EFFICIENT PATTERN-BASED QUERYING OF TREND LINE VISUALIZATIONS

BY

ZESHENG WANG

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Aditya G Parameswaran

ABSTRACT

Finding visualizations with desired patterns is a common goal during data exploration. However, due to the limited expressiveness and flexibility of existing visual analytics systems, pattern-based querying of visualizations has largely been a manual process. We present ShapeSearch, a system that enables users to express their desired patterns using multiple flexible mechanisms—including natural language and visual regular expressions— and automates the search via an optimized execution engine. Internally, the system leverages an expressive ShapeQuery *algebra* that supports a range of operators and primitives for representing ShapeSearch queries. We will describe how the various components of ShapeSearch help accelerate scientific discovery by automating the search for meaningful patterns in multiple domains such as genomics and material science.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to first express my sincere gratitude to my advisor Professor Aditya Parameswaran for his immense knowledge, continuous guidance of my master thesis research, and also for his inspiration and influence on me throughout the year that I worked with him.

My sincere thanks also goes to his Ph.D. student Tarique Siddiqui, my mentor, for his motivation, patience and support. I've learned many important lessons from him. I could not have imagined having a better advisor and mentor for my master thesis.

I would also like to thank Paul Luh for his collaboration with me on my research all the time. Without his precious support and hard work in front-end and NLP interface, it would not be possible for me to finish this thesis.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 RELATED WORK 5

CHAPTER 3 SYSTEM OVERVIEW 7
- 3.1 Usage Scenario 7
- 3.2 System Architecture 7
- 3.3 Shape Query Algebra 8
- 3.4 Natural Language Translation 13

CHAPTER 4 QUERY EXECUTION AND OPTIMIZATION 15
- 4.1 Query Execution 15
- 4.2 Optimizations 19

CHAPTER 5 USER AND CASE STUDY 27
- 5.1 User Study 27
- 5.2 Case Study : Genomic Data Analysis 30

CHAPTER 6 EXPERIMENTS AND EVALUATION 33
- 6.1 Overall Results 34
- 6.2 Varying Characteristics of ShapeSearch Queries 35

CHAPTER 7 CONCLUSION AND FUTURE WORK 38

REFERENCES 39

CHAPTER 1: INTRODUCTION

While visual analytics systems such as Tableau [1] and Spotfire provide intuitive mechanisms to specify and generate visualizations, they do not provide ways to automate the search for desired visual patterns or trends. Due to this, analysts often spend hours examining many visualizations, all to find those that satisfy some desired visual pattern, e.g., a product whose sales is decreasing over time. Recent work, such as our tool Zenvisage [2] and Google Correlate [3], as well as older tools such as TimeSearcher [4] and Query-by-Sketch [5] aim to alleviate this burden by providing a "canvas" for users to sketch a visual pattern (or to drag-and-drop an existing visualization), with the system automating the search for visualizations that match that pattern, using an appropriate distance metric. We characterize such tools as *visual query systems* [6]. While these tools are a useful starting point in supporting the automated search for visualizations that match desired patterns, they offer limited flexibility in the pattern specification mechanism. In particular, if, instead of finding a product whose sales is decreasing over time, the analyst wanted to find a product whose sales is decreasing over some 3 month window (without specifying when), or if the analyst wanted to find a product whose sales has many increasing and decreasing portions (without specifying when these portions occur, their magnitude or their width), a precise "sketch" on a canvas can prove to be too rigid of a specification of the desired pattern. In such cases, the analyst may want more flexible mechanisms to specify the pattern of interest. Consider the following real-world examples:

Example 1.1 ***Health Informatics*** In clinical settings, wearable sensors are widely used for monitoring and collecting a range of vital health indicators such as temperature, heart rate, respiration rate, glucose level, for analysis and diagnosis of clinical conditions and behavioral interpretation. Based on their experiences over the years, doctors have identified a set of prototypical patterns and trends that are symbolic of specific critical conditions (Cao et al., 2008). For example, in a respiratory failure condition, most of the time, the heart rate first fluctuates in a very small range, followed by the respiration rate steadily rising and then steadily falling. Similarly, during Myocardial infarction (MI), i.e, heart attack, most of the time, the heart rate first suddenly increases, and then steadily decreases, while at the same time the blood pressure steadily reduces. Existing healthcare analytics tools typically follow a data-driven approach, where they pre-process and cluster the data to identify a small set of frequent patterns, which are then indexed for experts to search quickly during analysis. Few tools support on-the-fly iterative exploration of desired patterns — the presence or absence of which significantly depends on the scale of granularity, regions of interest, and filtering constraints best known, and often interactively changed by domain experts while analyzing the data. As a result, due to the limited flexibility offered by current tools, experts often have

to adopt a more tedious approach where they step through tens of thousands of visualizations and manually search for relevant patterns.

Example 1.2 ***Cosmological Data Analysis*** Dark Energy Survey scientists study the history and makeup of our universe by understanding the properties of astronomical objects such as galaxies over time. One such task involves studying the changes in brightness of supernova (bright stellar explosions) to estimate distances in space: the brighter the explosion is, the closer it is to our planet. By studying the relationship between the distance and the time when the explosion occurred (signified by its redshift), one can figure out the rate at which the Universe has been expanding at different times. Over years, scientists have identified characteristic classes (transient types) of supernova based on changes in level shifts, peaks, and width of brightness over time; these classes of patterns and their variants are then used for searching for new supernova in millions of visualizations collected at regular intervals. Yet another example deals with finding new planetary or stellar objects that belong to different stars. For this, scientists manually analyze tens of thousands of visualizations corresponding to brightness over time trends for stars (collected via telescopes constantly pointing at stars) for different time intervals. A dip in brightness is symbolic of a stellar object passing between the star and the telescope. Moreover, the duration and the degree of dips are used for the classification of stellar objects, and studying their physical properties.

Example 1.3 ***Genomics*** DNA microarray technology allows the measurement of expression levels of thousands of genes under tens or hundreds of different conditions. Biomedical researchers routinely look for structural changes or regulation (e.g., rising and falling of expressions at different points in times) in gene expression patterns, that are characteristic of internal biological processes such as the cell cycle or circadian rhythms, or some external perturbation, such as influence of a drug or a presence of a disease such as breast cancer. Analysis of these expression profiles reveals distinct patterns of gene expression that correlate with differentiation of organs including the nervous system, liver, skin, lungs, and the digestive system; providing intriguing insights into the mechanisms of mammalian development, as well as into the evolution of developmental processes. On numerous occasions, the actual patterns are not known in advance, but are discovered during the exploration process. For instance, scientists often apply a pattern, which was just discovered in a group of genes, to another population.

Example 1.4 ***Stock Market Analysis*** Financial analysts working in the stock market use variants of a selected set of pattern templates when analyzing and predicting stock market performance. A large number of frequently used templates such as ‘uptrends’, ‘downtrends’, ‘flag’, ‘head and shoulders’, ‘double bottom’ and ‘cup and handle’ have been identified. Certain shapes such as ‘cup and handle’ and ‘flag’ depict future gains, while a particular set of shapes such as ‘double top’ and

'head and shoulders' are harbingers of downtrends. Although the significance of prototypical patterns is widely accepted, their exact shape or variant of interest depends on the context and the scale of observation, therefore applying preprocessing to mine and index patterns in advance is not always effective. In addition, analysts often look for a number of patterns simultaneously, combining them in arbitrary ways to better understand and predict future behavior. As a result, with the constant enlargement of the stock market, it is becoming ever-challenging for analysts to both flexibly and efficiently locate meaningful patterns.

To address these challenges, we developed ShapeSearch, a flexible pattern querying system that supports multiple mechanisms for helping users express and search for desired patterns, with the following contributions:

ShapeQuery **Algebra**. We developed a ShapeQuery *algebra* that abstracts key shape-based primitives and operators, encapsulating a variety of typical patterns that are often of interest in trend line visualizations. For developing this algebra, we used a corpus of real-world pattern queries, collected via Mechanical Turk.

Natural Language Interface. Since our typical end-users, such as our biomedical researchers, are often not proficient in programming, we built a natural language interface within ShapeSearch for the flexible specification of ShapeQueries, coupled with a sophisticated parser and translator for converting these queries to ShapeQuery algebra. Unlike structured query languages, end-users do not need to know the syntax and semantics of the internal representation. One downside is that natural language queries can often be incomplete, and have subjective and/or ambiguous interpretation. ShapeSearch leverages a mix of automated and user-driven ambiguity resolution mechanisms to tackle these issues.

Regular Expression and Sketching. In addition to natural language, ShapeSearch supports a regular-expression-based interface for expert users, as well as a sketching interface (similar to typical visual query systems). These three interfaces can be used simultaneously, based on the complexity of the desired pattern, and users can switch between them as needed. All three interfaces ultimately compile down to ShapeQueries.

Scalable Evaluation. Naively matching a ShapeQuery to each visualization in a large collection of visualizations, wherein each visualization can be composed of thousands of datapoints, can take a really long time. ShapeSearch employs a scalable evaluation engine that facilitates efficient but approximate and perceptually-aware matching of visualizations to ShapeQueries, leveraging optimizations such as pruning and reuse of intermediate results.

The outline of this thesis will be as follow: we will briefly discuss related works in Chapter 2 and then present an overview to the ShapeSearch system in Chapter 3. We will first provide a usage

scenario and introduce the architecture of the system in and then present the formal definition of ShapeQuery algebra. Both the regular expression interface and natural language interface will be discussed in Chapter 3 as well. In Chapter 4, we will dive into the details of the core of the ShapeSearch system to discuss the algorithms and optimizations we developed in order to provide efficient executions of ShapeQueries. We then evaluate and analyze our system for usability in Chapter 5 and for efficiency in Chapter 6. Finally we will talk about future work and then conclude in Chapter 7.

CHAPTER 2: RELATED WORK

Our work draws on prior work on sequence matching in temporal data, and querying languages and natural language interfaces for databases and visualizations. Most of these work focus on either the expressivity or the scalability of pattern matching, with only few work supporting both aspects simultaneously.

Symbolic Sequence Matching. There has been many related work in time series sequence matching, that can be divided into two categories: symbolic matching and precise similarity metrics based matching. Symbolic approaches [7–9] discretize a time series as a sequence of events, and use variants of string matching algorithms such as edit-distance or longest common subsequence to find similar similar sequence of events as present in the input time-series. Symbolic sequences normally consists of a fixed number of discrete symbols as opposed to continuous points that make the similarity search methods quite efficient. The major downside is that detailed information about each sequence is often difficult to faithfully represent using few symbols such that all possible shape queries with different levels of granularity can be answered accurately. For instance, a portion of time-series annotated with Pattern "UP", may represent a "DOWN" pattern when looked as part of a larger portion. ShapeSearch, on the other hand, follows an online approach that looks at the primitives and operators in each query independently to decide how to discretize and match visualizations.

Precise Similarity Metrics. Dynamic time warping [10, 11] and cross-correlation based measures have been popular techniques in the context of shape-based precise time series matching. The essential idea is to match one dimensional patterns while allowing for local stretching of the time parameterization. Besides the matching process being compute intensive (the complexity of matching is $O(MN)$ given two sequences of lengths M and N), DTW is sensitive to the actual values and can express only a limited number of ShapeSearch queries.

Both symbolic and DTW-based approaches, often build indexing mechanisms [12–15] for efficient query processing during the runtime. For instance, Shape Definition Language (SDL) [16] lets users search for trend lines with specific sequences using a structured keyword-based language, and each trendline in the database is annotated with supported keywords in advance for faster sequence matching.

However, the ability to dynamically process a diverse class of pattern queries, expressed via a variety of primitives and and operators supported in ShapeQuery algebra, over arbitrary subsets of data (via filtering) makes ShapeSearch more suitable as an interactive and ad hoc time-series pattern matching tool. For example, users can express search for more complex combination of patterns, besides a sequence, using AND, OR, ITERATOR, and POSITIONAL operators. More-

over, unlike in the existing work, indexing trendlines in advances with supported patterns is not effective for ad hoc vague queries (e.g., a product following a specific pattern in US in 2014 may not follow the same pattern when only considered filtered on the state of California). ShapeSearch, thus, leverages several query-ware optimizations for efficient pattern matching.

QueryBySketch Systems. Most of the existing tools, including Query-By-Sketch systems [2–4] leverage distance measures such as Euclidean that perform "exact" matching based on the proximity of the values between the query and target visualizations, or use measures such as Discrete Time Warping [17] and cross-correlation-based metric [18] that are not expressive enough for capturing additional local and global features of trend lines such as the slope of the trend, number of peak, the rate of change of values in the trend line, and their combinations.

Natural Language systems. A number of keyword and natural language interfaces for querying databases [19] and generating visualizations [20, 21] have been developed over previous years. However, since the underlying shape query algebra in ShapeSearch is different from SQL, existing parsing and translation strategies from existing work cannot be easily adapted. Recently, conversation-based systems such as AVA [22], and the visual data exploration language ZQL [2] 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.

CHAPTER 3: SYSTEM OVERVIEW

We now briefly describe the usage scenario of ShapeSearch for our genomics application, along with the systems architecture.

3.1 USAGE SCENARIO

Figure 3.1 depicts the interface of the system, with an example query from genomics where a biomedical researcher wants to search for genes whose expression values follow a specific pattern — first rising, then going down, and finally rising again. In order to search for this pattern, researcher first loads the dataset via form-based options on the left (Box 1), and then selects the space of visualizations to explore by setting category as gene, X axis as time, and Y axis as expression values. Next, the researcher enters her intended pattern search query using natural language (Box 2). When the researcher types her query, ShapeSearch also recommends potential phrases based on historical queries via an auto-complete functionality. Alternatively, if the query consists of a basic trend, she can draw the sketch as depicted in Box 3, or if it is too complex to be expressed via either natural language or sketch, she can issue a regular expression query (Box 2). On submitting, the query is parsed and translated to structured representation consisting of operators and primitives by shape query algebra. The structured representation is sent back to the interface and shown to user as part of correction panel (Box D). Simultaneously, in case the query had no ambiguity, the system executes it and visualizes the top-k matching results in the results panel (Box E). In case of ambiguity or incorrect translation, the researcher can make corrections with the help of correction panel, or rephrase her query.

3.2 SYSTEM ARCHITECTURE

Figure 3.2 depicts the architecture of ShapeSearch. The system is designed as a lightweight web-based application consisting of a front-end and a back-end. As described in previous section, the front-end supports an interactive interface (Figure 3.1) for composing shape search queries, as well as for displaying the matching visualizations. All queries are converted to a common ShapeQuery representation and issued to the back-end using a REST protocol, which then parses and translates it into an intermediate ShapeQuery representation. As described in subsequent sections, the back-end supports an ambiguity resolver that uses a set of rules for resolving syntactic and semantic ambiguities, for making corrections, and for adding missing values to the user queries. The corrected queries are forwarded to the user for validation and additional corrections. The validated

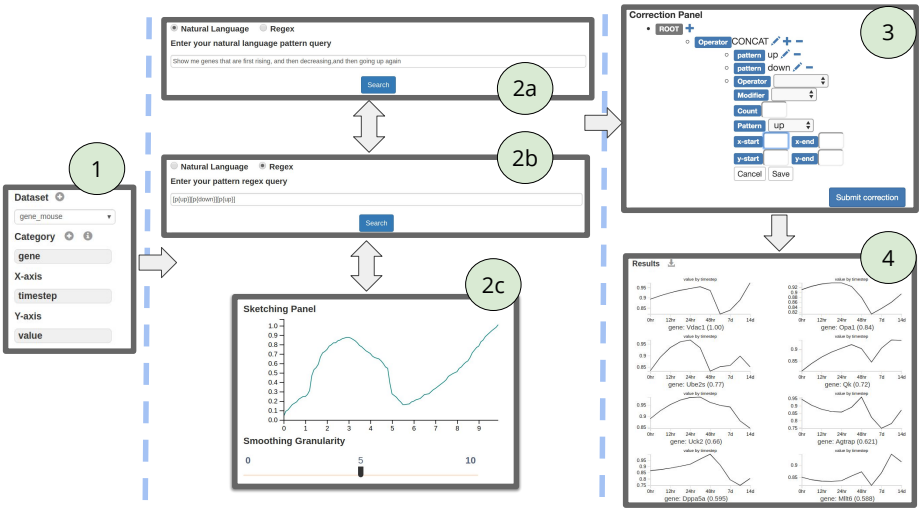

Figure 3.1: The ShapeSearch *Interface, consisting of six components. 1) Data upload and attribute selection, 2) Query specification: 2a) Natural language query interface 2b) Regular expression interface, and 2c) Sketching canvas, 3) Correction panel, and 4) Top-K results*

query is finally optimized and executed by the execution engine. The top visualizations that best match the ShapeQuery are finally returned to the front-end in the the JSON format.

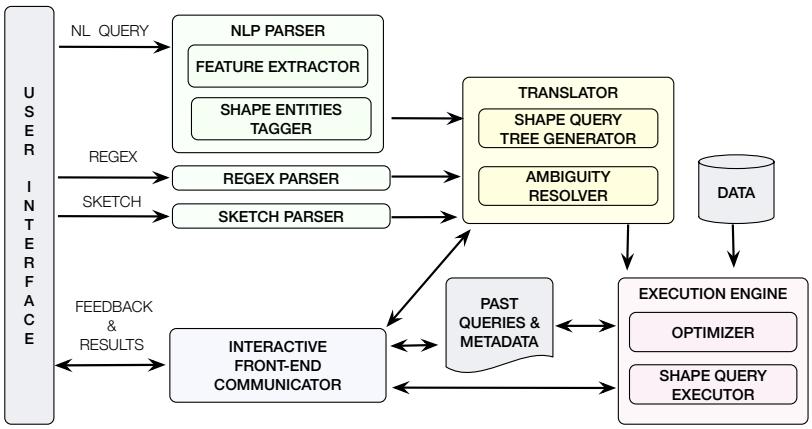

Figure 3.2: System Architecture

3.3 SHAPE QUERY ALGEBRA

In this section, we describe ShapeQuery algebra, a structured internal representation of a pattern search query, supporting a minimal set of primitives and operators that capture a wide range of ShapeSearch use-cases. All queries issued at the front-end are translated into a ShapeQuery before

being issued to the execution engine, which then optimizes and executes it for finding visualizations that best match the query.

The most interesting feature of ShapeQuery is its capability for fuzzy matching, allowing users to search for major trends or patterns in trendlines instead of local perturbations, fluctuations or noise. For example, biologists are often interested in locating genes whose expressions first fall and then rise, or ones that rise at least three times but fall not more than one time in a duration of 2 weeks, here they expect the system to ignore any minor increase or decrease in expression values over a small window of one or two days. ShapeQuery has been designed to make it natural to express such queries. Another important feature of ShapeQuery is that it has been designed to be efficiently executed within interactive response times, thereby helping in increased rates of insights and hypotheses via rapid exploratory analysis of multiple patterns over a large number of trendline visualizations.

Therefore, for efficient and fuzzy matching of pattern search queries, ShapeSearch represents a shape as a combination of one or more trends (e.g., rising followed by falling), where each trend is matched over a subregion of the visualization by fitting a line segment. A line-segment can capture a major trend in the visualization, and can be estimated efficiently using regression in time proportional to the number of the points in the visualization. We call the part of query representing an individual pattern, as ShapeSegment ($\mathbb{S}$). Moreover, multiple patterns or non-linear trends can be captured via combining ShapeSegments in different ways using *operators* $\mathbb{OP}$ to compose a ShapeQuery ($\mathbb{Q}$). As we will see later in the section, *operators* help in searching for a sequence of two or more patterns, two or more patterns within the same sub-region of the visualization, or one of many patterns in a given sub-region of the visualization among other combinations. Overall, a ShapeQuery can be represented as:

$$\mathbb{Q} = \mathbb{S}_1 \, \mathbb{OP} \, \mathbb{S}_2 \, \mathbb{OP} \, \mathbb{S}_n \tag{3.1}$$

For example, a user query "rising from X=2 to X=5 and then falling" can be translated into a ShapeQuery consisting of two ShapeSegments: [$\mathbb{S}.\mathbb{X} = 2$, $\mathbb{E}.\mathbb{X} = 5$,$\mathbb{P}$=$\mathbb{UP}$]$\otimes$[$\mathbb{P}$=$\mathbb{DOWN}$]. The first ShapeSegment captures "rising from X=2 and X=5', and the second ShapeSegment expresses a "falling" pattern. Moreover, the second ShapeSegment must "follow" the first one, i.e, the sub-region of the visualization where "falling" is searched must start from the end point of the region where "rising" is matched (X=5). As we will see in later part of the section, such a sequence is captured using *CONCAT* operator denoted by $\otimes$.

We, now, describe the shape primitives and operators that constitute the ShapeQuery algebra. Table 3.1 lists these primitives and operators.

3.3.1 Shape Primitives

A ShapeSegment can have three primitives — *LOCATION* ($\mathbb{L}$), *PATTERN* ($\mathbb{P}$), and *MODIFIER* ($\mathbb{M}$), and not all primitives are mandatory. For example, users can skip the *LOCATION* values, if they want to match a *PATTERN* anywhere in a trendline. Similarly, users can input the exact trendline to matched, or the endpoints of the ShapeSegments to match without specifying the *PATTERN*.

LOCATION ($\mathbb{L}$) defines the endpoints of the ShapeSegment between which a pattern is matched. It consists of four sub primitives, not all of which are mandatory: X start position $\mathbb{S}.\mathbb{X}$, X end position $\mathbb{S}.\mathbb{E}$, Y start position $\mathbb{S}.\mathbb{Y}$, and Y end position $\mathbb{E}.\mathbb{Y}$ For example, $\mathbb{Q}$ = [$\mathbb{S}.\mathbb{X} = 2$, $\mathbb{E}.\mathbb{X} = 10$, $\mathbb{S}.\mathbb{Y} = 10$, $\mathbb{E}.\mathbb{Y} = 100$] is a simple ShapeQuery for finding visualizations whose trend between X=2 to X=10 are similar to the line segment starting at (2,10) and ending at (10,100). Similar to QueryBySketch systems, users can also draw a trendline and issue a query to match it precisely. Such a query is represented as $\mathbb{Q}$ = [$\mathbb{V}$=((2,10), (3,14), ...,(10,100))], here the vector of (x,y) values for V are automatically translated from the user drawn sketch.

PATTERN ($\mathbb{P}$) defines a shape or a characteristic feature that the user is looking for in a sub-region of the visualization or over the entire visualization. The system supports a set of basic patterns that are commonly used for characterizing trendlines such as $\mathbb{UP}$, $\mathbb{DOWN}$, $\mathbb{FLAT}$, or they can specify the angle of the trend in degrees. For example, $\mathbb{Q}$ = [$\mathbb{P}$=$\mathbb{UP}$] finds trendlines that are increasing and $\mathbb{Q}$ = [$\mathbb{S}.\mathbb{X} = 2$, $\mathbb{E}.\mathbb{X} = 10$,$\mathbb{P}$=$\mathbb{UP}$] finds trendlines that are increasing from X=2 to 5. In these examples, the higher the slope of the line segment fitted over the visualization, the better the match. One can also specify the exact slope of the trendline to be matched. For instance, for $\mathbb{Q}$ = [$\mathbb{P} = 45$], the closer the slope of the trendline to 45°, the better the match. We will see subsequently how we can combine these basic shapes with operators to search for arbitrary complex shapes.

MODIFIER ($\mathbb{M}$) further defines the way or the number of times (or quantify) a pattern is matched. For example, one can ask for trendlines that rise sharply or ones that have at least two peaks. When $\mathbb{M} = \gg$ is combined with $\mathbb{P}$=$\mathbb{UP}$, it means that trends that rise sharply are a better match. Similarly, $\mathbb{M} = 2$ can be combined with $\mathbb{P}$=$\mathbb{UP}$ to find trendlines that rise twice. $\mathbb{M} = \{2, 5\}$ refers to the occurence of a pattern between 2 and 5 times, $\mathbb{M} = \{2, \}$ refers to at least 2 times, $\mathbb{M} = \{, 2\}$ refers to at most 2 times. *MODIFIER*, when used along with the *POSITION* operator (described later in the section) can help compare a ShapeSegment with previous or subsequent ShapeSegments in the query (e.g, the slope of second ShapeSegment must be more than that of first ShapeSegment, or rise by at least 2X relative to that of the first ShapeSegment).

3.3.2 Operators

So far, we have seen examples of ShapeQueries consisting of single ShapeSegment. Now, we will see how we can use operators to combine multiple ShapeSegments to search for complex shapes. The system supports six operators.

MATCH([ ]) is a unary operator that takes as input ShapeSegment, execute it over one or more subregions (visual segments) of visualization.

CONCAT ($\otimes$) operator specifies a sequence of two or more ShapeSegments. For example, one can search for genes that are first rising, then falling, and then increasing again.

AND ($\odot$) operator simultaneously matches two or more patterns in the same sub-region of the visualization. Unlike CONCAT, all of the patterns must be present in the same sub-region of the visualization. For example, one can look for genes whose expression values rise twice but do not fall more than once.

OR ($\oplus$) operators searches for one among many patterns in the same sub-region of the visualization, picking the one that matches the most. For example, one can search for genes whose expressions are either up-regulated or down-regulated.

OPPOSITE (!). A unary operation for matching the opposite of the shape expressed in the ShapeSegment. For example, instead of saying increasing or decreasing patterns, one can say not flat pattern.

ITERATOR (.) is used within the *LOCATION* primitive to iterative over points in the trendline. For examples, one can search for cities that have maximum rise in temperature over a width of 3 months using query $\mathbb{Q}$ = [$\mathbb{S}.\mathbb{X} = .$, $\mathbb{E}.\mathbb{X} = (.+3)$,$\mathbb{P}$=$\mathbb{UP}$]. Here the ITERATOR (.) iterates over all points in the trendlines trying out each point as the start X position, with the x end position set to the point that is 3 points ahead.

POSITION ($) is used within *PATTERN* to refer to the pattern in the previous or subsequent ShapeSegments in the ShapeQuery. Along with *POSITION*, *MODIFIER* can be set to $>$,$<$, or $=$ to ensure the slope of the pattern in the current ShapeSegment is more than, less than or equal to the slope of the pattern in the referred ShapeSegment. For example, $\mathbb{Q}$ = [$\mathbb{P}$=$\mathbb{UP}$][$\mathbb{P}$=\$1,$\mathbb{M}$=$<$] finds trendlines where the first part is rising, and the second part is either rising or falling but the slope is less than that of the first part. One can set $\mathbb{M}$=$< \frac{1}{2}$ to ensure the slope of second part is at most (1/2) of the first part. $- and $+ can be used to refer to the previous and subsequent ShapeSegments.

Symbol	Name	Description
$\mathbb{L}$	LOCATION	Primitive
$\mathbb{S}.\mathbb{X}$	START X VALUE	Location Sub-Primitive
$\mathbb{S}.\mathbb{Y}$	START Y VALUE	Location Sub-Primitive
$\mathbb{E}.\mathbb{X}$	END X VALUE	Location Sub-Primitive
$\mathbb{E}.\mathbb{Y}$	END Y VALUE	Location Sub-Primitive
$\mathbb{P}$	PATTERN	Primitive
θ	SLOPE	Pattern value
$\mathbb{U}$	UP	Pattern Value
$\mathbb{D}$	DOWN	Pattern Value
$\mathbb{F}$	FLAT	Pattern Value
$\mathbb{V}$	SKETCH	Pattern Value
$\mathbb{M}$	MODIFIER	Primitive
$>$	MORE / GRADUAL (UP)	Modifier value
$>>$	MUCH MORE / SHARPER (UP)	Modifier value
>2	ATLEAST 2X	Modifier value
$=$	SIMILAR	Modifier value
$<$	LESS / GRADUAL (DOWN)	Modifier value
$<<$	MUCH LESS / SHARPER (DOWN)	Modifier value
$=$	SIMILAR	Modifier value
[ ]	MATCH	Operator
$\otimes$	CONCAT	Operator
$\odot$	AND	Operator
$\oplus$	OR	Operator
!	OPPOSITE	Operator
.	ITERATOR	Operator (within location)
$	POSITION	Operator (within modifier)

Table 3.1: Symbols used in Shape Query Algebra

3.3.3 Grouping and Nesting

ShapeSearch allows ShapeSegments to be grouped to specify precedence of some parts of the query over others. For example, one can search for trends that are increasing followed by either flat or decreasing then increasing. In order to ensure that, the pattern "flat" or "decreasing then increasing" are considered together, one can use parenthesis ( ) for indicating precedence: $[P = UP] \otimes ([P = FLAT] \oplus ([P = DOWN] \otimes [P = UP]))$

Similarly, ShapeSearch allows ShapeSegment to be *nested* as a value for *PATTERN* primitive to search for complex shapes with the constraint that the *LOCATION* values within parent and child ShapeSegments must not conflict. For example, one can use nesting to search for peaks (UP followed by DOWN) with a width of 4 points in the x range 2 to 10 using the following query, $\mathbb{Q}$ = [$\mathbb{S}.\mathbb{X} = 2$, $\mathbb{E}.\mathbb{X} = 10$, $\mathbb{P}$=[$\mathbb{S}.\mathbb{X}$=., $\mathbb{E}.\mathbb{X} = . + 4$,$\mathbb{P}$=[[$\mathbb{P}$=$\mathbb{UP}$][$\mathbb{P}$=$\mathbb{DOWN}$]]]].

3.4 NATURAL LANGUAGE TRANSLATION

While regular expression and sketch-based queries are straight-forward to parse, there are multiple steps involved in the translation of natural language queries to their intermediate ShapeQuery representation.

Shape Primitives and Operators Recognition. A natural language query consists of a sequence of words, where each word either maps to one of the entities (primitives and operators) in the ShapeQuery, or is a noise word. We follow a three step process in tagging words to their corresponding entities. In the first step, we use a Stanford Parser [23] to annotate words to their corresponding POS tags, and to extract dependency tree among the words. Second, using a set of rules, we annotate each word to be either noise, or non-noise. For non-noise words, if they match with high confidence (e.g., edit distance <= 1) to a commonly used word stored for each of the entities, we add another annotation called "likely primitive". In the final step, we extract a set of predefined features (in addition to annotations) for each of the non-noise words, and use a training conditional-random field-based model to predict their corresponding entities. For training datasets, we collected and tagged 250 natural language queries via a Mechanical Turk study, where users were asked to describe patterns in trend line visualizations. On cross-validation, the model had an F1 score of 81%.

ShapeQuery Tree Generation. We use a context-free grammar to represent nesting and structural relationships between entities. Using the grammar, the tree generator groups related entities into ShapeSegment, and combines multiple ShapeSegments using operators to make the ShapeQuery. ShapeSegments make the leaves in the query tree, whereas operators make the intermediate nodes. If no valid ShapeQuery tree is identified, the ambiguity resolver module tries to predict the nearest possible valid ShapeQuery tree.

Ambiguity Resolution. Ambiguity can occur at various levels in the query, both in the structure (syntax) as well as the meaning (semantics). Common causes of structural inconsistencies are wrong phrasing of queries by users, grammatical errors, and missing connector words, all of which can lead to wrong tagging of entities. For example, there could be two patterns with the same shape segment, or multiple missing entities in the same segment. Similarly, semantic inconsistencies occur when there are multiple conflicting meaning of the same phrase. For example, a parsed ShapeSegment might represent an increasing pattern from y=10 to y=2. In case of ambiguities, the system tries to predict the best possible intermediate language representation, using a number of rules.

Identifying Patterns. For words tagged as pattern, we calculate the normalized edit distance score between the word and the set of synonym words for each of the supported patterns. If the lowest edit distance is less than a pre-defined threshold, we further calculate the average semantic

similarity (using wordnet synset) between the word and the synonyms of the pattern type. If the score is again less than the threshold, we throw an error.

ShapeQuery Tree Validation. As depicted in Figure 3.1, we display the parsed ShapeQuery at the interface for user to further edit or refine it. The validated query is finally sent to the execution engine.

CHAPTER 4: QUERY EXECUTION AND OPTIMIZATION

4.1 QUERY EXECUTION

Given a validated ShapeQuery, the goal of the query executor is to score each visualization, and return top k visualizations that best match the ShapeQuery. As depicted in the Figure 4.1, the execution engine supports a pipeline of physical operators that together generate and process a collection of visualizations according to the primitives and operators specified within the ShapeQuery. In this section, we give an overview of the pipeline and scoring methodology, followed by several optimizations that the engine performs during query execution in the next section.

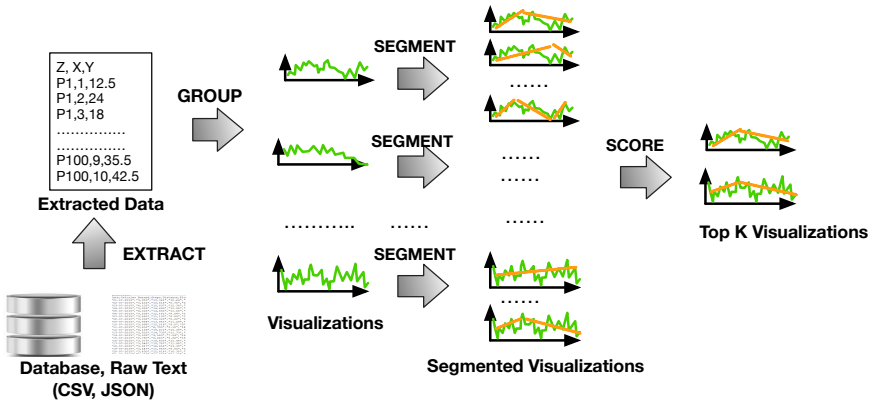

Figure 4.1: Query Exececution Pipeline

EXTRACT operator reads records from one or more data sources including relational databases and raw text files in CSV or JSON format based on Z, X, Y, and filter constraints, passing only only those records and attributes that are relevant to the downstream operators in the pipeline.

GROUP operator generates multiple visualizations simultaneously, one corresponding to each unique value of the Z attribute in records received from EXTRACT operator. Note that this functionality is different from the GroupBy operator supported in relational databases, where grouping is done per unique combination of $<Z,X,Y>$ values without any ordering or supporting the notion of visualization. Each visualization is first created as an ordered list of $<X,Y>$ values, with Y values aggregated if the user has specified aggregation function such as average, sum. Next, the visualization is approximated as a sequence of small line segments, with minimum width granularity. Users can set the minimum width granularity according to their preference; if not set ShapeSearch automatically figures out the minimum width granularity as *(X attribute range) / (number of pixels allocated along the X axis)*. As we will see subsequently, downstream operators in the pipeline construct longer segments out of these smaller segments based on the shape primitives specified in

the query. An important property of the GROUP operator is that it passes only five statistics, called *summarized statistics*, for each small-width segment: $\sum x_i$, $\sum y_i$, $\sum x_i.y_i$, $\sum x_i^2$, n in the visualization, this substantially reduces the amount of data to be processed by the downstream operators. Moreover, these statistics are sufficient for downstream operators to estimate longer segments, without any loss of accuracy.

Property 4.1 (Additivity) *Given two adjacent regions A and B of a visualization, a line segment over the combined region AB can be estimated using the summarized statistics over the individual regions A and B.*

Proof: A line segment is characterized by slope θ and intercept δ, given by the following formula:

$$\theta = \frac{(n * \sum x_i.y_i - \sum x_i \sum y_i)}{(n * \sum x_i^2 - (\sum x_i)^2)} \tag{4.1}$$

$$\delta = \sum y_i - \theta * \sum x_i \tag{4.2}$$

From these equations, it is easy to see that:

$$\theta_{AB} = \frac{((n_A * \sum x_{Ai}.y_{Ai} + n_B * \sum x_{Bi}.y_{Bi}) - (\sum x_{Ai} \sum y_{Ai}) * (\sum x_{Bi} \sum y_{Bi}))}{(n * \sum x_i^2 - (\sum x_i)^2)} \tag{4.3}$$

$$\delta_{AB} = (\sum y_{Ai} + \sum y_{Bi}) - \theta_{AB} * (\sum x_{Ai} + \sum x_{Bi}) \tag{4.4}$$

In summary, the group operator outputs a collection of visualizations, with each visualization consisting an ordered list of summarized statistics.

SEGMENT is computationally the most intensive operation in the pipeline; it takes visualizations from the Group operator as input, and represents each visualization with multiple line-segments, depending on the number of CONCAT operators in the ShapeQuery. We use the term *segmented visualization* to denote such a visualization.

In Figure 4.2, we depict one of the several segmented visualizations constructed for the ShapeQuery $a \otimes (b \oplus (c \otimes d))$, here a,b,c, and d denote ShapeSegments without the LOCATION primitives. This ShapeQuery aims to find visualizations where there is a pattern a in the beginning followed by either pattern b or a sequence of two patterns c and d. The Abstract Syntax Tree (AST) for the ShapeQuery and the input visualization are shown in the left side of the figure, while the segmented visualization along with annotated ShapeSegments and operators are shown in the right side. Segment operator traverses the AST from top to bottom and segments the visualization whenever a CONCAT operator is seen. For example, in the query, the CONCAT operator at the root segments the visualization into two parts: the first segment is matched with a, while the second segment is matched with $(b \oplus (c \otimes d))$. We use the term *SegmentGroup* to denote the parts of

the query such as a and $(b \oplus (c \otimes d))$ that are scored over different segments. Moving down the AST, we see the $\oplus$ operator between b and $c \odot d$, that scores them independently on the second segment and takes the one with higher Score. However, for scoring $c \otimes d$, the second segment is further split into two segments. Thus, overall for the above query, a segmented visualization consists of four segments, one for each of the following SegmentGroups: a,$(b \oplus (c \otimes d))$,c,d.

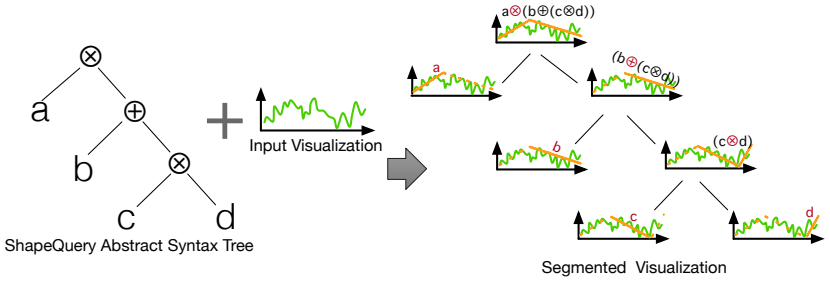

Figure 4.2: Segmentation

In summary, for a query with no CONCAT operator, the segmented visualization has only one line-segment. On the other hand, if there is CONCAT operator with k SegmentGroups, the segmented visualization has (k+1) line segments. *The key challenge here is that the location primitives are often not fully specified like in the example above, in such a case a naive segment operator follows an* exhaustive *approach where it segments each visualization using all possible ways: for a* ShapeQuery *consisting of k SegmentGroups operators, an exhaustive partitioning approach creates* $n^{(k-1)}$ *segmented visualizations, where n is the number of points in the visualization. Evaluating these many segmented visualizations within an interactive response time is infeasible even for a moderate number of points within a visualization and the number of SegmentGroups in the* ShapeQuery. We discuss our techniques for optimizing this in the next section.

SCORE operator scores each segmented visualization with respect to the ShapeQuery, and then for each visualization it picks the segmented visualization with the highest score. Finally, it outputs the overall top-k visualizations with highest scores.

4.1.1 Scoring

Each segmented visualization is scored in a bottom-up fashion starting with segments corresponding to leaf nodes in the ShapeQuery AST. For example, in Figure 4.2, we first compute the scores of the ShapeSegments a, b, c and d. The scores vary in between 1.0 (best match) to -1.0 (worst match), based on how closely the patterns in ShapeSegments match with the segments in the visualization. After computing the scores of the leaf nodes, scores of intermediate nodes are computed by combining the scores of the children nodes according to the definition of operators. The score of the root operator represents the overall score of the segmented visualization.

Scoring ShapeSegment. The score of a ShapeSegment depends on two factors: 1) Location matching, L: whether the segment satisfies the LOCATION primitive constraints in the ShapeSegment. L is set to 1 if the location primitive are satisfied, otherwise L=0; and 2) Pattern matching, P: how similar is the segment in the visualization with respect to the PATTERN primitive specified in the ShapeSegment. P varies between -1 (worst match) to 1 (best match). The score of the ShapeSegment, *score* = P if L=1 and *score* = -1 if L=0.

The scores for PATTERN primitives are computed using the slope of the line-segment. For instance, the score of UP increases linearly from -1 to 1 as the slope of the segment increases from -90° to +90°. Similarly for FLAT, the score is +1 when the slope is 0° and decreases linearly to -1 as slope increases to +90° or -90°. When the PATTERN is $\theta = x$, the score is maximized when the slope of the segment is x°, and decreases linearly to -1 as the difference between the segment slope and x° increases. Formally, scores are calculated as follows:

Pattern	Score
UP	$\frac{2 \cdot tan^{-1}(slope)}{\pi}$
DOWN	$-\frac{2 \cdot tan^{-1}(slope)}{\pi}$
FLAT	$(1.0 - \|\|\frac{4 \cdot tan^{-1}(slope)}{\pi}\|\|)$
$\theta = x$	$-\frac{2 \cdot tan^{-1}(\|slope - x\|)}{\pi}$
any	1
empty	-1
V	L2 Norm

Table 4.1: Formulas for Pattern Scoring

Note that when the PATTERN is V, where the user wants to perform a precise matching with her drawn trend, we calculate the score using L2 norm (Euclidean distance) between the drawn sketch and input visualization without any segmentation. The L2 norm can vary in a range of 0 to ∞, therefore we normalize it to a range of +1 to -1 using using the maximum (set to +1) and minimum (set to -1) L2 norms across all visualizations. ShapeSearch currently does not allow users to construct a ShapeQuery with both V and the other PATTERN primitives discussed above.

Combining scores using operators. Operators define how the scores ($score_1, score_2, ..., score_k$ in the table below) across ShapeSegments are combined, and are computed as follows:

Operator	Score
CONCAT	$\frac{(score_1+score_2+...+score_k)}{k}$
AND	$min(score_1, score_2, ...score_k)$
OR	$max(score_1, score_2, ...score_k)$
NOT	$-(score_1)$

Table 4.2: Formulas for Operator Scoring

For CONCAT, we take average of the scores across a sequence of patterns. AND looks for multiple patterns over the same segment, and to minimize the chances of any pattern not having a good score, it takes the minimum of all scores. OR accepts the best pattern among all, so it takes the maximum of all scores. It is easy to observe the following property from the definition of the above operators.

Property 4.2 (Boundedness) *The absolute value of the combined score from an operator is bounded between the minimum and maximum scores of the input* ShapeSegments.

Quantifier Scoring. When a ShapeSegment has a quantifier such as AT LEAST, AT MOST, or BETWEEN, we divide the segment into smaller sub-segments and score the pattern over each of the sub-segments. Then we count the number of sub-segments where the score is greater than certain threshold, by default it is set to zero). If the number does not satisfy the constraint (e.g., AT LEAST k, BETWEEN k and k'), we give the segment an overall score of -1, otherwise we calculate the score of the segment by taking the average of scores across top k'' sub-segments, where k'' is the minimum number that satisfies the constraint. We take the minimum since for higher number of sub-segments the average decreases.

4.2 OPTIMIZATIONS

In this section, we will discuss in detail about how the actual scoring is done by the ShapeSearch system. Specifically, we will introduce the algorithms we developed along with the optimizations we applied based on our observations.

4.2.1 Push Down Shape Primitives

Even a simple ShapeQuery often involves processing a large number of visualizations, with each visualization potentially resulting in many segmented visualizations. In order to minimize the number of such visualizations to score, we push down certain primitives within a ShapeQuery

to the lower operators in the pipeline. This optimization drastically reduces the query processing time by identifying and pruning visualizations that do not belong to the final results earlier in the pipeline.

First, LOCATION primitives are pushed down to the GROUP operator to filter visualizations which do not contain any values in the region specified by LOCATION primitives. Moreover, even for the visualizations that satisfy the primitives, it does not order or fit smaller segments over those regions that are not part of any ShapeSegment, and are therefore not used for scoring.

Similarly when ShapeSegments contain both LOCATION primitives, as well as PATTERN primitives such as UP or DOWN, we push these details one level down to the SEGMENT operator. Instead of creating all the segments for the visualization in a pattern-agnostic manner, the Segment operator prioritizes the construction of segments that satisfy location primitives first, and then immediately calculates their slopes to validate whether their (UP or DOWN) scores are positives. If the scores are negative, it prunes those visualizations. This optimization helps in avoiding creation and scoring of large number of low-utility (i.e., less relevant to patterns) segmented visualizations, that are definitely going to be rejected later.

4.2.2 Optimizing CONCAT Operator

4.2.2.1 Optimal Approach: Dynamic Programming

The exhaustive approach, we discussed earlier, examines all possible ways to segment a visualization, and is thus extremely slow even for a moderate number of points in the visualizations and number of ShapeSegments in the ShapeQuery (with a time complexity of $O(n^{k-1})$). We observe that scores over smaller parts of the visualization can be reused for scoring the larger parts of the visualizations, instead of re-generating and re-scoring the smaller parts many times. We state this property below.

Property 4.3 (OPTIMAL SUBSTRUCTURE) *Optimal segmentation for k SegmentGroups of a visualization over points 1 to n can be constructed from the optimal segmentation of k' and $(k-k')$ SegmentGroups over points $1-n'$ and $(n-n')$ for some $k' < k$ and $n' < n$. In other words, optimal segmentation for a sequence of patterns on a visualization can be constructed from optimal segmentations of subsequences of patterns over smaller regions of the visualization.*

Proof: We prove the above property by contradiction. Let $OPT(1,i,[1:j])$ be the score for the optimal segmentation over the first i values of the visualization on fitting 1 to j SegmentGroups. Assume a configuration where the scores over the smaller regions of the visualization are sub-optimal, i.e., $Score(1,n',[1:k']) < OPT(1,n',[1:k'])$ and $Score(n',n,[k':k]) < OPT(n',n,[k':k])$

then $score(1,k,[1:n])$ on applying CONCAT operation on scores $= \frac{Score(1,k',[1:n'])+Score(n',n,[k':k])}{k} < \frac{OPT(1,n',[1:k'])+OPT(n',n,[k':k])}{k} < OPT(1,n,[1:k])$. Therefore, scores of SegmentGroups over smaller regions of the visualization must be optimal.

Based on the above property, we have the following recurrence for the dynamic programming algorithm that we use for implementing CONCAT operator:

$$OPT(1,i,[1:j]) = \\ max_l\{CONCAT(Score(1,l,[1:j-1]),Score(l,i,[j-1:j]))\} = \\ max_l\{\frac{(j-1)*(Score(1,l,[1:j-1])+Score(l,i,[j-1:j]))}{j}\} \quad (4.5)$$

In other words, the optimal score for the first j SegmentGroups can be computed by using the optimal score of the first $(j-1)$ SegmentGroups, and the optimal score of jth SegmentGroup. The base case is $Score(i,i+1,[j:j]\ \forall\ i$ and j, when the difference between the start and end point of the visualization is one and the number of SegmentGroup is also 1. Here, we calculate the score for each SegmentGroup independently, and for $j > i$, we set the score to $-\infty$.

The correctness guarantee of the DP (Dynamic Programming) algorithm follows from the optimal substructure proof, and therefor we have the following theorem.

Theorem 4.1 *Given optimal substructure, finding the best visualization can be optimally done in $O(n^2k)$ using Dynamic Programming.*

Time Complexity. Our dynamic programming algorithm runs in polynomial time with respect to the number of points in the visualization and number of SegmentGroups in the ShapeQuery. If the length of the visualization is n, and the number of SegmentGroups is k, we need to calculate k scores for each of the n points; and for each point, we need to look at all previous points. Therefore, overall worst-case time complexity is $O(n^2k)$.

Given that time series sequences in practice are rather long, the dynamic programming approach, with its quadratic runtime, is unfortunately not fast enough for real-time pattern matching.

4.2.2.2 Our Approach: Segment-Tree-based Pattern Aware Scoring

In this section, we introduce a query-aware divide and conquer-based approach that provides results much faster (in linear time) than the optimal solution, while trading off minimal accuracy. In particular, we introduce a tree-based bottom-up algorithm, where the main idea is to divide the problem into smaller subproblems, solve the subproblems optimally for all possible

SegmentGroups mentioned in the query, and combine subproblems incrementally to derive the final solution. Before we discuss our algorithm, we introduce Segment-Tree, a data structure that our algorithm uses for deciding the order for scoring SegmentGroups.

Definition 4.1 (Segment-Tree) *A Segment-Tree is a balanced binary tree structure, where each node in the tree depicts a segment of the visualization, with the length of the segments increasing from top to bottom. The root node fits a single segment over the entire visualization, which recursively splits into multiple equal-size segments such that there are 2^L segments at level $L-1$.*

Figure 4.3 is an example of 4 level segment-tree for the ShapeQuery $a \otimes (b \oplus (c \otimes d))$ we discussed earlier. Note that we do not create the Segment-Tree structure in advance, rather we create nodes in the tree in a bottom-up fashion as we process the query, that is, the segment operator takes segments for leaf nodes as input from GROUP operator, and creates nodes at the upper levels in a bottom-up fashion. In other words, the Segment-Tree logically defines the order in which segments are to be created and scored by our algorithm.

High-Level Intuition: Our key insight is that for segmentation, we should avoid considering points, that are irrelevant to the patterns within the ShapeQuery. For example, consider a visualization of length 10, that is strictly increasing from 1 to 6, and strictly decreasing from 6 to 10. For scoring a ShapeQuery UP⊗DOWN where the user is looking for a pattern that is first increasing and then decreasing, the point at which we segment the visualization should involve a switch from increasing to decreasing pattern. Segmenting the visualization at a point in the earlier region (X=1 to 5) or in the later region (X=7 to 10) region is not useful since these regions do not have both increasing and decreasing trends.

Based on our analysis of patterns across domains such as genomics, finance we found that *a point that is not relevant to the patterns in the query over a local region around it, has lower likelihood of being relevant when considered as part of a wider or a more global region*. In the above example, when we look at a smaller region [4 to 8] for the pattern up and down, points other than 6 are irrelevant. These points will remain irrelevant when we look at a wider region say [2 to 9]. We exploit this observation in our algorithm as follows: on finding point x to be the optimal point in region $[m:n]$ for SegmentGroups S1 and S2, we reject other points in $[m:n]$ from the S1 and S2 perspective. Note that for some other pair of adjacent SegmentGroups S3 and S4, there can be another point y within $[m:n]$ that is allowed to be optimal. Therefore, we select all such points for different sub-patterns within the ShapeQuery, and reject the rest before moving on to the wider region. We formally state our assumption below.

Assumption 4.1 (Closure) *If a point x is the best split point for two adjacent SegmentGroups S*1 *and S*2 *in the* ShapeQuery *within the sequence $[i, j]$ ($i \leq x$ and $j \geq x$), then point x is also the best*

split point for S1 and S2 for any smaller sequence $[m,n]$ *around x* ($i \leq m \leq x$ *and* $j \geq n \geq x$). *Alternatively, if point x is the best split point in* $[m,n]$ *for* $S1$ *and* $S2$, *then no point other than x within* $[m:n]$ *can be the optimal split for* $S1$ *and* $S2$ *in a longer sequence* $[i,j]$. *There can still be a point outside* $[m:n]$ *which is more optimal than x in the longer sequence* $[i:j]$.

At a high level, our algorithm starts by looking at lower-level nodes of the Segment-Tree, that capture smaller regions in the visualization. For each node, we select those points (referred to as relevant points above) which result in the highest scores for one or more sub-patterns in the query. We, then, move to the higher level nodes, and repeat the search of relevant points among the ones selected earlier at children nodes. We do so until we reach the root node, which represents the entire visualization. The segments obtained from the points selected in the root node, are considered as the best fit for the ShapeQuery.

Algorithmic Details: Our algorithm takes as input the leaf nodes of the segment-tree, where each leaf node is the smallest segment of the visualization. The Segment-Tree algorithm starts by calculating scores for each of the SegmentGroups mentioned in the ShapeQuery for each of the leaf nodes. It, then, computes the scores of the parent nodes in the upper levels by merging the subsequences of ShapeSegments from the left and right children nodes, considering only those merged subsequences that exist in the ShapeQuery. Merging is performed according to the operator connecting the two subsequences in the ShapeQuery, as discussed in Section 4.1.1. If there are duplicate merged subsequences, we take the one with maximum score. This procedure is repeated level by level until we reach the root node of the Segment-Tree, where we retrieve the score for the sequence of ShapeSegments corresponding to the ShapeQuery.

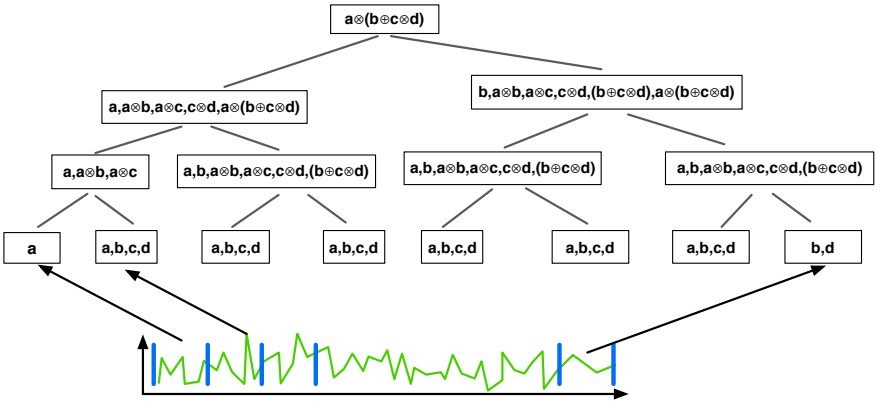

Figure 4.3: Segment-Tree based scoring of ShapeQuery $a \otimes (b \oplus (c \otimes d))$

Theorem 4.2 *Under the CLOSURE assumption, the Segment-Tree algorithm is optimal.*

Proof: We prove the above theorem via induction. For a single node Segment-Tree and a ShapeQuery with a single SegmentGroup, it's clear that the algorithm is optimal since there is only one

choice. Now assuming that the scores for subsequences of ShapeSegments in the left child node and right child node of a node N are optimal, we need to prove that the score computed for a merged subsequence S in node N are optimal. From the closure assumption, the selected points for SegmentGroups in S must be a subset of the points of subsequences in children nodes. While merging, our algorithm considers all possible valid sequences of points from the children nodes, and selects the subset where the score for the merged subsequence is maximum, therefore it must be optimal (using the CLOSURE assumption).

Time Complexity. Assuming there are n leaf nodes in the Segment-Tree, one corresponding to each smallest segment in the visualization out from the GROUP operator, there will be a total of $2n$ nodes in the tree. Moreover, for a CONCAT operator on k SegmentGroups, there is a maximum of k^2 possible subsequences of SegmentGroups, and therefore at each parent node, there can be a maximum of k^4 merge operations (from the cross product of the k^2 subsequences from the left and right node). Thus, the time complexity of the Segment-Tree algorithm is $O(nk^4)$ (k^4 operations at each node), which is linear in the number of points in the visualizations. The k^4 factor is acceptable since the number of SegmentGroups in the ShapeQuery is often very small (rarely going above 5). Moreover, in practice, not all combinations of subsequences results in a valid subsequence for a ShapeQuery, and so the actual merges are much less than k^4.

4.2.3 Two Stage Collective Pruning

Even though the Segment-Tree algorithm is linear in the number of the visualization points, the execution time for finding the top k visualizations can still be non-interactive, especially when the number of visualizations to explore is large. In order to further improve the performance, we apply a two-stage optimization to prune visualizations that are guaranteed to be not in top k. The optimization is based on the following observation.

Property 4.4 (Segment Score Boundedness) *The optimal score for a* ShapeSegment *over the entire Segment-Tree is bounded between the maximum and minimum possible scores of the* ShapeSegment *at any given level in the Segment-Tree as defined in Table 4.3.*

Proof (sketch): Consider two adjacent regions A and B represented by different nodes at a given level i in the Segment-Tree. Let the slopes of line-segments fitted over them be θ_1 and θ_2 respectively. Then, the slope θ of the line-segment fitted over the combined region AB will lie between θ_1 and θ_2, according to the property of lines. As discussed in Section 4.1.1, the scores of patterns are proportional to the slope of the line segment, and therefore the score over the parent node representing the combined region AB in level $(i-1)$ must also be bounded by the scores over the

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 level i nodes
FLAT	max across all level i nodes if all $\theta > 0$ or all $\theta < 0$; otherwise 1	min across level i nodes
$\theta = x$	max across all level i nodes if all $\theta > x$ or $\theta < x$; otherwise 1	min across level i nodes

Table 4.3: Bounds on overall scores for different patterns based on scores at a given level i in the Segment-Tree

children regions. Considering all nodes in level i, the optimal score in level $i-1$ is, thus, bounded by the maximum and minimum possible scores in level i. Moreover, as we move up the tree, we combine smaller regions to form bigger regions, and therefore the difference between the maximum and minimum bounds should be non-increasing. Hence, the overall optimal score for a given ShapeSegment should be bounded by its maximum and minimum score at any given level in the Segment-tree.

Now, from Property 6.2 and 6.3, at a given level in the Segment-Tree, we can get the upper and lower bounds on the scores of the CONCAT by using the maximum and minimum scores respectively for all ShapeSegments. Similar to the bounds on individual ShapeSegments, the difference between the bounds decreases as we move up the Segment-Tree. We formally state the following theorem.

Theorem 4.3 *Given the scores at a particular level in the Segment-Tree, the overall score for a CONCAT operator is bounded between the scores obtained by applying CONCAT operation on the maximum and minimum scores for each of the constituent* ShapeSegments. *Additionally, the difference between the upper and lower bounds on the score of decreases as we move up the Segment-Tree.*

We use the above theorem to come up with an optimal pruning algorithm. Often, the bounds on the overall score are often very loose at the bottom-most levels of the Segment-Tree, since the bottom level nodes are all small-length line-segments with varying slopes. Therefore, we precede the pruning stage with another light-weight stage to come up with tighter lower bounds for top k scores. We describe these two stages below.

Stage 1: Identifying tighter Lower Bounds. For each visualization, we first execute the ShapeQuery over a fixed subset of equidistant points in the visualization to quickly come up with the tight

lower bound on the overall score. This is done before we start computing scores for the leaf nodes in the segment-tree, to identify a good estimate on the lower bounds of the scores.

Stage 2: Refining and Pruning. In this phase, instead of processing each visualization completely in one go, we process visualizations in rounds. In each round, we process a few levels of Segment-Tree for all of the visualizations simultaneously, and incrementally refine the upper and lower bounds on their scores. Before moving on to the upper levels, we prune the visualizations that have lower bounds higher than the top-k upper bound score. Since the lower bounds estimated at the lower levels in Segment-Tree are often poor (much less than the ones estimated in Phase 1), we leverage the estimated bounds from stage 1 for more effective pruning in the initial rounds.

CHAPTER 5: USER AND CASE STUDY

In this chapter, we evaluate our ShapeSearch system with carefully designed user study, focusing on studying the usability of our system and gain insights from domain experts. We show that our system receives positive feedbacks from majority of the participants when performing real-world tasks comparing to other baseline system. We also provide a case study to detail the insights that we have gain through an open-ended evaluation with two domain experts from bioinformatics.

5.1 USER STUDY

We conducted a user study to evaluate the utility of ShapeSearch relative to QueryBySketch systems, on its ability to effectively support pattern matching tasks in trendlines. We also wanted to understand and categorize tasks for which users prefer each of the specification mechanisms: sketch, natural language, and regular expression. We first describe the user study methodology, followed by our key findings from the study.

5.1.1 User Study Methodology

Participants. We recruited 12 participants (7 male, 5 female) with varying degrees of expertise in data analytics. Out of the 12 participants, 7 were graduate students, 3 were undergraduate, 1 post-doctorate, and 1 university employee. All the participants reported working daily with data using tools such as Excel, Tableau, or programming languages such as R, Python, and Matlab.

Baseline. For the purposes of our study, we explicitly wanted to do a head-to-head qualitative and quantitative comparison of ShapeSearch tool, primarily the natural language and regular expression interfaces, with QueryBySketch systems such as Zenvisage, and thus we developed a baseline tool that replicated the capabilities of these systems with a styling scheme identical to the ShapeSearch tool to control for external factors. The tool allowed users to specify the X-axis, Y-axis, dimensions, and filters, and search for patterns via sketching on a canvas. Similar to the functionalities supported in tools like Zenvisage, users could specify either the Euclidean distance or DTW distance measure for pattern matching. In addition, they could zoom into a specific range

Tool	Accuracy (%)	Latency (s)	Preference (%)
QueryBySketch	79	184	9
ShapeSearch	91	114	91

Table 5.1: Overall results

	QueryBySketch		ShapeSearch		Preference			
Tasks	Accuracy (%)	Latency (s)	Accuracy (%)	Latency (s)	Sketch (%)	NL (%)	Regex (%)	NL or Regex (%)
Exact Trend Matching	96	88	89	112	83	25	50	67
Sequence Matching	81	135	98	75	17	75	67	92
Sub-Pattern Matching	85	115	93	100	50	17	83	83
Location Invariant Matching	74	117	95	90	42	67	83	100
Width Specific Matching	50	210	94	100	0	75	92	100
Multiple X or Y constraints	69	270	97	81	17	50	92	100
Trend Characterization	81	250	93	160	8	83	50	92
Complex Shape Matching	69	170	85	235	67	33	50	58

Table 5.2: Task Accuracy, Task Completion Time (Latency), and Tool Preferences

of the visualization to constrain the part of visualization for pattern matching.

Dataset and Tasks. We designed our user study tasks with the case studies from Section 1 in mind, on the Weather and Dow Jones stock datasets from the UCI repository [24] — we selected these datasets so that participants could understand and relate to the tasks they performed. At a high level, the tasks together involved the following eight commonly used pattern matching operations: 1) Exact trend matching (e.g., find trends similar to the trend of a city or stock) , 2) Sequence matching (e.g., find cities with temperature rising, remaining flat and falling), 3) Sub-pattern (motif) matching (e.g., find other stocks that depicted a common sub-pattern found in stocks X, Y, and Z (4) Location invariant matching (e.g., find cities with 2 peaks) 5) Width specific matching (e.g., find cities with maximum rise or fall in temperature over 3 months, peaks with a width of 3 months), 6) Multiple X or Y constraints (e.g., find stocks with prices rising in a range of 30 to 60 in march, then falling in the same range over the next month) of), 7) Trend Characterization (e,g., find stocks with typical weather patterns, stock price patterns), 8) Complex Shape Matching (e.g., stocks with head and shoulder pattern, cup-shaped patterns, W-shaped patterns). Together these tasks involved both exploratory search as well as direct / specific pattern search queries, which helped us better gauge the effectiveness of individual interfaces.

Study Protocol. The user study was conducted using a within-subjects study design, forming three phases. First, participants described their previous experience with data analytics tools. Next, participants performed exploration tasks using QueryBySketch (Tool A) and the ShapeSearch 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 measured their satisfaction levels and preferences, as well as answered open-ended questions on the strengths and weaknesses of interfaces with respect to each other and in comparison with other data analytics tools they used in their work.

Metrics. From data that we recorded during the study, we collected the following metrics: task completion time (latency), accuracy of participants‘ answers, and the usability ratings and satisfaction level from the survey results. In addition, we analyzed open-ended subjective answers given by answers during the interview sessions.

5.1.2 Key Findings

Now, we describe the key findings that we discovered through our analysis of the metrics collected during the study.

Accuracy and Speed. Table 5.1 depicts the overall accuracy of answers given by participants, and the time taken by participants to complete the tasks using QueryBySketch and ShapeSearch. ShapeSearch helps participants achieve higher accuracy (91%) over QuerySketch (79%) in about 40% lesser time, a significant improvement both in accuracy and task completion times. Overall the lower accuracy for QueryBySketch can be attributed to the limited mechanisms offered by the tool for expressing their patterns (described in a more detail below), and therefore participants had to browse through large collections of visualizations. On the other hand, ShapeSearch is able to accept more fine-grained user input, and show more relevant visualizations earlier in the results, and therefore participants were able to retrieve more accurate answers with less effort.

Expressivity. In order to further understand the significant difference in accuracy and task completion time between the two tools, we aggregated accuracy and latency results on a per-task basis as depicted in Table 5.2. We further analyzed the answers to the survey and interview questions, and found that accuracy and speed results were closely related to the expressivity or capabilities of the tool. The more expressive the tool, the more easily the desired pattern can be expressed within the query and hence less is the manual exploration involved in searching for specific visualizations.

While searching for exact trends or complex shapes that cannot be characterized using a combination of high-level trends, participants performed marginally better when using QueryBySketch tool. However, for other tasks that involve sequence matching, trend characterization, multiple X or Y constraints, location invariance, and fixed length patterns, they had to resort to multiple trial and steps. For queries involving such tasks, participants tended to compromise the accuracy of their results by searching for nearest alternate query that the tool could express. On the other hand, ShapeSearch helped participants achieve significantly better results in much less time for these queries. For queries involving subpattern or motif search, DTW distance helped automate the search to some extent, but the results were still worse than ShapeSearch. On a 5-point Likert scale ranging from strongly disagree (1) to strongly agree (5), participants rated QuerybySketch 2.8 and ShapeSearch 4.2 when asked about how they well could express the patterns they intended to search.

User Preference. When we asked participants about which tool they would prefer to use for pattern matching tasks in their workflow, 11 out of 12 participants preferred ShapeSearch over QueryBySketch. Participant P1 said "*It's hard to for me use TOOL A (QueryBySketch) to search for multiple patterns over disjoint parts of the visualization at the same time*". In addition, we also

explicitly asked users via survey questions to select one or more of the three specifications: sketch, NL, regex for each of the pattern matching task listed in Table 5.2. As shown in the table, users preferences were consistent with accuracy and speed results.

Regex vs Natural Language. Out of 11 participants who preferred ShapeSearch over QueryBySketch, 7 said they would opt for regex over natural language, if they had to choose one. This was surprising given 5 of these participants mentioned they did not have any prior experience with regular expressions like languages. 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*". 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 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 4.2.

Line Segment Approximations. When asked about the effectiveness of using line segments for approximating trend lines and using them the match pattern queries, the average response was positive with a rating of 4.3 on a scale of 5. Participant 4 said "*Green lines are good, they make me more confident, help me understand visualizations 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 visualization ...*".

Feedback on Improvements Participants suggested several improvements to make the tool more useful in their workflows. For example, a large number of participants wanted to support more mathematical patterns by default like concave, convex, exponential. For line fittings, participants suggested we use different colors for lines that match the queried pattern. Syntax for regex can be made simpler by removing square brackets, supporting better symbols for primitives and operators, and supporting validation and auto-correction. A few participants wondered if the tool could also support statistical measures such as entropy, correlations. Finally, a number of participants suggested that we could recommend queries and visualization based on learning from historical queries.

5.2 CASE STUDY : GENOMIC DATA ANALYSIS

For a more open-ended evaluation of ShapeSearch, we invited two researchers R1 (a 5th year graduate student) and R2 (a 2nd year graduate student) from bioinformatics. Both researchers

perform pattern analysis on genomic data on a daily basis using combination of spreadsheets and scripting languages such as R. We asked participants to send us their data so that we could ensure the ShapeSearch operates on their dataset as expected, and have it pre-loaded in our database before the study.

We started the study with a 15 minute introduction and demo of the ShapeSearch tool on the Weather data set that we had also used for the user-study, and asked participants to perform a few pattern matching tasks to help them familiarize with the tool, as well as to ensure they understood the functionalities of the tool.

Next, we encouraged participants to use ShapeSearch to explore their data. Since we were interested in seeing if they could create queries that reveal previously unknown insights, we requested our participants to think aloud, explain to us what kind of queries they were constructing, whether the results confirmed some already known fact, and how they currently performed similar exploration using existing tools. This also helped us ensure their mental model was matching with what the queries actually expressed. Each session went for 75 minutes.

5.2.1 Results

Usability. Both the participants were able to grasp the functionalities of the system after a 15 minute introduction and demo session without much difficulty. They answered questions that we asked to test their understanding during the end of the introduction with 100% accuracy. There were positive reactions from the participants during the introduction and demo like "oh, this feature is cool, ... something that we frequently do", "I like that we can enter and change our queries so quickly, and also see the results at the same time...". Both participants concurred that the tool was easy enough for them to use more often, unlike writing R scripts and Spreadsheets they currently used in her workflow. More importantly, using succinct declarative queries, they could interactively explore large number of gene groups, depicting a variety of gene expression patterns.

Insights. Participants made several insights while working on their own data.

Both R1 and R2 were able to query for genes with differential expressions over time. During the initial part of the study, R1 issued natural language queries to search for genes that suddenly start expressing themselves as some point, and then gradually stop expressing. These expression changes in genes signify an effect of external stimulus such as a chemical or a treatment. Before the treatment, the affected genes are stable with low expression, immediately after the treatment they suddenly get expressed, and then as the effect of treatment subsides, the expression reduces gradually. Thereafter, R1 was interested in understanding the variations in expression rates. For instance, expressions in certain groups of genes rise and fall much faster, while in others expression changes are gradual. In order to search for these patterns, she interactively adjusted the width of

patterns in her queries. R1 could also see groups of genes that show similar changes in expression over time, indicating they regulated similar cell mechanisms.

Another group of genes that R2 was interested in searching using regex were the ones that increase with a slope of 45° until a certain point, after which they remain high and flat. Then, he wanted to look at those genes that depicted the inverse behavior, ones that start high and then gradually reduce their expression and remain low and flat. These changes are often symbolic of fundamental or permanent changes in cell mechanisms (e.g., aging). For instance, stem cells have two functions: self-renewal during early stages they do when they repeatedly generate themselves and show higher expression, and differentiation during later stages when they differentiate into other types of cells such as muscle cells, and have low expression. While exploring these patterns, R2 discovered two genes, namely *gbx2* and *klf5*, and told that these two genes indeed have similar functionality at any point in time and are actively researched by scientists working in stem cells. He could further discover other genes such as *spry4* and said not many researchers know that *spry4* had similar functionalities to that of *gbx2* and *klf5*.

During the end of her study, R1 wanted to see if the tool can help her find genes with unexpected expression patterns. She mentioned that it is rare to see a gene with two peaks in their expressions within a short window. However, on searching for such genes via natural language, she found a gene named "pvt1" having two peaks within a short time duration of 10 points, which was strange for her. She thought either it's because of some preprocessing error, or some rare activity happening in the cell. Surprised by this result, she went ahead and searched for other patterns that she thought were outliers (e.g., three peaks, always increasing).

Natural Language vs Regex. R1 said she will start with natural language, and then make the query more complex with the help of correction panel, which she thought provides a form-based mechanism for expressing regular expression. Finally, she said she will directly use regex only when the pattern she wants to search is too long, and involves multiple constraints.

R2, on the other hand, preferred regex over natural language and 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 how the system would execute the query.

Challenges and Problems. Participants faced a few challenges during exploration. They wanted to switch back and forth between queries without them having to remember and reissue queries. In addition to improving the coloring of the fitted lines, they wanted to understand in more detail on how the matching scores were computed and if they could tweak the scoring according to their needs.

CHAPTER 6: EXPERIMENTS AND EVALUATION

In this chapter, we evaluate our ShapeSearch execution engine on real datasets, focusing on runtime and accuracy. We show that the Segment-Tree based algorithms from Section 4.2 provide substantial speedups as compared to the optimal dynamic programming approach, with reasonable trade offs in accuracy. We also varied the characteristics of ShapeSearch queries to observe their impact on optimizations.

Algorithms. We evaluate the performance of the following algorithms, (i) Dynamic Programming, the optimal approach from Section 6.2.1 (ii) Segment Tree: the pattern-aware scoring approach from Section 6.2.2 (iii) Segment Tree with Pruning: the two-stage collective pruning approach from Section 6.3. (iv) Greedy: we also implemented a greedy approach for scoring, where we start with segmenting visualizations into k (i.e, number of SegmentGroups) equal sized segments, and then move each of the break points to the left or right by half the distance with their adjacent break points. We repeat this until we don't see any improvement in the overall score. In all experiments, we consider Dynamic Programming to be our baseline.

Datasets and Queries We evaluate ShapeSearch on five publicly available real-world datasets — Weather, Worms, 50 Words, Haptics from the UCI repository [24], and Real Estate dataset from zillow.com [25]. We summarize the characteristics of these datasets in Table 6.1. We issued multiple ShapeSearch queries with the following templates: $a \otimes b \otimes c$, $a \otimes (b \oplus (c \otimes d))$, $a \otimes b \otimes c \otimes d$, where a, b, c, and d represent a ShapeSegment without LOCATION primitives, and PATTERN primitive set to one of the following: UP, DOWN, FLAT, or $\theta = 45^\circ$ — for each ShapeSegment we tried all possible pattern primitives, and selected those queries which had at least 30 matching visualizations (those with scores > 0) in the dataset. We compared algorithms on top-k results, with maximum values of k set to 20. Thus, setting the threshold to 30 for selecting queries ensured that queries had sufficient matching visualizations.

Setup. 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, ignoring the results from

Name	Number of Visualizations	Number of X points
Weather	144	366
Worms	258	900
50 Words	905	270
Real Estate	1777	138
Haptics	463	1092

Table 6.1: Real World Dataset Characteristics

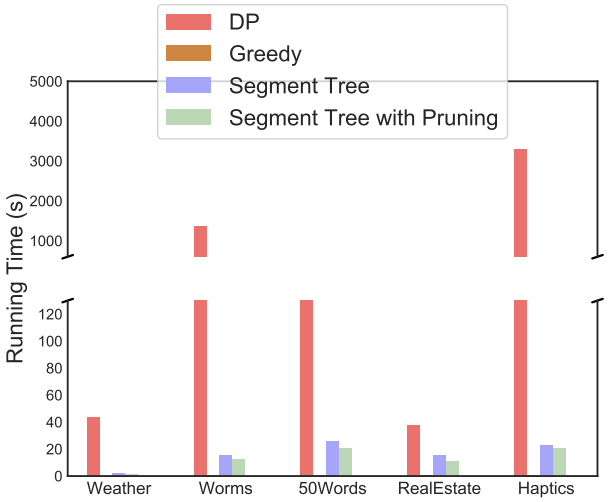

Figure 6.1: Average running time over five datasets

the first trial and taking the average across the rest of the five trials.

6.1 OVERALL RESULTS

Runtime. Figure 6.1 depicts the runtime performance for all the algorithms on each of the dataset. For a given algorithm, runtime numbers for all queries were close to each other, so we plot the average runtime of all queries on a dataset. We observed that compared to Dynamic Programming (DP), Segment Tree provides a 2X to 100X improvement in runtime. As discussed in Section 4.2.2.1, the runtime for DP is quadratic in the number of points in the visualization, and therefore it takes multiple 10s of seconds even for visualizations with only a few hundred points, and ShapeSearch queries involving 3 or 4 patterns. On the other hand, the runtime of Segment Tree is only linear in the number of points, and thus can scale to visualizations with even thousands of points. Segment Tree with Pruning further helps in reducing runtime by an additional 10-30% by identifying and pruning early low utility visualizations that look much different from the issued ShapeSearch query and are guaranteed to be not selected in final top K visualizations. The Greedy algorithm has the least runtime (< 2s) among all the algorithms. This is because it tries only a few segmentations before reaching a local maximum. As we discuss next, this improvement in runtime comes at the cost of extremely low accuracy in results.

Accuracy. We also examine the accuracy in the results for Segment Tree and Greedy approaches with respect to the results of optimal Dynamic Programming (DP) approach, which we consider as the ground truth. Table 6.2 depicts the results for top-k visualizations, where we vary k from 2 to 20. Each cell in the table consists of two numbers: the first number depicts the deviation in % of the score of kth visualization that Segment Tree or Greedy approach choses with respect to

	Top 2		Top 5		Top 8		Top 10	
Data Set	Greedy	Segment Tree	Greedy	Segment Tree	Greedy	Segment Tree	Greedy	Segment Tree
Weather	27.19 (0)	0.00 (2)	35.63 (1.5)	0.64 (4)	18.47 (3.5)	7.82 (7.5)	25.61 (4)	5.16 (9)
Worms	16.41 (0)	0.00 (2)	12.99 (0)	0.00 (4.5)	26.60 (1)	0.00 (7.5)	12.53 (2)	0.00 (9.5)
50 Words	5.60 (0.5)	0.61 (1)	0.77 (2)	1.92 (3)	2.87 (2)	3.83 (7.5)	13.81 (3)	2.65 (9.5)
Real Estate	2.18 (1)	1.54 (1)	1.80 (1.5)	3.21 (4)	6.29 (2.5)	1.62 (7)	7.84 (3.5)	1.32 (8)
Haptics	1.09 (0)	0.73 (2)	1.77 (0)	0.01 (4.5)	1.73 (0)	0.05 (8)	1.96 (0.5)	0.04 (9.5)
	Top 12		Top 15		Top 18		Top 20	
Data Set	Greedy	Segment Tree	Greedy	Segment Tree	Greedy	Segment Tree	Greedy	Segment Tree
Weather	19.66 (5)	0.99 (10.5)	5.78 (5.5)	2.44 (13.5)	14.00 (6.5)	0.00 (16.5)	6.54 (7)	0.33 (18.5)
Worms	11.56 (3)	0.01 (11.5)	23.98 (3.5)	1.77 (13.5)	27.00 (5)	0.00 (16)	5.10 (5.5)	0.63 (17.5)
50 Words	11.65 (4)	2.22 (11)	5.29 (4)	3.82 (13)	9.80 (4.5)	1.03 (16.5)	3.66 (5.5)	2.97 (18.5)
Real Estate	3.57 (4)	2.27 (10)	0.81 (4)	2.30 (13)	2.91 (4.5)	2.77 (15)	10.11 (6)	2.46 (16.5)
Haptics	1.51 (0.5)	0.04 (11.5)	1.89 (0.5)	0.00 (14.5)	0.99 (0.5)	0.01 (16)	1.74 (1)	0.01 (18)

Table 6.2: Average accuracy over five datasets. The first number measures the deviation in % the shape of visualization of visualizations chosen by algorithms with respect to optimal ones selected by DP. The second number in parenthesis lists the number of common visualizations in top-k with the optimal top-k

the score of kth visualization that DP selects, indicating how off the shape of chosen visualization from Segment Tree or Greedy is from the one from DP. The second number (in parenthesis) is the number of visualizations selected by Segment-Tree or Greedy that are missing in top k visualizations of DP. We observe that the deviation in shapes of visualizations selected by Segment-Tree is extremely low (<2% on average) as compared to optimal ones. In fact, the top k visualizations output by Segment Tree are almost same as those output by DP (never off by more than 2 visualizations). These numbers and the runtime results discussed above indicates that **Segment Tree approach can provide much faster results compared to optimal approach with almost similar accuracy**. Greedy approach, on the other hand, incurs much more severe penalty in accuracy, outputting visualizations which are much different from the optimal ones.

6.2 VARYING CHARACTERISTICS OF SHAPESEARCH QUERIES

We also evaluated the efficacy of our Segment Tree-based optimizations with respect to three different characteristics of a ShapeSearch query: (i) n, the number of data points in each visualization, (ii) k, the number of patterns in the ShapeSearch query and (iii) N, the number of visualizations in dataset. To control for all variables except these characteristics, we used Worms (for varying n), Weather (for varying k) and Real Estate (for varying N. First, for varying n, we process each visualization in Worm dataset to take the first 50, 100, 200, ..., 900 points to create ten sub datasets and then run experiments on each of them. Similarly, for varying N, we take the first 100, 200, ..., 1000 visualizations in the Real Estate dataset to create ten sub datasets. For the first and third experiments, we issue the same queries as used in last section, while for varying k experiments,

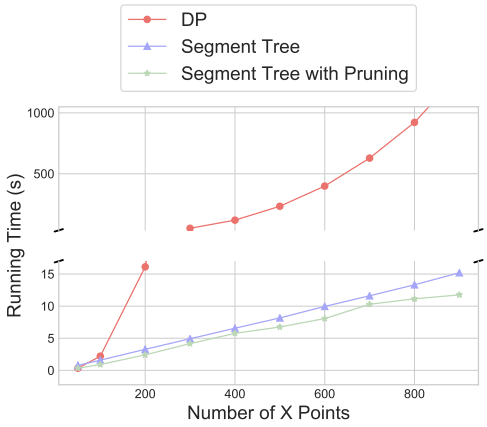

Figure 6.2: Running time vs number of X Points

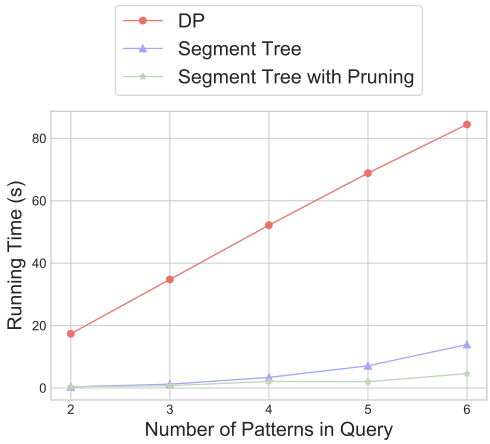

Figure 6.3: Running time vs number of patterns in query

we compose queries of length to 2 to 6 via concatenation of queries used in the previous section. For these experiments, we exclude the discussion on Greedy approach because it's extremely low accuracy makes it undesirable over Segment Tree approach, despite having faster execution times.

Impact of number of data points. Figure 6.2 shows the performance of DP, Segment Tree, and Segment Tree with Pruning on Worm dataset as we increase the number of data points in the visualizations from 50 to 1000. With the increase in data points, the overall runtimes increased for all approaches because the amount of processing per ShapeSearch query increased. Nevertheless, Segment Tree showed better performance than Dynamic Programming after 100 data points due to fact that the runtime for Segment Tree approach is less sensitive (linear) to the number of data points than that of Dynamic Programming (quadratic). In the meantime, Segment Tree with pruning further performed better than Segment Tree since it skips the processing of upper levels of the Segment-Tree for low utility visualizations.

Impact of number of patterns. We varied number of patterns in the query from 2 to 6 as depicted in Figure 6.3. As the number of patterns in the query grew, the overall runtimes for different

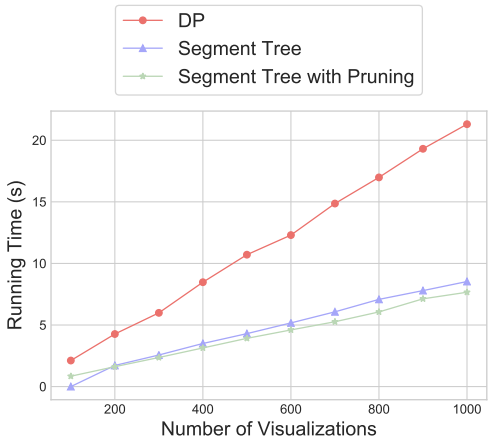

Figure 6.4: Running time vs number of visualizations

approaches also grew. Even though the runtimes for SegmentTree grew much faster (k^4) than DP (k) the overall times were much smaller since the number of data points in visualizations plays a more dominant role. In addition, the growth for Segment Tree with pruning is more flat compared to that for Segment Tree because more patterns in the query help prune more visualization early by improving the pruning threshold, thereby offsetting the impact of the increase in processing time per visualization.

Impact of number of visualizations. We increased the number of independent visualizations from 100 to 1000 with a step size of 100 to observe the impact on runtimes of our optimizations; the results are depicted in Figure 6.4. We found that while the overall runtime for all approaches did grow linearly with the number of visualizations in the dataset as expected, the runtimes for Segment Tree with Pruning tend to be better than that for Segment Tree, and the gap between the two increases as the number of visualizations in the collection increase. This is because Segment Tree with pruning can prune a large number of utility visualizations with respect to the issued query, the number of which increases as the size of the collection goes higher.

CHAPTER 7: CONCLUSION AND FUTURE WORK

In this thesis, we developed ShapeSearch, a visual analytics system to address the challenges encountered in interactively exploring a large collection of trendlines. Specifically, our system supports multiple mechanisms for helping users intuitively express and search for desired patterns. We first proposed a ShapeQuery *algebra* that abstracts key shape-based primitives and operators that are often of interest in real-world trend line visualizations. Then we developed both a natural language interface and a regular-expression-based interface to accommodate user queries with different levels of complexity. A sketching interface, which is similar to those from typical visual query systems, is also provided for better usability. Finally, we explored several algorithms for executing ShapeQueries and propose our efficient Segment-Tree-based algorithms. We then evaluated them via multiple experiments and also performed a user study to evaluate the end-to-end usability of the ShapeSearch system.

In the future, we wish to further explore and improve the efficiency of our system for better interactivity and also provide complete support to the entire ShapeQuery algebra for all of the discussed execution algorithms, so that users can have the flexibility to switch between different options and choose the best combination of interface and algorithm for their use cases.

REFERENCES

[1] "Tableau public (www.tableaupublic.com/)," [Online; accessed 3-March-2014]. [Online]. Available: www.tableaupublic.com/

[2] 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.

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

[4] 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. International Society for Optics and Photonics, 2005, pp. 175–187.

[5] M. M. Zloof, "Query-by-example: A data base language," *IBM Systems Journal*, vol. 16, no. 4, pp. 324–343, 1977. [Online]. Available: http://dx.doi.org/10.1147/sj.164.0324

[6] D. J.-L. Lee, J. Lee, T. Siddiqui, J. Kim, K. Karahalios, and A. Parameswaran, "Accelerating scientific data exploration via visual query systems," *arXiv preprint arXiv:1710.00763*, 2017.

[7] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, *Fast subsequence matching in time-series databases*. ACM, 1994, vol. 23, no. 2.

[8] 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*. Citeseer, 1995, pp. 490–501.

[9] 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*. IEEE, 1996, pp. 536–545.

[10] 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.

[11] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh, "Searching and mining trillions of time series subsequences under dynamic time warping," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2012, pp. 262–270.

[12] 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*. Springer, 2000, pp. 347–357.

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

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

[15] 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*. ACM, 1999, pp. 282–286.

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

[17] L. Rabiner, A. Rosenberg, and S. Levinson, "Considerations in dynamic time warping algorithms for discrete word recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 26, no. 6, pp. 575–582, 1978.

[18] J. Paparrizos and L. Gravano, "k-shape: Efficient and accurate clustering of time series," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1855–1870.

[19] 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.

[20] 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*. ACM, 2015, pp. 489–500.

[21] 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*. ACM, 2016, pp. 365–377.

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

[23] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit." in *ACL (System Demonstrations)*, 2014, pp. 55–60.

[24] "Uci repository," [Online; accessed 30-Oct-2015]. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/

[25] "Zillow real estate data (http://www.zillow.com/research/data/)," [Online; accessed 1-Feb-2016]. [Online]. Available: http://www.zillow.com/research/data/