertini. "How Deceptive Are Deceptive Visualizations?: An Empirical Analysis of Common Distortion Techniques". In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI '15. Seoul, Republic of Korea: ACM, 2015. ISBN: 978-1-4503-3145-6. DOI: 10.1145/2702123.2702608 (pages 63, 78).

[154] Y. Park, M. Cafarella, and B. Mozafari. "Visualization-aware sampling for very large databases". In: *Conference on Data Engineering*. May 2016. DOI: 10.1109/ICDE.2016.7498287 (page 106).

[155] Yongjoo Park, Michael Cafarella, and Barzan Mozafari. "Visualization-aware sampling for very large databases". In: *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE. 2016 (page 15).

[156] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* (2011) (page 91).

[157] Alexandre Perrot, Romain Bourqui, Nicolas Hanusse, Frédéric Lalanne, and David Auber. "Large interactive visualization of density functions on big data infrastructure". In: *Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE. 2015. DOI: 10.1109/LDAV.2015.7348077 (page 135).

[158] Nicola Pezzotti, Boudewijn Lelieveldt, Laurens van der Maaten, Thomas Hollt, Elmar Eisemann, and Anna Vilanova. "Approximated and User Steerable tSNE for Progressive Visual Analytics". In: *IEEE Transactions on Visualization and Computer Graphics*. INFOVIS '16. IEEE, 2015. DOI: 10.1109/TVCG.2016.2570755 (pages 136, 138, 149).

[159] William A. Pike, John Stasko, Remco Chang, and Theresa A. O'Connell. "The science of interaction". In: *Information Visualization* (2009). ISSN: 1473-8716. DOI: 10.1057/ivs.2009.22 (pages 1, 10, 24, 27).

[160] Peter Pirolli and Stuart Card. "The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis". In: *Proceedings of international conference on intelligence analysis*. 2005 (page 133).

[161] Jim Pivarski. *diana-hep/histbook: 1.0.0*. June 2018. DOI: 10.5281/zenodo.1284427 (page 54).

[162] W. Playfair. *The statistical breviary; shewing the resources of every state and kingdom in Europe*. J. Wallis, 1801 (page 9).

[163] J. Poco, A. Mayhua, and J. Heer. "Extracting and Retargeting Color Mappings from Bitmap Images of Visualizations". In: *IEEE Transactions on Visualization and Computer Graphics* (Jan. 2018). ISSN: 1077-2626. DOI: 10.1109/TVCG.2017.2744320 (page 68).

[164] Jorge Poco and Jeffrey Heer. "Reverse-Engineering Visualizations: Recovering Visual Encodings from Chart Images". In: *Computer Graphics Forum* (2017). ISSN: 1467-8659. DOI: 10.1111/cgf.13193 (page 56).

[165] Hoifung Poon and Pedro Domingos. "Unsupervised Semantic Parsing". In: *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*. EMNLP '09. Singapore: Association for Computational Linguistics, 2009. ISBN: 978-1-932432-59-6 (pages 69, 70).

[166] *Project Jupyter*. https://jupyter.org/index.html. 2015 (page 56).

[167] Z. Qu and J. Hullman. "Keeping Multiple Views Consistent: Constraints, Validations, and Exceptions in Visualization Authoring". In: *IEEE Transactions on Visualization and Computer Graphics* (Jan. 2018). ISSN: 1077-2626. DOI: 10.1109/TVCG.2017.2744198 (pages 63, 97, 165).

[168] J. R. Quinlan. "Learning logical definitions from relations". In: *Machine Learning* (Aug. 1990). ISSN: 1573-0565. DOI: 10.1007/BF00117105 (page 97).

[169] Matthew Richardson and Pedro Domingos. "Markov logic networks". In: *Machine Learning* (Feb. 2006). ISSN: 1573-0565. DOI: 10.1007/s10994-006-5833-1 (pages 70, 79).

[170] Christopher Root and Todd Mostak. "MapD: A GPU-powered Big Data Analytics and Visualization Platform". In: *ACM SIGGRAPH 2016 Talks*. SIGGRAPH '16. Anaheim, California: ACM, 2016. ISBN: 978-1-4503-4282-7. DOI: 10.1145/2897839.2927468 (pages 57, 60, 103, 118).

[171] Steven F Roth and Joe Mattis. "Data characterization for intelligent graphics presentation". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 1990 (pages 69, 86).

[172] Dan Saber. *A Dramatic Tour through Python's Data Visualization Landscape (including ggpy and Altair)*. https://bit.ly/2sUHcJu. Oct. 2016 (page 56).

[173] Bahador Saket, Alex Endert, and Cagatay Demiralp. "Task-Based Effectiveness of Basic Visualizations". In: *IEEE Vis (Proc. InfoVis)* (2018). ISSN: 1077-2626. DOI: 10.1109/TVCG.2018.2829750 (pages 65, 67, 90).

[174] Bahador Saket, Dominik Moritz, Halden Lin, Victor Dibia, Cagatay Demiralp, and Jeffrey Heer. *Beyond Heuristics: Learning Visualization Design*. 2018 (pages 97, 100).

[175] A. Sarikaya, M. Gleicher, and D. A. Szafir. "Design Factors for Summary Visualization in Visual Analytics". In: *Computer Graphics Forum* (June 2018). DOI: 10.1111/cgf.13408 (pages 92, 94).

[176] Alper Sarikaya and Michael Gleicher. "Scatterplots: Tasks, Data, and Designs". In: *IEEE Transactions on Visualization and Computer Graphics* (Jan. 2018). ISSN: 1077-2626. DOI: 10.1109/TVCG.2017.2744184 (pages 13, 24).

[177] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. "Vega-Lite: A Grammar of Interactive Graphics". In: *IEEE Transactions on Visualization and Computer Graphics* (2017). DOI: 10.1109/tvcg.2016.2599030 (pages 8, 23, 60, 64, 68, 96, 98).

[178] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. "Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization". In: *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2015). DOI: 10.1109/TVCG.2015.2467091 (pages 12, 22, 23, 25–27, 50, 58, 111, 112).

[179] 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*. ACM. 2014 (pages 23, 27, 37, 40, 50, 51, 57).

[180] D. W. Scott. "On optimal and data-based histograms". In: *Biometrika* (1979) (page 15).

[181] Vidya Setlur, Sarah E. Battersby, Melanie Tory, Rich Gossweiler, and Angel X. Chang. "Eviza: A Natural Language Interface for Visual Analysis". In: *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. UIST '16. Tokyo, Japan: ACM, 2016. ISBN: 978-1-4503-4189-9. DOI: 10.1145/2984511.2984588 (page 96).

[182] *Shiny by RStudio*. https://shiny.rstudio.com. June 2016 (pages 23, 28).

[183] B. Shneiderman. “Dynamic queries for visual information seeking”. In: *IEEE Software* (Nov. 1994). DOI: 10.1109/52.329404 (pages 10, 105).

[184] Ben Shneiderman. “The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations”. In: *Proceedings of the 1996 IEEE Symposium on Visual Languages*. VL ’96. Washington, DC, USA: IEEE Computer Society, 1996. ISBN: 0-8186-7508-X (page 10).

[185] Ronell Sicat, Jiabao Li, JunYoung Choi, Maxime Cordeil, Won-Ki Jeong, Benjamin Bach, and Hanspeter Pfister. “DXR: A Toolkit for Building Immersive Data Visualizations”. In: *IEEE Vis (Proc. InfoVis)*. 2018 (page 56).

[186] Parag Singla and Pedro Domingos. “Discriminative Training of Markov Logic Networks”. In: *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2*. AAAI’05. Pittsburgh, Pennsylvania: AAAI Press, 2005. ISBN: 1-57735-236-x (pages 70, 80).

[187] Parag Singla and Pedro Domingos. “Entity Resolution with Markov Logic”. In: *Proceedings of the Sixth International Conference on Data Mining*. ICDM ’06. Washington, DC, USA: IEEE Computer Society, 2006. ISBN: 0-7695-2701-9. DOI: 10.1109/ICDM.2006.65 (page 70).

[188] Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, and Yannis Kotidis. “Dwarf: Shrinking the PetaCube”. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’02. Madison, Wisconsin: ACM, 2002. ISBN: 1-58113-497-5. DOI: 10.1145/564691.564745 (page 17).

[189] A. M. Smith and M. Mateas. “Answer Set Programming for Procedural Content Generation: A Design Space Approach”. In: *IEEE Transactions on Computational Intelligence and AI in Games* (Sept. 2011). ISSN: 1943-068X. DOI: 10.1109/TCIAIG.2011.2158545 (page 65).

[190] Adam M. Smith, Eric Butler, and Zoran Popovic. “Quantifying over play: Constraining undesirable solutions in puzzle design”. In: *Proceedings of the Foundations of Digital Games, FDG*. 2013 (page 69).

[191] Adam M Smith, Erik Andersen, Michael Mateas, and Zoran Popović. “A case study of expressively constrainable level design automation tools for a puzzle game”. In: *Proceedings of the International Conference on the Foundations of Digital Games*. ACM. 2012. DOI: 10.1145/2282338.2282370 (page 69).

[192] Tableau Software. *Tableau Desktop*. https://tableau.com/products/desktop (page 105).

[193] Tableau Software. *Tableau Public*. https://public.tableau.com/ (page 108).

[194] Timo Soininen and Ilkka Niemelä. "Developing a Declarative Rule Language for Applications in Product Configuration". In: *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*. PADL '99. London, UK, UK: Springer-Verlag, 1998. ISBN: 3-540-65527-1 (page 69).

[195] Arjun Srinivasan and John Stasko. "Natural language interfaces for data analysis with visualization: Considering what has and could be asked". In: *Proceedings of EuroVis*. 2017 (page 96).

[196] John Stasko, Carsten Görg, and Zhicheng Liu. "Jigsaw: Supporting Investigative Analysis Through Interactive Visualization". In: (2008). ISSN: 1473-8716 (page 133).

[197] Charles D Stolper, Adam Perer, and David Gotz. "Progressive visual analytics: User-driven visual exploration of in-progress analytics". In: INFOVIS '14 (2014). DOI: 10.1109/TVCG.2014.2346574 (page 136).

[198] C. Stolte, D. Tang, and P. Hanrahan. "Polaris: a system for query, analysis, and visualization of multidimensional relational databases". In: *IEEE Transactions on Visualization and Computer Graphics* (2002). ISSN: 1077-2626. DOI: 10.1109/2945.981851 (pages 10, 12, 20, 134).

[199] Chris Stolte, Diane Tang, and Pat Hanrahan. "Polaris: A System for Query, Analysis, and Visualization of Multidimensional Databases". In: *Commun. ACM* (Nov. 2008). ISSN: 0001-0782. DOI: 10.1145/1400214.1400234 (pages 23, 25, 31).

[200] Michael Stonebraker. "The Case for Shared Nothing". In: *Database Engineering* (1986) (page 17).

[201] H. A. Sturges. "The choice of a class interval". In: *Journal of the American Statistical Association* (1926) (page 15).

[202] Deborah F Swayne, Duncan Temple Lang, Andreas Buja, and Dianne Cook. "GGobi: evolving from XGobi into an extensible framework for interactive data visualization". In: *Computational Statistics & Data Analysis* (2003) (page 27).

[203] D. A. Szafir. "Modeling Color Difference for Visualization Design". In: *IEEE Transactions on Visualization and Computer Graphics* (Jan. 2018). ISSN: 1077-2626. DOI: 10.1109/TVCG.2017.2744359 (page 63).

[204] Danielle Albers Szafir, Steve Haroz, Michael Gleicher, and Steven Franconeri. "Four types of ensemble coding in data visualizations". In: *Journal of Vision* (2016). ISSN: 1534-7362. DOI: 10.1167/16.5.11 (pages 13, 67).

[205] Bureau of Transportation Statistics. *On-Time : Reporting Carrier On-Time Performance (1987-present)*. Accessed: 2019-08-12. 2019 (pages 101, 111, 121, 122, 151).

[206] Edward R. Tufte. *The Visual Display of Quantitative Information*. Cheshire, CT, USA: Graphics Press, 1986. ISBN: 0-9613921-0-X (page 9).

[207] J. W. Tukey and M. B. Wilk. "Data Analysis and Statistics: An Expository Overview". In: *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*. AFIPS '66 (Fall). San Francisco, California: ACM, 1966. DOI: 10.1145/1464291.1464366 (page 11).

[208] John W Tukey. *Exploratory data analysis*. Reading, Mass., 1977 (pages 1, 11, 133).

[209] John W. Tukey. "We Need Both Exploratory and Confirmatory". In: *The American Statistician* (1980). DOI: 10.1080/00031305.1980.10482706 (page 11).

[210] Cagatay Turkay, Erdem Kaya, Selim Balcisoy, and Helwig Hauser. "Designing Progressive and Interactive Analytics Processes for High-Dimensional Data Analysis". In: *IEEE Transactions on Visualization and Computer Graphics*. INFOVIS '16. IEEE, 2017. DOI: 10.1109/TVCG.2016.2598470 (pages 136, 138, 149).

[211] Jacob VanderPlas, Brian Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. "Altair: Interactive Statistical Visualizations for Python". In: *Journal of Open Source Software* (Dec. 10, 2018). ISSN: 2475-9066 (pages 54, 56, 59, 165).

[212] *vegalite: R ggplot2 "bindings" for Vega-Lite*. https://github.com/hrbrmstr/vegalite. Aug. 2017 (page 56).

[213] *VegaLite.jl: Julia bindings to Vega-Lite*. https://github.com/fredo-dedup/VegaLite.jl. Aug. 2017 (page 56).

[214] *Vegas: The missing MatPlotLib for Scala + Spark*. https://github.com/aishfenton/Vegas. Aug. 2017 (page 56).

[215] Richard L Villars, Carl W Olofson, and Matthew Eastwood. "Big data: What it is and why you should care". In: *White Paper, IDC* (2011) (page 2).

[216] Richard Wesley, Matthew Eldridge, and Pawel T. Terlecki. "An analytic data engine for visualization in tableau". In: *Proceedings of the 2011 international conference on Management of data - SIGMOD '11*. ACM. 2011. ISBN: 9781450306614. DOI: 10.1145/1989323.1989449 (page 20).

[217] Hadley Wickham. "A layered grammar of graphics". In: *Journal of Computational and Graphical Statistics* (2010) (pages 23, 25).

[218] Hadley Wickham. "Bin-summarise-smooth : A framework for visualising large data". In: *InfoVis 2013* (2013) (pages 13, 15, 105, 135).

[219] Hadley Wickham. *Ggplot2: Elegant Graphics for Data Analysis*. 2nd. Springer Publishing Company, Incorporated, 2009. ISBN: 0387981403, 9780387981406 (page 12).

[220] Hadley Wickham and Lisa Stryjewski. "40 years of boxplots". In: (2011) (page 136).

[221] Adalbert Wilhelm. "User interaction at various levels of data displays". In: *Computational statistics & data analysis* (2003) (page 28).

[222] Leland Wilkinson. *The grammar of graphics*. Springer Science & Business Media, 2006 (pages 12, 19, 25, 60).

[223] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. "Towards a general-purpose query language for visualization recommendation". In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics - HILDA '16*. 2016. ISBN: 9781450342070. DOI: 10.1145/2939502.2939506 (pages 8, 60, 63–65, 67, 68, 73, 86).

[224] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. "Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations". In: *IEEE Transactions on Visualization and Computer Graphics* (Jan. 2016).

DOI: 10.1109/tvcg.2015.2467191 (pages 8, 12, 21–23, 29, 31, 56, 59, 60, 63, 67, 68, 73, 97, 133, 158).

[225] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. "Voyager 2: Augmenting Visual Analysis with Partial View Specifications". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (2017). DOI: 10.1145/3025453.3025768 (pages 8, 56, 60, 67).

[226] Jo Wood, Alexander Kachkaev, and Jason Dykes. "Design Exposition with Literate Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* (2018). DOI: 10.1109/TVCG.2018.2864836 (page 56).

[227] Eugene Wu, Leilani Battle, and Samuel R Madden. "The case for data visualization management systems: vision paper". In: *Proceedings of the VLDB Endowment* (2014) (page 20).

[228] J. S. Yi, Y. a. Kang, and J. Stasko. "Toward a Deeper Understanding of the Role of Interaction in Information Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* (2007). ISSN: 1077-2626. DOI: 10.1109/TVCG.2007.70515 (pages 1, 25, 50).

[229] Ji Soo Yi, Youn-ah Kang, John T. Stasko, and Julie A. Jacko. "Understanding and Characterizing Insights: How Do People Gain Insights Using Information Visualization?" In: *Proceedings of the Workshop on BEyond Time and Errors: Novel evaLuation Methods for Information Visualization*. BELIV '08. ACM, 2008. ISBN: 978-1-60558-016-6. DOI: 10.1145/1377966.1377971 (page 134).

[230] E. Zgraggen, A. Galakatos, A. Crotty, J. Fekete, and T. Kraska. "How Progressive Visualizations Affect Exploratory Analysis". In: *IEEE Transactions on Visualization and Computer Graphics* (2017). ISSN: 1077-2626. DOI: 10.1109/TVCG.2016.2607714 (pages 103, 106).