TOWARDS EXPRESSIVE AND SCALABLE VISUAL DATA EXPLORATION

BY

TARIQUE ASHRAF SIDDIQUI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Assistant Professor Aditya Parameswaran, Chair
Professor Jiawei Han
Professor Karrie Karahalios
Dr. Çağatay Demiralp, Megagon Labs

ABSTRACT

Data visualization is the primary means by which data analysts explore patterns, trends, and insights in their data. Despite of their growing popularity, existing visualization tools (e.g., Tableau, PowerBI, Excel) are limited in their ability to automatically find desired visualizations or insights. As a result, the process of visual data exploration is manually-intensive and time-consuming, and becomes simply unsustainable as the complexity and scale of the dataset increases. In this dissertation, we address the shortcomings of existing visualization tools by facilitating expressive and scalable data exploration. In particular, we propose two systems: 1) Zenvisage—for effortlessly and efficiently finding visualizations with specific patterns or insights among large collections, and 2) ShapeSearch—for finding visualizations based on fine-grained and fuzzy patterns. Both Zenvisage and ShapeSearch draw heavily from use-cases in a variety of domains including biology, battery science, and cosmology, and provide expressive visual primitives to capture a large variety of data exploration needs. Backed by formal algebra and semantics, the visual primitives help operate on collections of visualizations (e.g., by composing, filtering, comparing, matching, and sorting) based on visual trends and patterns. Furthermore, these systems support built-in recommendations, and multiple flexible query specification mechanisms, including intuitive interactions and natural language, simultaneously catering to the needs of both novice and expert analysts. To automatically parse and execute visual queries efficiently, Zenvisage and ShapeSearch support a suite of optimizations, that can traverse and evaluate a large number of visualizations within interactive response times. We document performance results, as well as results from multiple user- and case-studies that demonstrate that users are able to effectively use Zenvisage and ShapeSearch to eliminate error-prone and tedious exploration and directly identify desired visualizations.

To my parents, for their love and support.

ACKNOWLEDGMENTS

My Ph.D. work would not have been possible without the support of an incredible number of people who have personally or professionally taught me many things, given me unique opportunities, and believed in me.

First and foremost, I would like to thank my advisor, Professor Aditya Parameswaran, for being everything I could ever ask for in an advisor. Aditya has played an instrumental role in my professional and personal growth ever since he took me under his wing. Every interaction with Aditya has been an absolute pleasure: discussing high-level ideas, designing algorithms and building systems, debugging pesky performance issues, learning how to write effectively, figuring out internships and job opportunities, or talking about life in general. During all these years, Aditya gave me the complete freedom to work on the problems I liked; even supporting my other projects with external researchers that go beyond this dissertation. His teachings, distilled from our interactions, will stay with me for many years to come. As I move into the next phase of my career, I hope I could be as energetic, lively, and collaborative as Aditya is, and someday be able to write and give talks as persuasively and elegantly as he can.

At the Univeristy of Illinois, Urbana-Champaign, I have had the opportunity to interact with and learn from some of the best minds in Computer Science. They have influenced the way I think and I am thankful for my interactions with every one of them. In particular, the work in this dissertation would not have been possible without the guidance of Professor Karrie Karahalios, who has been an amazing advisor and collaborator throughout my Ph.D.. Many practical inputs from her have substantially improved the user experience and design of the systems presented in this dissertation. I am fortunate to have worked with Professor Jiawei Han during the early years of my Ph.D. I learnt a lot not only from his deep insights in the areas of heterogeneous networks and text mining but also from his down-to-earth attitude. He has been the most understanding and caring person I have ever had a chance to work with. I would like to thank Çağatay Demiralp, who along with Karrie and Jiawei, served on my committee, and provided valuable feedback during our discussions that ultimately made this dissertation stronger and richer.

During my internships at Microsoft and Tableau, I have had the privilege of learning from some of the eminent database and visualization researchers in the industry. I am thankful to Alekh Jindal for introducing me to the wonderful area of cloud databases and exposing me to many interesting problems at the intersection of machine learning and databases, these have since shaped my research tastes. I am incredibly grateful to Surajit Chaudhuri and Vivek Narasayya for their mentoring and constant encouragement over the past year, and for so many interesting discussions

that deepened my understanding of databases systems, and taught me new ways and styles of doing impactful research on industry-scale problems. I am also thankful to Justin Talbot and Jock Mackinlay for a great and productive summer at Tableau Research, during the early years of my Ph.D.. My interactions with them equipped me with the deeper insights into the problems intersecting databases and visualization, something that continues to benefit me even today.

Many outstanding students have contributed their time, efforts, and insights into the work presented in this dissertation, and shaped my research methodology in their unique ways. In particular, I would like to thank Albert Kim, John Lee, Doris Lee, Edward Xue, Paul Luh, and Zesheng Wang for so fruitful and productive collaboration. The work on Zenvisage and ShapeSearch would have been impossible without their contribution, and I feel privileged to have had the opportunity to work with them.

I am proud to have spent time and learnt from my fellow students, who have been on this journey with me. In addition to those mentioned above, I would like to thank Sajjadur Rahman, Silu Huang, Mangesh Bendre, Vipul Venkataraman, Liqi Xu, Ayush Jain, Yihan Gao, Stephen Macke, Doris Xin, Tana Wattanawaroon, and other members of the group for their stimulating discussions and help over the years. Together, we have explored new projects; brainstormed countless ideas; shared frustrations and advice; traveled to conferences; enjoyed fun activities and games; and so much more. No matter where life takes us, I hope we always remain friends.

Navigating graduate school would not have been as easy without the help of the wonderful staff in the Computer Science Department. I would like to especially thank (in alphabetical order): Maggie Metzger Chapell, Viveka Kudaligama, Kara Lynn MacGregor, and Lisa Yanello for being extremely supportive every time I reached out.

I am blessed to have a loving and supportive family who instilled in me the desire to strive for the best, right from my childhood. No words can express my gratitude to my parents, Asma and A A Siddiqui, who made immense sacrifices for my education. Indeed, this dissertation is a culmination of their dedication and sacrifices. I am thankful to my siblings, Tahir and Tamanna, and my sister-in-law, Zeenat, for being a strong pillar of support, and for being always there — celebrating my successes and encouraging me through disappointments. Last but not the least, I would like to sincerely thank Arshna, my last two years with her have not only been very blissful but also productive. She has been very calm, supportive, and sweet even though I could not spend enough time with her.

Finally, I would like to extend my gratitude to all those whose names I may have missed by mistake. Many good things have happened to me during the course of my Ph.D., and I have no doubt I have been quite lucky.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Our Vision 3
1.2 Thesis Contributions and Outline 3

CHAPTER 2 RELATED WORK 6
2.1 Visual Analytic Systems 6
2.2 Visualization Recommendation Systems 6
2.3 Similarity Metrics for Visualizations 7
2.4 Symbolic Pattern Mining and Querying 7
2.5 Statistical Packages and Programming Libraries 8
2.6 OLAP Browsing Tools 8
2.7 Data Mining Languages 9
2.8 Overall Takeaways 9

CHAPTER 3 ZENVISAGE SYSTEM OVERVIEW 10
3.1 The Front-End 10
3.2 The Back-End 15
3.3 Recommendations via Representative and Outlier Search. 16
3.4 Overall Takeaways 17

CHAPTER 4 ZQL: AN EXPRESSIVE VISUALIZATION QUERY LANGUAGE 18
4.1 Overview 18
4.2 Formalization 18
4.3 Expressiveness 28
4.4 Visual Exploration Algebra Operators 32
4.5 Proof of Visual Exploration Completeness 34
4.6 Discussion of Capabilities and Limitations 44
4.7 Overall Takeaways 45

CHAPTER 5 ZENVISAGE QUERY OPTIMIZATION 46
5.1 Basic Translation 46
5.2 Optimizations 49
5.3 Experimental Study 52
5.4 Overall Takeaways 57

CHAPTER 6 ZENVISAGE USER EVALUATION 59
6.1 Analyst Interviews and Task Selection 59
6.2 User Study Methodology 60
6.3 Key Findings 63
6.4 Overall Takeaways and Future Work 64

CHAPTER 7 SHAPESEARCH SYSTEM OVERVIEW 66
7.1 Motivation 66
7.2 Overview of Our Approach 68
7.3 ShapeSearch System Overview 69
7.4 ShapeSearch Algebra 71
7.5 Formal Semantics of ShapeQuery 76
7.6 Scoring Methodology 77
7.7 Parsing and Translation 80
7.8 Additional Related Work 83
7.9 Overall Takeaways 84

CHAPTER 8 FUZZY SHAPE MATCHING 86
8.1 The Dynamic Programming Algorithm 86
8.2 A Pattern-Aware Bottom-Up Approach 88
8.3 Additional Optimizations 94
8.4 Performance Evaluation 95
8.5 Overall Takeaways 99

CHAPTER 9 SHAPESEARCH USER EVALUATION 100
9.1 User Study 100
9.2 Case Study: Genomics 104
9.3 Overall Takeaways and Future Work 106

CHAPTER 10 CONCLUSION AND FUTURE WORK 108

REFERENCES 111

CHAPTER 1: INTRODUCTION

With massive volumes of data available in every sphere of human endeavor, there is a pressing need for tools that make it simpler for analysts — both novice and expert — to extract insights, test hypotheses, and make sense of trends and patterns in large datasets. Visual data exploration is typically the first step in studying large datasets [1]. Visual analytics tools such as Microsoft Excel, Qlikview, and Tableau allow analysts to easily specify and generate a desired visualization by selecting attributes and the data subset of interest. While these tools are highly effective at creating user-specified visualizations, as the scale and complexity of datasets increase finding visualizations with interesting trends or patterns becomes extremely cumbersome and challenging.

To illustrate further, let us look at the challenges of several groups who have been hobbled by the ineffectiveness of current data exploration tools.

Example 1.1 (Genomics Data Analysis) Clinical researchers at NIH-funded center at the University of Illinois and Mayo Clinic are interested in studying data from clinical trials, in the context of gene and protein data. One such task involves finding pairs of genes that *visually explain the differences* in clinical trial outcomes. Current tools require the researchers to generate and manually evaluate tens of thousands of scatter plots for whether the outcomes (positive vs. negative) can be clearly distinguished in the scatter plot.

Another more fine-grained task is to study changes in gene expression for investigating the impact of drugs on disease treatment. When influenced by an external factor, a gene can get induced (up-regulated), or repressed (down-regulated), or can have both induced or repressed patterns within a certain time window. Moreover, there can be further variations in the scale, repetitions, or the rate of change of these patterns depending on the type and influence of external factors. Based on their domain understanding and past experience, researchers first hypothesize the expected change in expression that an affected gene should depict, and then manually inspect them for the hypothesized patterns.

Example 1.2 (Battery Science) Battery scientists at Carnegie Mellon University perform visual exploration of datasets containing solvent properties at various scales—molecular, meso, and continuum—to design better batteries. A specific task may involve finding solvents with some desired behavior: e.g., those whose solvation energy of Li^+ vs. the boiling point is an *increasing trend*. To do this using current visual analytic tools, these scientists manually examine these plots for each of the thousands of solvents.

Example 1.3 (Cosmological Data Analysis) Dark Energy Survey scientists study the history and

makeup of our universe by understanding the variation in properties of astronomical objects (e.g., galaxies) over time. For example, astronomers monitor the brightness (luminosity) of stars over time to search for new planetary objects—a dip in brightness is symbolic of a planetary object passing between the star and the telescope, and the width and the degree of dips are used for characterizing these planetary objects. Based on their domain understanding, astronomers apply on-the-fly filters (on values, specific attributes, and so on), or modify the scale (e.g., from seconds to days to months) to change the shape of the trend. Scientists currently identify such insights by either manually searching for visualizations with desired variations in properties, or write custom scripts in Python or R. Often, writing code for each new use-case and manually optimizing them becomes as tedious as manual searching of visualizations to find patterns.

In all these examples, the recurring theme is the manual examination of a large number of generated visualizations for one or more specific visual patterns. We outline the key characteristics of these examples that highlight why the process of data exploration is often tedious, time-consuming, and increasingly unsustainable with current visualization tools.

1. Too many visualizations to explore. Data analysts often explore subsets of data for interesting patterns, by manually specifying visualizations formed by different combination of attributes. Even for a modest dataset with a small number of attributes, the number of visualizations that need to be examined is often in the hundreds or thousands. This need to manually specify and examine every visualization hampers rapid analysis and exploration.

2. Approximate notion of desired patterns. Many application domains such as bioinformatics, astronomy, and finance require non-traditional database support for large data sequences such as time series where analysts often search for patterns that fit some approximate notion of what they want. They are often not interested in the specific details as much as the overall shape of the patterns.

3. Ad-hoc. Based on their domain understanding and what they discover during the data exploration, data analysts typically look for visualizations with previously unknown patterns and apply constraints on-the-fly. For instance, biologists often search for a pattern in a group of genes, that is similar to a pattern just discovered in another group. Existing pattern search tools, on the other hand, usually require heavy precomputation and indexing, thereby limiting ad-hoc exploration. For such cases, generating visualizations can take hours, impeding interaction, preventing exploration, and delaying the extraction of insights.

1.1 OUR VISION

Our goal is to build systems that can capture the aforementioned characteristics and *automatically "fast-forward" to the desired insights,* thereby minimizing the significant burden on the part of the data analysts in scenarios like the ones described above. In order to achieve this goal, we focus on three aspects.

1. Expressive visual primitives. We note that one of the major hindrances with existing visual analytic tools is that they do not support enough mechanisms to let users specify the patterns they are interested in, apart from the visualization generation and encoding aspects. However, data exploration needs in most of these scenarios can be captured within a common set of operations (e.g., comparison, sorting, and filtering) on collections of visualizations based on trends or patterns. Thus, in this thesis, we focus on developing a minimal set of operators and primitives that can capture the tasks that often require tedious manual examination.

2. Flexible specification mechanisms and built-in recommendations. Since a substantial fraction of our targeted end-users, such as our biomedical researchers, are often not proficient in programming, we need to support interactive and more natural shortcuts for common interaction "idioms" as well as support built-in recommendations to guide users towards their desired goals.

3. Automated and scalable execution. For fast-forwarding to desired insights, we should be able to automatically compile and execute the visual primitives, traversing over large collections of visualizations to find the ones that match the criteria. Often, even a single visual query can lead to the generation of thousands of visualizations; executing each one independently as an aggregate query would take several hours, rendering the tool somewhat useless. In this thesis, we focus on developing a suite of optimizations that are optimized for visual queries and can lead to substantial improvements over the naive schemes adopted within existing visualization tools.

1.2 THESIS CONTRIBUTIONS AND OUTLINE

This dissertation is divided into two parts, covering two new systems — Zenvisage and ShapeSearch, both implementing expressive visual data exploration primitives and scalable query processing algorithms for accelerating the search of desired visual trends. The first part, consisting of Chapters 3 to 6, covers the Zenvisage system, and the second part, consisting of Chapters 7 to 9, presents the ShapeSearch system. The outline of the dissertation is as follows:

In Chapter 2, Related Work, we survey the related work from the visualization, database and data-mining that our work draws on, as well as explain the limitations of existing tools.

PART I

In Chapter 3, Zenvisage System Overview, we introduce our system, Zenvisage, and describe the user-experience of someone using Zenvisage, as well as give an overview of the different components of the system.

In Chapter 4, ZQL: An Expressive Visualization Query Language, we introduce ZQL, a visual query language that forms the core of Zenvisage. We describe how ZQL is powerful enough to capture the use cases discussed in this chapter. We describe the syntax and semantics of ZQL and formalize the notion of a *visual exploration algebra*, an analog of relational algebra, describing a core set of capabilities for any language that supports efficient visual data exploration. We demonstrate that ZQL is *complete* in that it subsumes these capabilities.

In Chapter 5, Zenvisage Query Optimization, we explain how Zenvisage automatically translates and executes ZQL over large collections of visualizations. We show that Zenvisage can leverage any relational database system as a back-end, and develop a number of simple optimizations that accelerates the execution of ZQL queries by minimizing the number of issued SQL queries.

In Chapter 6, Zenvisage User Evaluation, we present findings from our user study focused on evaluating the effectiveness and usability of Zenvisage.

The contents of Chapter 3 have appeared in our CIDR'17 [2] paper, and the contents of Chapters 4 to 6 have appeared in our VLDB'17 [3] paper as well as an extended technical report [4]. The work was done in collaboration with Albert Kim, John Lee, and several masters and undergraduate students. I was responsible for the design and development of the first version of the system, and jointly responsible for the query language, query optimizations, user studies, and performance experiments.

PART II

In Chapter 7, ShapeSearch Overview, we motivate the need for flexible and fuzzy pattern matching, and introduce our system, ShapeSearch, a flexible and efficient shape matching system for trendlines. ShapeSearch supports flexible query specification mechanisms including regular expressions and natural language. We describe the system design, followed by the ShapeQuery algebra, comprising the primitives and operators that help express a wide variety of use-cases. We also describe how we translate natural language queries into the ShapeQuery algebra.

In Chapter 8, Fuzzy Shape Matching, we introduce the problem of fuzzy shape matching in ShapeSearch and present algorithms for this problem along with query-aware optimizations and perceptually-aware scoring methodologies for comparing and ranking shapes of visualizations.

In Chapter 9, ShapeSearch User Evaluation, we present a user study and a genomics case study, focusing on evaluating the expressiveness, effectiveness, and usability of ShapeSearch.

The contents of Chapters 7 to 9 are going to appear in our SIGMOD'20 paper, and have also appeared in our short VLDB'18 demo paper [5] as well as our extended technical report [6]. The work was done in collaboration with Zesheng Wang and Paul Luh. I was responsible for the design and development of the system, algebra, visual regular expression, natural language parser, user study, case-study as well as the performance experiments.

In Chapter 10, Conclusions and Future Work, we conclude and present open future research directions stemming from this line of work.

CHAPTER 2: RELATED WORK

Our work in this dissertation draws on many prior papers in the areas of visualization, databases, and data mining.

2.1 VISUAL ANALYTIC SYSTEMS

Querying data through direct manipulation to modify the shape of a desired output visualization is an intuitive, yet powerful method of data exploration. For this reason, researchers have designed a number of new interfaces that allow users to do just this. Our work extends prior research in "sketch-to-query" systems, such as QuerySketch and Google Correlate, which allow users to draw or modify an existing visualization to form a query, which the system then uses to browse the dataset for similar-looking visualizations [7, 8]. Wattenberg was the first to create a functional prototype of such a system [8]. Later, Google Labs launched the "Search by Drawing" feature in Google Correlate, which allows users to compare trends in search terms within a specific time frame [9]. Ryall et al. and Holz et al. have developed similar interfaces that allow users to sketch on a graph to identify matching trends and patterns [10, 11]. Hochheiser et al. took a slightly different approach and developed TimeSearcher, a tool that allows users to directly set constraints by drawing rectangular boxes on data graphs [12]. TimeSearcher's additional functions include pattern inversions and "leaders and laggards" to help identify trends and patterns in the data. The aforementioned papers evaluate their systems with a single data type, and a single set of visualizations of interest; we extend these papers by implementing more general systems to accommodate multiple modes of specification, multiple data types, multiple sets of visualizations, and multiple data sets. We add necessary customization capabilities to the sketching interface that can adapt to various forms of needs of analyst needs and inputs—for instance, applying search to only a subset of points, providing a representative visualization as a starting point with the option to consider or ignore scale when needed, specifying a function as an input, supporting drag-and-drop, drawing a new trend, or modifying an existing trend.

2.2 VISUALIZATION RECOMMENDATION SYSTEMS

There has been extensive research on how to provide users with appropriate visualizations with minimal user input. Data analysts frequently work with large amounts of data in various formats, and are constantly called upon to make decisions about what exactly to visualize. Pioneering research by Mackinlay et al. envisioned and developed an "artificial assistant" that generated ef-

fective visualizations for its users [13, 14]. Recent work by Wongsuphasawat et al. has extended this work through Voyager [15], a system that recommends multiple charts based on statistical and perceptual measures. Several researchers have focused on identifying outliers in data. Vartak et al. tackled this problem with SeeDB, a system for visualizing data that deviates from reference trends [16]. Profiler and Scorpion, developed by Kandel et al. and Wu et al. respectively, focus on providing insights to users by working from anomalies [17, 18]. Building on previous work in this area, we devised a simple visualization-recommendation system for users, and examined how they took advantage of this feature during their everyday workflow with their own datasets.

2.3 SIMILARITY METRICS FOR VISUALIZATIONS

Identifying similar trends in data is difficult. The definition of "similar" is subjective by nature, varying across personal experience and across different data types. Many researchers have explored the categorization of time-series data based on the shape of the line graph [12, 19], and similar approaches have also been applied to scatter plots [20, 21]. Computational techniques have also been developed to identify similar trends for users. One widely used approach labels two trends similar if their computed Euclidean Distance among all the trends in the data is the smallest value. Other popular techniques include Dynamic Time Warping (DTW) [22], which warps time axes to align pairs of trends. Google Correlate uses a combination of the Approximate Nearest Neighbor system [23] and a custom distance function to balance precision and speed [7]. Additionally, there has been a plethora of work on indexing and retrieving for a fixed set of time-series, typically via the Euclidean distance metric [24–30]. In short, much of the previous work in this field has focused on the technical aspects of identifying shapes and patterns in a fixed set of time series, without an appropriate user interface. Our research is general in that it does not require pre-indexing of time series (nor is it restricted to time series), and instead adapts dynamically to any candidate set of visualizations selected by the user during ad-hoc visual data exploration. In addition, we incorporate a novel customizable interface for the searching and specification of desired patterns, and provides recommendations, and has been tested via case studies with users to observe what patterns they identify in their own data, and focuses on how such visual query systems will complement their workflow.

2.4 SYMBOLIC PATTERN MINING AND QUERYING

. There are a number of symbolic approaches [31–33] that discretize a time series into a sequence of symbols (or events), and use variants of string matching algorithms such as edit-distance

or longest common subsequence to find similar sequences present in the input trendline. Having a fewer number of discrete symbols as opposed to continuous points also helps in making similarity search efficient. Shape Definition Language (SDL) [34], like ShapeSearch, allows users to search for trendlines with specific pattern sequences via a structured keyword-based input. Like other indexing-based approaches [33, 35–38], SDL segments each time-series into fixed-length segments, and indexes them with closest matching pattern symbols in advance—trading-off flexibility and precision for efficiency. Similarly, SPIRIT [39] is a regular-expression-based tool that incorporates user-controlled constraints for mining frequent patterns in sequence databases. The major downside with this line of work (and a key distinguishing factor compared to our work) is that the detailed information about each trendline is difficult to faithfully abstract in advance to answer all possible shape queries. Moreover, a time-series can have varying shape for different filters and aggregations, often applied on-the-fly during ad hoc data exploration (as we illustrate in Chapter 7). Therefore, instead of representing a trendlines as a sequence of patterns, or using pre-built indexes, in this work, we develop online query-aware optimizations for efficient pattern matching. Allowing real-time pattern matching over arbitrary subsets of data with varying granularities and aggregations, makes our work more suitable for ad hoc data exploration scenarios.

2.5 STATISTICAL PACKAGES AND PROGRAMMING LIBRARIES

Statistical analysis tools [40–43] support complex data mining primitives, but require extensive knowledge of the underlying algorithms and parametrization, limiting their use to experts, e.g., a user can select to use decision trees, with a certain depth and splitting criteria on their dataset using any of these tools. Programming libraries such as Weka [44] and Scikit-learn [45] support embedding machine learning and data mining within programs. Indeed, complex statistical tasks such as classification, regression, and dimensionality reduction are hugely important, but require programming ability or an understanding of machine learning and statistics to be useful. In this work, we instead build systems that are substantial generalization over existing data exploration tools, aimed at identifying the patterns and trends that hold in the data, by non-programmers, which can be then used to train a machine learning algorithm later on.

2.6 OLAP BROWSING TOOLS

There has been some limited work on interactive browsing of data cubes, e.g., [46, 47]. While we may be able to reuse some of the metrics from that line of work, the work focuses on suggestions for raw aggregates (cells) to examine that are informative given past browsing, or those

that show a generalization or explanation of a specific cell, and not on two (or more) dimensional visualizations, or the full data exploration capabilities (e.g., searching for trends, patterns, outliers) that we support in our proposed systems..

2.7 DATA MINING LANGUAGES

There has been some limited work in data mining query languages, all from the early 90s, on association rule mining (DMQL [48], MSQL [49]), or on storing and retrieving models on data (OLE DB [50]), as opposed to a visual data exploration language aimed at identifying visual trends or insights.

2.8 OVERALL TAKEAWAYS

In conclusion, despite the wealth of data exploration and mining tools, selecting the"right" view on the data that reveals the "desired" insight still remains laborious and time-consuming. In particular, while visual analytics tools have made it much easier for business analysts to analyze data, the onus is on the user to manually specify the view they want to see, and then repeat this until they get the desired view. Similarly, databases are powerful and efficient but the rigidity of the syntax limits users ability to express queries like "show me visualizations where the sales is increasing over years". And finally, data mining and statistical tools are hard to use, especially by novice users, since these tools often involve extensive programming, manual optimizations (not desirable for ad-hoc querying), as well as an understanding of which data mining tool applies to what purpose.

In the subsequent chapters, we introduce two systems, namely Zenvisage and ShapeSearch, aimed towards addressing the limitations of existing tools. In particular, we discuss the research challenges and their solutions underlying these systems that facilitate expressive and scalable means for finding visualizations with the desired insights.

CHAPTER 3: ZENVISAGE SYSTEM OVERVIEW

In this chapter, we introduce Zenvisage, a visual data exploration system that accelerates the search for desired visual patterns. Zenvisage allows users to search for patterns by drawing, selecting, or modifying a pattern of interest. We first describe the user experience and various features offered through the front-end interface, and then explain how the Zenvisage back-end enables these features.

3.1 THE FRONT-END

The Zenvisage front-end is designed as a lightweight web application that runs completely within a user's browser. The application provides an intuitive graphical interface to compose queries for exploring trends and insights in data, and visualizes the output results from the back-end, taking into consideration data properties and perceptual principles.

3.1.1 User Experience

Since Zenvisage is meant to be an end-user-facing interactive data exploration tool, the user experience of the tool is hugely important in determining the utility and usability of the tool. Here, we describe the experience of an individual using Zenvisage.

In Figure 3.1, we show Zenvisage loaded with the real estate dataset.

Attribute Selection. The first step is attribute selection (Box 1). Here the user can specify the desired X axis attribute, and the desired Y axis attribute for the visualization or visualizations that the user is interested in. In this case, the user has specified the X axis is quarters (in other words, time), and the Y axis is the sold price. (By default Zenvisage assumes average as the aggregation applied to the Y axis, but this can be changed by clicking on the gear symbol next to Zenvisage.) Additionally, the user specifies the category: this is variable indexing the space of candidate visualizations the user is operating over. Here, the selected category is "metro"—indicating a metro area or township.

Summarization of Typical and Outlier Trends. As soon as the user selects the X, Y and category, immediately, Zenvisage populates Box 2 with typical or representative trends across categories, and outliers. In this case, there are three typical trends that were found across different metros (i.e. categories): one corresponding to a spike in the middle (Panama City), one to a gradual increasing trend (San Jose), and one to a trend that increased and then decreased (Reno)—most of the other

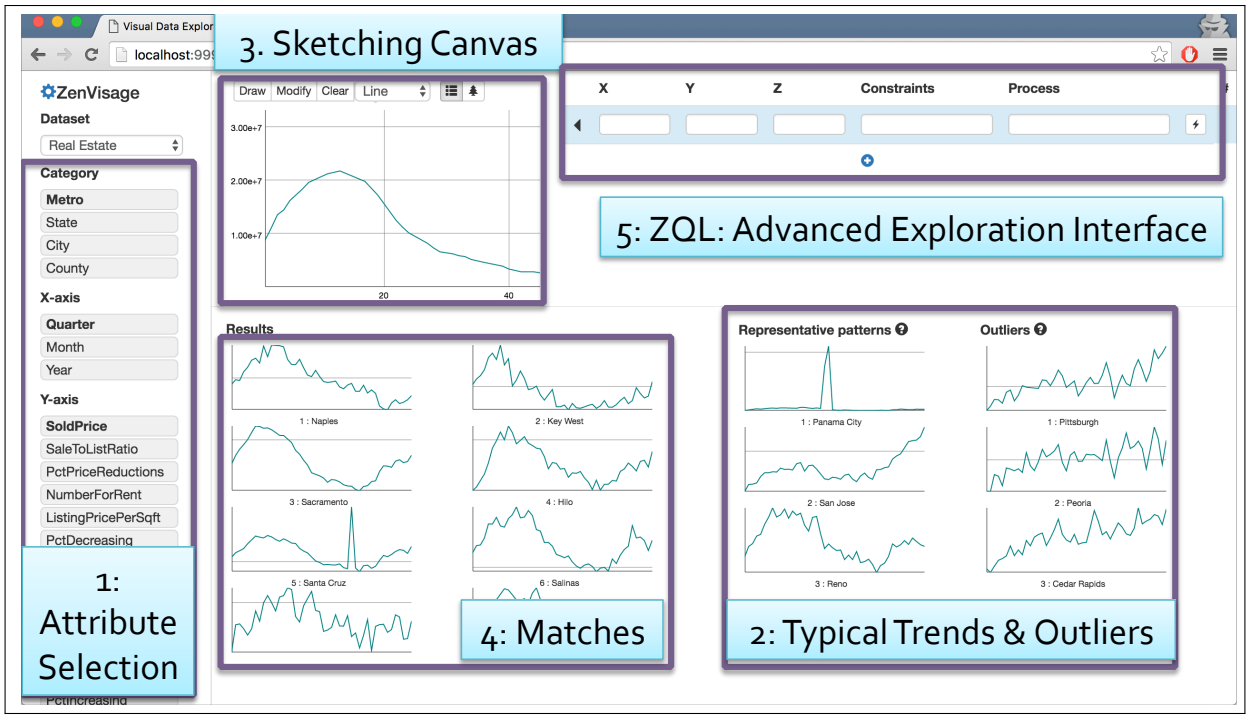


Figure 3.1: Zenvisage *interactive visual query interface*

trends were found to be similar to one of these three. The outlier visualizations (Pittsburgh, Peoria, Cedar Rapids) have a large number of seemingly random spikes.

Drawing or Drag-and-Drop Canvas. Then, in Box 3, the editable canvas, the user can either draw a shape that they are looking for, or alternatively drag and drop one of the displayed visualizations into the canvas. In this manner, the user indicates that they would like to see a similarity search starting from the shape or pattern that they have drawn or dragged onto the canvas. (Zenvisage also supports a dissimilarity search, the opposite of a similarity search, once again a non-default option hidden away behind the gear symbol.) The user is also free to edit the drawn pattern. In this figure, the user has drawn a trend which is gradually increasing up, then gradually decreasing after that.

Similarity Search Results. As soon as the user completes an interaction in Box 3, Box 4 is populated with results corresponding to visualizations (on varying the category) that are most similar to the trend in Box 3, ordered by similarity. We describe how the similarity search results are computed in the next Section. As yet another example of similarity search, see Figure 3.2, where the user has drawn a gradually increasing trend in the canvas area (or dragged an existing visualization onto the area), and the results returned below, corresponding to San Jose, Denver, and Honolulu are matches of increasing trends.

ZQL **Specification Interface.** Lastly, the user can specify a multi-line ZQL (Zenvisage query

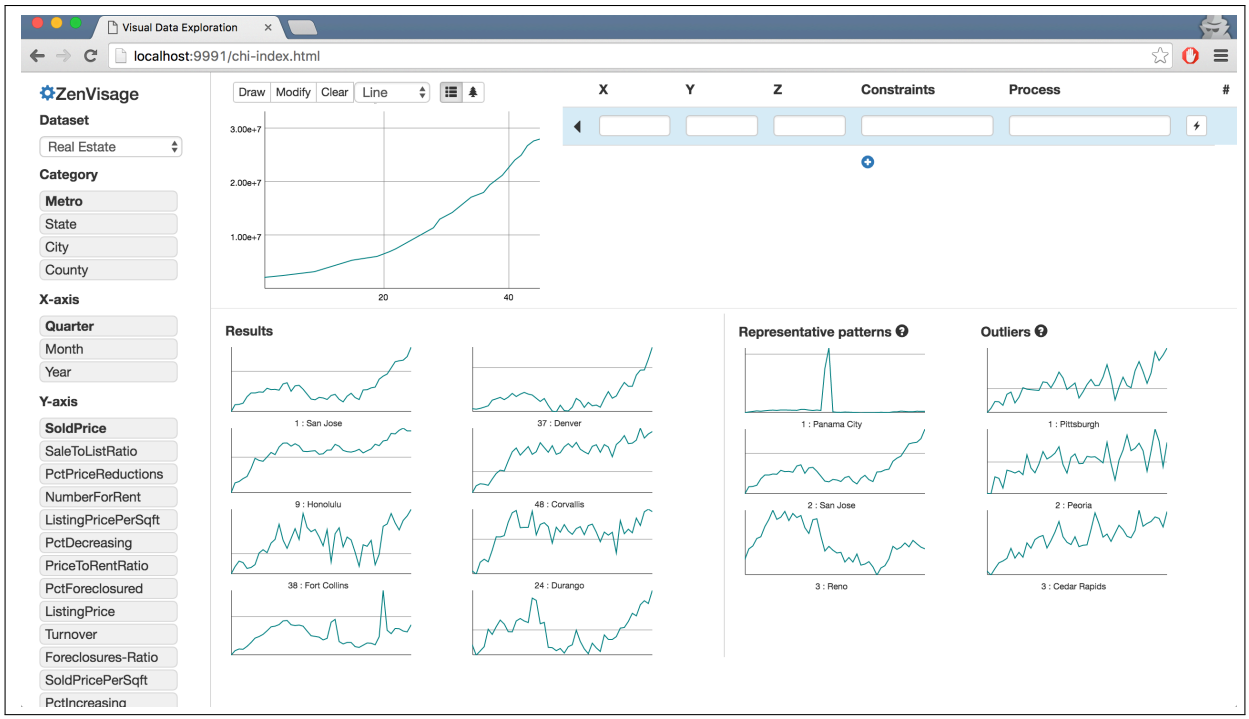


Figure 3.2: Finding cities with similar sales over year trends to a user-drawn trend

language) query in Box 5. ZQL is a high level language that aims to automate the manual visual data exploration process by allowing users to specify their desired visualization objective in a few lines. Instead of providing the low-level data retrieval and manipulation operations, users operate at the level of sets of visualizations, and compare, sort, filter, and transform visualizations as well as attributes — eventually visualized on either the X or Y axis. We provide details about our formal syntax, the expressiveness and power, and completeness of ZQL in the next chapter. Once the user completes the action, this request triggers a recomputation and redisplaying of the results shown in Box 4.

Starting from this point, the user is free to switch back and forth from ZQL to the simple interaction mode, depending on whether the user has complicated requirements or simple ones.

3.1.2 Key Components

Sketching Canvas. We created the drawing canvas using dygraph [51], the same charting library behind Google Correlate [9]. The drawing panel acts as the most frequently used method for exploring data in Zenvisage. Users can form and issue queries through direct manipulation of the drawing panel. Once the trends panel has been populated, the user can either sketch on the drawing panel, or drag-and-drop one of the representative or outlier trends from the trends panel

into the drawing panel to issue a query. When either of these actions is completed, the graph in the drawing panel is passed on to the server, which then searches for graphs that are similar to the input graph. We also equipped the drawing panel with a range selector, which allows users to make comparisons based on a subset of points instead of on the entire range of the domain. This drawing panel can be further customized as described in the Settings panel below.

Custom Query Builder. If the user would like the full expressive power of ZQL, the user may choose to use the front-end's custom query builder, which allows users to directly specify their query in the ZQL query format, depicted in Figure 7.2. The builder contains all the columns necessary for writing a ZQL query. Users write their query in a row-wise manner. For each row, they can either drag and drop the attributes from the building blocks panel on to a cell in the ZQL table. If a row requires a user-drawn input, the user may select the row and use the drawing box to draw the trend she is looking for. Iterators and outputs of previous rows may also be reused by dragging and dropping them to the current cell. Users also have the option to run and see the output of a specific row row by clicking on the "Submit Active Row" button.

Result Visualizer. The Zenvisage front-end's visualizer makes use of Vega-lite [52] grammar and the Vega visualization specification for mapping the query results to effective visualizations. The Zenvisage front-end determines the default visual encodings, such as as size and position, and uses the visualizer to map the output data according to the Vega-lite grammar.

Other Settings. Other options provided in the settings panel include an aggregation method. This setting determines how attributes are grouped into a metric: either by average value or total value. Users also can choose the number of results they want to retrieve; to display scatter plots instead of line charts; and to ignore the x-range for all computations. When the Consider x-range option is toggled on, instead of comparing trends based on x-values, the drawing panel divides the x-range into ten equally spaced bins. Essentially, it simplifies the x-range considerably, by averaging or adding all the numbers that fall into the same bin. The similarity algorithms are then carried on using the same binning system. Lastly, we provide users with an Input equation option, where they can enter equations such as $y = 2x + 10$ to input trends on the drawing panel rather than drawing them by hand.

3.1.3 Similarity Measures

The settings panel also allows users to choose between three different similarity metrics. Currently, the three metrics Zenvisage provides are Euclidean Distance, DTW, and Segmentation, each of which is described below.

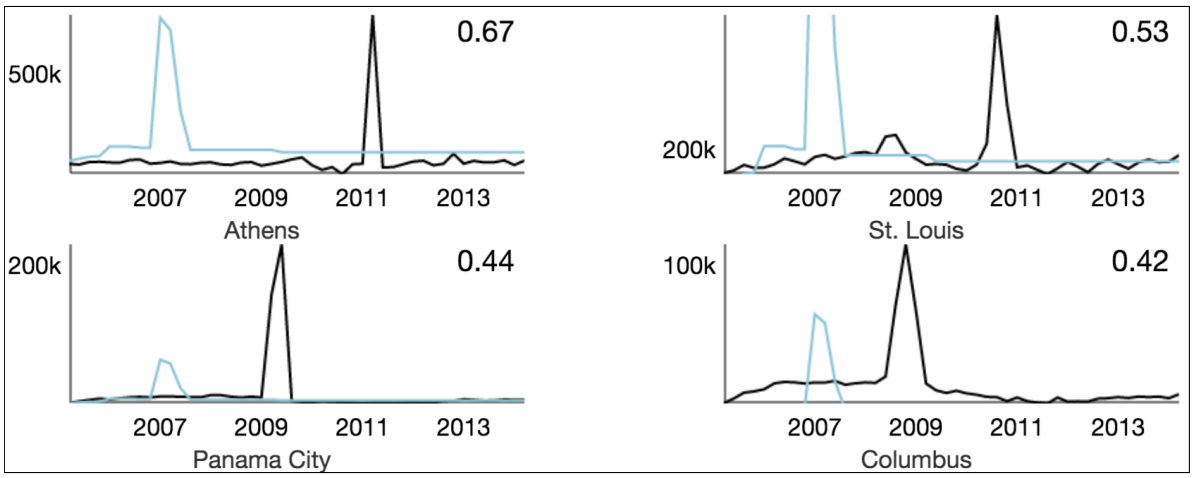


Figure 3.3: Four graphs returned from drawing a peak in the drawing panel. The trend drawn by the user is visible in teal for comparison. DTW is able to detect the similarity between the x-shifted trends.

Euclidean Distance: The Euclidean Distance between two lines is calculated by taking the square root of the sum of the differences for each x value in the data. In the case of x values in the two trends that do not overlap perfectly, we use interpolation to generate the necessary points to calculate the distance.

Dynamic Time Warping (DTW). DTW [53] measures the similarity between two lines by warping their axes to align them more closely. For instance, two trends may look similar in shape, but computing the Euclidean Distance between them may reveal that they are not similar at all. However, DTW can account for such differences, as depicted in Figure 3.3. After the algorithm aligns the two trends, the calculated distance will be minimized if the overall shape is similar.

Segmentation. While the DTW metric is very good at identifying trends with a similar shape, it is unfortunately very expensive to compute, and is sensitive to noise in the trends. We developed a custom similarity metric which combines the segmentation method described in [54] with DTW [22]. Our method is as follows. When an array of n points is provided as an input, each point is connected to create $n-1$ segments. Then, the segments are incrementally merged in bottom-up fashion one at a time—if the costs of merging them are below a user specified (or system-default) threshold. This is repeated until none of the adjacent segments can be merged further (see Figure 3.4). Once this process is complete, we use DTW to compute the distances between the trends.

We chose the above three similarity metrics to capture the breadth of existing metrics. While Euclidean Distance may not be perfect for all scenarios, it is a fast and simple method that works well if all the data is on a similar scale and has an overlapping range. With DTW, which warps time axes to align the trends, users can query based on the overall shape of the trend, largely ignoring

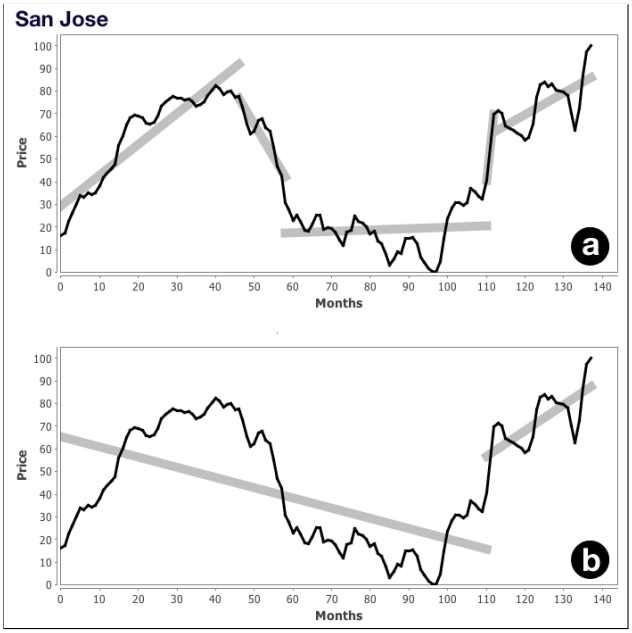


Figure 3.4: An example of the segmentation applied to the sales price of homes in San Jose with two different error thresholds. (a) has a smaller error threshold than (b), and thus represents the graph using five segments, compared to two of (b). This method helps look at overall shape of the trends, instead of being biased by small noise in the trends.

the problem the Euclidean Distance metric faces when the range of the data is not uniform. DTW is a popular method for identifying shapes, with many researchers having created variations of it based on their needs [53, 55, 56]. The segmentation method was developed in-house to address the performance problems of DTW, to capture broader trends in data than DTW can, and to reduce the effect of noise and increase speed while sacrificing accuracy.

3.2 THE BACK-END

All user inputs at the interface are internally translated and composed into one or more ZQL queries by the query builder module at the front-end before being sent to the back-end for processing. The front-end talks to the back-end through a REST interface, and all of the data transfers happen in a JSON format. The Zenvisage back-end is responsible for running all of the computations necessary for generating output visualizations that match user-specified insights. It is developed completely in Java and runs within an embedded Jetty web-server [57].

The back-end consists of a ZQL parser, an optimizer, and a query executor, and is capable of processing any ZQL query. At a high-level, the parser reads in the ZQL query is a textual format, parses the query and validates its structure, and checks the database catalog for the existence of the referenced columns and operators including the functional primitives. If everything succeeds,

the parser creates a graph of computation, consisting of SQL queries for retrieving data for each visualization from the underlying storage engine, following by processing (e.g., trend matching, comparing, filtering) to be applied on visualizations for selecting the ones to be output to the users. We dig deeper into the details of ZQL query processing, including optimizations that allow Zenvisage to scale over large collections of visualizations in the next chapter.

Storage. Since Zenvisage uses standard SQL queries for retrieving data for visualizations, it can work over any database. The current version, however, supports PostgreSQL as a row-store and Vertica as a column store options. We have also implemented a bitmap-based in-memory database for providing interactive response times, using a state-of-the-art implementation of bitmaps called Roaring Bitmaps [58]. Most of the queries in Zenvisage are ad hoc and unpredictable, making it infeasible to precompute or index the result visualizations in advance—rendering typical time series indexing and retrieval techniques such as those adopted in Google Correlate or TimeSearcher inapplicable. At the same time, traditional databases with conventional B-tree based indices result in high memory and computation overhead.

By making use of bitmaps, our in-memory database exploits bit-level parallelism that significantly accelerate queries involving arbitrary and complex selection predicates and aggregation, precisely the kind of queries that one would encounter in arbitrary visual data exploration. As compared to conventional bitmaps, Roaring Bitmap is up to $10\times$ faster and has higher compression rates. We follow a column oriented storage model, and as a default policy, we create indices for all categorical columns and leave measure columns un-indexed. The un-indexed columns are stored as an array, and indexed columns have a Roaring Bitmap for every distinct value of that column.

3.3 RECOMMENDATIONS VIA REPRESENTATIVE AND OUTLIER SEARCH.

In addition to returning the pattern or similarity search results, the Zenvisage back-end also recommends a set of interesting visualizations—the representative and outlier visualizations—to help users further understand the trends in the data.

Initially, our implementation for the representative trends was to simply apply the k-means algorithm on the set of candidate visualizations, and then display the cluster centers or centroids. However, we found that when we have many visualizations that are quite similar to each other—i.e., that they all have a similar "shape", and are therefore not very interesting. Instead, we develop a new algorithm based on a small modification to the k-means algorithm: if the desired number of visualizations is k, we invoke the k-means algorithm with a larger $k' > k$ first. Then, after we identify the k' cluster centroids, we merge these cluster centroids greedily based on which pairs

have the smallest distance between each other, until we get k visualizations. These visualizations are then the representative trends that are output to the user. We have found that this approach, while a small modification to k-means, yields much more visually appealing and "diverse" results.

As described previously, our outlier visualizations are found by summing up the distance for each candidate visualization to the k visualizations that were found to be most representative; then the k visualization candidates that are the most distant from these k representative visualizations are deemed to be the outliers.

3.4 OVERALL TAKEAWAYS

In this chapter, we introduced Zenvisage, giving an overview of the key front-end and back-end components of the system. Zenvisage is a general-purpose visual exploration system, that supports multiple modes of specification, including a sketching canvas where users can draw a pattern of interest, or drag and drop an existing visualization, with the system finding visualizations that are similar to the queried pattern. In addition, Zenvisage also provides high-level recommendations, in the form of representative and outlier visualizations, to help users get started. We worked with potential users across multiple domains on a regular basis to gather feedback on our system and understand how they envision themselves using it. Overall, Zenvisage appears to be an ideal fit for the challenge of visualization exploration described in Chapter 1— in that it automates the painful manual exploration of visualizations to find desired patterns.

In the next chapter, we dig deeper into the Zenvisage Query Language (ZQL) and corresponding algebra, that together form an important component of Zenvisage.

CHAPTER 4: ZQL: AN EXPRESSIVE VISUALIZATION QUERY LANGUAGE

In this chapter, we describe the syntax and semantics of Zenvisage query language (ZQL), the forms the core of the Zenvisage system. ZQL provides a set of operations on collections of visualizations that help automate the search for desired patterns. We also formalize the notion of visual exploration algebra, analogous to relational algebra, describing a core set of capabilities for any language that supports visual data exploration, and demonstrate that ZQL is complete in that it subsumes these capabilities.

4.1 OVERVIEW

Zenvisage query language or ZQL, provides users with a powerful mechanism to operate on collections of visualizations. In fact, ZQL treats visualizations as a *first-class citizen*, enabling users to operate at a high level on collections of visualizations much like one would operate on relational data with SQL. For example, a user may want to filter out all visualizations where the visualization shows a roughly decreasing trend from a collection, or a user may want to create a collection of visualizations which are most similar to a visualization of interest. Regardless of the query, ZQL provides an intuitive, yet flexible specification mechanism for users to express the desired patterns of interest (in other words, their *exploration needs*) using a small number of ZQL lines. Overall, ZQL provides users the ability to compose collections of visualizations, filter them, and sort and compare them in various ways.

ZQL draws heavy inspiration from the Query by Example (QBE) language [59] and uses a similar table-based specification interface. Although ZQL components are not fundamentally tied to the tabular interface, we found that our end-users felt more at home with it; many of them are non-programmers who are used to spreadsheet tools like Microsoft Excel. Users may either directly write ZQL, or they may use the Zenvisage front-end, which supports interactions that are transformed internally into ZQL.

We now provide a formal introduction to ZQL in the rest of this chapter. We introduce many sample queries to make it easy to follow along, and we use a relatable fictitious product sales-based dataset throughout this chapter in our query examples—we will reveal attributes of this dataset as we go along.

4.2 FORMALIZATION

For describing ZQL, we assume that we are operating on a single relation or a star schema where the attributes are unique (barring key-foreign key joins), allowing ZQL to seamlessly support

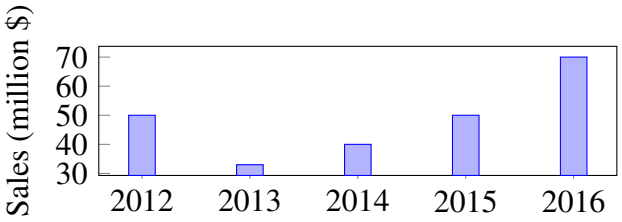


Figure 4.1: Sales over year visualization for the product chair.

Name	X	Y	Z	Viz
*f1	'year'	'sales'	'product'.'chair'	bar.(y=agg('sum'))

Table 4.1: Query for the bar chart of sales over year for the product chair.

Name	X	Y	Z	Viz
*f1	'year'	'sales'	'product'.*	bar.(y=agg('sum'))

Table 4.2: Query for the bar chart of sales over year for each product.

natural joins. In general, ZQL could be applied to arbitrary collections of relations by letting the user precede an attribute A with the relation name R, e.g., $R.A$. For ease of exposition, we focus on the single relation case.

The concept of visualizations. We start by defining the notion of a visualization. We use a sample visualization in Figure 4.1 to guide our discussion. Of course, different visual analysis tasks may require different types of visualizations (instead of bar charts, we may want scatter plots or trend lines), but across all types, a visualization is defined by the following five main components: (i) the x-axis attribute, (ii) the y-axis attribute, (iii) the subset of data used, (iv) the type of visualization (e.g., bar chart, scatter plot), and (v) the binning and aggregation functions for the x- and y- axes.

Visualization collections in ZQL: ZQL has four columns to support the specification of visualizations that the five aforementioned components map into: (i) ***X***, (ii) ***Y***, (iii) ***Z***, and (iv) ***Viz***.

Table 4.1 gives an example of a valid ZQL query that uses these columns to specify a bar chart visualization of overall sales over the years for the product chair (i.e., the visualization in Figure 4.1)—ignore the Name column for now. The details for each of these columns are presented subsequently. In short, the x-axis (X) is the attribute year, the y-axis (Y) is the attribute sales, and the subset of data (Z) is the product chair, while the type of visualization is a bar chart (bar), and the binning and aggregation functions indicate that the y axis is an aggregate (agg)—the sum of

Name	X	Y	Z	Viz	Process
Identifier	Visualization Collection				Operation

Table 4.3: ZQL *query structure.*

Name	X	Y	Z	Viz
...	...	{'sales', 'profit'}	...	...

Table 4.4: Query for the sales and profit bar charts for the product chair (missing values are the same as that in Table 4.1)

sales.

In addition to specifying a single visualization, users may often want to retrieve multiple visualizations. ZQL supports this in two ways. Users may use multiple rows, and specify one visualization per row. The user may also specify a *collection* of visualizations in a single row by iterating over a collection of values for one of the X, Y, Z, and Viz columns. Table 4.2 gives an example of how one may iterate over all products (using the notation * to indicate that the attribute product can take on all possible values), returning a separate sales bar chart for each product.

High-level structure of ZQL**.** Starting from these two examples, we can now move onto the general structure of ZQL queries. Overall, each ZQL query consists of multiple rows, where each row operates on collections of visualizations. Each row contains three sets of columns, as depicted in Table 4.3: (i) the first column corresponds to an identifier for the visualization collection, (ii) the second set of columns defines the visualization collection, while (iii) the last column corresponds to some operation on the visualization collection. All columns can be left empty if needed (in such cases, to save space, for convenience, we do not display these columns). For example, the last column may be empty if no operation is to be performed, like it was in Table 4.1 and 4.2. We have already discussed (ii); now we will briefly discuss (i) and (iii), corresponding to ***Name*** and ***Process*** respectively.

Identifiers and operations in ZQL**.** The ***Process*** column allows the user to operate on the defined collections of visualizations, applying high-level filtering, sorting, and comparison. The ***Name*** column provides a way to label and combine specified collections of visualizations, so users may refer to them in the Process column. Thus, by repeatedly using the X, Y, Z, and Viz columns to compose visualizations and the Process column to process those visualizations, the user is able derive the exact set of visualizations she is looking for. Note that the result of a ZQL query is the *data* used to generate visualizations. The Zenvisage front-end then uses this data to render the visualizations for the user to peruse.

Name	X	Y	Z	Viz
...	{'year', 'month'}	{'sales', 'profit'}	...	...

Table 4.5: Query for the sales and profit bar charts over years and months for chairs (missing values are the same as in Table 4.1).

Name	X	Y	Z	Z2	Viz
...	...	...	...	'location'.'US'	...

Table 4.6: Query which returns the overall sales bar chart for the chairs in US (all missing values are the same as that in Table 4.1).

4.2.1 X, Y, and Z

The X and Y columns specify the attributes used for the x- and y- axes. For example, Table 4.1 dictates that the returned visualization should have 'year' for its x-axis and 'sales' for its y-axis. As mentioned, the user may also specify a collection of values for the X and Y columns if they wish to refer to a collection of visualizations in one ZQL row. Table 4.4 refers the collection of both sales-over-years and profit-over-years bar charts for the chair—the missing values in this query ("...") are the same as Table 4.1. As we can see, a collection is constructed using {}. If the user wishes to denote all possible values, the shorthand * symbol may be used, as is shown by Table 4.2. In the case that multiple columns contain collections, a Cartesian product is performed, and visualizations for every combination of values is returned. For example, Table 4.5 would return the collection of visualizations with specifications: {(X: 'year', Y: 'sales'), (X: 'year', Y: 'profit'), (X: 'month', Y: 'sales'), (X: 'month', Y: 'profit')}. Additionally, ZQL allows composing multiple attributes in the X and Y columns by supporting Polaris table algebra [4] over the operators: +, x, /

With the Z column, the user can select which subset of the data they wish to construct their visualizations from. ZQL uses the ⟨*attribute*⟩.⟨*attribute-value*⟩ notation to denote the selection of data. Consequently, the query in Table 4.1 declares that the user wishes to retrieve the sales bar chart only for the chair product. Note that unlike the X and Y columns, both the attribute and the attribute value must be specified for the Z column; otherwise, a proper subset of the data would not be identified.Collections are allowed for both the attribute and the attribute value in the Z column. Table 4.2 shows an example of using the * shorthand to specify a collection of bar charts, one for each product. A Z column which has a collection over attributes might look like: {'location', 'product'}.* (i.e., a visualization for every product and a visualization for every lo-

Name	X	Y	Viz
*f1	'weight'	'sales'	bin2d.(x=nbin(20), y=nbin(20))

Table 4.7: Query which returns the heat map of sales vs. weights across all transactions.

cation). In addition, the Z column allows users to specify predicate constraints using syntax like 'weight'.[? < 10]; this specifies all items whose weight is less than 10 lbs. To evaluate, the ? is replaced with the attribute and the resulting expression is passed to SQL's WHERE clause. The predicate constraint syntax has an analogous predicate collection syntax, which creates a collection of the values which satisfy the condition. 'weight'.[? < 10] specifies that the resulting visualizations must only contains items with less than 10 lbs, 'weight'.{? < 10} creates a collection of values, one for each item which is less than 10 lbs.

ZQL supports multiple constraints on different attributes through the use of multiple Z columns. In addition to the basic Z column, the user may choose to add Z2, Z3, ... columns depending on how many constraints she requires. Table 4.6 gives an example of a query which looks at sales plots for chairs only in the US. Note that Z columns are combined using conjunctive semantics.

4.2.2 Viz

The Viz column decides the visualization type, binning, and aggregation functions for the row. Elements in this column have the format: ⟨*type*⟩.⟨*bin+aggr*⟩. All examples so far have been bar charts with no binning and SUM aggregation for the y-axis, but other variants are supported. The visualization types are derived from the Grammar of Graphics [60] specification language, so all plots from the geometric transformation layer of ggplot [61] (the tool that implements Grammar of Graphics) are supported. For instance, scatter plots are requested with point and heat maps with bin2d. As for binning, binning based on bin width (bin) and number of bins (nbin) are supported for numerical attributes—we may want to use binning, for example, when we are plotting the total number of products whose prices lie within 0-10, 10-20, and so on.

Finally, ZQL supports all the basic SQL aggregation functions such as AVG, COUNT, and MAX. Table 4.7 is an example of a query which uses a different visualization type, heat map, and creates 20 bins for both x- and y- axes.

Like the earlier columns, the Viz column also allows collections of values. Similar to the Z column, collections may be specified for both the visualization type or the binning and aggregation. If the user wants to view the same data binned at different granularities, she might specify a bar chart with several different bin widths: bar.(x={bin(1), bin(5), bin(10)}, y=agg('sum')). On the other hand, if the user wishes to view the same data in different visualizations, she might write: {bar.(y=agg('sum')), point.()}.

Name	X	Y	Z
f1	'year'	'sales'	'product'.'chair'
f2	'year'	'profit'	'location'.'US'
*f3 <− f1 + f2			'weight'.[? < 10]

Table 4.8: Query which returns the sales for chairs or profits for US visualizations for all items less than 10 lbs.

The Viz column allows users powerful control over the structure of the rendered visualization. However, there has been work from the visualization community which automatically tries to determine the most appropriate visualization type, binning, and aggregation for a dataset based on the x- and y- axis attributes [52, 62]. Thus, we can frequently leave the Viz column blank and Zenvisage will use these rules of thumb to automatically decide the appropriate setting for us. With this in mind, we omit the Viz column from the remaining examples with the assumption that Zenvisage will determine the "best" visualization structure for us.

4.2.3 Name

Together, the values in the X, Y, Z, and Viz columns of each row specify a collection of visualizations. The Name column allows us to label these collections so that they can be referred to be in the Process column. For example, f1 is the label or identifier given to the collection of sales bar charts in Table 4.2. The * in front of f1 signifies that the collection is an output collection; that is, ZQL should return this collection of visualizations to the user.

However, not all rows need to have a * associated with their Name identifier. A user may define intermediate collections of visualizations if she wishes to further process them in the Process column before returning the final results. In the case of Table 4.8, f1 and f2 are examples of intermediate collections.

Also in Table 4.8, we have an example of how the Name column allows us to perform high-level set-like operations to combine visualization collections directly. For example, f3 <− f1 + f2 assigns f3 to the collection which includes all visualizations in f1 and f2 (similar to set union). This can be useful if the user wishes to combine variations of values without considering the full Cartesian product. In our example in Table 4.8, the user is able to combine the sales for chairs plots with the profits for the US plots without also having to consider the sales for the US plots or the profits for chairs plots; she would have to do so if she had used the specification: (Y: {'sales', 'profit'}, Z: {'product'.'chair', 'location'.'US'}).

An interesting aspect of Table 4.8 is that the X and Y columns of the third row are devoid of values, and the Z column refers to the seemingly unrelated weight attribute. The values in the X,

Name	X	Y	Z	Process
f1	'year'	'profit'	v1 <− 'product'.*	
f2	'year'	'sales'	v1	v2 <− argmax$_{v1}[k=10]D(f1,f2)$
*f3	'year'	'profit'	v2	

Table 4.9: Query which returns the top 10 profit visualizations for products which are most different from their sales visualizations.

Y, Z, and Viz columns all help to specify a particular collection of visualizations from a larger collection. When this collection is defined via the Name column, we no longer need to fill in the values for X, Y, Z, or Viz, except to select from the collection—here, ZQL only selects the items which satisfy the constraint, weight $<$ 10.

Other set-like operators include f1 - f2 for set minus and f1 ^ f2 for intersection.

4.2.4 Process

The real power of ZQL as a query language comes not from its ability to effortlessly specify collections of visualizations, but rather from its ability to operate on these collections somewhat declaratively. With ZQL's processing capabilities, users can filter visualizations based on trend, search for similar-looking visualizations, identify representative visualizations, and determine outlier visualizations. Naturally, to operate on collections, ZQL must have a way to iterate over them; however, since different visual analysis tasks might require different forms of traversals over the collections, we expose the iteration interface to the user.

Iterations over collections. Since collections may be composed of varying values from multiple columns, iterating over the collections is not straight-forward. Consider Table 4.9—the goal is to return profit by year visualizations for the top-10 products whose profit by year visualizations look the most different from the sales by year visualizations. This may indicate a product that deserves special attention. While we will describe this query in detail below, at a high level the first row assembles the visualizations for profit over year for all products (f1), the second row assembles the visualizations for sales over year for all products (f2), followed by operating (via the Process column) on these two collections by finding the top-10 products who sales over year is most different from profit over year, while the third row displays the profit over year for those top-10 products. A array-based representation of the visualization collections f1 and f2, would look like the following:

$$\texttt{f1} = \left\{ \begin{array}{c} \texttt{X: 'year', Y: 'profit'} \\ \hline \texttt{Z: 'product.chair'} \\ \texttt{Z: 'product.table'} \\ \texttt{Z: 'product.stapler'} \\ \vdots \end{array} \right\} \texttt{f2} = \left\{ \begin{array}{c} \texttt{X: 'year', Y: 'sales'} \\ \hline \texttt{Z: 'product.chair'} \\ \texttt{Z: 'product.table'} \\ \texttt{Z: 'product.stapler'} \\ \vdots \end{array} \right\} \tag{4.1}$$

We would like to iterate over the products, the Z dimension values, of both f1 and f2 to make our comparisons. Furthermore, we must iterate over the products in the *same order* for both f1 and f2 to ensure that a product's profit visualization correctly matches with its sales visualization. Using a single index for this would be complicated and need to take into account the sizes of each of the columns. While there may be other ways to architect this iteration for a single attribute, it is virtually impossible to do when there are multiple attributes that are varying. Instead, ZQL opts for a more powerful *dimension-based* iteration, which assigns each column (or dimension) a separate iterator called an *axis variable*. This dimension-based iteration is a powerful idea that extends to any number of dimensions. As shown in Table 4.9, axis variables are defined and assigned using the syntax: $\langle variable \rangle$<– $\langle collection \rangle$; axis variable v1 is assigned to the Z dimension of f1 and iterates over all product values. For cases in which multiple collections must traverse over a dimension in the same order, an axis variable must be shared across those collections for that dimension; in Table 4.9, f1 and f2 share v1 for their Z dimension, since we want to iterate over the products in lockstep.

Operations on collections. With the axis variables defined, the user can then formulate the high-level operations on collections of visualizations as an optimization function which maximizes/minimizes for their desired pattern. Given that $\texttt{argmax}_x[k = 10]\ g(x)$ returns the top-10 x values which maximizes the function $g(x)$, and $D(x,y)$ returns the "distance" between x and y, now consider the expression in the Process column for Table 4.9. Colloquially, the expression says to find the top-10 v1 values whose $D(f1, f2)$ values are the largest. The $f1$ and $f2$ in $D(f1, f2)$ refer to the collections of visualizations in the first and second row and are bound to the current value of the iteration for v1. In other words, for each product v1' in v1, retrieve the visualizations f1[z: v1'] from collection f1 and f2[z: v1'] from collection f2 and calculate the "distance" between these visualizations; then, retrieve the 10 v1' values for which this distance is the largest—these are the products, and assign v2 to this collection. Subsequently, we can access this set of products, as we do in the Z column of the third line of Table 4.9.

Name	X	Y	Z	Process
f1	'year'	'sales'	v1 <− 'product'.*	v2 <− $\text{argmax}_{v1}[t < 0]T(f1)$
*f2	'year'	'sales'	v2	

Table 4.10: Query which returns the sales visualizations for all products which have a negative trend.

Name	X	Y	Z	Process
f1	'year'	'sales'	'product'.'chair'	
f2	'year'	'sales'	v1 <− 'product'.(* - 'chair')	v2 <− $\text{argmin}_{v1}[k = 10]D(f1, f2)$
*f3	'year'	'sales'	v2	

Table 4.11: Query which returns the sales visualizations for the 10 products whose sales visualizations are the most similar to the sales visualization for the chair.

Formal structure. More generally, the basic structure of the Process column is:

$$\langle argopt \rangle_{\langle axvar \rangle}[\langle limiter \rangle]\langle expr \rangle \quad \text{where}$$

$$\begin{aligned}
\langle expr \rangle &\rightarrow \left(\max \mid \min \mid \sum \mid \prod\right)_{\langle axvar \rangle} \langle expr \rangle \\
&\rightarrow \langle expr \rangle \ (+ \mid - \mid \times \mid \div) \ \langle expr \rangle \\
&\rightarrow T(\langle nmvar \rangle) \\
&\rightarrow D(\langle nmvar \rangle, \langle nmvar \rangle) \\
\langle argopt \rangle &\rightarrow (\text{argmax} \mid \text{argmin} \mid \text{argany}) \\
\langle limiter \rangle &\rightarrow (k = \mathbb{N} \mid t > \mathbb{R} \mid p = \mathbb{R}) \qquad (4.2)
\end{aligned}$$

where $\langle axvar \rangle$ refers to the axis variables, and $\langle nmvar \rangle$ refers to collections of visualizations. $\langle argopt \rangle$ may be one of argmax, argmin, or argany, which returns the values which have the largest, smallest, and any expressions respectively. The $\langle limiter \rangle$ limits the number of results: $k = \mathbb{N}$ returns only the top-k values; $t > \mathbb{R}$ returns only values who are larger than a threshold

Name	X	Y	Z	Process
f1	'year'	'sales'	v1 <− 'product'.*	
f2	'year'	'sales'	v2 <− 'product'.*	v3 <− $\text{argmax}_{v1}[k = 10]\sum_{v2} D(f1, f2)$
*f3	'year'	'sales'	v3	

Table 4.12: Query which returns the sales visualizations for the 10 products whose sales visualizations are the most different from the others.

value t (may also be smaller, greater than equal, etc.); $p = \mathbb{R}$ returns the top p-percentile values. T and D are two simple *functional primitives* supported by ZQL that can be applied to visualizations to find desired patterns:

- $[T(f) \rightarrow \mathbb{R}]$: T is a function which takes a visualization f and returns a real number measuring some visual property of the trend of f. One such property is "growth", which returns a positive number if the overall trend is "upwards" and a negative number otherwise; an example implementation might be to measure the slope of a linear fit to the given input visualization f. Other properties may measure the skewness, or the number of peaks, or noisiness of a visualization.

- $[D(f, f') \rightarrow \mathbb{R}]$: D is a function which takes two visualizations f and f' and measures the distance (or dissimilarity) between these visualizations. Examples of distance functions may include a pointwise distance function like Euclidean distance, Earth Mover's Distance, or the Kullback-Leibler Divergence. The distance D could also be measured using the difference in the number of peaks, or slopes, or some other property.

ZQL supports many different implementations for these two functional primitives, and the user is free to choose any one. If the user does not select one, Zenvisage will automatically detect the "best" primitive based on the data characteristics. Furthermore, if ZQL does not have an implementation of the T or D function that the user is looking for, the user may write and use their own function.

Concrete examples. With just dimension-based iteration, the optimization structure of the Process column, and the functional primitives T and D, we found that we were able to support the majority of the visual analysis tasks required by our users. Common patterns include filtering based on overall trend (Table 4.10), searching for the most similar visualization (Table 4.11), and determining outlier visualizations (Table 4.12). Table 4.10 describes a query where in the first row, the variable `v2` selects all products whose trend is decreasing, and the second row visualizes these product's sales over year. Table 4.11 starts with the visualization sales over year for chair in the first row, then in the second row computes the visualizations of sales over year for all products, and in the process column computes the similarity with chair, assigning the top 10 to `v2`, and the third row visualizes the sales over year for these products. Table 4.12 starts with the visualization collection of sales over year for all products in the first row, followed by another collection of the same in the second row, and in the process column computes the sum of pairwise distances, assigning the 10 products whose visualizations are most distant to others to `v3`, after which they are visualized. Table 4.13 features a realistic query inspired by one of our case studies. The overall goal of the query is to find the products which have positive sales and profits trends in locations

Name	X	Y	Z	Z2	Z3	Process
f1	'year'	'sales'	v1 <– 'location'.*			v2 <– argany$_{v1}[t<0]T(f1)$
f2	'year'	'profit'	v3 <– 'category'.*			v4 <– argany$_{v3}[t<0]T(f2)$
f3	'year'	'profit'	v5 <– 'product'.*	'location'.[? IN v2]	'category'.[? IN v4]	v6 <– argany$_{v5}[t>0]T(f3)$
f4	'year'	'sales'	v5	'location'.[? IN v2]	'category'.[? IN v4]	v7 <– argany$_{v5}[t>0]T(f4)$
*f5	'year'	{'profit', 'sales'}	v6 ^ v7			

Table 4.13: Query which returns the profit and sales visualizations for products which have positive trends in profit and sales in locations and categories which have overall negative trends.

and categories which have overall negative trends; the user may want to look at this set of products to see what makes them so special. Rows 1 and 2 specify the sales and profit visualizations for all locations and categories respectively, and the processes for these rows filter down to the locations and categories which have negative trends. Then rows 3 and 4 specify the sales and profit visualizations for products in these locations and categories, and the processes filter the visualizations down to the ones that have positive trends. Finally, row 5 takes the list of output products from the processes in rows 3 and 4 and takes the intersection of the two returning the sales and profits visualizations for these products.

Pluggable functions. While the general structure of the Process column does cover the majority of the use cases requested by our users, users may want to write their own functions to run in a ZQL query. To support this, ZQL exposes a Java-based API for users to write their own functions. In fact, we use this interface to implement the *k*-means algorithm for ZQL. While the pluggable functions do allow virtually any capabilities to be implemented, it is preferred that users write their queries using the syntax of the Process column; pluggable functions are considered black-boxes and cannot be automatically optimized by the ZQL compiler.

4.3 EXPRESSIVENESS

We now formally quantify the expressive power of ZQL. To this end, we formulate an algebra, called the *visual exploration algebra*, like relational algebra, with a basic set of operators that we believe all visual exploration languages should be able to express. At a high level, the operators of our visual exploration algebra operate on sets of visualizations and are not mired by the data representations of those visualizations, nor the details of how the visualizations are rendered. Instead, the visual exploration algebra is primarily concerned with the different ways in which visualizations can be selected, refined, and compared with each other.

Given a defined measure T for calculating the overall trend of a visualization, a distance metric D for a pair of visualizations, and an algorithm R to identify the most representative visualizations from a set of visualizations, a visual exploration language L is defined to be *visual exploration*

complete $VEC_{T,D,R}(L)$ with respect to T, D, and R if it supports all the operators of the visual exploration algebra. These operators may optionally take these functions T, D, and D, as input parameters. These functions T, D, and R (also defined previously) are "exploration functions" without which the resulting algebra would have been unable to manipulate visualizations in the way we need for data exploration. Unlike relational algebra, which does not have any "black box" functions, visual exploration algebra requires these functions for operating on visualizations effectively. That said, these three functions are flexible and configurable and up to the user to define (or left as system defaults). Next, we formally define the visual exploration algebra operators and prove that ZQL is visual exploration complete.

4.3.1 Ordered Bag Semantics

In visual exploration algebra, relations have bag semantics. However, since users want to see the most relevant visualizations first, ordering is critical. So, we adapt the operators from relational algebra to preserve ordering information.

Thus, we operate on *ordered bags* (i.e., a bag that has an inherent order). We describe the details of how to operate on ordered bags below. We use R, S to denote the ordered bags. We also use the notation $R = [t_1, \ldots, t_n]$ to refer to an ordered bag, where t_i are the tuples.

The first operator that we define is an indexing operator, much like indexing in arrays. The notation $R[i]$ refers to the ith tuple within R, and $R[i:j]$ refers to the ordered bag corresponding to the list of tuples from the ith to the jth tuple, both inclusive. In the notation $[i:j]$ if either one of i or j is omitted, then it is assumed to be 1 for i, and n for j, where n is the total number of tuples.

Next, we define a union operator $\cup$: $R \cup S$ refers to the concatenation of the two ordered bags R and S. If one of R or S is empty, then the result of the union is simply the other relation. We define the union operation first because it will come in handy for subsequent operations.

We define the σ operator like in relational algebra, via a recursive definition:

$$\sigma_\theta(R) = \sigma_\theta([R[1]]) \cup \sigma_\theta(R[2:]) \tag{4.3}$$

where σ_θ when applied to an ordered bag with a single tuple ([t]) behaves exactly like in the relational algebra case, returning the same ordered bag ([t]) if the condition is satisfied, and the empty ordered bag ([]) if the condition is not satisfied. The π operator for projection is defined similarly to σ in the equation above, with the π operator on an ordered bag with a single tuple simply removing the irrelevant attributes from that tuple, like in the relational algebra setting.

Then, we define the $\setminus$ operator, for ordered bag difference. Here, the set difference operator operates on every tuple in the first ordered bag and removes it if it finds it in the second ordered

bag. Thus:

$$R \setminus S = ([R[1]] \setminus S) \cup (R[2:] \setminus S) \tag{4.4}$$

where $[t] \setminus S$ is defined like in relational algebra, returning $[t]$ if $[t]$ is not in S, and [] otherwise. The intersection operator $\cap$ is defined similarly to $\cup$ and $\setminus$.

Now, we can define the duplicate elimination operator as follows:

$$\delta(R) = [R[1]] \cup (R[2:] \setminus [R[1]]) \tag{4.5}$$

Thus, the duplication elimination operator preserves ordering, while maintaining the first copy of each tuple at the first position that it was found in the ordered bag.

Lastly, we have the cross product operator, as follows:

$$R \times S = ([R[1]] \times S) \cup (R[2:] \times S) \tag{4.6}$$

where further we have

$$[t] \times S = ([t] \times [S[1]]) \cup ([t] \times S[2:]) \tag{4.7}$$

where $[t] \times [u]$ creates an ordered bag with the result of the cross product as defined in relational algebra.

Given these semantics for ordered bags, we can develop the visual exploration algebra.

4.3.2 Basic Notation

Assume we are given a k-ary relation $\mathcal{R}$ with attributes $(A_1, A_2, \ldots, A_k)$. Let $\mathcal{X}$ be the unary relation with attribute X whose values are the names of the attributes in $\mathcal{R}$ that can appear on the x-axis. If the x-axis attributes are not specified by the user for relation $\mathcal{R}$, the default behavior is to include all attributes in $\mathcal{R}$: $\{A_1, \ldots, A_k\}$. Let $\mathcal{Y}$ be defined similarly with Y for attributes that can appear on the y-axis. Given $\mathcal{R}$, $\mathcal{X}$, and $\mathcal{Y}$, we define $\mathcal{V}$, the *visual universe*, as follows: $\mathcal{V} = \nu(\mathcal{R}) = \mathcal{X} \times \mathcal{Y} \times \left(\times_{i=1}^{k} \pi_{A_i}(\mathcal{R}) \cup \{*\} \right)$ where π is the projection operator from relational algebra and $*$ is a special wildcard symbol, used to denote all values of an attribute. At a high level, the visual universe specifies all subsets of data that may be of interest, along with the intended attributes to be visualized. Unlike relational algebra, visual exploration algebra mixes schema and data elements, but in a special way in order to operate on a collection of visualizations.

Any subset relation $V \subseteq \mathcal{V}$ is called a *visual group*, and any $k+2$-tuple from $\mathcal{V}$ is called a *visual source*. The last k portions (or attributes) of a tuple from $\mathcal{V}$ comprise the *data source* of the visual source. Overall, a visual source represents a visualization that can be rendered from

a selected data source, and a set of visual sources is a visual group. The X and Y attributes of the visual source determine the x- and y- axes, and the selection on the data source is determined by attributes $A_1, \ldots, A_k$. If an attribute has the wildcard symbol $*$ as its value, no subselection is performed on that attribute for the data source. For example, the third row is a visual source that represents the visualization with year as the x-axis and sales as the y-axis for chair products. Since the value of location is $*$, all locations are considered valid or pertinent for the data source. In relational algebra, the data source for the third row can be written as $\sigma_{\text{product=chair}}(\mathcal{R})$. The $*$ symbol therefore attempts to emulate the lack of presence of a selection condition on that attribute in the σ operator of the relational algebra. Readers familiar with OLAP will notice the similarity between the use of the symbol $*$ here and the GROUPING SETS functionality in SQL.

Note that infinitely many visualizations can be produced from a single visual source, due to different granularities of binning, aggregation functions, and types of visualizations that can be constructed, since a visualization generation engine can use a visualization rendering grammar like ggplot [61] that provides that functionality. Our focus in defining the visual exploration algebra is to specify the inputs to a visualization and attributes of interest as opposed to the aesthetic aspects and encodings. Thus, for our discussion, we assume that each visual source maps to a singular visualization. Even if the details of the encoding and aesthetics are not provided, standard rules may be applied for this mapping. Furthermore, a visual source does not specify the data representation of the underlying data source; therefore the expressive power of visual exploration algebra is not tied to any specific backend data storage model. The astute reader will have noticed that the format for a visual source looks fairly similar to the visual components of ZQL; this is no accident. In fact, we will use the visual components of ZQL as a proxy to visual sources when proving that ZQL is visual exploration complete.

4.3.3 Exploration Functions

Earlier, we mentioned that a visual exploration algebra is visual exploration complete with respect to three exploration functions T, R, and D. Here we define the types of these exploration functions and describe them in more detail.

The function $T : \mathcal{V} \to \mathcal{R}$ returns a real number given a visual source. This function can be used to assess whether a trend: defined by the visualization corresponding to a specific visual source, is "increasing", or "decreasing", or satisfies some other fixed property. Many such T can be defined and used within the visual exploration algebra.

The function $D : \mathcal{V} \times \mathcal{V} \to \mathcal{R}$ returns a real number given a pair of visual sources. This function can be used to compare pairs of visualizations (corresponding to the visual sources) with respect to each other. The most natural way to define D is via some notion of distance, e.g., Earth Mover's

or Euclidian distance, but once again, the definition can be provided by the user or assumed as a fixed black box.

The function $R : \mathcal{V}^n \rightarrow \mathcal{R}^n$ given a list of visual sources returns a list of values, one corresponding to each of the visual sources. This function can take in an arbitrarily long list of visual sources, and returns a score for each one. This function's intended use is for ascertaining representativeness, i.e., how representative do we believe the visualization corresponding to a specific visual source to be. The function could equally well be used for ascertaining outlierness, or some other global property that requires consideration of the entire set of visualizations with respect to each other.

4.4 VISUAL EXPLORATION ALGEBRA OPERATORS

Similar to how operators in ordered bag algebra operate on and result in ordered bags, operators in visual exploration algebra operate on and result in visual groups. Many of the symbols for operators in visual exploration algebra are also derived from relational algebra, with some differences. To differentiate, operators in visual exploration algebra are superscripted with a v (e.g., σ^v, τ^v). The unary operators for visual exploration algebra include *(i)* σ^v for selection, *(ii)* τ^v for sorting a visual group based on the trend-estimating function T, *(iii)* μ^v for limiting the number of visual sources in a visual group, *(iv)* δ^v for duplicate visual source removal, and *(v)* ζ^v for finding the most representative visual sources of a visual group based on algorithm R . The binary operators include *(i)* $\cup^v$ for union, *(ii)* $\setminus^v$ for difference, *(iii)* β^v for replacing the attribute values of the visual sources in one visual group's with another's, *(iv)* ϕ^v to reorder the first visual group based on the visual sources' distances to the visual sources of another visual group based on metric D, and *(v)* η^v to reorder the visual sources in a visual group based on their distance to a reference visual source from a singleton visual group based on D. These operators are described below, and listed in Table 4.14:

Unary Operators. $\sigma^v_\theta(V)$**:** σ^v selects a visual group from V based on selection criteria θ, like ordered bag algebra. However, σ^v has a more restricted θ; while $\vee$ and $\wedge$ may still be used, only the binary comparison operators $=$ and $\neq$ are allowed. As an example,
$\sigma^v_\theta(\mathcal{V})$ where $\theta =$ (X='year' $\wedge$ Y='sales' $\wedge$ year=$*$ $\wedge$ month=$*$ $\wedge$ product $\neq *$ $\wedge$ location='US' $\wedge$ sales=$*$ $\wedge$ profit=$*$) from Table 4.15 on $\mathcal{V}$ would result in the visual group of time vs. sales visualizations for different products in the US.

In this example, note that the product is specifically set to not equal $*$ so that the resulting visual group will include all products. On the other hand, the location is explicitly set to be equal to US. The other attributes, e.g., sales, profit, year, month are set to equal $*$: this implies that the visual groups are not employing any additional constraints on those attributes. (This may be useful, for

Operator	Name	Derived from Bag Algebra	Meaning	Unary/Binary
σ^v	Selection	Yes	Subselects visual sources	Unary
τ^v	Sort	No	Sorts visual sources in increasing order	Unary
μ^v	Limit	Yes	Returns first k visual sources	Unary
δ^v	Dedup	Yes	Removes duplicate visual sources	Unary
ζ^v	Representative	No	Selects k representative visual sources	Unary
$\cup^v/\setminus^v/\cap^v$	Union/Diff/Int	Yes	Returns the union of/differences between/intersection of two visual groups	Binary
β^v	Swap	No	Returns a visual group in which values of an attribute in one visual group is replaced with values of the same attribute in another visual group	Binary
ϕ^v	Dist	No	Sorts a visual group based on pairwise distance to another visual group	Binary
η^v	Find	No	Sorts a visual group in increasing order based on their distances to a single reference visual source	Binary

Table 4.14: Visual Exploration Algebra Operators

example when those attributes are not relevant for the current visualization or set of visualizations.) As mentioned before, visual groups have the semantics of ordered bags. Thus, σ^v operates on one tuple at a time in the order they appear in V, and the result is in the same order the tuples are operated on.

$\tau^v_{F(T)}(V)$**:** τ^v returns the visual group sorted in an increasing order based on applying $F(T)$ on each visual source in V, where $F(T)$ is a procedure that uses function T. For example, $\tau^v_{-T}(V)$ might return the visualizations in V sorted in decreasing order of estimated slope. This operator is not present in the ordered bag semantics, but may be relevant when we want to reorder the ordered bag using a different criterion. The function F may be any higher-order function with no side effects. For a language to visual exploration complete, the language must be able to support any arbitrary F.

$\mu^v_k(V)$**:** μ^v returns the first k visual sources of V ordered in the same way they were in V. μ^v is equivalent to the `LIMIT` statement in SQL. μ^v is often used in conjunction with τ^v to retrieve the top-k visualizations with greatest increasing trends (e.g. $\mu^v_k(\tau^v_{-T}(V))$). When instead of a number k, the subscript to μ^v is actually $[a:b]$, then the items of V that are between positions a and b in V are returned. Thus μ^v offers identical functionality to the $[a:b]$ in ordered bag algebra, with the convenient functionality of getting the top k results by just having one number as the subscript. Instead of using μ^v, visual exploration algebra also supports the use of the syntax $V[i]$ to refer to the ith visual source in V, and $V[a:b]$ to refer to the ordered bag of visual sources from positions a to b.

$\delta^v(V)$ **:** δ^v returns the visual sources in V with the duplicates removed, in the order of their first appearance. Thus, δ^v is defined identically to ordered bag algebra.

$\zeta^v_{R,k}(V)$**:** ζ^v returns the k-most representative visual sources from V based on representative-finding algorithm R. The returned results may be in any order. Unlike ordered bag algebra, which does not have this functionality, visual exploration algebra has a special purpose operator that uses the black box function R to return representative visual sources, from among all of V: implicitly, this black box function can compare all of V to itself.

Binary Operators. $V \cup^v U$ **|** $V \setminus^v U$ **|** $V \cap^v U$**:** Returns the union / difference / intersection of V and U. These operations are just like the corresponding operations in ordered bag algebra.

$\beta^v_A(V,U)$**:** β^v returns a visual group in which values of attribute A in V are replaced with the values of A in U. Formally, assuming A_i is the ith attribute of V and V has n total attributes: $\beta^v_{A_i}(V,U) = \pi_{A_1,...,A_{i-1},A_{i+1},...,A_n}(V) \times \pi_{A_i}(U)$. This can be useful for when the user would like to change an axis: $\beta_{\mathrm{X}}(V, \sigma^v_{\mathrm{X=year}}(\mathcal{V}))$ will change the visual sources in V to have year as their x-axis. β^v can also be used to combine multiple dimensions as well. If we assume that V has multiple Y values, we can do $\beta^v_{\mathrm{X}}(V, \sigma^v_{\mathrm{X}\neq *}(\mathcal{V}))$ to have the visual sources in V vary over both X and Y. This operator allows us to start with a set of visualizations and then "pivot" to focus on a different attribute, e.g., start with sales over time visualizations and pivot to look at profit. Thus, the operator allows us to transform the space of visual sources.

$\phi^v_{F(D),A_1,...,A_j}(V,U)$**:** ϕ^v sorts the visual sources in V in increasing order based on their distances to the corresponding visual sources in U. More specifically, ϕ^v computes $F(D)(\sigma^v_{A_1=a_1\wedge...\wedge A_j=a_j}(V), \sigma^v_{A_1=a_1\wedge...\wedge A_j=a_j}(U))\forall a_1,...,a_j \in \pi_{A_1,...,A_j}(V)$ and returns an increasingly sorted V based on the results. If $\sigma^v_{A_1=a_1\wedge...\wedge A_j=a_j}$ for either V or U ever returns a non-singleton visual group for any tuple $(a_1,...,a_j)$, the result of the operator is undefined. This operator supports comparative queries where we find the x- and y-axes for which the visualizations are most different: $\phi^v_{-D,X,Y}(V,U)$.

$\eta^v_{F(D)}(V,U)$**:** η^v sorts the visual sources in V in increasing order based on their distances to a single reference visual source in singleton visual group U. Thus, $U = [t]$. η^v computes $F(D)(V[i],U[1])$ $\forall i \in \{1,\ldots,|V|\}$, and returns a reordered V based on these values, where $F(D)$ is a procedure that uses D. If U has more than one visual source, the operation is undefined. η^v is useful for queries in which the user would like to find the top-k most similar visualizations to a reference: $\mu^v_k(\eta^v_D(V,U))$, where V is the set of candidates and U contains the reference. Once again, this operator is similar to τ^v, except that it operates on the results of the comparison of individual visual sources to a specific visual source.

4.5 PROOF OF VISUAL EXPLORATION COMPLETENESS

We now attempt to quantify the expressiveness of ZQL within the context of visual exploration algebra and the three exploratory functions T, D, and R. More formally, we prove the following

X	Y	year	month	product	location	sales	profit
year	sales	$*$	$*$	chair	US	$*$	$*$
year	sales	$*$	$*$	table	US	$*$	$*$
year	sales	$*$	$*$	stapler	US	$*$	$*$
year	sales	$*$	$*$	printer	US	$*$	$*$
$\vdots$							

Table 4.15: Results of performing unary operators on $\mathcal{V}$: $\sigma^{v}_{\theta}(\mathcal{V})$ where $\theta = (X=$'year' $\wedge$ $Y=$'sales' $\wedge$ year$=* \wedge$ month$=* \wedge$ product $\neq * \wedge$location='US' $\wedge$ sales$=* \wedge$ profit$=*$)

.

theorem:

Theorem 4.1. *Given well-defined exploratory functions T, D, and R,* ZQL *is visual exploration complete with respect to T, D, and R: $VEC_{T,D,R}$(ZQL) is true.*

Our proof for this theorem involves two major steps:

Step 1. We show that a visual component of ZQL has as much expressive power as a visual group of visual exploration algebra, and therefore a visual component in ZQL serves as an appropriate proxy of a visual group in visual exploration algebra.

Step 2. For each operator in visual exploration algebra, we show that there exists a ZQL query which takes in visual components semantically equivalent to the visual group operands and produces visual components semantically equivalent to the resultant visual group.

Lemma 4.1. *A visual component of* ZQL *has at least as much expressive power as a visual group in visual exploration algebra.*

Proof. A visual group V, with n visual sources, is a relation with $k+2$ columns and n rows, where k is the number of attributes in the original relation. We show that for any visual group V, we can come up with a ZQL query q which can produce a visual component that represents the same set of visualizations as V.

Name	X	Y	Z1	...	Zk
f1	$\pi_{\mathtt{X}}(V[1])$	$\pi_{\mathtt{Y}}(V[1])$	$E_{1,1}$	...	$E_{1,k}$
⋮	⋮	⋮	⋮	⋮	⋮
fn	$\pi_{\mathtt{X}}(V[n])$	$\pi_{\mathtt{Y}}(V[n])$	$E_{n,1}$	...	$E_{n,k}$
*fn+1=f1+...+fn					

Table 4.16: ZQL *query q which produces a visual component equal in expressivity to visual group V.*

Query q has the format given by Table 4.16, where $V[i]$ denotes the ith tuple of relation V and:

$$E_{i,j} = \begin{cases} \text{“ ”} & \text{if } \pi_{A_j}(V[i]) = * \\ A_j.\pi_{A_j}(V[i]) & \text{otherwise} \end{cases} \tag{4.8}$$

Here, A_j refers to the jth attribute of the original relation. The ith visual source of V is represented with the fi from q. The X and Y values come directly from the visual source using projection. For the Zj column, if the A_j attribute of visual source has any value than other than $*$, we must filter the data based on that value, so $E_{i,j} = A_j.\pi_{A_j}V[i]$. However, if the A_j attribute is equal to $*$, then the corresponding element in fi is left blank, signaling no filtering based on that attribute.

After, we have defined a visual component fi for each ith visual source in V, we take the sum (or concatenation) across all these visual components, and the resulting fn+1 becomes equal to the visual group V.

Lemma 4.2. *$\sigma^v_\theta(V)$ is expressible in* ZQL *for all valid constraints θ and visual groups V.*

Proof. We prove this by induction.

The full context-free grammar (CFG) for θ in σ^v_θ can be given by:

$$\theta \to E \mid E \wedge E \mid E \vee E \mid \varepsilon \tag{4.9}$$

$$E \to C \mid (E) \mid E \wedge C \mid E \vee C \tag{4.10}$$

$$C \to T_1 = B_1 \mid T_1 \neq B_1 \mid T_2 = B_2 \mid T_2 \neq B_2 \tag{4.11}$$

$$T_1 \to X \mid Y \tag{4.12}$$

$$B_1 \to A_1 \mid ... \mid A_k \tag{4.13}$$

$$T_2 \to A_1 \mid ... \mid A_k \tag{4.14}$$

$$B_2 \to \text{string} \mid \text{number} \mid * \tag{4.15}$$

where ε represents an empty string (no selection), and X, Y, and A_1, ..., A_k refer to the attributes of $\mathcal{V}$.

To begin the proof by induction, we first show that ZQL is capable of expressing the base expressions $\sigma^v_C(V)$: $\sigma^v_{T_1=B_1}(V)$, $\sigma^v_{T_1\neq B_1}(V)$, $\sigma^v_{T_2=B_2}(V)$, and $\sigma^v_{T_2\neq B_2}(V)$. The high level idea for each of these proofs is to be come up with a *filtering visual group* U which we take the intersection with to arrive at our desired result: $\exists U, \sigma^v_C(V) = V \cap^v U$.

In the first two expressions, T_1 and B_1 refer to filters on the X and Y attributes of V; we have the option of either selecting a specific attribute ($T_1 = B_1$) or excluding a specific attribute ($T_1 \neq B_1$). Tables 4.17 and 4.18 show ZQL queries which express $\sigma^v_{T_1=B_1}(V)$ for $T_1 \to X$ and $T_1 \to Y$ respectively. The ZQL queries do the approximate equivalent of $\sigma^v_{T_1=B_1}(V) = V \cap^v \sigma^v_{T_1=B_1}(\mathcal{V})$.

Name	X	Y	Z1	...	Zk
f1	-	-	-	...	-
f2=f1		y1 <− _	v1 <− A_1._	...	vk <− A_k._
f3	B_1	y1	v1	...	vk
*f4=f1^f3					

Table 4.17: ZQL *query which expresses* $\sigma^v_{X=B_1}(V)$.

Name	X	Y	Z1	...	Zk
f1	-	-	-	...	-
f2=f1	x1 <− _		v1 <− A_1._	...	vk <− A_k._
f3	x1	B_1	v1	...	vk
*f4=f1^f3					

Table 4.18: ZQL *query which expresses* $\sigma^v_{Y=B_1}(V)$.

We have shown with Lemma 4.1 that a visual component is capable of expressing a visual group, so we assume that f1, the visual component which represents the operand V, is given to us for both of these tables. Since we do not know how f1 was derived, we use - for its axis variable columns. The second rows of these tables derive f2 from f1 and bind axis variables to the values of the non-filtered attributes. Here, although the set of visualizations present in f2 is exactly the same as f1, we now have a convenient way to iterate over the non-filtered attributes of f1. The third row combines the specified attribute B_1 with the non-filtered attributes of f2 to form the *filtering visual component* f3, which expresses the filtering visual group U from above. We then take the intersection between f1 and the filtering visual component f3 to arrive at our desired visual component f4, which represents the resultant visual group $\sigma^v_{T_1=B_1}(V)$. Although, we earlier said

that we would come up with f3 $= \sigma^v_{T_1=B_1}(\mathcal{V})$, in truth, we come up with f3 $= B_1 \times \pi_{Y,A_1,...,A_k}(V)$ for $T_1 \to X$ and f3 $= \pi_{X,A_1,...,A_k}(V) \times B_1$ for $T_1 \to Y$ because they are easier to express in ZQL; regardless we still end up with the correct resulting set of visualizations.

Tables 4.19 and 4.20 show ZQL queries which express $\sigma^v_{T_1 \neq B_1}(V)$ for $T_1 \to X$ and $T_1 \to Y$ respectively. Similar to the queries above, these queries perform the approximate equivalent of $\sigma^v_{T_1 \neq B_1}(V) = V \cap^v \sigma^v_{T_1 \neq B_1}(\mathcal{V})$. We once again assume f1 is a given visual component which represents the operand V, and we come up with a filtering visual component f3 which mimics the effects of (though is not completely equivalent to) $\sigma^v_{T_1 \neq B_1}(\mathcal{V})$. We then take the intersection between f1 and f3 to arrive at f4 which represents the resulting $\sigma^v_{T_1 \neq B_1}(V)$.

Name	X	Y	Z1	...	Zk
f1	-	-	-	...	-
f2=f1	x1 <- _	y1 <- _	v1 <- A_1._	...	vk <- A_k._
f3	x2 <- x1.range - {B_1}	y1	v1	...	vk
*f4=f1^f3					

Table 4.19: ZQL *query which expresses* $\sigma^v_{X \neq B_1}(V)$.

Name	X	Y	Z1	...	Zk
f1	-	-	-	...	-
f2=f1	x1 <- _	y1 <- _	v1 <- A_1._	...	vk <- A_k._
f3	x1	y2 <- y1.range - {B_1}	v1	...	vk
*f4=f1^f3					

Table 4.20: ZQL *query which expresses* $\sigma^v_{Y \neq B_1}(V)$.

The expressions $\sigma^v_{T_2=B_2}$ and $\sigma^v_{T_2 \neq B_2}$ refer to filters on the $A_1,...,A_k$ attributes of V. Specifically, T_2 is some attribute $A_j \in \{A_1,...,A_k\}$ and B_2 is the attribute value which is selected or excluded. Here, we have an additional complication to the proof since any attribute A_j can also filter for or exclude $*$. First, we show ZQL is capable of expressing $\sigma^v_{T_2=B'_2}$ and $\sigma^v_{T_2 \neq B'_2}$ for which $B'_2 \neq *$; that is B'_2 is any attribute value which is not $*$. Tables 4.21 and 4.22 show the ZQL queries which express $\sigma^v_{T_2=B'_2}(V)$ and $\sigma^v_{T_2 \neq B'_2}(V)$ respectively. Note the similarity between these queries and the queries for $\sigma^v_{T_1=B_1}(V)$ and $\sigma^v_{T_1 \neq B_1}(V)$.

For $\sigma^v_{T_2=*}(V)$ and $\sigma^v_{T_2 \neq *}(V)$, Tables 4.23 and 4.24 show the corresponding queries. In Table 4.23, we explicitly avoid setting a value for Zj for f3 to emulate $A_j = *$ for the filtering visual component. In Table 4.24, f3's Zj takes on all possible values from $A_j.*$, but that means that a value is set for Zj (i.e., $T_2 \neq *$).

Now that we have shown how to express the base operations, we next assume ZQL is capable of expressing any arbitrary complex filtering operations $\sigma^v_{E'}$ where E' comes from Line 4.10 of the

Name	X	Y	Z1	...	Zj	...	Zk
f1	-	-	-	...	-	...	-
f2=f1	x1 <− _	y1 <− _	v1 <− A_1._	...		...	vk <− A_k._
f3	x1	y1	v1	...	B'_2	...	vk
*f4=f1^f3							

Table 4.21: ZQL *query which expresses* $\sigma^v_{A_j=B'_2}(V)$ *when* $B'_2 \neq *$.

Name	X	Y	Z1	...	Zj	...	Zk
f1	-	-	-	...	-	...	-
f2=f1	x1 <− _	y1 <− _	v1 <− A_1._	...	vj <− A_j._	...	vk <− A_k._
f3	x1	y1	v1	...	uj <− vj.range - {B'_2}	...	vk
*f4=f1^f3							

Table 4.22: ZQL *query which expresses* $\sigma^v_{A_j\neq B'_2}(V)$ *when* $B'_2 \neq *$.

CFG. Specifically, we assume that given a visual component f1 which expresses V, there exists a filtering visual component f2 for which $\sigma^v_{E'}(V) =$ f1^f2. Given this assumption, we now must take the inductive step, apply Line 4.10, and prove that $\sigma^v_{E\to(E')}(V)$, $\sigma^v_{E\to E'\wedge C}(V)$, and $\sigma^v_{E\to E'\vee C}(V)$ are all expressible in ZQL for any base constraint C.

Name	X	Y	Z1	...	Zj	...	Zk
f1	-	-	-	...	-	...	-
f2=f1	x1 <− _	y1 <− _	v1 <− A_1._	...		...	vk <− A_k._
f3	x1	y1	v1	...		...	vk
*f4=f1^f3							

Table 4.23: ZQL *query which expresses* $\sigma^v_{A_j=*}(V)$.

Name	X	Y	Z1	...	Zj	...	Zk
f1	-	-	-	...	-	...	-
f2=f1	x1 <− _	y1 <− _	v1 <− A_1._	...		...	vk <− A_k._
f3	x1	y1	v1	...	vj <− A_j.*	...	vk
*f4=f1^f3							

Table 4.24: ZQL *query which expresses* $\sigma^v_{A_j\neq *}(V)$.

$\sigma^v_{E\to(E')}(V)$: This case is trivial. Given f1 which represents V and f2 which is the filtering visual component for E', we simply the intersect the two to get f3=f1^f2 which represents $\sigma^v_{E\to(E')}$.

$\sigma^v_{E\to E'\wedge C}$: Once again assume we are given f1 which represents V and f2 which is the filtering visual component of E'. Based on the base expression proofs above, we know that given any base constraint C, we can find a filtering visual component for it; call this filtering visual component

f3. We can then see that f2^f3 is the filtering visual component of $E \to E' \wedge C$, and f4=f1^(f2^f3) represents $\sigma^v_{E \to E \wedge C}(V)$.

$\sigma^v_{E \to E' \vee C}$**:** Once again assume we are given f1 which represents V, f2 which is the filtering visual component of E', and we can find a filtering visual component f3 for C. We can then see that f2+f3 is the filtering visual component of $E \to E' \vee C$, and f4=f1^(f2+f3) represents $\sigma^v_{E \to E \vee C}(V)$.

With this inductive step, we have shown that for all complex constraints E of the form given by Line 4.10 of the CFG, we can find a ZQL query which expresses $\sigma^v_E(V)$. Given this, we can finally show that ZQL is capable of expressing $\sigma^v_\theta(V)$ for all θ: $\sigma^v_{\theta \to E}(V)$, $\sigma^v_{\theta \to E \wedge E')}(V)$, $\sigma^v_{\theta \to E \vee E'}(V)$, and $\sigma^v_{\theta \to \varepsilon}(V)$.

$\sigma^v_{\theta \to E}(V)$**:** This case is once again trivial. Assume, we are given f1 which represents V, and f2, which is the filtering visual component of E, f3=f1^f2 represents $\sigma^v_{\theta \to E}(V)$.

$\sigma^v_{\theta \to E \wedge E'}(V)$**:** Assume, we are given f1 which represents V, f2, which is the filtering visual component of E, and f3, which is the filtering visual component of E'. f2^f3 is the filtering visual component of $\theta \to E \wedge E'$, and f4=f1^(f2^f3) represents $\sigma^v_{\theta \to E \wedge E'}(V)$.

$\sigma^v_{\theta \to E \vee E'}(V)$**:** Assume, we are given f1 which represents V, f2 which is the filtering visual component of E, and f3 which is the filtering visual component of E'. f2+f3 is the filtering visual component of $\theta \to E \vee E'$, and f4=f1^(f2+f3) represents $\sigma^v_{\theta \to E \vee E'}(V)$.

$\sigma^v_{\theta \to \varepsilon}(V)$**:** This is the case in which no filtering is done. Therefore, given f1 which represents V, we can simply return f1.

Lemma 4.3. $\tau^v_{F(T)}(V)$ *is expressible in* ZQL *for all valid functionals* F *of* T *and visual groups* V.

Name	X	Y	Z1	...	Zk	Process
f1	-	-	-	...	-	
f2=f1	x1 <- _	y1 <- _	v1 <- A_1._	..	vk <- A_k._	x2, y2, u1, ..., uk <- $argmin_{x1,y1,v1,...,vk}[k = \infty]F(T)(f2)$
*f3	x2	y2	u1	...	uk	

Table 4.25: ZQL *query q which expresses* $\tau^v_{F(T)}(V)$.

Proof. Assume f1 is the visual component which represents V. Query q given by Table 4.25 produces visual component f3 which expresses $\tau^v_{F(T)}(V)$.

Lemma 4.4. $\mu^v_{[a:b]}(V)$ *is expressible in* ZQL *for all valid intervals* $a : b$ *and visual groups* V.

Name	X	Y	Z1	...	Zk	Process
f1	-	-	-	...	-	
*f2=f1[a:b]						

Table 4.26: ZQL *query q which expresses* $\mu^v_{[a:b]}(V)$.

Proof. Assume f1 is the visual component which represents V. Query q given by Table 4.26 produces visual component f2 which expresses $\mu^{v}_{[a:b]}(V)$.

Lemma 4.5. *$\zeta^{v}_{R,j}(V)$ is expressible in* ZQL *for all valid numbers j and visual groups V.*

Name	X	Y	Z1	...	Zk	Process
f1	-	-	-	...	-	
f2=f1	x1 <- _	y1 <- _	v1 <- A_1._	...	vk <- A_k._	x2, y2, u1, ..., uk <- $R(j,x1,y1,v1,...,vk,f2)$
*f3	x2	y2	u1	...	uk	

Table 4.27: ZQL *query q which expresses $\zeta^{v}_{R,j}(V)$.*

Proof. Assume f1 is the visual component which represents V and R' is the corresponding representative-finding algorithm in ZQL. Query q given by Table 4.27 produces visual component f3 which expresses $\mu^{v}_{[a:b]}(V)$.

Lemma 4.6. *$\delta^{v}(V)$ is expressible in* ZQL *for all valid visual groups V.*

Name	X	Y	Z1	...	Zk	Process
f1	-	-	-	...	-	
*f2=f1.range						

Table 4.28: ZQL *query q which expresses $\delta^{v}(V)$.*

Proof. Assume f1 is the visual component which represents V. Query q given by Table 4.28 produces visual component f2 which expresses $\delta^{v}(V)$.

Lemma 4.7. *$V \cup^{v} U$ is expressible in* ZQL *for all valid visual groups V and U.*

Name	X	Y	Z1	...	Zk	Process
f1	-	-	-	...	-	
f2	-	-	-	...	-	
*f3=f1+f2				...		

Table 4.29: ZQL *query q which expresses $V \cup^{v} U$.*

Proof. Assume f1 is the visual component which represents V and f2 represents U. Query q given by Table 4.29 produces visual component f3 which expresses $V \cup^{v} U$.

Lemma 4.8. *$V \setminus^{v} U$ is expressible in* ZQL *for all valid visual groups V and U.*

Proof. Assume f1 is the visual component which represents V and f2 represents U. Query q given by Table 4.30 produces visual component f3 which expresses $V \setminus^{v} U$. The proof for $\cap^{v}$ can be shown similarly.

Name	X	Y	Z1	...	Zk	Process
f1	-	-	-	...	-	
f2	-	-	-	...	-	
*f3=f1-f2				...		

Table 4.30: ZQL *query q which expresses* $V \setminus^v U$.

Lemma 4.9. $\beta^v_A(V,U)$ *is expressible in* ZQL *for all valid attributes A in* $\mathcal{V}$ *and visual groups V and U.*

Name	X	Y	Z1	...	Zk
f1	-	-	-	...	-
f2	-	-	-	...	-
f3=f1		y1 <- _	v1 <- A_1._	...	vk <- A_k._
f4=f2	x1 <- _				
*f5	x1[2]	y1[1]	v1[1]	...	vk[1]

Table 4.31: ZQL *query q which expresses* $\beta^v_A(V,U)$ *where* $A = X$.

Name	X	Y	Z1	...	Zk
f1	-	-	-	...	-
f2	-	-	-	...	-
f3=f1	x1 <- _		v1 <- A_1._	...	vk <- A_k._
f4=f2		y1 <- _			
*f5	x1[1]	y1[2]	v1[1]	...	vk[1]

Table 4.32: ZQL *query q which expresses* $\beta^v_A(V,U)$ *where* $A = Y$.

Proof. Assume f1 is the visual component which represents V and f2 represents U. There are three cases we must handle depending on the value of A due to the structure of columns in ZQL: *(i)* $A = X$ *(ii)* $A = Y$ *(iii)* $A = A_j$ where A_j is an attribute from the original relation $\mathcal{R}$. For each of the three cases, we produce a separate query which expresses β^v. For $A = X$, the query given by Table 4.31 produces f5 which is equivalent to $\beta^v_X(V,U)$ We use the superscripts in the last row so that cross product conforms to the ordering defined in Section 4.4. For $A = Y$, the query given by Table 4.32 produces f5 which is equivalent to $\beta^v_Y(V,U)$, and for $A = A_j$, the query given by Table 4.33 produces f5 which is equivalent to $\beta^v_{A_j}(V,U)$.

Lemma 4.10. $\phi^v_{F(D),A_1,\ldots,A_j}(V,U)$ *is expressible in* ZQL *for all valid attributes* $A_1,\ldots,A_j$ *and visual groups V and U.*

Name	X	Y	Z1	...	Zj-1	Zj	Zj+1	...	Zk
f1	-	-	-	...	-	-	-	...	-
f2	-	-	-	...	-	-	-	...	-
f3=f1	x1 <- _	y1 <- _	v1 <- A_1._	...	vj-1 <- A_{j-1}._		vj+1 <- A_{j+1}._	...	vk <- A_k._
f4=f2						uj <- A_j._			
*f5	x1[1]	y1[1]	v1[1]	...	vj-1[1]	uj[2]	vj+1[1]	...	vk[1]

Table 4.33: ZQL *query q which expresses* $\beta^v_A(V,U)$ *where* $A = A_j$ *and* A_j *is an attribute from* $\mathcal{R}$

Name	X	Y	Z1	...	Zj	Process
f1	-	-	-	...	-	
f2	-	-	-	...	-	
f3=f1			v1 <- A_1._	...	vj <- A_j._	
f4=f2.order			v1 ->	...	vj ->	u1, ..., uj <- $argmin_{v1,...,vj}[k=\infty]F(D)(f3,f4)$
*f5=f1.order			u1 ->	...	uj ->	

Table 4.34: ZQL *query q which expresses* $\phi^v_{F(D),A_1,...,A_j}(V,U)$.

Proof. Assume f1 is the visual component which represents V, and f2 represents U. Without loss of generality, assume the attributes we want to match on $(A_1,...,A_j)$ are the first j attributes of $\mathcal{R}$. Query q given by Table 4.34 produces visual component f5 which expresses $\phi^v_{F(D),A_1,...,A_j}(V,U)$. In the table, we first retrieve the values for $(A_1,...,A_j)$ using f3 and reorder f2 based on these values to get f4. We then compare the visualizations in f3 and f4 with respect to $(A_1,...,A_j)$ using the distance function $F(D)$ and retrieve the increasingly sorted $(A_1,...,A_j)$ values from the *argmin*. We are guaranteed that visualizations in f3 and f4 match up perfectly with respect to $(A_1,...,A_j)$ since the definition in Section 4.4 allows exactly one visual source to result from any $\sigma^v_{A_1=a_1\wedge...\wedge A_j=a_j}$. Finally, we reorder f1 according to these values to retrieve f5.

Lemma 4.11. $\eta^v_{F(D)}(V,U)$ *is expressible in* ZQL *for all valid functionals* F *of* D *and visual groups* V *and singleton visual groups* U.

Proof. Assume f1 is the visual component which represents V and f2 represents U. Query q given by Table 4.35 produces visual component f4 which expresses $\eta^v_{F(D)}(V,U)$.

Name	X	Y	Z1	...	Zk	Process
f1	-	-	-	...	-	
f2	-	-	-	...	-	
f3=f1	x1 <- _	y1 <- _	v1 <- A_1._	...	vk <- A_k._	x2, y2, u1, ..., uk <- $argmin_{x1,y1,v1,...,vk}[k=\infty]F(D)(f3,f2)$
*f4	x2	y2	u1	...	uk	

Table 4.35: ZQL *query q which expresses* $\eta^v_{F(D)}(V,U)$.

Although we have come up with a formalized algebra to measure the expressiveness of ZQL, ZQL is actually more expressive than visual exploration algebra. For example, ZQL allows the user to nest multiple levels of iteration in the Process column

Nevertheless, visual exploration algebra serves as a useful minimum metric for determining the expressiveness of visual exploration languages. Other visual analytics tools like Tableau are capable of expressing the selection operator σ^v in visual exploration algebra, but they are incapable of expressing the other operators which compare and filter visualizations based on summarization functions T, D, and R. General purpose programming languages with analytics libraries such as Python and Scikit-learn [45] are visual exploration complete since they are Turing-complete, but ZQL's declarative syntax strikes a novel balance between simplicity and expressiveness which allows even non-programmers to become data analysts as we will see in Chapter 9.1.

4.6 DISCUSSION OF CAPABILITIES AND LIMITATIONS

Although ZQL can capture a wide range of visual exploration queries, it is not limitless. Here, we give a brief description of what ZQL can and cannot do.

ZQL's primary goal is to support queries over visualizations—which are themselves aggregate group-by queries on data. Using these queries, ZQL can compose a collection of visualizations, filter them in various ways, compare them against benchmarks or against each other, and sort the results. The functions T and D, while intuitive, support the ability to perform a range of computations on visualization collections—for example, any filter predicate on a single visualization, checking for a specific visual property, can be captured under T. With the pluggable functions, the ability to perform sophisticated computation on visualization collections is enhanced even further. Then, via the dimension-based iterators, ZQL supports the ability to chain these queries with each other and compose new visualization collections. These simple set of operations offer unprecedented power in being able to sift through visualizations to identify desired trends. These features also distinguish ZQL from visualization languages languages such as D3 and Vega and tools such as Voyager. While the latter support more expressive features for specifying visual encodings and aesthetic properties for a given visualization, ZQL complements them with new primitives for operating over a collection of visualization based on patterns or trends.

Since ZQL already operates one layer above the data—on the visualizations—it does not support the *creation of new derived data*: that is, ZQL does not support the generation of derived attributes or values not already present in the data. The new data that is generated via ZQL is limited to those from binning and aggregating via the Viz column. This limits ZQL's ability to perform prediction—since feature engineering is an essential part of prediction; it also limits ZQL's ability to compose visualizations on combinations of attributes at a time, e.g., $\frac{A1}{A2}$ on the X axis. Among other drawbacks of ZQL: ZQL does not support (i) recursion; (ii) any data modification; (iii) non-foreign-key joins nor arbitrary nesting; (iv) dimensionality reduction or other changes to the attributes; (v) other forms of processing visualization collections not expressible via T, D

or the black box; (vi) multiple-dimensional visualizations; (vii) intermediate variable definitions; (viii) merging of visualizations (e.g., by aggregating two visualizations); and (ix) statistical tests.

Finally, to facilitate ease of specification, ZQL lets users omit visualization specification details as well as supports black-box primitives T and D with defaults parameters for pattern searching and visualization comparison. The downside of this is that it makes it harder for users to debug their queries. There remains important future work for balancing manual and automated query specification. One important future research work could be to let users compose T and D using more fine-grained primitives. Furthermore, we can provide more transparency and understanding on what Zenvisage does internally, e.g., how Zenvisage arrived at recommendations as well as display more interpretable outputs such as similarity scores of the visualizations.

4.7 OVERALL TAKEAWAYS

In this chapter, we describe how Zenvisage query language (ZQL) captures the data exploration needs with its supported operations on collections of visualizations. These operations include: composing collections of visualizations as well as filtering visualizations, comparing visualizations, and sorting visualizations based on some conditions. The conditions include similarity or dissimilarity to a specific pattern, "typical" or anomalous behavior, or the ability to provide explanatory or discriminatory power.

In the next chapter, we discuss how Zenvisage optimizes and executes ZQL queries over large datasets.

CHAPTER 5: ZENVISAGE QUERY OPTIMIZATION

An important challenge in building Zenvisage is the backend that supports the execution of ZQL. A single ZQL query can lead to the generation of tens of thousands of visualizations — executing each one independently as an aggregate query, would take several hours, rendering the tool somewhat useless. In this chapter, we discuss Zenvisage's execution engine, which is responsible for compiling and executing ZQL queries. The ZQL compiler translates ZQL queries into a combination of SQL queries to fetch the visualization collections and processing tasks to operate on them. We present a basic graph-based translation for ZQL and then provide several optimizations to the graph which reduce the overall runtime considerably.

5.1 BASIC TRANSLATION

Every valid ZQL query can be transformed into a query plan in the form of a directed acyclic graph (DAG). The DAG contains *c*-nodes (or *collection nodes*) to represent the collections of visualizations in the ZQL query and *p*-nodes (or *process nodes*) to represent the optimizations (or *processes*) in the Process column. Directed edges are drawn between nodes that have a dependency relationship. Using this query plan, the ZQL engine can determine at each step which visualization collection to fetch from the database or which process to execute. The full steps to build a query plan for any ZQL query is as follows: (i) Create a *c*-node or *collection node* for every collection of visualizations (including singleton collections). (ii) Create a *p*-node or *processor node* for every optimization (or *process*) in the Process column. (iii) For each *c*-node, if any of its axis variables are derived as a result of a process, connect a directed edge from the corresponding *p*-node. (iv) For each *p*-node, connect a directed edge from the *c*-node of each collection which appears in the process. Following these steps, we can translate our realistic query example in Table 4.13 in Chapter 4 to its query plan presented in Figure 5.1. The *c*-nodes are annotated with f#, and the *p*-nodes are annotated with p# (the *i*th *p*-node refers to the process in the *i*th row of the table). Here, f1 is a root node with no dependencies since it does not depend on any process, whereas f5 depends on the results of both p3 and p4 and have edges coming from both of them.

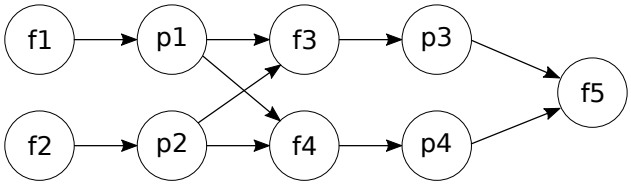


Figure 5.1: The query plan for the query presented in Table 4.13.

Once the query plan has been constructed, the ZQL engine can execute it using the simple algorithm presented in in Algorithm 5.1.

Algorithm 5.1. *Algorithm to execute* ZQL *query plan:*

1. *Search for a node with either no parents or one whose parents have all been marked as done.*
2. *Run the corresponding task for that node and mark the node as done.*
3. *Repeat steps 1 and 2 until all nodes have been marked as done.*

For *c*-nodes, the corresponding task is to retrieve the data for visualization collection, while for *p*-nodes, the corresponding task is to execute the process.

***c*-node translation:** At a high level, for *c*-nodes, the appropriate SQL group-by queries are issued to the database to compose the data for multiple visualizations at once. Specifically, for the simplest setting where there are no collections specified for X or Y, a SQL query in the form of:

```
SELECT X, A(Y), Z, Z2, ...
WHERE C(X, Y, Z, Z2, ...)
GROUP BY X, Z, Z2, ...
ORDER BY X, Z, Z2, ...
```

is issued to the database, where X is the X column attribute, Y is the Y column attribute, A(Y) is the aggregation function on Y (specified in the Viz column), Z, Z2, ... are the attributes/dimensions we are iterating over in the Z columns, while C(X, Y, Z, Z2, ...) refers to any additional constraints specified in the Z columns. The ORDER BY is inserted to ensure that all rows corresponding to a visualization are grouped together, in order. As an example, the SQL query for the *c*-node for f1 in Table 4.12 would have the form:

```
SELECT year, SUM(sales), product
GROUP BY year, product
ORDER BY year, product
```

If a collection is specified for the y-axis, each attribute in the collection is appended to the SELECT clause. If a collection is specified for the x-axis, a separate query must be issued for every X attribute in the collection. The results of the SQL query are then packed into a *m*-dimensional

array (each dimension in the array corresponding to a dimension in the collection) and labeled with its f# tag.

***p*-node translation:** At a high level, for *p*-nodes, depending on the structure of the expression within the process, the appropriate pseudocode is generated to operate on the visualizations. To illustrate, say our process is trying to find the top-10 values for which a trend is maximized/minimized with respect to various dimensions (using T), and the process has the form:

$$\langle argopt \rangle_{v0}[k = k'] \left[\langle op_1 \rangle_{v1} \left[\langle op_2 \rangle_{v2} \cdots \left[\langle op_m \rangle_{vm} T(f1) \right] \right] \right] \tag{5.1}$$

where $\langle argopt \rangle$ is one of argmax or argmin, and $\langle op \rangle$ refers to one of $(\max | \min | \sum | \prod)$. Given this, the pseudocode which optimizes this process can automatically be generated based on the actual values of $\langle argopt \rangle$, $\langle op \rangle$, and the number of operations. In short, for each $\langle op \rangle$ or dimension traversed over, the ZQL engine generates a new nested for loop. Within each for loop, we iterate over all values of that dimension, evaluate the inner expression, and then eventually apply the overall operation (e.g., max, $\sum$). For Equation 5.1, the generated pseudocode would look like the one given by Listing 5.1. Here, f refers to the visualization collection being operated on by the *p*-node, which the parent *c*-node should have already retrieved.

```
f = make_ndarray(SQL(...))
tmp0 = make_array(size=len(v0))
for i0 in [1 .. len(v0)]:
  tmp1 = make_array(size=len(v1))
  for i1 in [1 .. len(v1)]:
    tmp2 = make_array(size=len(v2))
    for i2 in [1 .. len(v2)]:
      ...
        tmpm = make_array(size=len(vn))
        for im in [1 .. len(vn)]:
          tmpm[im] = T(f[0, i1, i2, ..., im])
        tmpm-1[im-1] = opm(tmpm)
      ...
    tmp1[i1] = op1(tmp2)
  tmp0[i0] = op0(tmp1)
return argopt(tmp0)[:k']
```

Listing 5.1: Pseudocode for a process in the form of Equation 5.1.

Although this is the translation for one specific type of process, it is easy to see how the code generation would generalize to other patterns.

5.2 OPTIMIZATIONS

We now present several optimizations to the previously introduced basic translator. In preliminary experiments, we found that the SQL queries for the *c*-nodes took the majority of the runtime for ZQL queries, so we concentrate our efforts on reducing the cost of computing *c*-nodes. However, we do present one *p*-node-based optimization for process-intensive ZQL queries. We start with the simplest optimization schemes, and add more sophisticated variations later.

5.2.1 Parallelization

One natural way to optimize the graph-based query plan is to take advantage of the multi-query optimization (MQO) [63] present in databases and issue in parallel the SQL queries for independent *c*-nodes—the *c*-nodes for which there is no dependency between them. With MQO, the database can receive multiple SQL queries at the same time and share the scans for those queries, thereby reducing the number of times the data needs to be read from disk.

To integrate this optimization, we make two simple modifications to Algorithm 5.1. In the first step, instead of searching for a single node whose parents have all been marked done, search for *all* nodes whose parents have been marked as done. Then in step 2, issue the SQL queries for all *c*-nodes which were found in step 1 in parallel at the same time. For example, the SQL queries for `f1` and `f2` could be issued at the same time in Figure 5.1, and once `p1` and `p2` are executed, SQL queries for `f3` and `f4` can be issued in parallel.

5.2.2 Speculation

While parallelization gives the ZQL engine a substantial increase in performance, we found that many realistic ZQL queries intrinsically have a high level of interdependence between the nodes in their query plans. To further optimize the performance, we use *speculation*: the ZQL engine pre-emptively issues SQL queries to retrieve the superset of visualizations for each *c*-node, considering all possible outcomes for the axis variables. Specifically, by tracing the provenance of each axis variable back to the root, we can determine the superset of all values for each axis variable; then, by considering the Cartesian products of these sets, we can determine a superset of the relevant visualization collection for a *c*-node. After the SQL queries have returned, the ZQL engine proceeds through the graph as before, and once all parent *p*-nodes for a *c*-node have been

evaluated, the ZQL engine isolates the correct subset of data for that *c*-node from the pre-fetched data.

For example, in the query in Table 4.13 in Chapter 4, f3 depends on the results of p1 and p2 since it has constraints based on v2 and v4; specifically v2 and v4 should be locations and categories for which f1 and f2 have a negative trend. However, we note that v2 and v4 are derived as a result of v1 and v3, specified to take on all locations and categories in rows 1 and 2. So, a superset of f3, the set of profit over year visualizations for various products for all locations and categories (as opposed to just those that satisfy p1 and p2), could be retrieved pre-emptively. Later, when the ZQL engine executes p1 and p2, this superset can be filtered down correctly.

One downside of speculation is that a lot more data must be retrieved from the database, but we found that blocking on the retrieval of data was more expensive in runtime than retrieving extra data. Thus, speculation ends up being a powerful optimization which compounds the positive effects of parallelization.

5.2.3 Query Combination

From extensive modeling of relational databases, we found that the overall runtime of concurrently running issuing SQL queries is heavily dependent on the number of queries being run in parallel. Each additional query constituted a T_q increase in the overall runtime (e.g., for our settings of PostgreSQL, we found T_q = ~900ms). To reduce the total number of running queries, we use *query combination*; that is, given two SQL queries Q_1 and Q_2, we combine these two queries into a new Q_3 which returns the data for both Q_1 and Q_2. In general, if we have Q_1 (and Q_2) in the form of:

```
SELECT X1, A(Y1), Z1
WHERE C1(X1, Y1, Z1)
GROUP BY X, Z1
ORDER BY X, Z1
```

we can produce a combined Q_3 which has the form:

```
SELECT X1, A(Y1), Z1, C1, X2, A(Y2), Z2, C2
WHERE C1 or C2
GROUP BY X1, Z1, C1, X2, Z2, C2
ORDER BY X1, Z1, C1, X2, Z2, C2
```

where C1 = C1(X1, Y1, Z1) and C2 is defined similarly. From the combined query Q_3,

it is possible to regenerate the data which would have been retrieved using queries Q_1 and Q_2 by aggregating over the non-related groups for each query. For Q_1, we would select the data for which C1 holds, and for each (X1, Z1) pair, we would aggregate over the X2, Z2, and C2 groups.

While query combining is an effective optimization, there are limitations. We found that the overall runtime also depends on the number of unique group-by values per query, and the number of unique group-by values for a combined query is the product of the number of unique group-by values of the constituent queries. Thus, the number of average group-by values per query grows super-linearly with respect to the number of combinations. However, we found that as long as the combined query had less than M_G unique group-by values, it was more advantageous to combine than not (e.g., for our settings of PostgreSQL, we found M_G = 100k).

Formulation 5.1. Given the above findings, we can now formulate the problem of deciding which queries to combine as an optimization problem: *Find the best combination of* SQL *queries that minimizes:* $\alpha\times$*(total number of combined queries)* + $\sum_i$ *(number of unique group-by values in combined query i), such that no single combination has more than* M_G *unique group-by values.*

The cost of adding a thread, α, is generally more than M_G—for instance, in PostgreSQL we found $\alpha >$ 100k (M_G) for different experimental settings. By further assuming that the cost of processing all group by values $< M_G$ is same, we can simplify the problem to finding the minimum number of combined queries such that the maximum number of group by values per combined query is less than M_G. We prove that the solution to this problem is NP-HARD by reduction from the PARTITION PROBLEM.

Proof. Let $g_1, g_2, ...g_n$ be the group by values for the queries $Q_1, Q_2, ..., Q_n$ we want to combine. We want to find minimum number m of combined queries, such that each combined query G_i has at most M_G maximum group by values. Recall that in the Partition problem, we are given an instance of n numbers $a_1, a_2, ..., a_n$, and we are asked to decide if there is a set S such that $\sum_{a_i \subset S} a_i = \sum_{a_i \not\subset S} a_i$. Let $A = \sum a_i$ and consider an instance of Query Combination problem with $g_i = M_G^{\frac{2\times a_i}{A}}$. With this setting, it is easy to see that the answer to the Partition instance is YES if and only if the minimum number of combined queries is 2.

Wrinkle and Solution. However, a wrinkle to the above formulation is that it assumes no two SQL queries share a group-by attribute. If two queries have a shared group-by attribute, it may be more beneficial to combine those two, since the number of group-by values does not increase upon combining them. Overall, we developed the metric $EFGV$ or the effective increase in the number of group-by values to determine the utility of combining query Q' to query Q: $EFGV_Q(Q') = \prod_{g \in G(Q')} \#(g)^{[[g \notin G(Q)]]}$ where $G(Q)$ is the set of group-by values in Q, $\#(g)$ calculates the number

of unique group-by values in g, and $[[g \notin G(Q)]]$ returns 1 if $g \notin G(Q)$ and 0 otherwise. In other words, this calculates the product of group-by values of the attributes which are in Q' but not in Q. Using the $EFGV$ metric, we then apply a variant of agglomerative clustering [64] to decide the best choice of queries to combine. As we show in the experiments, this technique leads to very good performance.

5.2.4 Cache-Aware Execution

Although the previous optimizations were all I/O-based optimizations for ZQL, there are cases in which optimizing the execution of p-nodes is important as well. In particular, when a process has multiple nested for loops, the cost of the p-node may start to dominate the overall runtime. To address this problem, we adapt techniques developed in high-performance computing—specifically, cache-based optimizations similar to those used in matrix multiplication [65]. With cache-aware execution, the ZQL engine partitions the iterated values in the for loops into blocks of data which fit into the L3 cache. Then, the ZQL engine reorders the order of iteration in the for loops to maximize the time that each block of data remains in the L3 cache. This allows the system to reduce the amount of data transfer between the cache and main memory, minimizing the time taken by the p-nodes.

5.3 EXPERIMENTAL STUDY

In this section, we evaluate the runtime performance of the ZQL engine. We present the runtimes for executing both synthetic and realistic ZQL queries and show that we gain speedups of up to $3\times$ with the optimizations We also varied the characteristics of a synthetic ZQL query to observe their impact on our optimizations. Finally, we show that disk I/O was a major bottleneck for the ZQL engine, and if we switched our back-end database to a column-oriented database and cache the dataset in memory, we can achieve interactive run times for datasets as large as 1.5GB.

Setup. All experiments were conducted on a 64-bit Linux server with 8 3.40GHz Intel Xeon E3-1240 4-core processors and 8GB of 1600 MHz DDR3 main memory. We used PostgreSQL with working memory size set to 512 MB and shared buffer size set to 256MB for the majority of the experiments; the last set of experiments demonstrating interactive run times additionally used Vertica Community Edition with a working memory size of 7.5GB.

PostgreSQL Modeling. For modeling the performance on issuing multiple parallel queries with varying number of group by values, we varied the number of parallel queries issued (#Q) from 1 to 100, and the group by values per query (#V) from 10 to 100000, and recorded the response times

	Query Description	# *c*-nodes	# *p*-nodes	# *T*	# *D*	#V	# SQL Queries: NO-OPT	# SQL Queries: SMARTFUSE
1	Plot the related visualizations for airports which have a correlation between arrival delay and traveled distances for flights arriving there.	6	3	670	93,000	18,642	6	1
2	Plot the delays for carriers whose delays have gone up at airports whose average delays have gone down over the years.	5	4	1,000	0	11,608	4	1
3	Plot the delays for the outlier years, outlier airports, and outlier carriers with respect to delays.	12	3	0	94,025	4,358	8	2

Table 5.1: Realistic queries for the airline dataset with the # of c-nodes, # of p-nodes, # of T functions calculated, # of D functions calculated, # of visualizations explored, # of SQL *queries issued with* NO-OPT*, and # of* SQL *queries issued with* SMARTFUSE *per query.*

(T). We observed that the time taken for a batch of queries was practically linearly dependent to both the number of queries as well as the group by values. Fitting a linear equation by performing multiple regression over the observed data, we derived the following cost-model,

$$T(ms) = 908 \times (\#Q) + 1.22 \times \frac{(\#V)}{100} + 1635 \tag{5.2}$$

As per the above model, adding a thread leads to the same rise in response time as increasing the number of group by values by 75000 over the existing threads in the batch. In other words, it is better to merge queries with small number of group by values. Moreover, since there is a fixed cost (1635 ms) associated with every batch of queries, we tried to minimize the number of batches by packing as many queries as possible within the memory constraints.

Optimizations. The four versions of the ZQL engine we use are: (i) NO-OPT: The basic translation. (ii) PARALLEL: Concurrent SQL queries for independent nodes. (iii) SPECULATE: Speculating and pre-emptively issuing SQL queries. (iv) SMARTFUSE: Query combination with speculation. In our experiments, we consider NO-OPT and the MQO-dependent PARALLEL to be our baselines, while SPECULATE and SMARTFUSE were considered to be completely novel optimizations. For certain experiments later on, we also evaluate the performance of the caching optimizations from Chapter 5.2.4 on SMARTFUSE.

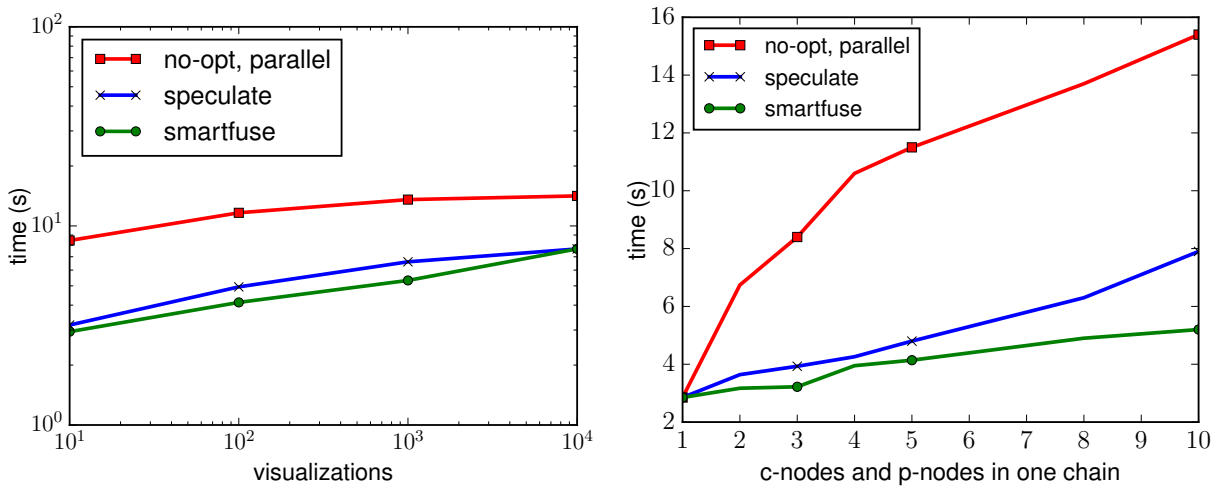


Figure 5.2: Effect of number of visualizations (left) and length of the chain (right) on the overall runtimes.

5.3.1 Realistic Queries

For our realistic queries, we used 20M rows of a real 1.5GB airline dataset [66] which contained the details of flights within the USA from 1987-2008, with 11 attributes. On this dataset, we performed 3 realistic ZQL queries inspired by the case studies in our introduction. Descriptions of the queries can be found in Table 5.1.

Figure 5.2 (left) depicts the runtime performance of the three realistic ZQL queries, for each of the optimizations. For all queries, each level of optimization provided a substantial speedup in execution time compared to the previous level. *Simply by going from* NO-OPT *to* PARALLEL*, we saw a 45% reduction in runtime. From* PARALLEL *to* SPECULATE *and* SPECULATE *to* SMARTFUSE*, we saw 15-20% reductions in runtime*. A large reason for why the optimizations were so effective was because ZQL runtimes are heavily dominated by the execution time of the issued SQL queries. In fact, we found that for these three queries, 94-98% of the overall runtime could be contributed to the SQL execution time. We can see from Table 5.1, SMARTFUSE always managed to lower the number of SQL queries to 1 or 2 after our optimizations, thereby heavily impacting the overall runtime performance of these queries.

5.3.2 Varying Characteristics of ZQL Queries

We were interested in evaluating the efficacy of our optimizations with respect to four different characteristics of a ZQL query: (i) the number of visualizations to explore, (ii) the complexity of the ZQL query, (iii) the level of interconnectivity within the ZQL query, and (iv) the complexity of

the processes. To control for all variables except these characteristics, we used a synthetic chain-based ZQL query to conduct these experiments. Every row of the chain-based ZQL query specified a collection of visualizations based on the results of the process from the previous row, and every process was applied on the collection of visualizations from the same row. Therefore, when we created the query plan for this ZQL query, it had the chain-like structure. Using the chain-based ZQL query, we could then (i) vary the number of visualizations explored, (ii) use the length of the chain as a measure of complexity, (iii) introduce additional independent chains to decrease interconnectivity, and (iv) increase the number of loops in a *p*-node to control the complexity of processes.

To study these characteristics, we used a synthetic dataset with 10M rows and 15 attributes (10 dimensional and 5 measure) with cardinalities of dimensional attributes varying from 10 to 10,000. By default, we set the input number of visualizations per chain to be 100, with 10 values for the X attribute, number of *c*-nodes per chain as 5, the process as T (with a single for loop) with a selectivity of .50, and number of chains as 1.

Impact of number of visualizations. Figure 5.2 (left) shows the performance of NO-OPT, SPECULATE, and SMARTFUSE on our chain-based ZQL query as we increased the number of visualizations that the query operated on. The number of visualizations was increased by specifying larger collections of Z column values in the first *c*-node. We chose to omit PARALLEL here since it performs identically to NO-OPT. With the increase in visualizations, the overall response time increased for all versions because the amount of processing per SQL query increased. SMARTFUSE showed better performance than SPECULATE up to 10k visualizations due to reduction in the total number of SQL queries issued. However, at 10k visualization, we reached the threshold of the number of unique group-by values per combined query (100k for PostgreSQL), so it was less optimal to merge queries. At that point, SMARTFUSE behaved similarly to SPECULATE.

Impact of the length of the chain. We varied the length of the chain in the query plan (or the number of rows in the ZQL query) to simulate a change in the complexity of the ZQL query and plotted the results in Figure 5.2 (right). As the number of nodes in the query plan grew, the overall runtimes for the different optimizations also grew. However, while the runtimes for both NO-OPT and SPECULATE grew at least linearly, the runtime for SMARTFUSE grew sublinearly due to its query combining optimization. While the runtime for NO-OPT was much greater than for SPECULATE, since the overall runtime is linearly dependent on the number of SQL queries run in parallel, we see a linear growth for SPECULATE.

Impact of the number of chains. We increased the number of independent chains from 1 to 5 to observe the effect on runtimes of our optimizations; the results are presented in Figure 5.3

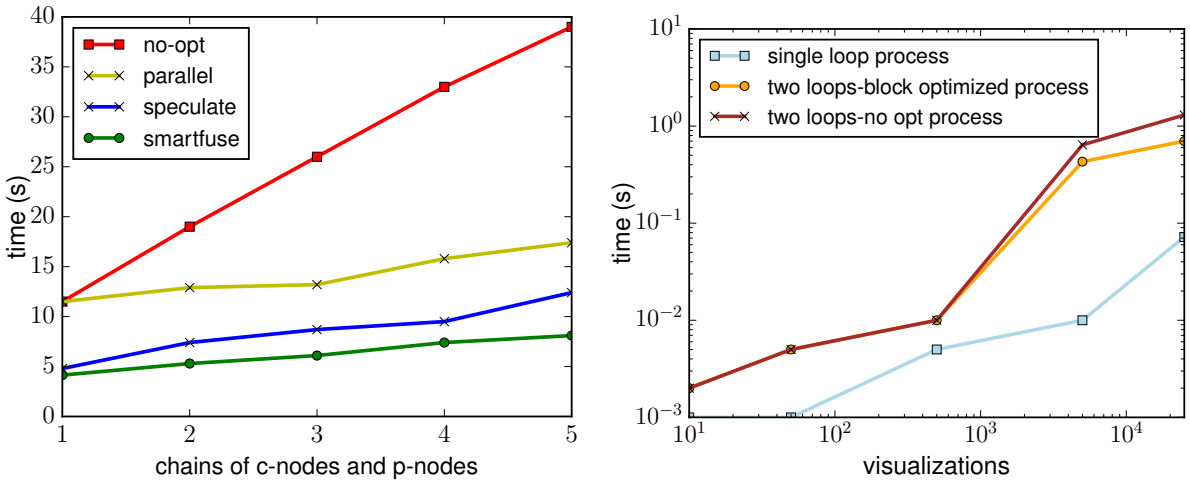


Figure 5.3: Effect of number of independent chains (left) and the number of loops in a p-node (right) on the overall runtimes.

(left). While NO-OPT grew linearly as expected, all PARALLEL, SPECULATE, and SMARTFUSE were close to constant with respect to the number of independent chains. We found that while the overall runtime for concurrent SQL queries did grow linearly with the number of SQL queries issued, they grew much slower compared to issuing those queries sequentially, thus leading to an almost flat line in comparison to NO-OPT.

Impact of process complexity. We increased the complexity of processes by increasing the number of loops in the first *p*-node from 1 to 2. For the single loop, the *p*-node filtered based on a positive trend via T, while for the double loop, the *p*-node found the outlier visualizations. Then, we varied the number of visualizations to see how that affected the overall runtimes. Figure 5.3 (right) shows the results. For this experiment, we compared regular SMARTFUSE with cache-aware SMARTFUSE to see how much of a cache-aware execution made. We observed that there was not much difference between cache-aware SMARTFUSE and regular SMARTFUSE below 5k visualizations when all data could fit in cache. After 5k visualizations, not all the visualizations could be fit into the cache the same time, and thus the cache-aware execution of the *p*-node had an improvement of 30-50% as the number of visualizations increased from 5k to 25k. However, this improvement, while substantial, is only a minor change in the overall runtime.

5.3.3 Interactivity

The previous figures showed that the overall execution times of ZQL queries took several seconds, even with SMARTFUSE, thus perhaps indicating ZQL is not fit for interactive use with large datasets. However, we found that this was primarily due to the disk-based I/O bottleneck of SQL

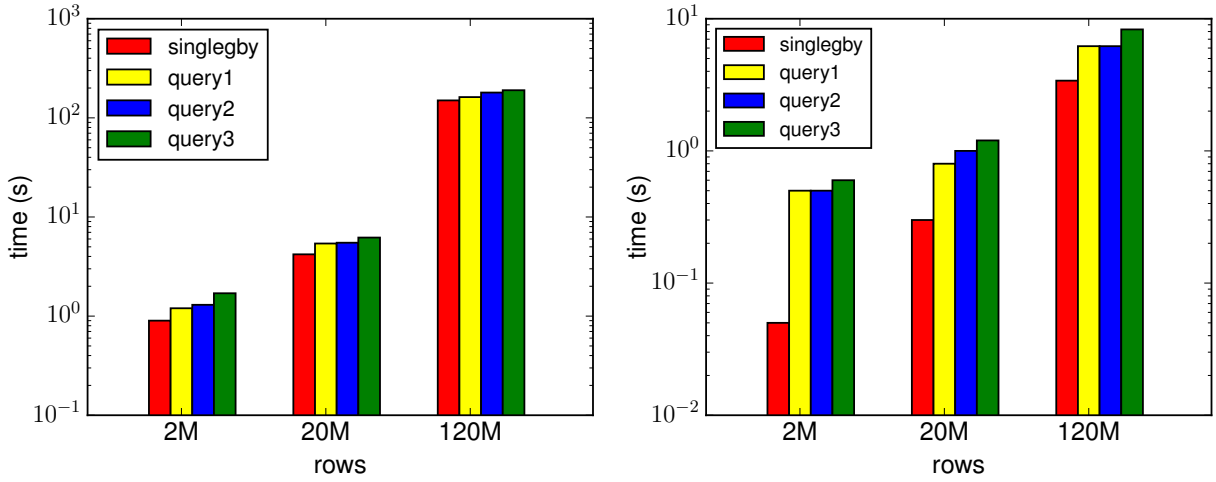


Figure 5.4: SMARTFUSE *on PostgreSQL (left) and Vertica (right)*

queries. In Figure 5.4 (left), we show the SMARTFUSE runtimes of the 3 realistic queries from before on varying size subsets of the airline dataset, with the time that it takes to do a single group-by scan of the dataset. As we can see, the runtimes of the queries and scan time are *virtually the same*, indicating that SMARTFUSE comes very close to the optimal I/O runtime (i.e., a "fundamental limit" for the system).

To further test our hypothesis, we ran our ZQL engine with Vertica with a large working memory size to cache the data in memory to avoid expensive disk I/O. The results, presented in Figure 5.4 (right), showed that there was a 50× speedup in using Vertica over PostgreSQL with these settings. Even with a large dataset of 1.5GB, we were able to achieve sub-second response times for many queries. Furthermore, for the dataset with 120M records (11GB, so only 70% could be cached), we were able to reduce the overall response times from 100s of seconds to less than 10 seconds. Thus, once again, Zenvisage returned results in a small multiple of the time it took to execute a single group-by query. Overall, SMARTFUSE is interactive on moderate sized datasets on PostgreSQL, or on large datasets that can be cached in memory and operated on using a columnar database—which is standard practice adopted by visual analytics tools [67]. Improving on interactivity is impossible due to fundamental limits to the system; in the future, we plan to explore returning approximate answers using samples, since even reading the entire dataset is prohibitively slow.

5.4 OVERALL TAKEAWAYS

In this chapter, we described ZQL, the underlying query language for Zenvisage for operating over collections of visualizations at a time. Unlike relational query languages that operate directly on data, ZQL operates on collections of visualizations, which are themselves aggregate queries

on data. ZQL conceptually captures two types of operations: composition and processing. The composition operation enables us to compose a collection of visualizations, while the processing operation allows us to operate on a collection of visualizations, by filtering, comparing, or sorting these visualizations. ZQL is a synthesis of desiderata after discussions with data scientists from a variety of domains, and we show that ZQL is expressive — it builds on a visual exploration algebra that we develop and has nice theoretical properties

An important challenge in building Zenvisage is the backend that supports the execution of ZQL. A single ZQL query can lead to the generation of 10,000s of visualization — executing each one independently as an aggregate query, would take several hours, rendering the tool somewhat useless. Zenvisage 's query optimizer operates as a wrapper over any traditional relational database system. This query optimizer compiles ZQL queries down to a directed acyclic graph of operations on collections of visualizations, followed with the optimizer using a combination of intelligent speculation and combination, to issue queries to the underlying database. We also demonstrated that the underlying problem is NP-HARD. Our query optimizer leads to substantial improvements over the naive schemes adopted within relational database systems for multi-query optimization.

In the next chapter, we present an user evaluation of Zenvisage focused on evaluating its effectiveness and usability.

CHAPTER 6: ZENVISAGE USER EVALUATION

In this chapter, we present a user study that we conducted to evaluate the utility of Zenvisage for data exploration versus two types of systems—first, visualization tools, similar to Tableau, and second, general database and data mining tools, which also support interactive analytics to a certain extent. In preparation for the user study, we conducted interviews with data analysts to identify the typical exploration tasks and tools used in their present workflow. Using these interviews, we identified a set of tasks to be used in the user study for Zenvisage. We describe these interviews first, followed by the user study details.

6.1 ANALYST INTERVIEWS AND TASK SELECTION

We hired seven data analysts via Upwork [68], a freelancing platform—we found these analysts by searching for freelancers who had the keywords `analyst` or `tableau` in their profile. We conducted one hour interviews with them to understand how they perform data exploration tasks. The interviewees had 3—10 years of prior experience and explained every step of their workflow; from receiving the dataset to presenting the analysis to clients. The rough workflow of all interviewees identified was the following: first, data cleaning is performed; subsequently, the analysts perform data exploration; then, the analysts develop presentations using their findings. We then drilled down onto the data exploration step.

We first asked the analysts what types of tools they use for data exploration. Analysts reported nine different tools—the most popular ones included Excel (5), Tableau (3), and SPSS (2). The rest of the tools were reported by just one analyst: Python, SQL, Alteryx, Microsoft Visio, Microsoft BI, SAS. Perhaps not surprisingly, analysts use both visualization tools (Tableau, Excel, BI), programming languages (Python), statistical tools (SAS, SPSS), and relational databases (SQL) for data exploration.

Then, to identify the common tasks used in data exploration, we used a taxonomy of abstract exploration tasks proposed by Amar et al. [69]. Amar et al. developed their taxonomy through summarizing the analytical questions that arose during the analysis of five different datasets, independent of the capabilities of existing tools or interfaces. The exploration tasks in Amar et al. include: filtering (f), sorting (s), determining range (r), characterizing distribution (d), finding anomalies (a), clustering (c), correlating attributes (co), retrieving value (v), computing derived value (dv), and finding extrema (e). When we asked the data analysts which tasks they use in their workflow, the responses were consistent in that all of them use all of these tasks, except for three exceptions—c, reported by four participants, and e, d, reported by six participants.

Given these insights, we selected a small number of appropriate tasks for our user study encom-

passing eight of the ten exploration tasks described above: f, s, r, d, a, c, co, v. The other two—dv and e—finding derived values and computing extrema, are important tasks in data analysis, but existing tools (e.g., Excel) already provide adequate capabilities for these tasks, and we did not expect Zenvisage to provide additional benefits.

6.2 USER STUDY METHODOLOGY

The goal of our user study was to evaluate Zenvisage with other tools, on its ability to effectively support data exploration.

Participants. We recruited 12 graduate students as participants with varying degrees of expertise in data analytics. Table 6.1 depicts the participants' experience with different categories of tools.

Tools	Count
Excel, Google spreadsheet, Google Charts	8
Tableau	4
SQL, Databases	6
Matlab,R,Python,Java	8
Data mining tools such as weka, JNP	2
Other tools like D3	2

Table 6.1: Participants' prior experience with data analytic tools

Baselines. For the purposes of our study, we explicitly wanted to do a head-to-head qualitative and quantitative comparison with visual analytics tools, and thus we developed a baseline tool to compare Zenvisage against directly. Further, via qualitative interviews, we compared Zenvisage versus against other types of tools, such as databases, data mining, and programming tools. Our baseline tool was developed by replicating the visualization selection capabilities of visual analytics tools with a styling scheme identical to Zenvisage to control for external factors. The tool allowed users to specify the x-axis, y-axis, dimensions, and filters. The tool would then populate all visualizations meeting the specifications.

Comparison Points. There are no tools that offer the same functionalities as Zenvisage. Visual analytics tools do not offer the ability to search for specific patterns, or issue complex visual exploration queries; data mining toolkits do not offer the ability to search for visual patterns and are instead tailored for general machine learning and prediction. Since visual analytics tools are closer

in spirit and functionality to Zenvisage, we decided to implement a visual analytics tool as our baseline. Thus, our baseline tool replicated the basic query specification and output visualization capabilities of existing tools such as Tableau. We augmented the baseline tool with the ability to specify an arbitrary number of filters, allowing users to use filters to drill-down on specific visualizations. This baseline visualization tool was implemented with a styling scheme similar to Zenvisage to control for external factors. As depicted in Figure 6.1, the baseline allowed users to visualize data by allowing them to specify the x-axis, y-axis, category, and filters. The baseline tool would populate all the visualizations, which fit the user specifications, using an alpha-numeric sort order. In addition to task-based comparisons with this baseline, we also explicitly asked participants to compare Zenvisage with existing data mining and visual analytics tools that they use in their workflow.

Dataset. We used a housing dataset from Zillow.com [70], consisting of housing sales data for different cities, counties, and states from 2004-15, with over 245K rows, and 15 attributes. We selected this dataset since participants could relate to the dataset and understand the usefulness of the tasks.

Tasks. We designed the user study tasks with the case studies from Chapter 1 in mind, and translated them into the housing dataset. Further, we ensured that these tasks together evaluate eight of the exploration tasks described above—f, s, r, d, a, c, co, and v. One task used in the user study is as follows: "Find three cities in the state of NY where the sold price vs year trend is very different from the state's overall trend." This query required the participants to first retrieve the trend of NY (v) and characterize its distribution (d), then separately filter to retrieve the cities of NY (f), compare the values to find a negative correlation (co), sort the results (s), and report the top three cities on the list.

Study Protocol. The user study was conducted using a within-subjects study design [71], forming three phases. First, participants described their previous experience with data analytics tools. Next, participants performed exploration tasks using Zenvisage (Tool A) and the baseline tool (Tool B), with the orders randomized to reduce order effects. Participants were provided a 15-minute tutorial-cum-practice session per tool to get familiarized before performing the tasks. Finally, participants completed a survey that both measured their satisfaction levels and preferences, along with open-ended questions on the strengths and weaknesses of Zenvisage and the baseline, when compared to other analytics tools they may have used. The average study session lasted for 75 minutes on average. Participants were paid ten dollars per hour for their participation. After the study, we reached out to participants with backgrounds in data mining and programming, and asked if they could complete a follow-up interview where they use their favorite analytics tool for

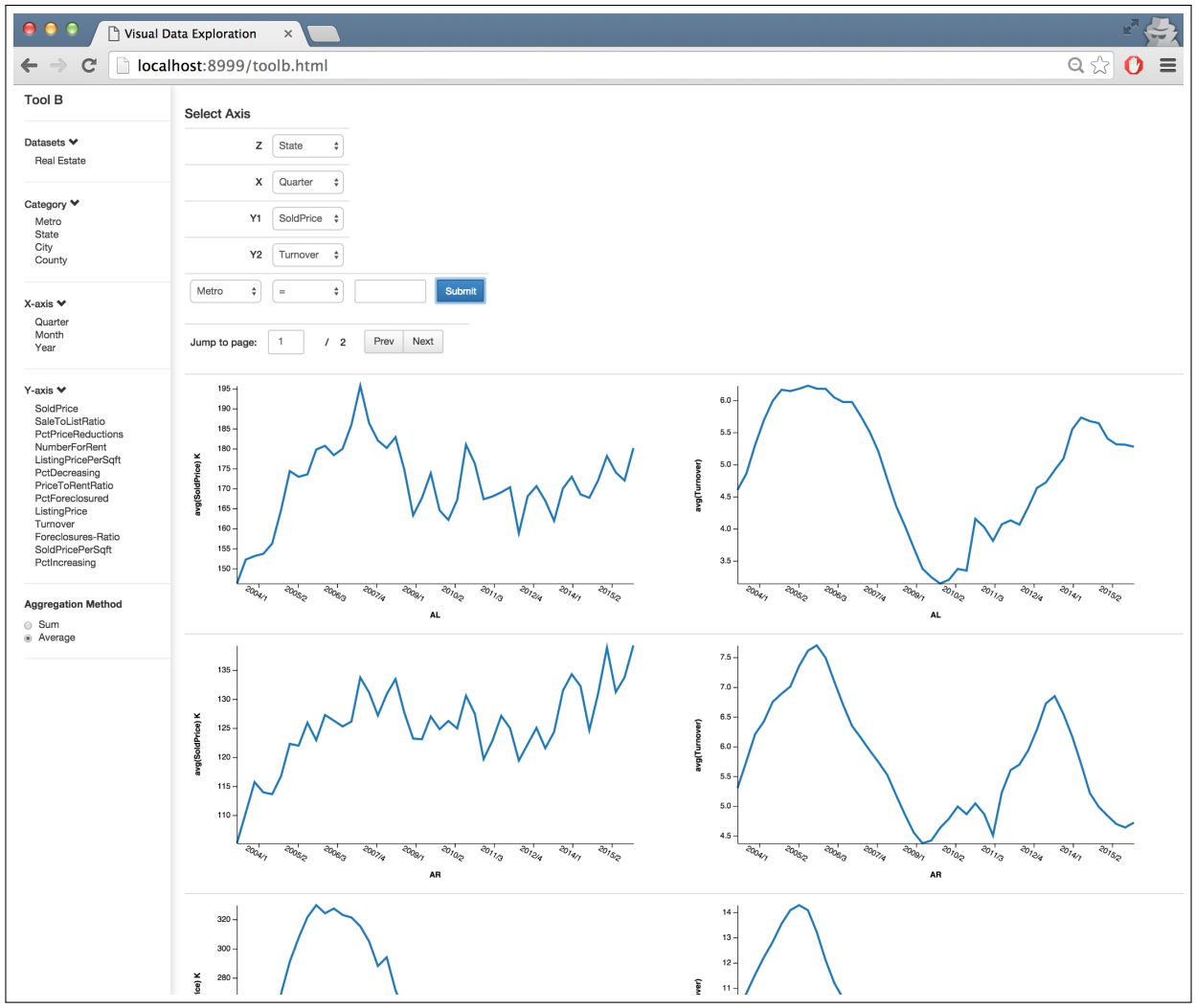


Figure 6.1: The baseline interface implemented for the user study.

performing one of the tasks.

Metrics. Using data that we recorded, we collected the following metrics: completion time, accuracy, and the usability ratings and satisfaction level from the survey results. In addition, we also explicitly asked participants to compare Zenvisage with tools that they use in their workflow. For comparisons between Zenvisage and general database and data mining tools via follow-up interviews, we used the number of lines of code to evaluate the differences.

Ground Truth. Two expert data analysts prepared the ground truth for each the tasks in the form of ranked answers, along with score cut-offs on a 0 to 5 scale (5 highest). Their inter-rater agreement, measured using Kendall's Tau coefficient, was 0.854. We took the average of the two scores to rate the participants' answers.

6.3 KEY FINDINGS

Three key findings emerged from the study and are described below. We use μ, σ, χ^2 to denote average, standard deviation, and Chi-square test scores, respectively.

Finding 6.1. Zenvisage **enables faster and more accurate exploration than existing visualization tools.** Since all of our tasks involved generating multiple visualizations and comparing them to find desired ones, participants were not only able to complete the tasks *faster*—μ=115s, σ=51.6 for Zenvisage vs. μ=172.5s, σ=50.5 for the baseline—but also more *accurately*—μ=96.3%, σ=5.82 for Zenvisage vs. μ=69.9%, σ=13.3 for the baseline. A one-way between-subjects ANOVA, followed by a post-hoc Tukey's test [72], we found that Zenvisage had statistically significant faster task completion times compared to the baseline interface, with p value of 0.0069. The baseline required considerable manual exploration to complete the same task, explaining the high task completion times. In addition, participants frequently compromised by selecting suboptimal answers before browsing the entire list of results for better ones, explaining the low accuracy. On the other hand, Zenvisage was able to automate the task of finding desired visualizations, considerably reducing manual effort. Also of note is the fact that the accuracy with Zenvisage was close to 100%—indicating that a short 15 minute tutorial on ZQL was enough to equip users with the knowledge they needed to address the tasks—and that too, within 2 minutes (on average).

When asked about using Zenvisage vs. the baseline in their current workflow, 9 of the 12 participants stated that they would use Zenvisage in their workflow, whereas only two participants stated that they would use our baseline tool ($\chi^2 = 8.22$, p<0.01). When the participants were asked how, one participant provided a specific scenario: *"If I am doing my social science study, and I want to see some specific behavior among users, then I can use tool A [*Zenvisage *] since I can find the trend I am looking for and easily see what users fit into the pattern."* (P7). In response to the survey question *"I found the tool to be effective in visualizing the data I want to see"*, the participants rated Zenvisage higher (μ=4.27, σ=0.452) than the baseline (μ=2.67, σ=0.890) on a five-point Likert scale. A participant experienced in Tableau commented: *"In Tableau, there is no pattern searching. If I see some pattern in Tableau, such as a decreasing pattern, and I want to see if any other variable is decreasing in that month, I have to go one by one to find this trend. But here I can find this through the query table."* (P10).

Finding 6.2. Zenvisage **complements existing database and data mining systems, and programming languages.** When explicitly asked about comparing Zenvisage with the tools they use on a regular basis for data analysis, all participants acknowledged that Zenvisage adds value in data exploration not encompassed by their tools. ZQL augmented with inputs from the sketching canvas proved to be extremely effective. For example P8 stated: *"you can just [edit] and draw to*

find out similar patterns. You'll need to do a lot more through Matlab to do the same thing." Another experienced participant mentioned the benefits of not needing to know much programming to accomplish certain tasks: *"The obvious good thing is that you can do complicated queries, and you don't have to write SQL queries... I can imagine a non-cs student [doing] this."* (P9). When asked about the specific tools they would use to solve the user study tasks, all participants reported a programming language like Matlab or Python. This is despite half of the participants reporting using a relational database regularly, and a smaller number of participants (2) reporting using a data mining tool regularly. Additionally, multiple participants, even those with extensive programming experience, reported that Zenvisage would take less time and fewer lines of code for certain data exploration tasks. (Indeed, we found that all participants were able to complete the user study tasks in under 2 minutes.) In follow-up email interviews, we asked a few participants to respond with code from their favorite data analytics tool for the user study tasks. Two participants responded — one with Matlab code, one with Python code. Both these code snippets were much longer than ZQL: as a concrete example, the participant accomplished the same task with 38 lines of Python code compared to 4 lines of ZQL. While comparing code may not be fair, the roughly order of magnitude difference demonstrates the power of Zenvisage over existing systems.

Finding 6.3. Zenvisage **can be improved.** While the participants looked forward to using custom query builder in their own workflow, a few of them were interested in directly exposing the commonly-used trends/patterns such as outliers, through the drag and drop interface. Some were interested in knowing how they could integrate custom functional primitives (we could not cover it in the tutorial due to time constraints). In order to improve the user experience, participants suggested adding instructions and guidance for new users as part of the interface. Participants also commented on the unrefined look and feel of the tool, as well as the lack of a diverse set of usability related features, such as bookmarking and search history, that are offered in existing systems.

6.4 OVERALL TAKEAWAYS AND FUTURE WORK

In this chapter, we presented a user study, comparing Zenvisage with a custom-made manual visual analytics tool to assess how Zenvisage can aid in visualizing interesting patterns, trends, or insights from large datasets. We also conducted additional interviews with data analysts to identify the low level tasks and tools used in their typical workflow, and explored how Zenvisage can help with their everyday activities. Our findings show that that Zenvisage enables much faster task completion times, as well as helps retrieve more accurate results than the manual visualization tools. Even those with considerable experience with programming languages such as Python and Matlab believe that Zenvisage takes less time and fewer lines of code for certain data exploration

tasks, and can be a valuable addition to their analytics workflow.

Zenvisage has also found usage in domains such as genomics, astronomy, and material science [73]. In genomics, scientists have used Zenvisage for understanding the gene expression profiles of breast cancer cells that exhibit induced, transient, and repressed patterns after a particular treatment. Scientists were able to gain novel scientific insights, such as finding characteristic gene expression profiles that confirmed the results of a related publication. In astronomy, researchers have used Zenvisage for studying common patterns among stars that exhibit planetary transits versus stars that don't. And in material science, Zenvisage has been used for identifying battery solvents with favorable properties and mass production potential through studying how changes in certain chemical properties correlate to changes in other chemical properties.

In addition, we found that Zenvisage can be be used beyond the exploratory phase of analysis, for data verification, debugging preliminary datasets, and performing sanity-checks on downstream models. For example, during their analysis, Zenvisage allowed astrophysicists to debug a mislabelled feature, leading them to find that the way data was aggregated was erroneous on a collaborator's dataset.

Nonetheless, there are several open questions that we need to address to further improve the usability of Zenvisage. First, we need to provide more transparency and understanding on what Zenvisage does internally, e.g., how Zenvisage arrives at recommendations, displaying more interpretable outputs such as similarity scores of the visualizations, as well as providing more system-level control to users, e.g., setting a minimum similarity threshold for displaying the search, as well as the ability to tune the smoothing algorithms and parameters.

Another barrier to adoption of Zenvisage involves the limited support for data preparation tasks such as schema understanding, data cleaning, imputing missing values, data integration, and facilitating export of visualizations for downstream analysis. A more tighter integration between these pre-processing tasks and the visual analysis operations that are currently supported in Zenvisage can eliminate detour and distraction and lead to faster insight generation.

Finally, many users want Zenvisage to support more sophisticated features that are typically used in analytics workflows. For example, users want to dynamically create subsets of data for each visualizations, rather than using the fixed categorical attribute. Other feature requests includes uploading the values of pattern to be searched via a CSV file, using an input equation, e.g. , $y = x^2$ for querying, and data smoothing.

CHAPTER 7: SHAPESEARCH SYSTEM OVERVIEW

In this chapter, we present ShapeSearch, a pattern-based querying system for trendlines that extends the capabilities of Zenvisage by providing more flexible mechanisms for specifying desired patterns of interest.

We, first, start with the motivation for ShapeSearch, outlining the common characteristics of pattern-based queries that are difficult to satisfy using existing tools (Section 7.1). We, then, provide a brief overview of our key ideas (Section 7.2), and the system design (Section 7.3) that show how ShapeSearch satisfies the desired characteristics. In the remaining sections, we dig deeper into the key components of ShapeSearch. First, we give an overview of a novel ShapeQuery algebra (Section 7.4), along with its formal semantics (Section 7.5), and scoring mechanisms (Section 7.6), form the.core of ShapeSearch. We, then, describe the parser that we built from scratch for parsing natural language-based shape queries (Section 7.7). Finally, we compare with additional prior work that are related to ShapeSearch (Section 7.8).

7.1 MOTIVATION

Identifying patterns in trendlines is an integral part of data exploration—routinely performed by domain experts to make sense of their datasets, gain new insights, and validate their hypotheses. For example, clinical data analysts examine trends of health-indicators such as temperature and heart-rate for diagnosis of medical conditions; astronomers study the variation in properties of galaxies over time to understand the history and makeup of the Universe; biologists analyze gene expression patterns over time to study biological processes; and financial analysts study patterns in stock trends data to predict future behaviour. Due to the lack of extensive programming experience, these domain experts typically perform manual exploration, tediously examining trendlines one at a time until they find ones that match their desired shape or pattern, e.g., gene expressions that rise and then become stable.

Limitations of existing tools. Recent work including Zenvisage let users interactively search for desired patterns [9, 74, 75]. However, as we will discuss below, these tools expect users to search in highly *constrained* ways, and, in addition, are *overly rigid* in how they assess a match. Most tools expect users to specify a complete and exact trendline as an input usually by sketching it on a canvas, followed by computing distances between this exact trendline and several candidate trendlines to identify matches. As a result, these tools are unable to support search when the desired shape is *under-specified or approximate*, e.g., finding stocks whose prices are decreasing for some time, followed by a sharp rise, with the position and intensity of movements being left

unspecified, or when the desired shape is *complex*, e.g., finding gene expression profiles where there is an unspecified number of peaks and valleys followed by a flattening out. Some data mining tools provide the ability to search for patterns in time series, e.g., [34,39], but require heavy precomputation, limiting ad-hoc exploration, in addition to suffering from the same limitations in flexibility as the visualization tools. Yet another alternative for domain experts with programming expertise is to write code to perform this flexible match, but writing code for each new use-case, followed by manual optimization is often as tedious as manual searching of visualizations to find patterns.

7.1.1 Characterizing Shape Search Queries

The design of ShapeSearch has been motivated by case studies and use-cases from domains such as genomics, astronomy, battery science, and finance.

We also collected a corpus of about 250 natural language queries via Mechanical Turk (mturk), where we asked crowd workers to describe patterns in trendline visualizations collected from real world datasets. We highlight the key characteristics of pattern matching tasks, based on our discussions with domain experts and analysis of mturk queries below.

Fuzzy Matching. Domain experts typically search for patterns (i) that are *approximate*, and are often not interested in the specific details or local fluctuations as much as the overall shape, and (ii) they often *do not* specify or even know the exact location of the occurrence of patterns. For example, biologists routinely look for structural changes in gene expression, e.g., rising and falling at different times (Figure 7.1a). Such changes characterize internal biological processes such as the cell cycle or circadian rhythms, or external perturbation, such as the influence of a drug or presence of a disease. Similarly, many crowd workers tend to describe trendlines using high level patterns such as *increasing and then decreasing*, without being precise about locations and/or features of the changes.

Combination of Multiple Simple Patterns. We notice that both domain experts as well as crowd workers often describe complex patterns using a *combination of multiple simple ones*. Each individual pattern is typically described using words such as "increasing", "stable", "falling", which are easy to state in natural language but hard to specify using existing query languages. Moreover, pattern matching tasks in many domains often go beyond finding a sequence of patterns, requiring arbitrary combinations, e.g., disjunction, conjunction or quantification, with varying location or width constraints. Examples include finding stocks with at least 2 peaks within a span of 6 months, e.g., the so-called "double/triple top" patterns that indicate future downtrends [76], or finding cities where the temperature rises from November to January and falls during May to July

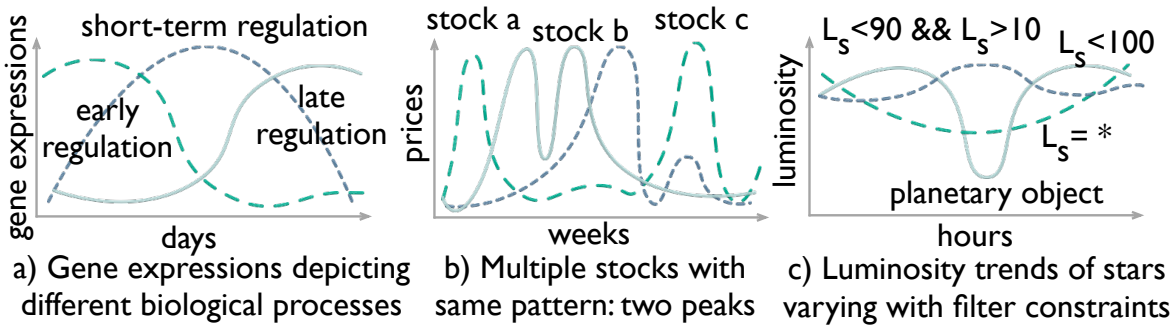


Figure 7.1: Shapes characterizing various real world phenomena

such as Sydney.

Ad hoc and Interactive. Pattern-based queries are often defined *on-the-fly* during analysis, based on other patterns observed. For instance, biologists often search for a pattern in a group of genes similar to a pattern recently discovered in another group [73]. Similarly, astronomers monitor the shape of the luminosity trends of stars over time to search for and characterize new planetary objects (Figure 7.1c). For example, a dip in brightness often indicates a planetary object passing between the star and the telescope. In order to limit comparison of patterns over similar duration (i.e., the X axis) or over value ranges (i.e, the Y axis), it is common to apply constraints while pattern matching. Examples include searching for changes in buying and selling patterns of stock or house prices in a specific range or duration. As such, some tools, e.g., TimeSearcher [75], allow interactive specification of constraints, however the pattern matching is still precise or value-based.

7.2 OVERVIEW OF OUR APPROACH

To satisfy the aforementioned characteristics, ShapeSearch incorporates three novel ideas.

(a) ShapeSearch incorporates an expressive *shape query algebra* that abstracts key shape-based primitives and operators for expressing a variety of desired trendline patterns. The most powerful feature of this algebra is its capability for "fuzzy" matching, allowing approximate and inexact pattern specification, without compromising on the needs of occasional precise queries. We developed this algebra after discussions with domain experts, as well as studying mturk pattern queries, as mentioned earlier.

(b) Unfortunately, naïvely executing these fuzzy queries is extremely slow, requiring an expensive

Mechanism	Intuitiveness	Control	Expressiveness
Natural language	high	low	high
Sketch	high	high	low
Regex	low	high	high

Table 7.1: Comparison between specification mechanisms

evaluation of all possible ways of matching each candidate trendline to the query to select the best one. We propose a dynamic programming-based optimal algorithm that reuses computations to provide substantial speed-ups, and show that even this algorithm can be prohibitively slow for interactive ad-hoc exploration. Thus, we develop a novel perceptually-aware bottom-up algorithm that incrementally prunes the search space based on patterns specified in the query, providing a **40**$\times$ speedup with over 85% accuracy compared to the optimal approach.

(c) Finally, to accommodate a range of needs without sacrificing on the expressiveness of the algebra, ShapeSearch supports three query specification mechanisms (Table 7.1): sketching on a canvas, natural language, and regular expressions (regex for short). All specification mechanisms are translated to the same shape query algebra representation, and can be used interchangeably, as user needs evolve.

7.3 SHAPESEARCH SYSTEM OVERVIEW

ShapeSearch provides powerful yet flexible mechanisms for users to search for trendline visualizations with a desired shape. We first present an overview of the front-end interface and user experience, and then describe the back-end.

Front-end. ShapeSearch supports an interactive interface for composing shape queries, as well as for displaying the result visualizations. Figure 7.2 depicts this interface, with an example query on genomics data discussed in the introduction. Here, the user is interested in searching for genes that get suppressed due to the influence of a drug, depicted by a specific shape in their gene expression — first rising, then going down, and finally rising again — with three patterns: up, down, and up, in a sequence. To search for this shape, the user first loads the dataset [77]), via form-based options on the left (Figure 7.2 Box 1), and then selects the space of visualizations to explore by setting the x axis as time, the y axis as expression values, and the category as gene. Each value of the category attribute results in a candidate visualization with the given x and y axis. Thus, the category attribute defines the space of visualizations over which we match the shape. ShapeSearch supports three mechanisms for shape specification — natural language, regular expressions (regex

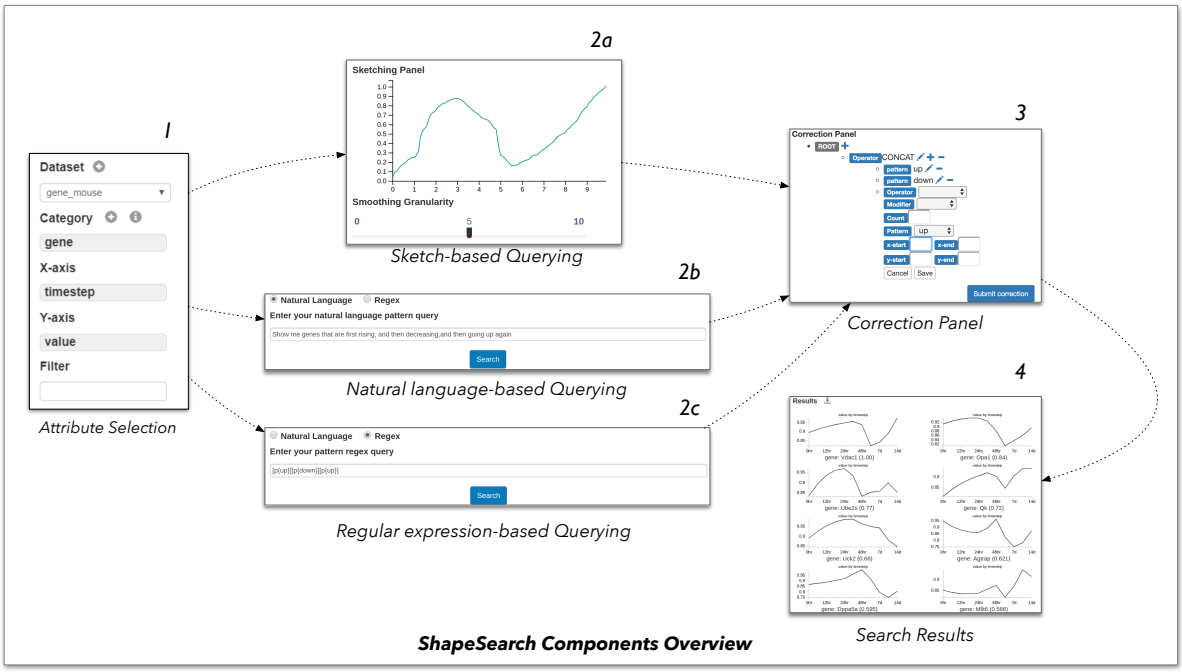


Figure 7.2: ShapeSearch *Interface, consisting of six components. 1) Data upload, attributes selection, and applying filter constraints 2) Query specification: 2a) Sketching canvas 2b) Natural language query interface, and 2c) Regular expression interface, 3) Correction panel, and 4) Results panel*

for short), and sketching on a canvas:

Sketching on Canvas. By drawing the desired shape as a sketch on the canvas (Figure 7.2 Box 2a), the user can search for visualizations that are *precisely* similar (using a distance measure such as Euclidean distance or Dynamic Time Warping [78]). As soon as the user finishes sketching, ShapeSearch outputs visualizations that are precisely similar to the drawn sketch in the results panel (Figure 7.2 Box 4).

Natural Language (NL). For searching for visualizations that approximately match patterns, users can use natural language. For instance, as in Figure 7.2 Box 2b, the desired shape in the aforementioned genomics example can be expressed as *"show me genes that are rising, then going down, and then increasing"*. Similarly, scientists analyzing cosmological data can easily search for supernovae (bright stellar explosions) using *"find me objects with a sharp peak in luminosity"*. We describe in Section 7.7 how ShapeSearch translates natural language queries to a structured internal representation.

Regular Expression (regex). For queries that involve complex combinations of patterns that are difficult to express using natural language or sketch, the user can issue a regular expression-

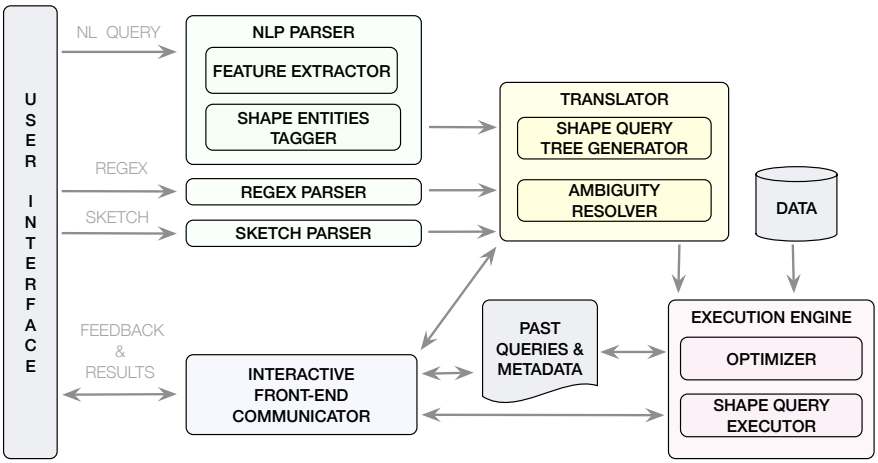


Figure 7.3: System Architecture.

like query that directly maps to the structured internal representation, consisting of ShapeSearch primitives and operations, described in detail in Section 7.4.

During exploration, users can choose specification mechanisms interchangeably based on the complexity of the query. For both NL as well as regex, ShapeSearch, additionally supports auto-complete functionality to guide users towards their target query. All queries are issued to the back-end using a REST protocol. We use the term *user query* to refer to the submitted query using any of the specification mechanisms.

Back-end. Figure 7.3 depicts the back-end components of ShapeSearch. The back-end parses and translates the user query into a ShapeQuery, a structured internal representation of the query consisting of operators and primitives supported in our algebra (Section 7.4). The back-end supports an ambiguity resolver that uses a set of rules for automatically resolving syntactic and semantic ambiguities, as well as forwards the parsed query to the user for further corrections and validation (Figure 7.2 Box 3). The validated query is finally optimized and executed by the execution engine, and the top visualizations that best match the ShapeQuery are presented to the user in the results panel (Figure 7.2 Box 4).

We give an overview of ShapeQuery, a structured query algebra, motivated from use-cases in real domains as well as our analysis of the crowdsourced pattern queries.

7.4 SHAPESEARCH ALGEBRA

We give an overview of ShapeQuery, a structured query algebra, motivated from use-cases in real domains as well as our analysis of the crowdsourced pattern queries.

Symbol	Name	Type
x.s	START X VALUE	Location Sub-Primitive
y.s	START Y VALUE	Location Sub-Primitive
x.e	END X VALUE	Location Sub-Primitive
y.e	END Y VALUE	Location Sub-Primitive
v	SKETCH	Location Sub-Primitive
p	PATTERN	Primitive
$\otimes$	CONCAT	Operator
$\odot$	AND	Operator
$\oplus$	OR	Operator

Table 7.2: Primitives and Operators in ShapeQuery

The ShapeQuery algebra consists of a minimal set of primitives and operators for declaratively expressing a rich variety of patterns, while supporting the three characteristics of pattern-matching tasks described in the introduction. At a high level, a ShapeQuery represents a *shape* as a combination of multiple *simple patterns*. A simple pattern can either be precise with specific location constraints, e.g., matching $y = x$ between $x = 2$ to $x = 6$, or fuzzy, e.g., roughly increasing, where the notion of the pattern is approximate and its location unspecified. Each simple pattern along with its precise or imprecise constraints is called a ShapeSegment. Complex shapes, e.g., rising and then falling, are formed by combining multiple ShapeSegments using one or more *operators*. One can search for multiple patterns in a sequence (concat, $\otimes$) or matching the same sub-region of the trendline (and, $\odot$), or one of many patterns matching a sub-region (or, $\oplus$), described later.

As an example,"rising from x=2 to x=5 and then falling" can be translated into a ShapeQuery [x.s=2,x.e=5, p=up]$\otimes$[p=down] consisting of two ShapeSegments separated by a $\otimes$ operator. The first ShapeSegment captures "rising from $x = 2$ to $x = 5$"; the second expresses a "falling" pattern. Since the second must "follow" the first, the two ShapeSegments are combined using the CONCAT operator, denoted by $\otimes$. We now describe the shape primitives and operators that constitute the ShapeQuery algebra. Table 7.2 lists these primitives and operators.

7.4.1 Shape Primitives and Operators

A ShapeSegment is described using two high level primitives: LOCATION and PATTERN. ShapeSearch allows users to skip one or more of these primitives in their query. The LOCATION values can be skipped in order to match the PATTERN anywhere in the trendline. Similarly, users can input the exact trendline to match, or the endpoints of the ShapeSegments to match without

Pattern	ShapeQuery
Increasing from 2 to 5 and then decreasing	[p=up, x.s=2, x.e=10]⊗[p=down]
Decreasing or increasing anywhere	[p = *]⊗(p=up⊕p=down) ⊗[p = *]
Increasing at 45, decreasing at 60 and then becomes flat	[p = 45]⊗[p = -60]⊗[p = flat]
Decreasing over a width of 3 points:	[x.s=., x.e=.+3, p=down]
Increasing at least once and at most 5	[p=up, q=1,5]
W shaped pattern	[p=-45]⊗[p = 60]⊗[p=-45]⊗[p = 60]
Specific sketch	[v = (2:10,3:14,...,10:100)]
Shape whose trend is increasing relatives its own trend before some point in the past (e.g, inverted bell shaped)	[p=down]⊗[p > $−.p]

Table 7.3: Examples of ShapeQueries

specifying the PATTERN. We describe each of these supported primitives.

Specifying LOCATION. LOCATION defines the endpoints of the sub-region of the trendline between which a pattern is matched: starting X/Y coordinate (x.s/y.s), ending X/Y coordinate (x.e/y.e). For example, [x.s=2,x.e=10, y.s=10,y.e=100] is a simple ShapeQuery to find trendlines whose trend between x=2 to x=10 is similar to the line segment from $(2, 10)$ to $(10, 100)$. Users can also draw a sketch to find trendlines similar to the sketch, a functionality supported in other tools alluded to in the introduction [3, 9, 75]. ShapeSearch translates the pixel values of the user-drawn sketch to the domain values of the X and Y attributes, and adds the transformed vector of (x, y) values as a vector v in the ShapeQuery. As an example, the ShapeQuery [v=(2:10,3:14,...,10:100)] finds trendlines that have precisely similar values to v using a distance measure, e.g., Euclidean distance, or dynamic time warping [78].

Specifying PATTERN. PATTERN defines a trend or a semantic feature in a sub-region of the trendline. A number of basic semantic patterns, commonly used for characterizing trendlines, are supported, such as *up*, *down*, *flat*, or the slope (θ) in degrees. For example [p=up] finds trendlines that are increasing, [p=45] finds trendlines that are increasing with a slope of about 45°, and [x.s=2,x.e=10, p=up] finds trendlines that are increasing from $x = 2$ to 10. Finally, one can use p=* to match any pattern and p=empty to ensure that there are no points over the sub-region.

Combining PATTERN**.** ShapeQuery supports three operators to combine ShapeSegments:

- CONCAT (⊗) specifies a sequence of two or more ShapeSegments. For example, using [p=up]⊗[p=down] one can search for genes that are first rising, and then falling. Note that ⊗ is one of the most frequently used operations, and we sometimes omit ⊗ between ShapeSegments, e.g, [p=up][p =down], to make it succinct to describe.

- AND (⊙) simultaneously matches multiple patterns in the same sub-region of the trendline. Unlike CONCAT, all of the patterns must be present in the same sub-region. For example, one can look for genes whose expression values rise twice but do not fall more than once within the same sub-region.

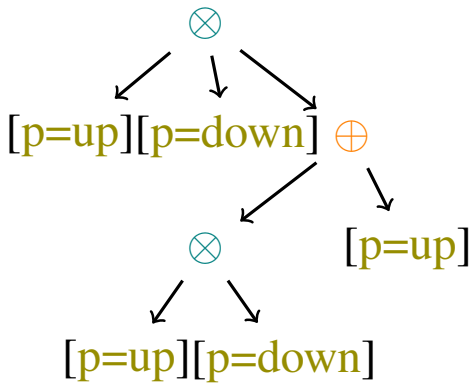


Figure 7.4: ShapeQuery *AST*

- OR (⊕) searches for one among many patterns in the same sub-region of the trendline, picking the one that matches the sub-region best. For example, one can search for genes whose expressions are either *up*- or *down*-regulated.

Note that when the same operator is specified consecutively, ShapeSearch fuses them into one, hence all operators can take two or more operands. For example, [p=up] ⊗[p=down] ⊗[p=down] is parsed as a single ⊗ operation with three operands [p=up], [p=down], and [p=down].

Multiple operations are often used in a given ShapeQuery. ShapeSearch follows left to right precedence order for execution of the operations. However, sub-expressions can be nested using parentheses () to specify precedence as in mathematical expressions. In Figure 7.4, we depict how ShapeSearch parses a complex ShapeQuery [p=up]⊗[p=down] ⊗(([p=up]⊗[p=down])⊕[p=flat]) into an Abstract Syntax Tree (AST) representation.

Comparing Patterns. In some cases, one may want to compare the pattern in a ShapeSegment with the preceding or succeeding ShapeSegments. To support such use cases, ShapeSearch (i) allows a ShapeSegment to refer to the previous or the next ShapeSegment using $+ or $− respectively, and (ii) compare patterns between the current and referred ShapeSegment using operations $>$, $<$, or $=$. For example, astronomers can issue a ShapeQuery [p=up]⊗[p < $−.p] with x=time and y=luminosity (brightness) to search for celestial objects that were initially moving rapidly towards earth, but after some point either slowed down or started moving away. The second ShapeSegment [p < $−.p] ensures that the slope of brightness over time is less than that in the previous sub-region [p=up].

Similarly, one can set p $<\frac{1}{2}$$−.p to ensure the slope of second sub-region is $\leq \frac{1}{2}$ of the first. To avoid ambiguity in position reference and for efficient execution, ShapeSearch restricts $-based references to a simple CONCAT operation, i.e., across a sequence of patterns at the same level of nesting.

Expressing complex patterns. The aforementioned basic primitives and operators are powerful enough to express more complex ShapeSearch use-cases. We discuss three such complex patterns below, along with shortcuts for their easy specification.

1. Searching shapes of specific width. In some cases, users want to find specific shapes irrespective of their start location. For example, one may want to search for cities with maximum rise in temperature over a width of 3 months. To express such queries, ShapeSearch supports the ITERATOR (.), e.g., [x.s=.,x.e=x.s+3,p=up] that iterates over all points in the trendline, setting each point as the start x position, with the x end position set to 3 units ahead. Internally, for a trendline of length n, this query can be rewritten as an OR operation over $(n-3+1)$ ShapeSegments, where, for the ith ShapeSegment, x.s=i and x.e=i+3.

2. Searching for sub-patterns. One can easily perform sub-pattern matching using [p=*]. For example, [p=*][p=up][p= down] [p=*] searches for a peak anywhere in the trendline.

3. Constraining patterns. In some cases, users want to search for patterns but with multiple constraints on X and Y axes. For example, financial analysts can use the ShapeQuery [p=up, x.s=1, x.e=6,y.s=100,y.e=200]⊗[p=down, x.s =7,x.e=12,y.s=200,y.e=100] to search for stock whose values increased between 100 and 200 over the first 6 months, and then decreased between 200 to 100 over the remaining 6 months.

4. Quantifiers. One can search for trendlines where a pattern occurs a specific number of times using quantifiers, denoted by q. For example, [p=up,q={1,2}] can be used to search for trendlines where there is an increasing pattern at least once and at most twice. Quantifiers can be internally rewritten using an OR of one or more CONCAT operations. For example, the above query is rewritten as ([p=*] ⊗[p=up]⊗[p=*])⊕([p=*]⊗[p=up]⊗[p=*] ⊗[p=up]⊗[p=*]).

5. Nesting A combination of patterns can be constrained to be within a specific sub-region by specifying them as a value of the PATTERN primitive. For example, to search for stocks that increased anytime between February to October, we can use nesting as follows: [x.s=2,x.e=10, p=([p=*][p=up][p=*])]. This can be rewritten using CONCAT operations as follows: [x.s=2, p=*]⊗[p=up]⊗[p=*]⊗[x.s=10,p=*].

6. Complex Shapes. Complex shapes can be searched by breaking them into multiple ShapeSegments, where each ShapeSegment specifies a pattern with a specific slope. For instance, a "W-shaped" pattern can be searched using [p=-60] ⊗[p=60]⊗[p=-60]⊗[p=60]. Similarly, in the following query, by setting different values for a, b and c, one can search for peaks of varying sharpness: [p=*][p=up,x.s=.,x.e=x.s+a] [p=flat,x.s=$-.x.e,x.e=x.s+b][p=flat,x.s=$-.x.e,x.e=x.s+c] [p=*].

7. Specifying both precise and fuzzy patterns. As mentioned earlier, one can draw a sketch of a shape and embed it within a ShapeSegment using the sketch (v) LOCATION primitive. Thus, by combining such ShapeSegments with those that involve semantic patterns such as p=up, e.g.,

[p=up]⊗[v=(10:10, 11:14, ...,20:100)]⊗[p=down], we issue a ShapeSearch query that involves both precise and fuzzy pattern matching.

8. Scale invariant matching. One can automatically search for shapes at varying granularity of x-scales and degree of smoothing. For example, for [p=∗]⊗[p=up] ⊗[p=down]⊗[p=∗], ShapeSearch searches for [p=up] and then [p=down] pattern at all possible scales and selects the one that leads to the best match. To do so, ShapeSearch uses efficient algorithms that we describe in the subsequent sections.

As ShapeSearch evolves, it may support additional shortcuts to simplify the writing of frequently used complex patterns.However, all of the complex patterns as well as the shortcuts can be expressed using basic primitives and operations for their execution. Thus, we omit further discussion of complex patterns and limit ourselves to the semantics and efficient scoring of basic primitives and operators.

7.5 FORMAL SEMANTICS OF SHAPEQUERY

We now formally define the semantics of ShapeQuery. Given three dataset attributes x, y, and z, ShapeSearch first generates a collection of trendlines V, one for each unique value of the z attribute. Each trendline is a sequence of (x, y) values ordered by x. A ShapeQuery Q operates on one trendline, V_i, at a time, and returns a real number, called *score*, between -1 to $+1$, i.e., $Q : V_i \rightarrow score$; $score \in [-1,1]$. The value of *score* describes how closely V_i matches Q, with $+1$ the best possible match, and -1 the worst.

The ShapeQuery Q operates on V_i with the help of ShapeSegments $(S_1, S_2, \ldots, S_n)$ and operators $(O_1, O_2, \ldots, O_m)$. Each ShapeSegment, S_i operates on $V_i^{p,q}$, a sub-region of V_i starting at $p = x.s$ and ending at $q = x.e$ and returns a $score_i \in [-1,1]$ using scoring functions we describe subsequently. A common subclass of ShapeQueries are *fuzzy* ShapeQueries. A fuzzy ShapeQuery is a sequence of ShapeSegments where there is at least one ShapeSegment with missing or multiple possible values for $x.s$ or $x.e$. Thus, for fuzzy ShapeQueries, we try all possible values of p and q, selecting the sub-region that leads to the best score. One or more ShapeSegments are combined using operators such as ⊗, ⊙, ⊕. Formally, an operator O_i takes as input the scores $score_1, score_2, \ldots, score_n$ from its n input ShapeSegments and outputs another $score_i$ using scoring functions that capture the behavior of the operators. When combined via AND or OR operators, ShapeSegments may operate on overlapping sub-regions $V_i^{p,q}$, however, for CONCAT, the sub-regions must not overlap since CONCAT specifies a sequence of patterns. Next, we describe our scoring methodology.

P	Score
up	$\frac{2 \cdot tan^{-1}(slope)}{\pi}$
down	$-\frac{2 \cdot tan^{-1}(slope)}{\pi}$
flat	$(1.0 - \|\frac{4 \cdot tan^{-1}(slope)}{\pi}\|)$
$\theta = x$	$(1.0 - \|\frac{2 \cdot tan^{-1}(slope - x))}{(\pi - \|tan^{-1}(x\|\|)}\|)$
$*$	1
empty	-1
v	L_2 norm (configurable)

Table 7.4: Pattern Scores

O	Score
$\otimes$	$\sum_{i=1}^{k} score_i / k$
$\odot$	$min(score_1, ..., score_k)$
$\oplus$	$max(score_1, ..., score_k)$

Table 7.5: Operator Scores

7.6 SCORING METHODOLOGY

For supporting interactive response times, ShapeSearch needs to *efficiently* and *effectively* compute the match between a ShapeQuery Q and a trendline V_i.

To satisfy both efficiency and effectiveness, ShapeSearch approximates each sub-region with a line, using the slope to quantify how closely it captures any given ShapeSegment. The line-based quantification is robust to noise or minor fluctuations, as is often intended in ShapeQueries. At the same time, lines are extremely fast to compute, requiring only a single pass on the data. As we explain shortly, lines over larger sub-regions can be quickly inferred from lines over over smaller ones, without additional passes. As the complexity of a pattern increases, the number of lines required to approximate it also increases. However, even for complex shapes, a small number of line segments is sufficient. Our study of patterns (e.g., double-bottom, triple-top) in finance [**?**] as well as mturk queries reveal that the maximum number of lines is usually small or less than 6.

As depicted in Table 7.4, ShapeSearch uses different scoring functions for each pattern primitive that transforms the slope to a value in $[-1,1]$ using a tan^{-1} function. For example, for an *up* pattern, the function returns a score between $[0,1]$ for all trendlines with slope from 0° to 90°, a score of $[-1,0]$ for slopes $< 0^\circ$ (opposite of *up*). Moreover, a change in trend from 10° to 30° is visually more noticeable than from 60° to 80°, thus we capture this behavior using tan^{-1} where the rate of increase in output decreases as the value of slope increases. Finally, we apply normalization,

Algorithm 7.1 ShapeQuery Scoring

Input: L: a sub-region of trendline, Q: a ShapeQuery sub-expression, $ScrFunc$: scoring functions from Tables 7.4 and 7.5
Output: score

```
1:  procedure ExecShapeQuery(L, Q, ScrFunc)
2:      if Q.root is a ShapeSegment then
3:          hasValidLoc <- CheckLocationConstraints(L, Q)
4:          hasValidLineFit <- CheckGoodnessofFit(L)
5:          if (hasValidLoc && hasValidLineFit) is False then
6:              return -1;
7:          end if
8:          return ScrFunc(L, operator, Q.root)
9:      end if
10:     operator ← Q.root.operator
11:     operands ← operator.children
12:     k = operands.size
13:     operandscores ← []
14:     if operator ∈ {⊙,⊕} then
15:         for each child in operands do
16:             operandscores.append(ExecShapeQuery(L,child))
17:         end for
18:         return ScrFunc(operator, operandscores)
19:     end if
20:     if operator ∈ {⊗} then
21:         candscores = []
22:         for each segmentation {L_1, L_2, ..., L_k} of L do
23:             sgtscores ← []
24:             for each child in operands do
25:                 sgtscores.append(ExecShapeQuery(L_i,child))
26:             end for
27:             sgtscores← ScoreComparators(sgtscores,Q.root)
28:             candscores.append(ScrFunc(operator,sgtscores))
29:         end for
30:         return max(candscores)
31:     end if
32: end procedure
```

such as multiplying by $2/\pi$, to re-scale the output of tan^{-1} between -1 and 1. Thus, depending on the specified pattern primitive, ShapeSearch uses the corresponding scoring function to compute the score for that ShapeSegment. For a given ShapeSegment, if the location constraints are not met, we assign a score of -1, and ignore the rest of the primitives.

We state the following observation regarding the scoring of a single ShapeSegment.

Observation 7.1. *The scoring of a* ShapeSegment*, as part of a* ShapeQuery *Q without comparisons, on a sub-region L can be done using the slope of the corresponding single line segment and the x.s, x.e, y.s, and y.e values of the sub-region, independent of other sub-regions.*

For a shape input as a sketch, users sometimes intend to perform precise matching. For such ShapeSegments we compute the score using L2 norm (Euclidean distance) between the drawn sketch and the trendline without fitting a line segment. The L2 norm can vary from 0 to ∞; therefore, we normalize the distance within [1, -1] using using Max-Min normalization. In addition, ShapeSearch allows users to use sketch for fuzzy matching where ShapeSearch fits a minimum number of lines to the sketch given an error threshold (adjustable via a slider), and automatically constructs a CONCAT operation of ShapeSegments, with one ShapeSegment for each line with the pattern corresponding the slope of the line.

In addition to providing domain-specific pattern types (UDPs), ShapeSearch also allows users to override the default scoring methodology by letting them define their own scoring functions. For seamless integration, user-defined scoring functions must take a sun-region as input, and output a score within $[-1, 1]$.

For two contiguous ShapeSegments compared using $-references, ShapeSearch returns a single score as if they were one single ShapeSegment evaluated over their combined sub-region. Internally, ShapeSearch evaluates each of ShapeSegments over their corresponding sub-region independently and combines scores across the CONCAT appropriately. The score of the ShapeSegment that uses that $ reference is set to $+1$ if the constraint is satisfied, otherwise it is set to -1. For ease of explanation, we refer them as a single ShapeSegment for the rest of the paper.

The scores across ShapeSegments are combined using the scoring functions for the operations. Note that, in general, as depicted in Figure 7.4, the operands of an operator can be sub-expressions involving other operators. Nevertheless, as depicted in Table 7.5, the scoring functions for operators are more straightforward as they directly capture the semantic behavior of the operators. For instance, CONCAT matches a sequence of patterns, therefore, the scoring function takes average of the scores of its operands to give equal weightage to each operand. AND matches multiple patterns over the same sub-region, so to avoid any ShapeSegment not having a good match, we take the minimum of all scores across its operands. On the other hand, OR picks the best among all matches, so it takes the maximum across all scores. From these definitions, we state the following

observations:

Observation 7.2. *The scoring of AND or OR operations with k operands on a sub-region L can be done by scoring each of the k operands independently on the sub-region L.*

Observation 7.3. *The scoring of* CONCAT *with k operands on sub-region L can be done by dividing sub-region L into all possible sequences of k sub-regions, followed by scoring operand i on sub-region i.*

Note that the scoring of an operand can be done independently of others. We used the term *segmentation* to refer to a division of a sub-region into L sub-regions.

Ensuring goodness of fit. It is possible that a line poorly approximates a given segment of the trendline. Therefore, we use a configurable (via a slider) threshold parameter to suggest how much error they can tolerate. For measuring the goodness of fit, we compute the standard R^2 error [79], also called coefficient of determination, of the line, between 0 to 1, with higher values indicating lower errors and better fit. ShapeSearch gives a score of -1 to a ShapeSegment for a given sub-region if R^2 is less than the threshold.

Overall algorithm. Algorithm 7.1 outlines the steps for scoring a ShapeQuery. At the start, the algorithm takes the entire trendline V_i as L, the Abstract Tree Representation (AST) of ShapeQuery as Q, and the list of scoring functions $ScrFunc$ as in Tables 7.4 and 7.5 as inputs. If the root node of the ShapeQuery tree is a ShapeSegment, ShapeSearch directly computes the score of the ShapeSegment on the sub-region using scoring functions after checking the location and goodness of fit constraints (lines 2-9). If the root node is $\odot$ or $\oplus$, ShapeSearch invokes each of the operands (i.e., child sub-trees) to compute their scores on the sub-region independently, combining the scores as per their scoring functions (lines 14-18). However, if the root node is a CONCAT with k operands, i.e., child sub-trees, ShapeSearch segments L into all possible k sub-regions: $L_1, L_2, ..., L_k$, and then for each segmentation, invokes the ith operand on ith segment (lines 20-30). Finally, the maximum score across all segmentations is output.

7.7 PARSING AND TRANSLATION

We now provide an overview of the key steps involved in parsing. We use the following natural language query collected from MTurk for illustration: "show me the trendlines that are increasing from 2 to 5 and then decreasing".

Step 1. Primitives and Operators Recognition. Given a natural language query, the first step is to map words to their corresponding shape primitives and operators. We follow a two-step process. First, using the Part-of-Speech (POS) tags and word-level features, we classify each word in the

Type	Features
POS Tags	pos-tag,pos-tag-,pos-tag+
Words	word-, word+, word–,word++
Predicted entities	predicted-entity,predicted-entity+,predicted-entity-, d(predicted-entity+),d(predicted-entity-)
Space and time prepositions	time-preposition+,time-preposition-,space-preposition+,space-preposition-,d(time-preposition+), d(time-preposition-),d(space-preposition+),d(space-preposition-)
Punctuation	d(,+),d(,-),d(;+),d(;-),d(.+),d(.-)
Conjunctions	d(and+),d(or-),d(and then+)
Miscellaneous	d(x),d(y),d(next),ends(ing),ends(ly), length(query)

Table 7.6: Features used during natural language parsing (d(x) denotes distance in terms of number of words between current word and x), x+ denotes next x, x- denotes previous x.

query as either noise or non-noise. For example, words $\in$ {determiner, preposition stop-words} are more likely to be noise, while words $\in$ {noun, adjective, adverb, number, transition words, conjunction} may refer to a primitive or operator. Next, given a sequence of non-noise words, we use a linear-chain conditional-random field model (CRF) [80] (a probabilistic graphical model used for modeling sequential data, e.g., POS tagging) to predict their corresponding primitives and operator. For example, the above query is tagged as "show (noise) me (noise) the (noise) trendlines (noise) that (noise) are (noise) increasing (p) from (noise) 2 (x.s) to (noise) 5 (x.e) and then ($\otimes$) decreasing (p)"..

We train the CRF model [80] on the same 250 natural language queries that we used for characterizing trendline patterns. We extract a set of features (listed in Table 7.6) for each non-noise word in the sequence. In addition, ShapeSearch stores "synonyms" for each primitve and operator (e.g., "increasing" for up, "next" for CONCAT), and if a non-noise words closely matches with them (e.g., with edit distance $<= 2$), we add the matched primitive or operator as a feature called *predicted-entity*. This idea is inspired from the concept of "boostrapping" in weakly-supervised learning approaches [81, 82], and helps improve the overall accuracy. We implemented the model using the Python CRF-Suite library [83] with parameter settings: *L1 penalty:1.0, L2 penalty:0.001, max iterations: 50, feature.possible-transitions: True*. On 5-fold cross-validation over the crowd-sourced queries, the model had an $F1$ score of 81% (*precison* = 73%, *recall* = 90%).

Step 2. Identifying Pattern Value. For each of the words predicted of type p, e.g., increasing and decreasing in the above query, we additionally map them to the corresponding semantic pattern supported in ShapeSearch, e.g., "increasing" is mapped to p=up. For this mapping, ShapeSearch

Ambiguity (example queries with predicted entitites)	**Resolution**
A1: Conflicting LOCATION and PATTERN in a ShapeSegment (e.g., [decreasing (p) from 4 ($x.s$) to 8 ($x.e$)])	R1: Change the sub-primitive of LOCATION from x to y or y to x. R2: Swap the start and end positions of LOCATION.
A2: Multiple p in the same ShapeSegment (e.g., [increasing (p) from 2 ($x.s$) to 5 ($x.e$) with decreasing (p)] next ($\otimes$))	R1: Move one of the ps to the adjacent ShapeSegment with missing p. R2: split the ShapeSegment into two new ShapeSegments with an OR operator between them
A3: Overlapping ShapeSegments with $\otimes$(e.g., increasing (p) from 4 ($x.s$) to 8 ($x.e$) and then ($\otimes$) decreasing (p) from 8 ($x.s$) to 0 ($x.e$)	R1: Change x to y, if y values missing. If y values already present, replace $\otimes$ with $\odot$ operator.

Table 7.7: Common Ambiguities and their Resolution

computes the similarity between the specified word and synonyms of the supported patterns, first using edit distance and then using wordnet [84]. The semantic pattern with the highest similarity between any of its synonyms and the specified word is selected.

Step 3. ShapeQuery Generation and Ambiguity Resolution. Next, we group primitives and operators into a ShapeQuery. ShapeSearch first groups all the primitives between two operators into a single ShapeSegment. For instance, for the above query, the primitives are grouped as follows: [increasing (p=up), 2 (x.s), 5 (x.e)] and then ($\otimes$) [decreasing (p=down)]. In some cases, this may lead to incorrect grouping of primitives, e.g., two patterns in the same ShapeSegment. Moreover, there could be semantic ambiguity because of wrong entity tagging, e.g., decreasing (p=up) from 5 (y.s) to 10 (y.e) where x.s and x.e values are wrongly tagged as y.s and y.e respectively. ShapeSearch uses rule-based transformations that try to reorder and change the types of entities to get a correct and meaningful ShapeQuery. In Table 7.7, we list three common ambiguities (A1, A2, A3) and a sequence of rules (e.g., R1, R2) that are applied in order to resolve these.

Step 4. Parsed ShapeQuery Validation. The parsed ShapeQuery is sent to the front-end, and displayed as part of the correction panel (Box 4 in Figure 7.2) for users to correct, further refine or edit the query if needed. The correction panel is a form-based representation of the ShapeQuery, where users can add, delete, or modify ShapeQuery primitives and operators with the help of drop-down menu options, or filling in text-boxes. While the ShapeQuery is validated by the user, the system executes the parsed query in parallel, and resulting visualizations are displayed in the result panel. If the users choose to modify the ShapeQuery, the validated query is executed again and results are updated.

7.8 ADDITIONAL RELATED WORK

In Chapter 2, we already discussed a number of prior work in visual analytics and data mining that our work draws on. Here, we discuss the additional work that are related to our techniques we introduce in this chapter.

In Table 7.8, we compare ShapeSearch capabilities and expressiveness with three representative systems from these areas: (1) Zenvisage, our general purpose visual querying tool that we discussed from Chapters 3–5, (2) Qetch [74], a recent sketch-based system, and (3) Shape Definition Language (SDL), a symbolic pattern searching language for trendlines.

At a high-level, ShapeSearch builds on the system capabilities of visual querying systems as well as expressiveness of symbolic pattern languages, while extending both to suit the needs of real domain users. Our user study in Chapter 9 compares ShapeSearch with (1) and (2) in terms of usability and effectiveness. We summarize key differences with these systems and others below.

Visual querying tools [3, 8–10, 85] help search for visualizations with a desired shape by taking as input a sketch of that shape. Most of these tools perform precise point-wise matching using measures such as Euclidean distance or DTW. A few tools such as TimeSearcher [75] let users apply soft or hard constraints on the x and y range values via boxes or query envelopes, but do not support mechanisms for specifying shape primitives beyond location constraints. Qetch improves upon these systems by supporting a custom similarity metric that is robust to distortions in the user sketch, in addition to supporting a "repeat" operator for finding recurring patterns. However, eeee Qetch and other visual querying tools have limited expressiveness when it comes to fuzzy pattern match needs. Furthermore, ShapeSearch introduces a novel algebra that improves extensibility by acting as a common "substrate" for various input mechanisms, along with an optimization engine that efficiently matches patterns against a large collection of trendlines.

Symbolic sequence matching papers approach the problem of pattern matching by employing offline computation to chunk trendlines into fixed length blocks, encoding each block with a symbol that describes the pattern in that block [31–34, 39]. The most relevant one of these papers is on the Shape Definition Language (SDL) [34], which encodes each block using "up", "down", and "flat" patterns, much like ShapeSearch, and supports a language for searching for patterns based on their sequence or the number of occurrences. Since SDL operates on pre-chunked-and-labeled trendlines, the problem is one of matching regular expressions against string sequences (one per pre-labeled trendline). Therefore, SDL cannot rank these trendlines, instead only returning a boolean score for whether the pattern matches the string sequence. This limits the expressiveness of SDL (Table 7.8), especially when the patterns are more complex, as well as when they don't align perfectly well with the boundaries of the blocks used for chunking. Moreover, since the trendlines are pre-labeled and indexed, SDL does not support on-the-fly pattern matching where

Aspect	Zenvisage	Qetch	SDL	ShapeSearch
System Capabilities				
Precise Pattern	✓✓	✓✓	✗	✓✓
Fuzzy Pattern	✗	✓	✓✓	✓✓
Specification	sketch	sketch	regex	sketch, NL, Regex
Auto Smoothing	✗	✓✓	✓	✓
Algorithm	ED, DTW	Custom	Custom	Custom
Ad hoc Patterns	✓	✓	✗	✓✓
Normalization	✓	✓✓	✗	✓(z-score)
Indexing Needed	✓	✓	✗	✓
Scalability	✓✓	✓	✓	✓✓
Extensibility	✗	✗	✗	✓✓
Query Expressivity				
Range Constraints	✓	✓	✗	✓✓
Sub-Pattern Matching	✓	✓	✓	✓✓
Sequence Matching	✗	✗	✓✓	✓✓
Width Selection	✗	✗	✗	✓✓
Multi- X or Y Constraints	✗	✗	✗	✓
Quantifiers	✗	✓(repeat)	✓✓	✓✓
Iteration	✗	✗	✗	✓
Nesting	✗	✗	✗	✓
Back/Forward Reference	✗	✗	✗	✓

Table 7.8: ShapeSearch *vs. related systems capabilities*

the same trendline can change shapes based on filters or aggregation constraints. ShapeSearch, on the other hand, adopts a more online query-aware ranking of trendlines without requiring precomputation, and is thus more suited for ad-hoc data exploration scenarios.

There is also a large body of work on keyword- and natural language-based interfaces for querying databases [86] and generating visualizations [87, 88]. However, since the underlying shape query algebra in ShapeSearch is different from SQL, parsing and translation strategies from existing work cannot be easily adapted. Recently, conversation-based systems such as AVA [89] have been proposed. However, these systems provide a high level framework for facilitating and automating the insight search, but consider pattern matching tasks over individual visualization as black-boxes.

7.9 OVERALL TAKEAWAYS

In this chapter, we motivated the need for flexible mechanisms for supporting fuzzy and ad hoc pattern matching needs, and proposed ShapeSearch towards this end. ShapeSearch supports a minimal set of primitives that allows users to compose rich variety of shape queries over trendlines,

and exposes multiple flexible mechanisms to let users easily specify their patterns of interest.

In the next chapter, we discuss how ShapeSearch optimizes the execution of fuzzy ShapeQueries, an important subclass of queries for which ShapeSearch needs to automatically and efficiently find the endpoints of one or more patterns in the trendline.

CHAPTER 8: FUZZY SHAPE MATCHING

The most interesting and powerful feature of ShapeSearch is its capability for "fuzzy" matching, allowing users to search for patterns without specifying exact locations, e.g., increasing followed by decreasing. Recall that a a *fuzzy* ShapeQuery *is one with atleast one* ShapeSegment *with multiple possible values for x.s or x.e*.

In the absence of exact location values for a CONCAT, ShapeSearch has to exhaustively score all possible segmentations to find the one with the best *score* (line 22-29 in Algorithm 1). This becomes prohibitively expensive as the number of points in the trendline increases. For example, a simple fuzzy ShapeQuery [p=up]⊗[p=down]⊗[p=up] on a trendline with 100 points can result in 10^4 possible segmentations for finding three segments that lead to the best score. More generally, for a CONCAT operator with k operands, the exhaustive approach creates $n^{(k-1)}$ segmentations, where n is the number of points in the trendline. We state this problem formally:

Problem 8.1 (Fuzzy CONCAT Scoring)**.** *Given a* CONCAT *operation with k operands and a subregion L of the trendline with n points, find the segmentation with k subregions where the score of* CONCAT *is maximum.*

8.1 THE DYNAMIC PROGRAMMING ALGORITHM

We first show that we can substantially reduce the number of segmentations for a a CONCAT operation operating on a sequence of ShapeSegments by reusing the scores from CONCAT operations over sub-sequences of ShapeSegments. We, next, show that this extends to the case when one or more operands of CONCAT are AND or OR expressions, but none of the operands internally involve nested CONCATs. Finally, we show how we can reuse computations when an AND or OR operand internally has a nested CONCAT or when an operand is a nested CONCAT. We start with the simplest case of a CONCAT on a sequence of ShapeSegments.

From Observation 2.3, it can be seen that for the CONCAT operation itself, the scoring of the jth operand on jth sub-region does not depend on the scoring of first $j-1$ operands on the first $j-1$ sub-regions. Thus, we can find the optimal segmentation of first $j-1$ operands over all smaller sub-regions and then combine them with the scores of jth operand on the remaining part of the sub-region to find the optimal segmentation.

Suppose the optimal segmentation of [p=up] ⊗[p=down] ⊗[p=flat] over sub-region $x = 1$ to 100 is when [p=up] is scored over the sub-region $x = 1$ to 45, [p=down] over $x = 46$ to 60, and [p=flat] over $x = 61$ to 100. Then, for another CONCAT operation involving a sub-sequence [p=up]⊗[p=down] over the sub-region $x = 1$ to 60, the optimal segmentation should have the

same sub-regions for [p=up] and [p=down] as in the previous CONCAT. This is because of the scoring of [p=flat] from $x = 61$ to 100 does not affect the scoring of [p=up]$\otimes$[p=down] over $x = 1$ to 60.

We use this idea to develop a faster dynamic programming algorithm (DP) for scoring CONCAT operations over ShapeSegments. Formally, let $OPT(1,t,(1:j-1))$ be the best score corresponding to the optimal segmentation over the sub-region between $x = 1$ to $x = t$ for first $j-1$ operands, and $SC(t+1,i,j)$ be the score of jth operand over the sub-region between $x = t+1$ and $x = i$. Then, the optimal segmentation $OPT(1,i,(1:j))$ for first j operands over $x = 1$ and $x = i$ can be computed using the following recursion:

$OPT(1,i,(1,j)) = \underset{t}{MAX}\{\frac{(j-1)\times OPT(1,t,(1:j-1))+SC(t+1,i,j))}{j}\}$

As base cases, we set $OPT(m,m+1,(j:j)) = SC(m,m+1,j)$.

Using the above recurrence, we develop a DP algorithm that reuses the intermediate results by memoizing the results of $OPT(1,i,(1,j))$ in a $2D$ array of size $n \times k$ and $SC(t+1,i,j)$ in $3D$ array of size $n \times n \times k$. It is to easy see that the DP algorithm considers $O(n^2k)$ segmentations to find the optimal score.

Theorem 8.1. *Finding the best segmentation for a* CONCAT *operation on* ShapeSegments *arguments can be done in* $O(n^2k)$ *using Dynamic Programming.*

AND/OR operands with no nested CONCAT**.** Let the ith operand of the CONCAT at the root be an AND/OR expression with no nested CONCAT. From Observation 3.2, and lines 17-22 in Algorithm 1, we can score operand i on sub-region i without any segmentation. Thus, the above theorem is also valid when an operand in the CONCAT operation is an AND/OR expression with no CONCAT operations internally.

AND/OR operand with a nested CONCAT **or directly nested CONCATs.** If the ith operand of the CONCAT at the root consists of a nested CONCAT (either under an AND or OR expression or directly), the ith sub-region needs to undergo further segmentation to find the optimal score for the nested CONCAT. For example, for scoring the ShapeQuery in Figure 7.4, the DP algorithm is first invoked for the CONCAT at the root node. The third operand for the root CONCAT is an OR which consists of another CONCAT and [p=flat] as its operands. Therefore, for every candidate sub-region for the third operand, another DP algorithm is invoked for the nested CONCAT. However, this is not a problem since we can reuse the score of ShapeSegments across invocations. For example, in Figure 7.4, CONCAT operations essentially involve scoring of ShapeSegments [p=down], [p=up] over all possible sub-regions. We, thus, score each ShapeSegment only once for a given sub-region, and reuse it across multiple invocations of the DP algorithm for each CONCAT. Moreover, the DP recursion involves $SC(t+1,i,j)$ for computing the cost of the operand (i.e.,

sub-expression) j from sub-region $t+1$ to i, which again can be shared across repeated invocations of same t, i, j.

Unfortunately, even though the DP algorithm is orders of magnitude faster than the exhaustive approach, we note that for trendlines with large number of points, even a ShapeQuery with a single CONCAT operation can be slow, because of its quadratic runtime. As we will see in Section 8.4, the DP algorithm takes 10s of seconds even for ShapeQueries with 3 or 4 ShapeSegments over trendlines with a few hundreds of points. We, next, discuss optimizations to further decrease the runtime of CONCAT operation on ShapeSegments.

8.2 A PATTERN-AWARE BOTTOM-UP APPROACH

The DP-based optimal approach scores all possible sub-regions for each operand in the CONCAT operation. For instance, consider a fuzzy ShapeQuery [p=up]⊗[p=down]⊗[p=flat] and a trendline L of 120 points. Here, for operand [p=up], the DP approach scores sub-regions of all possible sizes, starting from the smallest possible sub-region (x.s=1 to x.e=2) to (x.s=1 to x.e=116). Note that a sub-region requires at least 2 points to fit a line.

Greedy approach. Two sub-regions that differ only in a few points tend to have similar scores. For instance, the scores of [p=up] over sub-regions (x.s=1 to x.e=15) and (x.s=1 to x.e=16) are likely to be similar. Therefore, an optimization over DP is to consider only those sub-regions for each ShapeSegment that differ substantially in their sizes. For example, a greedy approach could be to start with sub-regions of equal size for each of the three ShapeSegments (i.e., 40 points each), and then greedily vary their sizes until we reach the maximum. One way of varying their sizes is to greedily extend one sub-region at a time, and proportionally shrink the others. For example, the next three configurations after starting with equal sizes could be: (60,30,30), (30,60,30), (30,30,60). We pick the best of these and then repeat the process. Clearly, this approach scores much fewer segmentations ($O(log(n^k))$), compared to $O(n^2)$ segmentations explored by the DP approach. However, as we show in our experiments (Section 8.4), such an approach leads to extremely poor accuracy.

Pattern-aware segmentation. The problem with the greedy approach is that it treats all points equally, and as possible candidates for endpoints of ShapeSegments. A better approach could be to select end points to be those where the slope (or pattern) changes drastically. We first illustrate our intuition, and then describe an algorithm that performs segmentation in a pattern-aware manner.

Intuition. As depicted in Figure 8.1, consider two sub-regions A on the left and B on the right for the trendline L. Say the trendline in sub-region A is inverted V-shaped, i.e., increasing until a point P and then decreasing. Now, for all possible segmentations where [p=up]'s sub-region

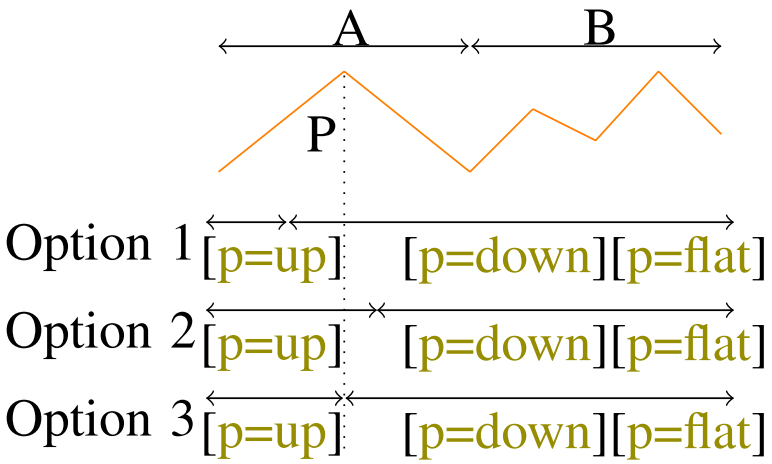


Figure 8.1: Pattern-aware selection of LOPs

lies completely in A, there are following possibilities for x.e of [p=up]: 1) [p=up]'s x.e point is before P. 2) [p=up]'s x.e point is after P. 3) [p=up]'s x.e point is at P.

Since [p=down] follows [p=up], we can see that option 1 that sets [p=up]'s x.e $< P$ is less likely to be optimal as that will lead to scoring of a part of [p=down] on an increasing trend. Similarly, x.e $> P$ is less optimal as that will lead to scoring of a part of [p=up] on a decreasing trend. Thus, if we have to (greedily) select one point in sub-region A for [p=up]'s x.e, P is likely a better choice. We call such a point as *locally optimal point* (LOP).

A Bottom-up algorithm. Based on the above intuition, we develop a much faster algorithm that uses the following assumption to reduce the number of segmentations.

Assumption 8.1 (Closure)**.** *If a point is not locally optimal for any of the sub-expressions in the* CONCAT *operation (i.e., a* CONCAT *on a sub-sequence of the operands), it cannot be* $x.s$ *or* $x.e$ *of a* ShapeSegment *in the optimal segmentation.*

That is, local optimality leads to global optimality. Because of this assumption, our proposed algorithm is approximate. However, our empirical results (Section 8.4) show that despite this assumption, the accuracy of the algorithm is very close to that of DP, while taking orders of magnitude less time.

Algorithm 8.1 outlines the steps for scoring a fuzzy ShapeQuery. At a high level, the algorithm starts by dividing the trendline into smaller contiguous sub-regions (line 2). Next, it selects locally optimal points (LOPs) over small sub-regions (line 12), followed by a bottom-up merging step that uses LOPs over small sub-regions to find LOPs over larger sub-regions.

Selection of LOPs. We define a point P to be a LOP in a sub-region A for the sub-expression S_i if it is either the x.e of the first ShapeSegment *or* x.s of the last ShapeSegment of S_i. For instance, in the above example, it is easy to see that a LOP P in sub-region A is the x.e value of [p=up] in the

optimal segmentation of [p=up]⊗[p=down] in A. Since a CONCAT operation with k operands can have (k^2) sub-sequences, there can be a maximum of $2.k^2$ LOPs in A. The SelectLOPs function (line 9) is used for selecting LOPs. It is a variant of Algorithm 7.1 that returns both the final score as well as the end points of lines that make the optimal segmentation.

Merging. Next, the algorithm incrementally merges nodes in a bottom-up fashion to select LOPs over larger sub-regions (lines 6 to 17). More specifically, the Merge function (line 23) merges a sub-sequence $t1 : s_i \otimes s_{i+1} ... \otimes s_{i+m-1}$ in the left child with a sub-sequence $t2 : s_j \otimes s_{j+1} ... \otimes s_{j+n-1}$ in the right child if (i) $s_{i+1} ... \otimes s_{i+m-1} \otimes s_j ... \otimes s_{j+n-1}$ is a subsequence of query Q, or (ii) $s_{i+1} ... \otimes s_{i+m-1} \otimes s_{j+1} ... \otimes s_{j+n-1}$ is a subsequence of query Q when $s_i = s_j$ (i.e., we consider the common boundary ShapeSegment only once). While the score of the merged subsequence for case (i) can be easily computed using the average of the scores of left and right subsequence weighted by their number of ShapeSegments, i.e, $\frac{(m)*score(t1)+(n)*score(t2)}{m+n}$, for case (ii) we rescore the common ShapeSegment by estimating the slope of a line from the x.s of the last ShapeSegment of t1 to x.e of the first ShapeSegment of t2. If the s_c is the score of common ShapeSegment, $t1^*$ and $t2^*$ are the scores of left and right subsequence without the common ShapeSegment, then the score of the merge subsequence is: $\frac{(m-1)*score(t1^*)+s_c+(n-1)*score(t2^*)}{m+n-1}$. When multiple sub-sequences in the children nodes generate the same sub-sequence in the parent node, we select the sub-sequences that result in maximum score after merging, i.e., the one with the most optimal segmentation (line 24–25), thereby pruning out LOPs corresponding to non-selected sub-sequences. This merging process is repeated at each intermediate node. Finally, at the root node, we select the points that result in the maximum score for the entire sequence of operands.

Figure 8.2 depicts the logical order for scoring ShapeQuery a⊗(b⊕(c⊗d)) over the sub-sub-regions. Here, a, b, c, and d represent a ShapeSegment. The SegmentTree algorithm starts by scoring individual ShapeSegments (e.g., a,b,c and d in a⊗(b⊕(c⊗d))) independently over each of leaf nodes as depicted in Figure 8.2. Next, it computes the scores of sub-sequences in the intermediate nodes using the merging process described below. For example, in Figure 8.2, node 4 depicts the sub-sequences formed by combining sub-sequences from nodes 1 and 2, and node 5 depicts the sub-sequences formed by combining sub-sequences from nodes 3 and 4. When multiple sub-sequences in the children nodes generate the same sub-sequence in the parent node, we select the sub-sequences that result in maximum score after concatenation (i.e, the one with the most optimal segmentation), thereby pruning out LOPs corresponding to non-selected sub-sequences. For example, at node 5, a⊗b can be computed from 1) a from node 3 and b from node 4, 2) a⊗b from node 3 and b from node 4, and 3) a from node 3 and a⊗b from node 4. Among the 3 concatenations, we pick the one that gives the maximum score.

Given the closure assumption, the merging process leads to optimal segmentation as follows:

Algorithm 8.1 Fuzzy Matching Algorithm

Input: L: a sub-region of trendline, Q: a CONCAT operation, $ScrFunc$: scoring functions from Tables 7.4 and 7.5
Output: score

```
1:  procedure ExecFuzzyQuery(L, Q, ScrFunc)
2:      subRegions ← ComputeSubRegions(L) // leaf nodes
3:      T ← ComputeSubSequences(Q)
4:      nodes ← Queue()
5:      // scoring of leaf segments
6:      for each s in subRegions do
7:          lops ← []
8:          for each t in T do
9:              lops[t] ← SelectLOPs(s,t)
10:         end for
11:         node ← [s.start, s.end, lops[t]]
12:         nodes.add(node)
13:     end for
14:     //  bottom-up processing
15:     while nodes.size() > 1 do
16:         s ← nodes.size()
17:         // pairwise merging of nodes at the same level
18:         while s > 0 do
19:             s1 ← nodes.deque(), s2 ← nodes.deque()
20:             mlops ← []
21:             for each t1,t2 in s1.lops.keys(),s2.lops.keys() do
22:                 score, lops ← Merge(L,s1[t1],s2[t2])
23:                 if score > mlops[t1⊗t2].score then
24:                     mlops[t1⊗t2 ] = {lops, score}
25:                 end if
26:             end for
27:             node ← [s1.start, s2.end, mlops]
28:             nodes.add(node)
29:             s = s − 1;
30:         end while
31:     end whilenode ← nodes.deque()
32:     return node.lops[Q].score
33: end procedure
```

Theorem 8.2. *Given the closure assumption, the bottom-up algorithm with k* CONCAT *operands is optimal with a time complexity of $O(nk^4)$, i.e., linear in the number of points in the trendlines.*

PROOF: We prove the above theorem via induction.

Base case. For a single node SegmentTree, there is no difference between the SegmentTree algorithm and DP, since the SegmentTree algorithm uses DP to select the LOPs for a single node.

Induction step. Let L and R be two sibling nodes in the SegmentTree consisting of optimal scores for each possible subsequence of operands in the CONCAT operations, and let P be their parent node. Let $S^L_{i(k-1)}$ be the score of sub-expression from operand i until k in L for the optimal segmentation of $(i-1)$th to kth operands in L and $S^R_{(k+1j}$ be the score of the sub-expression from operand $k+1$ until j for the optimal segmentation of kth to $j+1$th operands in R. Let S^P_{ij} be the score of sub-expression of operand i until j in P, formed by concatenation of operands $i-1$ until k in L and k until $j+1$ in R. As per the Closure assumption, the optimal segmentation corresponding to S^P_{ij} must include the optimal segmentation $i-1$ until k in L and k until $j+1$ in R. Since kth operand is common between L and R, we need to re-compute its score over the sub-region from `x.e` of $(k-1)$th operand in L and `x.s` of $(k+1)$th operand in R during concatenation. Let sc^P_k be the re-computed score of the kth ShapeSegment. Then, S_{ij} can thus be computed as:

$$S_{ij} = \underset{k}{MAX}\{\frac{(k-i)\times S^L_{i(k-1)})+sc^P_k+(j-k)*S^R_{(k+1,j)})}{(j-i+1)}\}$$

Since, for computing S_{ij}, we consider all possible combinations of optimal segmentations in L and R and pick the one that gives the maximum score, it must be optimal.

Conclusion. Thus, by the principle of induction, the SegmentTree algorithm must also be optimal over the entire SegmentTree.

Time Complexity. For a sub-region of n points, the maximum number of leaf nodes is $n/2$ (since we need at least 2 points per sub-region) and therefore the total number of nodes in the tree is n. At each of the leaf node, we estimate the scores of each ShapeSegment independently, taking $O(n\times k)$ operations across all leaf nodes. Each intermediate node involves a merge step, involving concatenation of subsequences from left node with the right node. For k operands in CONCAT, there can be a maximum of k^2 subsequences per node, requiring a total of k^4 concatenations. Moreover, each concatenation involves the computation of score sc^P_k of the kth ShapeSegment that intersects left and right child. The computation of sc^P_k involves the estimation of the slope of line from `x.e` of $(k-1)$th ShapeSegment in L to `x.s` of $k+1$th ShapeSegment in R, which can be done in constant time from statistics of kth ShapeSegment's sub-region in L and R (see Theorem 3.3). Thus, each merging step involves $O(k^4)$ operations. Overall, the SegmentTree algorithm takes $O(n/2\times k^4+nk)\approx O(nk^4)$ time, i.e., linear in the number of points in the sub-region. In practice, k^4 is not a problem, since not all combinations of sub-sequences lead to a valid sub-sequence in the CONCAT operands, therefore the actual number of merges are much fewer. Moreover, k is

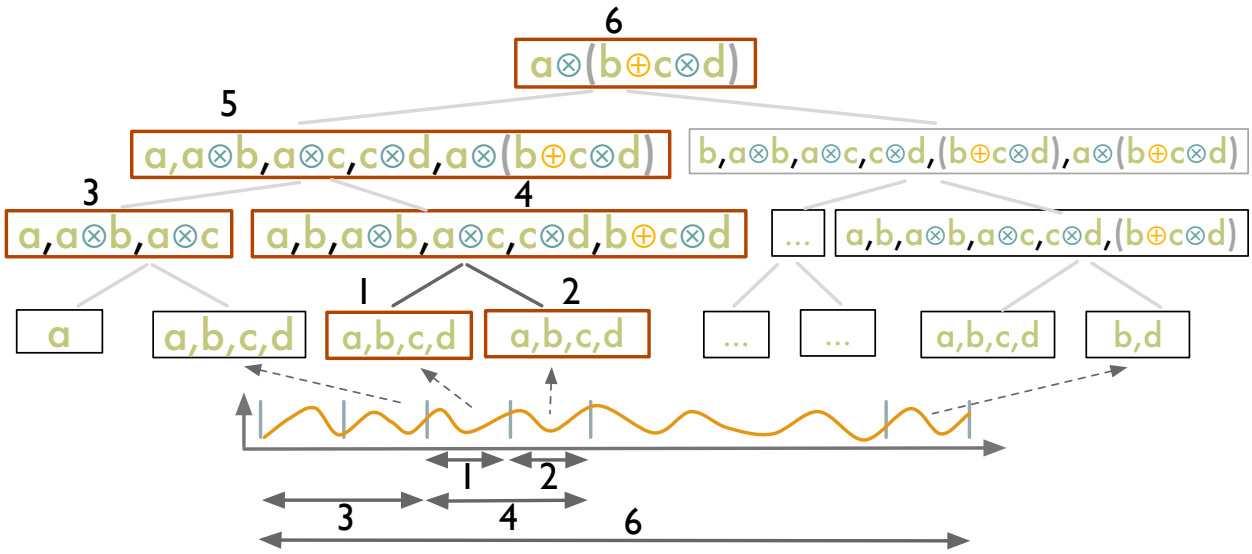


Figure 8.2: Bottom-up scoring of ShapeQuery

typically small (≤ 5). □.

8.2.1 Pruning Optimization

A large number of ShapeQueries are sequential pattern matching queries, consisting of only a single CONCAT operation on a sequence of simple patterns such as up, down, $\theta = x$. For such CONCAT operations, we can bound the final scores of trendlines and filter low-scoring trendlines without scoring them until the root node of the SegmentTree. We first describe our key observations.

Observation 8.1. Given a sub-region L comprising of sub-sub-regions: $L_1, L_2, ..., L_n$, the score of a ShapeSegment consisting of patterns up or down over L, $score_{up/down,L}$ is bounded between the maximum and minimum scores over any of the smaller sub-regions, i.e., $\underset{i}{MIN}(score_{up/down,L_i}) \leq score_{up/down,L} \leq \underset{i}{MAX}(score_{up/down,L_i})$. This is because the scores of up or down vary monotonically with the slope of the line, and the slope of the line over the large sub-region is always bounded between the maximum and minimum slopes of the lines over any smaller regions, $\underset{i}{MIN}(slope_{L_i}) \leq slope_L \leq \underset{i}{MAX}(slope_{L_i})$. However, when a slope (e.g., $\theta = x$) is specified as pattern, the above observation does not hold for $\underset{i}{MIN}(slope_{L_i}) \leq x \leq \underset{i}{MAX}(slope_{L_i})$, because the $score_{x,L}$ can be more than $\underset{i}{MAX}(score_{x,L_i})$ when $|x - slope_L| \leq \underset{i}{MIN}(x - slope_{L_i})$. For such cases, we set the upper bound to 1, the maximum possible score.

Observation 8.2. The score of an operator is bounded between the minimum and maximum scores of input ShapeSegments. This is clear from the scoring functions of operators as defined in Table 7.5.

Pattern	Max possible Score	Min possible Score
up	max across all level i nodes	min across all level i nodes
down	max across all level i nodes	min across all level i nodes
flat	max across all level i nodes if all $\theta > 0$ or all $\theta < 0$; otherwise 1	min across all level i nodes
$\theta = x$	max across all level i nodes if all $\theta > x$ or $\theta < x$; otherwise 1	min across all level i nodes

Table 8.1: Bounds on scores for different patterns based on scores at a given level i in the SegmentTree

Based on the above observations, we can derive the bounds on the final score of a ShapeSegment at the root node using the maximum and minimum scores of the ShapeSegment at a given level i in the SegmentTree. We summarize the bounds for each of the patterns in Table 8.1. Thus, instead of processing each trendline completely in one go, we process trendlines in rounds. In each round, we process one level of SegmentTree for all of the trendlines simultaneously, and incrementally refine the upper and lower bounds on their scores. Before moving on to the upper levels, we prune the trendlines that have their upper bound score lower than the current top-k lower bound scores. Overall, the pruning optimization helps avoid processing to complete for a large number of trendlines in the collection, and is particularly effective when the user is looking for trendlines with rare patterns.

8.3 ADDITIONAL OPTIMIZATIONS

ShapeSearch supports a couple of additional optimizations that result in faster scoring of trendlines.

Generating lines via Summary Statistics. For scoring a sub-region, ShapeSearch fits a line to approximate it. This is costly for fuzzy ShapeQueries where ShapeSearch needs to score sub-regions of varying sizes, fitting one line for every sub-region. We note that a summary of five statistics namely, $\sum x_i$, y_i, $\sum x_i.y_i$, $\sum x_i^2$, and n for a sub-region, is sufficient to compute the slope of the line over the sub-region as follows:

$$\theta = \frac{(n \times \sum x_i.y_i - \sum x_i \sum y_i)}{(n \times \sum x_i^2 - (\sum x_i)^2)}, \delta = \sum y_i - \theta \times \sum x_i. \tag{8.1}$$

Moreover, it is easy to see that the individual summaries over two sub-regions (A and B) are sufficient to compute the slope of the line over the combined region AB, without making additional

passes over the data.

$$\theta_{AB} = \frac{(n_A + n_B) * (\sum x_{Ai}.y_{Ai} + \sum x_{Bi}.y_{Bi}) - (\sum x_{Ai} + \sum x_{Bi}) * (\sum y_{Ai} + \sum y_{Bi})}{(n_A + n_B) \times \sum((x_{Ai})^2 + (x_{Bi})^2) - \sum(x_{Ai} + x_{Bi})^2} \tag{8.2}$$

Thus, the summary statistics help reduce data movement as well as the amount of data processed during segmentation. We summarize our finding using the following theorem.

Theorem 8.3 (Additivity)**.** *Given two adjacent segments A and B, a line segment over the combined segment AB can be estimated using linear regression on the summarized statistics over the individual segments A and B.*

Push-Down Optimizations. When a ShapeQuery consists of both fuzzy and non-fuzzy ShapeSegments, location constraints are pushed down to the trendline generation component to prune trendlines that do not have any value in the specified x ranges. Similarly, ShapeSearch avoids computing *summary statistics* over x ranges that are not used in the ShapeQuery (e.g., 0 to 50 in the above query), since the values over such ranges are ignored for segmentation and scoring. Overall, our empirical evaluation shows that these push-down optimizations significantly help in improving the overall response time of ShapeSearch.

8.4 PERFORMANCE EVALUATION

In this section, we evaluate the runtime and accuracy of ShapeSearch pattern matching algorithms. We first compare the runtime of the exhaustive pattern matching algorithm (Section 7.6) with four algorithms proposed in this chapter: (i) the dynamic programming-based (DP) algorithm, (ii) the Greedy algorithm, (iii) the SegmentTree algorithm, and (iv) the SegmentTree algorithm with pruning. We also compare with Dynamic Time Warping (DTW) [78], another dynamic-programming algorithm that is typically used for matching shapes in trendlines in systems like Zenvisage [3], to show the efficiency of ShapeSearch relative to existing systems. Next, we compare the accuracy of SegmentTree and Greedy with respect to the results of DP. Note that SegmentTree and Greedy are approximate while DP is an optimal algorithm and gives the same results as that of the exhaustive algorithm. Finally, we vary the characteristics of ShapeQueries to assess the impact of different factors on performance.

Datasets and Setup. Figure 8.2 depicts the five real-world datasets drawn from the UCI repository [90], and the list of queries we used for our experiments. Each dataset consists of trendlines

#	Dataset	|V|	$\|V_i\|$	Query	Runtime (sec)						Accuracy (%)	
					Exhaustive	DP	DTW	Segment Tree	Segment Tree+Prune	Greedy	Segment Tree	Greedy
1	Weather	144	366	(θ=45°⊗d⊗u⊗d)	290	52	11	5	2.8	0.9	85	25
2	Weather	144	366	((u⊕d)⊗f⊗u⊗d)	211	55	9	4	3.2	1.1	90	30
3	Weather	144	366	(f⊗u⊗d⊗f)	244	47	9	5	3.3	1.4	100	25
4	Worms	258	900	(d⊗(θ=45° ⊕θ=−20°)⊗f)	4737	76	53	10	7	2.2	90	35
5	Worms	258	900	(d⊗θ=45°⊗d)	4320	63	44	12	9	3.4	90	35
6	Worms	258	900	(u⊗d⊗u)	3953	68	42	9	6	2.5	90	20
7	50Words	905	270	(d⊗(u⊕(f⊗d)))	1046	105	28	7	5	1.1	90	25
8	50Words	905	270	(d⊗θ=45°⊗d)	954	122	32	7	5	1.9	100	40
9	50Words	905	270	((u⊕d)⊗(u⊕d)⊗f)	979	131	29	9	7	1.2	85	40
10	Housing	1777	138	(f⊗d⊗u⊗f)	165	58	40	14	12	1.5	80	15
11	Housing	1777	138	(u⊗d⊗u⊗f)	152	63	41	17	13	1.9	85	20
12	Housing	1777	138	(u⊗f⊗((θ=45°⊗ θ=60°)⊕(u⊗d)))	157	52	35	18	14	1.2	75	15
13	Haptics	463	1092	(u⊗d⊗f⊗u)	6869	890	62	16	12	3.1	90	40
14	Haptics	463	1092	(d⊗u⊗d⊗f)	7189	924	58	20	15	2.6	95	25

Table 8.2: Runtime and accuracy results

with a mix of shapes, and the datasets differ from each other in terms of number of trendlines ($|V|$) as well as their length ($|V_i|$). The queries were selected to have at least 20 trendlines with scores > 0 to ensure that the issued ShapeQueries were relevant to the dataset. All experiments were conducted on a 64-bit Linux server with 16 2.40GHz Intel Xeon E5-2630 v3 8-core processors and 128GB of 1600 MHz DDR3 main memory. Datasets were stored in memory, and we ran six trials for each query on each dataset.

8.4.1 Overall Run-time and Accuracy

Runtime Comparison. Figure 8.2 (Runtime) depicts the runtime for each of the query across all datasets. We see the time taken by the exhaustive algorithm is prohibitively large, rendering it no longer interactive. DP provides an order of magnitude speed-up over the exhaustive approach; however even DP can take 100s of seconds over trendlines with only a few hundred points. Both Greedy and SegmentTree provide a 2× to 40× improvement in runtime compared to DP, taking only a few seconds in the worst case. These algorithms explore a much fewer number of segmentations compared to the DP approach. We also see that these algorithms are about 10× faster than the DTW algorithm, whose runtime, like DP, varies quadratically with the number of points in the trendline. Finally, SegmentTree with Pruning further provides a speed-up of 10-30% by pruning low utility trendlines. Since the improvement in performance of SegmentTree and Greedy comes at the cost of accuracy, we next compare the accuracy.

Acccuracy Comparison. Figure 8.2 (Accuracy) depicts the accuracy of SegmentTree and Greedy

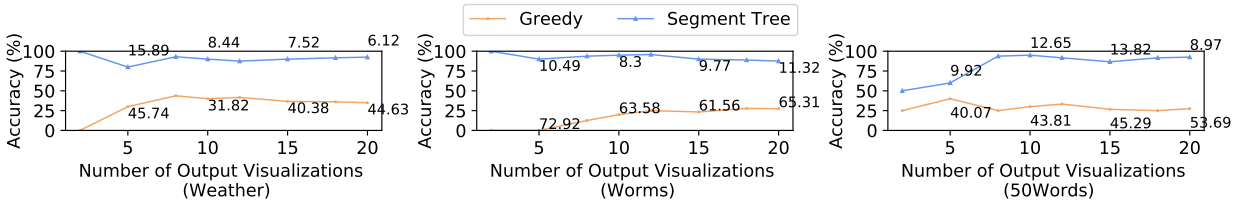


Figure 8.3: Accuracy with respect to DP over 3 *real datasets with varying number of output trendlines. Annotations denote the average deviation (in %) of the score of kth trendline chosen by algorithms with respect to the kth optimal trendline.*

relative to DP. We do not compare the accuracy of DTW with ShapeSearch algorithms since their scoring functions differ; instead we perform an user study in the next section to compare the effectiveness of ShapeSearch scoring functions with DTW and other similar metrics. We define accuracy here to be the number of trendlines picked by the algorithm that are also present in the top 20 trendlines selected by DP. We see that Greedy has a low accuracy ($< 30\%$), since it gets stuck on local optima. The accuracy of SegmentTree is closer to that of DP and is never off by more than 2 trendlines when we look at top 10 visualizations. Unlike Greedy, SegmentTree compares the local patterns in the trendlines and those specified in the ShapeQuery to select the segmentations that could result in high score.

Figure 8.3 depicts the accuracy results over top-k visualizations (with k varying from 2 to 20) for 3 of the the datasets. Annotations in each of the figures depict the average deviation in % of the score of kth visualization that an algorithm selects with respect to the score of the kth optimal visualization, indicating how off the shapes of selected visualizations are from optimal ones. We note that the accuracy of SegmentTree improves as the number of output visualizations increases, and is never off by more than 2 visualizations or have more than $> 12\%$ deviation in scores when we look at top 20 visualizations.

Overall, the runtime and accuracy results demonstrate that the SegmentTree **achieves comparable accuracy to that of DP in much less time**.

Next, we explore the impact of push-down optimizations on the overall performance of queries.

Impact of Push-Down Optimizations. We issue non-fuzzy queries, one query for each of the datasets, as depicted in Table 8.3. Figure 8.4 depicts the runtimes for ShapeSearch (note that all ShapeSearch algorithms behave similar for non-fuzzy queries) with and without push-down optimizations. We observe that non-fuzzy queries execute very quickly ($< 4s$ for over 1000 trendlines with more than 1000 points each), but pushdown optimizations help in further reduction of runtime in proportion to the selectivity of the LOCATION primitives in the query. For example, for ShapeQuery [p=up,x.s=60,x.e=80] on the Haptics dataset, pushdown optimizations help reduce the runtime from $3s$ to $< 1.2s$.

Name	Non-Fuzzy Queries
Weather	[p=down,x.s=1,x.e=4]⊗[p=up,x.s=4,x.e=10]⊗[p=down, x.s=10,x.e=12]
Worms	[p=down,x.s=50,x.e=100]
50 Words	[p=down,x.s=200,x.e=400]⊗[p=up,x.s=800,x.e=850]
Real Estate	[p=down,x.s=1,x.e=20]⊗[p=up,x.s=20,x.e=60] ⊗[p=down,x.s=60,x.e=138]
Haptics	[p=up,x.s=60,x.e=80]

Table 8.3: Non-Fuzzy Queries

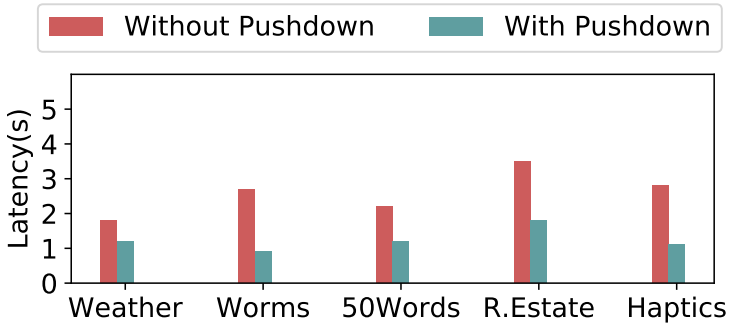


Figure 8.4: Average running time before and after push-down optimizations on non-fuzzy queries.

8.4.2 Varying ShapeQuery Characteristics

We evaluated the efficacy of our SegmentTree-based optimizations with respect to three different characteristics of ShapeQueries, as discussed below.

Impact of number of data points. Figure 8.5 shows the performance of algorithms as we increase the number of data points in trendlines for a fuzzy ShapeQuery (u ⊗d ⊗u ⊗d). With the increase in data points, the overall runtimes increases for all algorithms because of the increase in the number of segmentations. Nevertheless, SegmentTree shows better performance than DP after 100 data points since the SegmentTree approach is less sensitive (linear time) to the number of data points than that of DP (quadratic).

Impact of number of patterns. Figure 8.5 depicts the performance of fuzzy ShapeQueries with varying number of ShapeSegments (alternating *up* and *down* patterns) and issued over the weather dataset. As the number of ShapeSegments in the ShapeQuery grows, the overall runtimes of the algorithms also increases, with the runtimes for SegmentTree and SegmentTree with pruning growing much faster (k^4) than DP (k). However, the overall time for DP is still larger because the number of data points (366 in the weather dataset) plays a more dominant (n^2) role.

Impact of number of trendlines. We increased the number of trendlines from 100 to 1000 in the real-estate dataset with a step size of 100 and issued a fuzzy ShapeQuery ($u⊗d ⊗u⊗d$); the results are depicted in Figure 8.5. While the overall runtime for all approaches grows linearly with the number of trendlines, the gap between SegmentTree and SegmentTree with pruning grows wider.

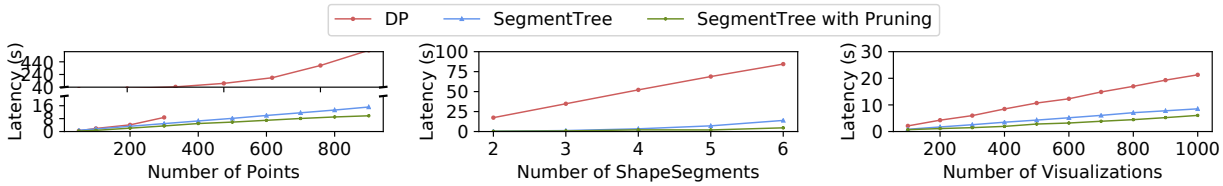


Figure 8.5: Runtimes on Varying Characterisitcs of ShapeQueries

This is because more trendlines get pruned as the size of the collection grows larger.

8.5 OVERALL TAKEAWAYS

In this chapter, we discussed the algorithms for executing fuzzy queries, a common subclass of ShapeQueries where there is at least one ShapeSegment with missing or multiple possible values for their starting and end locations in the trendline. Unfortunately, naïvely executing the fuzzy queries is extremely slow, requiring an expensive evaluation of all possible ways of matching each candidate trendline to the query to select the best one. We proposed a dynamic programming based optimal algorithm that reuses computation to provide substantial speed-ups, and show that even this algorithm can be prohibitively slow for interactive ad-hoc exploration. We then developed a novel perceptually-aware bottom-up algorithm that incrementally prunes the search space based on patterns specified in the query, providing a 40× speedup with over 85% accuracy, compared to the optimal approach.

Next, we present the results from the user evaluation of ShapeSearch, that we conducted to assess the usability and effectiveness of ShapeSearch.

CHAPTER 9: SHAPESEARCH USER EVALUATION

In this chapter, we first present a user study and a case study that we conducted to evaluate the evaluate the utility of ShapeSearch (relative to visual query systems [3, 9, 75]) on its ability to effectively support pattern matching tasks in trendlines. Next, we discuss the performance of ShapeSearch, focusing on overall runtime and accuracy.

9.1 USER STUDY

We conducted a user study to perform a qualitative and quantitative comparison of ShapeSearch with two baseline tools: our own system Zenvisage that we discussed from Chapters 3–6, and Qetch [74], both recent sketch-based systems for trendline pattern search. These systems allow users to sketch a pattern on a canvas, zoom in and out of the trendline to focus on a specific sub-region, and apply filtering and smoothing to match trendlines at varying granularities. While Qetch supports its own custom shape matching algorithm, Zenvisage allows users to choose between the Euclidean or DTW distance measures depending on the task. Qetch additionally supports a simple regex (via a *repeat* operator) to search for repeated occurrences of a sketched pattern. We disabled the sketching capability in ShapeSearch to isolate the benefits of the novel natural language and regex query mechanisms over sketch. ShapeSearch* denotes ShapeSearch with onatural languagey natural language- and regex-based querying mechanisms. We recruited 24 (14M/10F) participants with varying degrees of expertise in data analytics via flyers and mass-emails. We employed within-subjects study design between ShapeSearch* and each of the baseline tools, using two groups of 12 participants each. Note that by design, each participant encountered sketch capabilities only once—either in Zenvisage or Qetch. Participants were free to employ either natural language or regex for ShapeSearch*.

Dataset and Tasks. Based on the domain case studies from Chapter 7, as well as prior work in time series data mining [91–95] and visualization [3, 9, 75, 96, 97], we identified seven categories of pattern matching tasks, as depicted in Table 9.1. We designed these tasks on two real-world datasets: the Weather and the Dow Jones stock datasets from the UCI repository [90] that participants could easily understand and relate with. Together, the seven tasks spanned both exploratory search as well as targeted pattern-based data exploration, which helped us test the effectiveness of individual interfaces in various settings.

Ground Truth. For each of the tasks, three of the authors and 20 Mechanical Turk (mturk) workers, assigned a score between 0 (worst match) to 5 (best match) to each of the candidate trendlines.

Tasks	Description
Exact Trend Match (ET)	Find shapes similar to a specific shape, e.g., cities with weather patterns similar to that of NY, stock trends similar to Google's.
Sequence Match (SQ)	Find shapes with similar trend changes over time, e.g., cities with the following temperature trends over time: rise, flat, and fall, stocks with decreasing and then rising trends.
Common Trends (TC)	Summarizing common trends e.g., find cities with typical weather patterns, stock with typical price patterns.
Sub-pattern Match (SP)	Find frequently occurring sub-pattern, e.g., stocks that depicted a common sub-pattern found in stocks of Google and Microsoft, cities with 2 peaks in temperature over the year.
Width specific Match (WS)	Find shapes occurring over a specific window, e.g., cities with steepest rise or fall in temperature over 3 months, peaks with a width of 4 months.
Multiple X or Y constraints (MXY)	Find shapes with patterns over multiple disjoint regions of the trendline, e.g., stocks with prices rising in a range of 30 to 60 in march, then falling in the same range over the next month.
Complex Shape Matching (CS)	Find shapes involving trends along specific directions, and occurring over varying duration, e.g., stocks with head and shoulder pattern, cup-shaped patterns, W-shaped patterns.

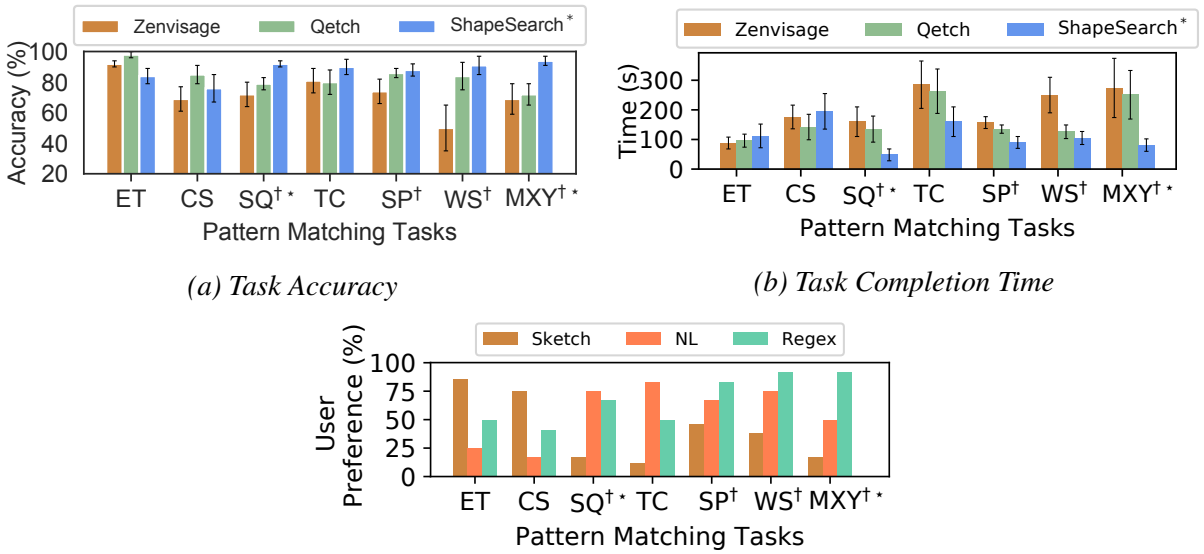
Table 9.1: Pattern Matching Tasks

Each mturk worker was presented with the task description in Table 9.1, along with a collection of trendlines, each of which they had to rate based on how closely the trendline matches the task description. We took the average of the ratings provided by the three authors and mturk workers to be the ground truth. We measure the participant's task accuracy as the (sum of the ground truth scores of the top K trendlines selected by the participant) $\times$ 100 / (sum of the top K ground-truth scores for the task). K varied between 2 to 5 per task.

9.1.1 Key Findings

We describe our key findings below.

Overall Task Accuracy and Completion Times. *As depicted in Figure 9.1a and Figure 9.1b,* ShapeSearch* *helped participants achieve higher accuracy and less time overall than Qetch and* Zenvisage*, and in particular for,* 5 *out of* 7 *tasks; however, for precise and complex shape matching tasks,* ShapeSearch* *performed worse than baselines due to the lack of sketch capabilities.* On average across all tasks, ShapeSearch* helped participants achieve an accuracy of 87%—8% more than Qetch and 17% more than Zenvisage—in about 30-40% less time, a significant improvement. While Zenvisage and Qetch involve less reasoning during query synthesis, they often lead to significantly more queries issued and manual browsing of trendlines for identifying the desired ones. ShapeSearch*, on the other hand, can accept more fine-grained user queries to rank relevant trendlines effectively, enabling participants to retrieve more accurate answers with less effort. In



(a) Task Accuracy

(b) Task Completion Time

(c) User Preferences

Figure 9.1: User study results († and ∗ denote that ShapeSearch *had statistically significant improvements (α = 5%) with respect to* Zenvisage *and Qetch respectively)*

order to better understand the differences between the tools, we separately analyze tasks where ShapeSearch* did better and worse than the baselines.

Settings Where ShapeSearch* **Wins**. Since sketch systems are based on precise matching, for sequence and sub-pattern matching tasks (SQ and SP), users drew multiple sketches for a given sequence or subsequence to find all possible instances. ShapeSearch*, however, is effective at automatically considering a variety of shapes that satisfy the same sequence or subsequence of patterns. Similarly, for tasks involving multiple constraints along the X and Y axes, or the width of patterns(TC, WS, MXY), a large majority of the participants gave more accurate results in less time with ShapeSearch*. ShapeSearch* supports a rich set of primitives for users to add multiple constraints on the patterns, including searching for patterns over multiple disjoint regions. While the users could zoom into a specific region of the trendline and sketch their desired patterns in the sketch systems, these capabilities were not sufficient to precisely specify all of the constraints at the same time. We believe that supporting visual widgets in the baseline tools that internally leverages the ShapeSearch primitives could remedy this issue.

Settings Where ShapeSearch* **Loses**. The opposite effect was observed (more time, less accurate with ShapeSearch*) when finding trendlines exactly similar to a given trendline (ET). This is understandable given that ShapeSearch* does not possess sketching capabilities which is a perfect fit for this task, and that ShapeSearch* regex scoring functions are targeted more towards approx-

imate and fuzzy pattern matching. For complex shapes (CS), Qetch performed the best, followed by ShapeSearch*, and then Zenvisage. Zenvisage performs the worst because the Euclidean and DTW measures used for matching shapes are sensitive to distortions in the sketch drawn by users for such complex shapes. Qetch, on the hand, applies corrections to distortions in shapes for better matching. For ShapeSearch* the results were mixed. We noted that the few participants who over-simplified the shape with fewer patterns (e.g., [p=down][p=up] for "cup-shaped" instead of [p=up][p=flat][p=up]) had poorer accuracy compared to those who used regex appropriately with correct sequence and width constraints. Overall, we find that complex shapes are easier to draw, harder to describe. We believe ShapeSearch with its sketching interface can potentially address these issues.

User Preferences and Limitations. In the end, we asked participants to complete a survey to gauge their preferences for the three mechanisms, sketch, natural language, and regex for each task. (Recall that each participant encounters a given specification mechanism in onatural languagey one tool .) We asked participants to select one or more of the three mechanisms they thought were most suited for each of the tasks they performed. They were allowed to select more than one if they felt multiple mechanisms were helpful. Figure 9.1c depicts the % of participants who selected the mechanism for each of the tasks. As depicted in the figure, user's preference results are correlated with their accuracy and completion times: most participants preferred the sketch-based interface for precise and complex shape-tasks, and natural language and regex for other tasks. When asked about their preferences in general, about 62% of the participants believed that the three interfaces integrated together would be most effective, 29% felt natural language and regex together without sketch would be sufficient for all pattern matching tasks, and onatural languagey 8% considered a sketch-based tool as sufficient, validating our design of a tool that goes beyond sketch capabilities. Participant P2 said "*Almost always, I will go with Tool B* [ShapeSearch*]. *I know exactly what I am searching [for] and what the tool is going to do, it is much more concise, I feel more confident in expressing my query pattern*".

About $2/3rd$ of the participants said they would opt for regex over natural language or sketch, if they had to choose one. Participant P8 said "*the concept for visual regex by itself is very powerful and could be helpful for most cases in general"*. This was surprising given than more than half of these participants had no prior experience with regular expressions-like languages. Participant P8 said "*the concept for visual regex by itself is very powerful, could be helpful for most cases in general*". Participant P4 said "*regex was very friendly to use, very powerful for a large number of usecases*". On the other hand, participant P1 who preferred natural language over regex said, "*Natural language and drawing [sketching-based querying] are good [sufficient] for most of the patterns, once or twice [rarely] I will search for complicated patterns with constraints, but I*

can then first use natural language and then fill the missing fields in the [form-based correction] panel below..." On a 5-point Likert scale ranging from strongly disagree (1) to strongly agree (5), participants gave a score of 3.9 when asked how effective ShapeSearch was in understanding and parsing their natural language queries. And when asked how easy it was to learn and apply regular expression, they gave a rating of 4.4.

Other findings. When asked about the effectiveness of using lines for matching trendlines, the average response was positive with a rating of 4.1 on a scale of 5. Participant P4 said "*Green lines are good, they make me more confident, help me understand trendlines especially [the] noisy ones without me having to spend too much time parsing signals. I can also see how my [query] pattern was fitted over the trendline ...*". Finally, participants also suggested several improvements to make ShapeSearch more useful in their workflows such as supporting more mathematical patterns by default like concave, convex, exponential, or statistical measures such as entropy; simplifying regex syntax by removing brackets; automatic regex validation and auto-correction; query and trendline recommendations, and using different colors for lines that correspond to different patterns (ShapeSegments) in the ShapeQuery.

Next, we present findings from a case study with bioinformatics researchers.

9.2 CASE STUDY: GENOMICS

To understand the use of ShapeSearch in a real-world setting, we conducted an open-ended evaluation of ShapeSearch via a case study with two bioinformatics researchers ($R1$ and $R2$). Both researchers were graduate students at a major university and perform pattern analysis on genomic data on a daily basis using a combination of spreadsheets and scripting languages such as R.

We started the study with a 15 minute introduction and a demo of the ShapeSearch tool on the Weather data set [90] that we had also used for the user-study, and asked participants to perform a few pattern matching tasks to help them familiarize themselves with the tool, as well as to ensure that they understood the functionalities of the tool. After introduction, participants used ShapeSearch to explore a popular mouse gene dataset [77] that they often analyze as part of their work. Since we were interested in seeing if they could create queries that reveal previously unknown insights, we requested our participants to think aloud, explain the kind of queries they were constructing, whether the results confirmed some already known fact, and how they currently performed such exploration using existing tools. This also helped us ensure that their mental model was matching with what the queries actually expressed. Each session lasted for about 75 minutes. We summarize our findings below.

9.2.1 Findings and Takeaways

I. *Both participants were able to grasp the functionalities of* ShapeSearch *after a 15 minute introduction and demo session without much difficulty*. During this session, the participants appreciated the ease of pattern search, saying "(*R*1) *oh, this feature [searching using combinations of patterns such as up and down] is cool, ... something that we frequently do*", "(*R*2) *I like that you can change your patterns [queries] that easily, and see the results in no time...*". Both participants concurred that ShapeSearch could be a valuable tool in a biologist's workflow, and can help perform faster pattern-based data exploration, compared to current R language scripting or spreadsheet approaches.

II. *Using succinct queries, participants could interactively explore a large number of gene groups, depicting a variety of gene expression patterns.* Both *R*1 and *R*2 were able to query for genes with differential expressions over time. *R*1 initially issued natural language queries to search for genes that suddenatural languagey start expressing themselves at some point, and then gradually stop expressing, i.e., flat, followed by increase, and then gradual decrease, a pattern signifying an effect of external stimulus such as a drug or a treatment. Thereafter, *R*1 was interested in understanding the variations in expression rates, e.g., identifying groups of genes that rise and fall much faster, or where changes are gradual. To search for these patterns, she interactively adjusted the width of patterns in her queries via regex. Finally, *R*1 also searched for groups of genes that show similar changes in expression over time, indicating they regulated similar cell mechanisms.

III. ShapeSearch *helped participants validate their hypotheses, and make new discoveries.* *R*2 used regex to explore a group of genes that increase with a slope of 45° until a certain point, and then remain high and stable (flat), as well as those with the inverse behavior (ones that start high and then gradually reduce their expression and remain low and flat). Such patterns are typically symbolic of permanent changes (e.g., due to aging) in cell mechanisms, often seen among genes in stem cells. While exploring these patterns, *R*2 discovered two genes, *gbx2* and *klf5*, in the results panel, that had similar expression patterns, and mentioned that the two genes indeed have similar functionality and are actively being investigated. Next to these two genes, he saw another gene *spry4* with almost similar expression, and hypothesized that the similarity in shape indicates that *spry4* possibly had similar functionalities to *gbx2* and *klf5*, something that is not well-known, and could lead to interesting discoveries if true.

IV. ShapeSearch *helped participants find genes with unexpected or outlier behaviors.* During the end of her study, *R*1 mentioned that it is rare to see a gene with two peaks in their expressions within a short window. However, on searching for this pattern via natural language, she found a gene named "*pvt1*" having two peaks within a short time duration of 10 time points. She found this

surprising, and said there could either be some preprocessing error, or some rare activity happening in the cell. She then searched for other unexpected patterns (e.g., three peaks, always increasing).

V. *Both natural language and regex are equally effective.* Comparing between natural language and regex, $R1$ said she could express most of her queries using natural language, and would use regex onatural languagey when the pattern is too long, and involves multiple constraints. $R2$, on the other hand, said he would use regex in all scenarios. He believed regex was not significantly difficult to learn, and helped him feel more in control and confident about what he was expressing, and whether the system was correctly inferring and executing his issued queries.

VI. *Participants faced a few challenges during exploration.* They wanted to switch back and forth between queries, so that they do not have to remember and reissue their previous queries. In addition to better presentation of the the fitted lines (e.g., coloring), they wanted to understand in more detail how the scores were computed, and if they could tweak the scoring according to their needs using visual widgets.

9.3 OVERALL TAKEAWAYS AND FUTURE WORK

In this chapter, we present a user-study and an open-ended case study with two genomics researchers for evaluating the expressiveness, effectiveness, and the usability of ShapeSearch. The findings from the user-study show that compared to sketch-based querying systems, ShapeSearch leads to more accurate and faster query results as well as have higher preference among users for tasks involving sequence-based patterns, sub-patterns, patterns involving multiple X and Y range constraints as well as those that involve complex combinations of patterns and location constraints. On the other hand, for searching for trendlines that precisely match the drawn pattern, sketch-based systems perform better. However, ShapeSearch with its sketching interface can potentially capture the precise shape-matching requirements.

We observe similar findings from the case study. ShapeSearch helped genomics researchers perform faster exploration of large number of gene groups, validate hypotheses, make new discoveries, as well as find genes with unexpected or outlier patterns.

Apart from providing more transparency on the approximation and scoring mechanisms as well as supporting additional patterns, there are a number of avenues for future work to further improve the capabilities of ShapeSearch. Currently, natural language, regex, and sketch cannot be used simultaneously to specify or refine a given pattern query, rather each specification leads to an independent query. We envision a more integrated interface where users can switch back and forth among natural language, regex and sketch to incrementally build and refine their queries. For example, users can start with a high level specification of the pattern by drawing a sketch, and then

incrementally refine it using either natural language (e.g., increase the slope of increasing), or add fine-grained constraints via regex.

Furthermore, ShapeSearch algebra (from Chapter 6) can be integrated with ZQL algebra (from Chapter 4) to simultaneously support both fine-grained and fuzzy pattern searching as well as operations such as comparison, filtering over collections of visualization. Recall from Chapter 4 that ZQL supports a user-defined functional primitive T which takes a visualization f and returns a real number measuring some visual property of the trend of f such as monotonicity, skewness. However, users needs to register T with Zenvisage in advance, and its execution is not optimized by the ZQL optimizer. Since a ShapeQuery also returns as real number in the range of -1 to 1, it can be easily used as a T primitive within a ZQL query. This will add more fine-grained pattern searching capabilities to ZQL, in addition to making such queries more amenable to various optimizations that we discussed in Chapter 8.

In the next chapter, we conclude and discuss a number of future directions stemming from our work in this dissertation.

CHAPTER 10: CONCLUSION AND FUTURE WORK

In this dissertation, we described two visual data exploration systems — Zenvisage and ShapeSearch, that synthesize techniques from databases, data mining, and visualization to fast-forward users to their desired visualizations. The research challenges and their solutions discussed in this dissertation play a significant role in the development of these systems.

In Chapter 3, we discussed the overall architecture and system development of Zenvisage as well as our first attempt towards addressing the visualization search problem via sketch-based interface and visualization recommendations. The remaining chapters covered two major extensions to the initial prototype, addressing the challenges of expressiveness (Chapters 4–6), and flexible and fuzzy pattern matching (Chapters 6–8).

In Chapter 4, we introduced a visualization exploration language — ZQL, which is based on an expressive algebra and forms the core of Zenvisage. In Chapter 5, we discussed how Zenvisage optimizes and executes ZQL queries interactively on large datasets. In Chapter 6, we reported findings from the user evaluation of both the sketching interface and ZQL with respect to manual visualization tools.

Then, in Chapter 7, we introduced a shape-based algebra coupled with multiple specification mechanisms for fuzzy pattern matching over trendlines in ShapeSearch. In Chapter 8, we discussed algorithms for efficient fuzzy shape matching over trendlines. In Chapter 9, we presented a user study as well as a case study, comparing ShapeSearch with exisiting systems for pattern matching over trendlines.

Overall, the automation in Zenvisage and ShapeSearch leads to (i) fewer man-hours to derive insights, (ii) less error as analysts are less likely to miss out on important trends or patterns, (iii) reduced frustration, tedium, and wastage of time of analysts, and (iv) identification of patterns or trends that the analysts were not otherwise aware of, and (v) higher data coverage by allowing traversal through a large number of visualizations at once. Furthermore, since these systems are intended to make knowledge hidden in data more evident and easier to access, they have already found usage within real world domains such as battery science, genomics, and ad analysis, and have lead to interesting findings.

Nonetheless, we are still far from solving the problem of visualization search and recommendation. There are several bottlenecks in our approach that we aim to address in future.

1. Facilitating Self-Service Data Preparation and Integration. Before users can start doing analysis, they need to spend considerable effort on pre-processing, profiling, cleaning, transforming, and integrating their data into a format amenable for importing into Zenvisage and ShapeSearch. Smarter and easier-to-use data preparation and integration support is critical to overcom-

ing the delays and inflexibility that frustrate users and hamper data exploration. Some important future steps include automatically learning the data formats as well as types of attributes such as categorical, quantitative; profiling to generate statistical summaries of data; eliminating duplicate data; spotting anomalies as well as interpolating missing values. In addition, users typically want to access and join a wider variety of datasets, but have to switch to another ETL tool or write code to do so manually. Augmenting tools with machine learning techniques to automatically join disparate but relevant data sets and creating a metadata catalog or glossary can be helpful as more users seek to drive data transformation and integration themselves through self-service BI and visual analytics tools, eliminating detours and distraction.

2. Intuitive Interactions for Supporting Novice Users. While our user studies showed that ZQL is easy to use, it may be unrealistic to expect everyone with interest in data analysis to learn ZQL. It remains to be seen if we can develop intuitive interactions that can translate automatically into lines of ZQL. For example, one novel direction could be to develop visual widgets and interactions that could let users compose two collections of visualizations and specify comparisons between the two collections along X, Y or Z attributes. These interactive primitives can potentially capture most two or three line ZQL queries, which form a large majority of overall queries issued to Zenvisage. Another method that we have been exploring is incremental composability: can we develop interaction primitives that correspond to a single line of ZQL, following which users can use multiple such interactions to generate a more complex query. These interaction primitives may include the use of drop-down menus. This would be akin to formulae in Microsoft Excel — which most analysts are comfortable with. We are also exploring the use of interactive querying interfaces, such as GestureDB [98] and DataPlay [99], which were originally developed for SQL, and try to adapt it to ZQL.

3. Conversational and Context-Driven Exploration. We observe that users often do not precisely know their questions when they start their analysis, instead they construct the questions as the exploration evolves. In addition, for complex analysis tasks, it is common for users to break them down into a sequence of simpler queries that build upon previous queries. Unfortunately, both Zenvisage and ShapeSearch currently treat each query independent of the past queries, thus users need to construct each query from scratch, impacting their flow of data exploration. A natural next step could be to represent the context of user exploration using a state machine consisting of partial queries as different possible states. The state can then be incrementally updated and executed as new queries arrive. Furthermore, we need to integrate the supported specification mechanisms such as sketch, natural language, regex such that users can seamlessly switch among them depending on the complexity of the individual queries.

4. Perceptually-Conscious Similarity and Outlier Search. Zenvisage makes use of similarity measures such as Euclidean distance and Dynamic Time Warping (DTW) for comparing two visualizations as well as for finding typical and outlier visualizations. However, typically these measures do not provide results that look visually similar to human analysts (especially when there is no perfect match); this is because they often provide the same "weight" to different features in a visualization. In particular, if there is a single dominant peak in a visualization, from the human perception standpoint, this is the main feature that should be used to assess similarity. Recent work has shown that similar issues arise in scatterplots as well, i.e., existing models of scatterplot similarity fail to accurately capture human visual perception [100]. Towards this end, we have been developing a similarity metric for trendlines that gives more importance to perceptual features. At a high level, the metric incorporates a weighted combination of points corresponding to extrema, the points in the local neighborhood of the extrema, as well as the local and global trends for measuring similarity. Our early experiments show that this metric performs better than other metrics listed on prototypical datasets, and at the same time, are much cheaper to compute. In future, we plan to further tune this metric based on our user study results.

5. In-Database Support for Common Analytics Operations. In Zenvisage's current architecture, the majority of the processing, e.g., comparison among visualizations, is done in its middleware, while limited amount of computation is pushed down to relational database management systems (RDBMS). The downside of this approach is that as the size of the dataset increases, the data transfer tends to dominate, resulting in an increase in latency. We find that a number of visual analytic operations involve comparisons between visualizations, which can be pushed inside RDBMS by expressing them using SQL queries. Unfortunately, the SQL queries for comparing visualizations tend to be complex and result in inefficient physical plans. In order to express such queries more easily, we envision developing new logical operations along with corresponding extensions to SQL that can simplify the writing of comparison-based visual analytic queries. For efficient execution, we can also develop new physical operators as well as algebraic equivalence rules and access patterns that can exploit the semantics of comparison operations to significantly improve the performance.

REFERENCES

[1] J. W. Tukey, "Exploratory data analysis," 1977.

[2] T. Siddiqui, J. Lee, A. Kim, E. Xue, X. Yu, S. Zou, L. Guo, C. Liu, C. Wang, K. Karahalios, *et al.*, "Fast-forwarding to desired visualizations with zenvisage.," in *CIDR*, 2017.

[3] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, and A. Parameswaran, "Effortless data exploration with zenvisage: an expressive and interactive visual analytics system," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 457–468, 2016.

[4] T. Térrific, "An $O(n \log n / \log \log n)$ sorting algorithm," Wishful Research Result 7, Fanstord University, Computer Science Department, Fanstord, California, Oct. 1988. This is a full TECHREPORT entry.

[5] T. Siddiqui, P. Luh, Z. Wang, K. Karahalios, and A. Parameswaran, "Shapesearch: flexible pattern-based querying of trend line visualizations," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1962–1965, 2018.

[6] T. Siddiqui, Z. Wang, P. Luh, K. Karahalios, and A. Parameswaran, "Shapesearch: A flexible and efficient system for shape-based exploration of trendlines," *arXiv preprint arXiv:1811.07977*, 2018.

[7] M. Mohebbi, D. Vanderkam, J. Kodysh, R. Schonberger, H. Choi, and S. Kumar, "Google correlate whitepaper," 2011.

[8] M. Wattenberg, "Sketching a graph to query a time-series database," in *CHI'01 Extended Abstracts*, pp. 381–382, 2001.

[9] M. Mohebbi, D. Vanderkam, J. Kodysh, R. Schonberger, H. Choi, and S. Kumar, "Google correlate whitepaper," 2011.

[10] K. Ryall, N. Lesh, T. Lanning, D. Leigh, H. Miyashita, and S. Makino, "Querylines: approximate query for visual browsing," in *CHI'05 Extended Abstracts*, pp. 1765–1768, 2005.

[11] C. Holz and S. Feiner, "Relaxed selection techniques for querying time-series graphs," in *UIST*, pp. 213–222, ACM, 2009.

[12] H. Hochheiser and B. Shneiderman, "Dynamic query tools for time series data sets: timebox widgets for interactive exploration," *Information Visualization*, vol. 3, no. 1, pp. 1–18, 2004.

[13] J. Mackinlay, "Automating the design of graphical presentations of relational information," *Acm Transactions On Graphics (Tog)*, vol. 5, no. 2, pp. 110–141, 1986.

[14] J. Mackinlay, P. Hanrahan, and C. Stolte, "Show me: Automatic presentation for visual analysis," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1137–1144, 2007.

[15] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer, "Voyager: Exploratory analysis via faceted browsing of visualization recommendations," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 1, pp. 649–658, 2016.

[16] M. Vartak, S. Rahman, S. Madden, A. Parameswaran, and N. Polyzotis, "Seedb: efficient data-driven visualization recommendations to support visual analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 13, pp. 2182–2193, 2015.

[17] S. Kandel, R. Parikh, A. Paepcke, J. M. Hellerstein, and J. Heer, "Profiler: integrated statistical analysis and visualization for data quality assessment," in *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pp. 547–554, ACM, 2012.

[18] E. Wu and S. Madden, "Scorpion: Explaining away outliers in aggregate queries," *Proceedings of the VLDB Endowment*, vol. 6, no. 8, pp. 553–564, 2013.

[19] C.-S. Perng, H. Wang, S. R. Zhang, and D. S. Parker, "Landmarks: a new model for similarity-based pattern querying in time series databases," in *Data Engineering, 2000. Proceedings. 16th International Conference on*, pp. 33–42, IEEE, 2000.

[20] T. N. Dang and L. Wilkinson, "Scagexplorer: Exploring scatterplots by their scagnostics," in *2014 IEEE Pacific Visualization Symposium*, pp. 73–80, IEEE, 2014.

[21] L. Wilkinson, A. Anand, and R. L. Grossman, "Graph-theoretic scagnostics.," in *INFOVIS*, vol. 5, p. 21, 2005.

[22] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE transactions on acoustics, speech, and signal processing*, vol. 26, no. 1, pp. 43–49, 1978.

[23] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 891–923, 1998.

[24] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra, "Locally adaptive dimensionality reduction for indexing large time series databases," *SIGMOD Rec.*, vol. 30, pp. 151–162, May 2001.

[25] D. Gunopulos and G. Das, "Time series similarity measures and time series indexing (abstract only)," *SIGMOD Rec.*, vol. 30, pp. 624–, May 2001.

[26] K.-P. Chan and A.-C. Fu, "Efficient time series matching by wavelets," in *Data Engineering, 1999. Proceedings., 15th International Conference on*, pp. 126–133, Mar 1999.

[27] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra, "Dimensionality reduction for fast similarity search in large time series databases," *Knowledge and Information Systems*, vol. 3, no. 3, pp. 263–286, 2001.

[28] K. Chakrabarti, E. Keogh, S. Mehrotra, and M. Pazzani, "Locally adaptive dimensionality reduction for indexing large time series databases," *ACM Trans. Database Syst.*, vol. 27, pp. 188–228, June 2002.

[29] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases," *SIGMOD Rec.*, vol. 23, pp. 419–429, May 1994.

[30] E. Keogh, "A decade of progress in indexing and mining large time series databases," in *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pp. 1268–1268, VLDB Endowment, 2006.

[31] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, *Fast subsequence matching in time-series databases*, vol. 23. ACM, 1994.

[32] R. A. K.-l. Lin and H. S. S. K. Shim, "Fast similarity search in the presence of noise, scaling, and translation in time-series databases," in *Proceeding of the 21th International Conference on Very Large Data Bases*, pp. 490–501, Citeseer, 1995.

[33] H. Shatkay and S. B. Zdonik, "Approximate queries and representations for large data sequences," in *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pp. 536–545, IEEE, 1996.

[34] R. A. G. Psaila and E. L. Wimmers Mohamed &It, "Querying shapes of histories," *Very Large Data Bases. Zurich, Switzerland: IEEE*, 1995.

[35] E. D. Kim, J. M. Lam, and J. Han, "Aim: Approximate intelligent matching for time series data," in *International Conference on Data Warehousing and Knowledge Discovery*, pp. 347–357, Springer, 2000.

[36] E. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping," *Knowledge and information systems*, vol. 7, no. 3, pp. 358–386, 2005.

[37] Y. Lamdan and H. J. Wolfson, "Geometric hashing: A general and efficient model-based recognition scheme," 1988.

[38] Y.-W. Huang and P. S. Yu, "Adaptive query processing for time-series data," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 282–286, ACM, 1999.

[39] M. N. Garofalakis, R. Rastogi, and K. Shim, "Spirit: Sequential pattern mining with regular expression constraints," in *VLDB*, vol. 99, pp. 7–10, 1999.

[40] "Knime." [Online; accessed 30-Oct-2015].

[41] "Rapidminer dataset." [Online; accessed 30-Oct-2015].

[42] "Sas." [Online; accessed 30-Oct-2015].

[43] "Spss." [Online; accessed 30-Oct-2015].

[44] G. Holmes, A. Donkin, and I. H. Witten, "Weka: A machine learning workbench," in *Conf. on Intelligent Information Systems '94*, pp. 357–361, IEEE, 1994.

[45] Pedregosa et al., "Scikit-learn: Machine learning in python," *The Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[46] S. Sarawagi, "Explaining differences in multidimensional aggregates," in *VLDB*, pp. 42–53, 1999.

[47] G. Sathe and S. Sarawagi, "Intelligent rollups in multidimensional olap data," in *VLDB*, pp. 531–540, 2001.

[48] J. Han *et al.*, "Dmql: A data mining query language for relational databases," in *Proc. 1996 SiGMOD*, vol. 96, pp. 27–34, 1996.

[49] T. Imielinski and A. Virmani, "A query language for database mining," *Data Mining and Knowledge Discovery*, vol. 3, no. 4, pp. 373–408, 2000.

[50] A. Netz *et al.*, "Integrating data mining with sql databases: Ole db for data mining," in *ICDE'01*, pp. 379–387, IEEE, 2001.

[51] "dygraphs." `https://www.dygraphs.com`, 2016. Accessed: July 18, 2016.

[52] K. Wongsuphasawat et al., "Voyager: Exploratory analysis via faceted browsing of visualization recommendations," *IEEE TVCG*, 2015.

[53] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series.," in *KDD workshop*, vol. 10, pp. 359–370, Seattle, WA, 1994.

[54] E. Keogh, S. Chu, D. Hart, and M. Pazzani, "An online algorithm for segmenting time series," in *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pp. 289–296, IEEE, 2001.

[55] K.-P. Chan and A. W.-C. Fu, "Efficient time series matching by wavelets," in *Data Engineering, 1999. Proceedings., 15th International Conference on*, pp. 126–133, IEEE, 1999.

[56] D. Rafiei and A. Mendelzon, "Similarity-based queries for time series data," in *ACM SIGMOD Record*, vol. 26, pp. 13–25, ACM, 1997.

[57] "jetty," http://www.eclipse.org/jetty/.

[58] S. Chambi, D. Lemire, O. Kaser, and R. Godin, "Better bitmap performance with roaring bitmaps," *Software: practice and experience*, 2015.

[59] M. M. Zloof, "Query-by-example: A data base language," *IBM Systems Journal*, vol. 16, no. 4, pp. 324–343, 1977.

[60] L. Wilkinson, *The grammar of graphics*. Springer Science & Business Media, 2006.

[61] H. Wickham, "ggplot: An implementation of the grammar of graphics," *R package version 0.4. 0*, 2006.

[62] J. Mackinlay, "Automating the design of graphical presentations of relational information," *ACM Trans. Graph.*, vol. 5, pp. 110–141, Apr. 1986.

[63] T. K. Sellis, "Multiple-query optimization," *ACM TODS*, vol. 13, no. 1, pp. 23–52, 1988.

[64] M. R. Anderberg, *Cluster analysis for applications: probability and mathematical statistics: a series of monographs and textbooks*, vol. 19. Academic press, 2014.

[65] K. Goto and R. A. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, p. 12, 2008.

[66] "Airline dataset." [Online; accessed 30-Oct-2015].

[67] P. Terlecki *et al.*, "On improving user response times in tableau," in *SIGMOD*, pp. 1695–1706, ACM, 2015.

[68] "Upwork (https://www.upwork.com/)." [Online; accessed 3-August-2016].

[69] R. Amar, J. Eagan, and J. Stasko, "Low-level components of analytic activity in information visualization," in *INFOVIS.*, pp. 111–117, IEEE, 2005.

[70] "Zillow real estate data." [Online; accessed 1-Feb-2016].

[71] K. S. Bordens and B. B. Abbott, *Research design and methods: A process approach* . McGraw-Hill, 2002.

[72] J. W. Tukey, "Comparing individual means in the analysis of variance," *Biometrics*, pp. 99–114, 1949.

[73] D. J.-L. Lee, J. Lee, T. Siddiqui, J. Kim, K. Karahalios, and A. Parameswaran, "You can't always sketch what you want: Understanding sensemaking in visual query systems," *IEEE transactions on visualization and computer graphics*, 2019.

[74] M. Mannino and A. Abouzied, "Expressive time series querying with hand-drawn scale-free sketches," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, p. 388, ACM, 2018.

[75] P. Buono, A. Aris, C. Plaisant, A. Khella, and B. Shneiderman, "Interactive pattern search in time series," in *Visualization and Data Analysis 2005*, vol. 5669, pp. 175–187, International Society for Optics and Photonics, 2005.

[76] "Investopedia." https://www.investopedia.com/terms/t/tripletop.asp.

[77] C. J. Bult, J. T. Eppig, J. A. Kadin, J. E. Richardson, J. A. Blake, and M. G. D. Group, "The mouse genome database (mgd): mouse biology and model systems," *Nucleic acids research*, vol. 36, no. suppl_1, pp. D724–D728, 2008.

[78] Wikipedia, "Dynamic time warping — wikipedia, the free encyclopedia," 2016. [Online; accessed 9-December-2016].

[79] "Coefficient of determination." https://bit.ly/2mRSB9A.

[80] J. Lafferty, A. McCallum, and F. C. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," 2001.

[81] Z. Kozareva, K. Voevodski, and S.-H. Teng, "Class label enhancement via related instances," in *Proceedings of the conference on empirical methods in natural language processing*, pp. 118–128, Association for Computational Linguistics, 2011.

[82] T. Siddiqui, X. Ren, A. Parameswaran, and J. Han, "Facetgist: Collective extraction of document facets in large technical corpora," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pp. 871–880, ACM, 2016.

[83] "Python crf-suite library (https://github.com/albertauyeung/python-crf-named-entity-recognition)." [Online; accessed 1-Oct-2018].

[84] R. P. Roetter, C. T. Hoanh, A. G. Laborte, H. Van Keulen, M. K. Van Ittersum, C. Dreiser, C. A. Van Diepen, N. De Ridder, and H. Van Laar, "Integration of systems network (sysnet) tools for regional land use scenario analysis in asia," *Environmental Modelling & Software*, vol. 20, no. 3, pp. 291–307, 2005.

[85] P. K. e. a. Muthumanickam, "Shape grammar extraction for efficient query-by-sketch pattern matching in long time series," in *Visual Analytics Science and Technology (VAST), 2016 IEEE Conference on*, pp. 121–130, IEEE, 2016.

[86] F. Li and H. Jagadish, "Constructing an interactive natural language interface for relational databases," *Proceedings of the VLDB Endowment*, vol. 8, no. 1, pp. 73–84, 2014.

[87] T. Gao, M. Dontcheva, E. Adar, Z. Liu, and K. G. Karahalios, "Datatone: Managing ambiguity in natural language interfaces for data visualization," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pp. 489–500, ACM, 2015.

[88] V. Setlur, S. E. Battersby, M. Tory, R. Gossweiler, and A. X. Chang, "Eviza: A natural language interface for visual analysis," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pp. 365–377, ACM, 2016.

[89] R. J. L. John, N. Potti, and J. M. Patel, "Ava: From data to insights through conversations.," in *CIDR*, 2017.

[90] "Uci repository." https://archive.ics.uci.edu/ml/datasets/.

[91] C. A. Ralanamahatana, J. Lin, D. Gunopulos, E. Keogh, M. Vlachos, and G. Das, "Mining time series data," in *Data mining and knowledge discovery handbook*, pp. 1069–1103, Springer, 2005.

[92] L. Ye and E. Keogh, "Time series shapelets: a new primitive for data mining," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 947–956, ACM, 2009.

[93] T.-c. Fu, "A review on time series data mining," *Engineering Applications of Artificial Intelligence*, vol. 24, no. 1, pp. 164–181, 2011.

[94] R. T. Olszewski, "Generalized feature extraction for structural pattern recognition in time-series data," tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2001.

[95] J. Han, M. Kamber, and J. Pei, *Data mining: concepts and techniques: concepts and techniques*. Elsevier, 2011.

[96] M. Correll and J. Heer, "Regression by eye: Estimating trends in bivariate visualizations," in *ACM Human Factors in Computing Systems (CHI)*, 2017.

[97] M. Correll and M. Gleicher, "The semantics of sketch: Flexibility in visual query systems for time series data," in *Visual Analytics Science and Technology (VAST), 2016 IEEE Conference on*, pp. 131–140, IEEE, 2016.

[98] A. Nandi, L. Jiang, and M. Mandel, "Gestural query specification," *PVLDB*, vol. 7, no. 4, pp. 289–300, 2013.

[99] A. Abouzied, J. M. Hellerstein, and A. Silberschatz, "Playful query specification with dataplay," *PVLDB*, vol. 5, no. 12, pp. 1938–1941, 2012.

[100] A. V. Pandey, J. Krause, C. Felix, J. Boy, and E. Bertini, "Towards understanding human similarity perception in the analysis of large sets of scatter plots," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pp. 3659–3669, 2016.