DOI:10.1145/3535337

To view the accompanying Technical Perspective, visit doi.acm.org/10.1145/3535336

Expressive Querying for Accelerating Visual Analytics

By Tarique Siddiqui, Paul Luh, Zesheng Wang, Karrie Karahalios, and Aditya G. Parameswaran

Abstract

Data visualization is the primary means by which data analysts explore patterns, trends, and insights in their data. Unfortunately, existing visual analytics tools offer limited *expressiveness* and *scalability* when it comes to searching for visualizations over large datasets, making visual data exploration labor-intensive and time-consuming. In this work, we introduce the problem of visualization search and highlight two underlying challenges of search enumeration and visualization matching. To address them, we first present our work on Zenvisage that helps enumerate large collections of visualizations and supports simple visualization matching with the help of an interactive interface and an expressive visualization query language. For more fine-grained and flexible visualization matching, including search for underspecified and approximate patterns, we extend Zenvisage to develop ShapeSearch. ShapeSearch supports a novel *shape querying algebra* that helps express a large class of pattern queries that are hard to specify with existing systems. ShapeSearch exposes multiple specification mechanisms: sketch, natural-language, and visual regular expressions that help users easily issue shape queries, while applying query-aware and perceptually-aware optimizations to efficiently execute them within interactive response times. To conclude, we discuss a number of open research problems to further improve the usability and performance of both Zenvisage and ShapeSearch.

1. INTRODUCTION

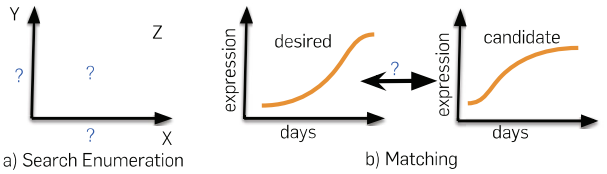
Data visualization is the primary means *via* which data analysts—many of whom have limited programming skills—explore their data. While the usability and visual encoding capabilities of data visualization tools such as Tableau and Excel have undergone a massive evolution over the years, when it comes to searching for patterns, trends, and insights in large and complex datasets, these tools are severely limited. The state of the art for data analysts, especially non-programmers, is to load their data into a visualization tool and repeatedly generate visualizations until the desired patterns or insights are identified. Unfortunately, this repeated process of manual examination to scour for desired insights becomes painful, tedious, and time consuming as the size and complexity of datasets increase. Even on moderately sized datasets, a data analyst may need to examine as many as tens of thousands of visualizations, all to test a single hypothesis, a severe impediment to data exploration. We characterize this problem of *visualization search* using examples from genomics data analysis.

Motivating example. *Genomic researchers often study genes, for example, how genes affect clinical trial outcomes, how the behavior of genes gets affected on specific medications. As an example, given a dataset consisting of clinical trial outcomes (positive vs. negative), researchers often want to find genes that can visually explain the differences in these outcomes. To do so, current tools require researchers to manually generate tens of thousands of scatter plots—with the x- and y-axes each referring to a gene, and each outcome depicted as a point in the scatterplot—to determine whether the outcomes can be clearly distinguished in the scatter plot.*

Similarly, researchers study changes in gene expression while investigating the impact of drugs on disease treatment. For doing so, they often explore trendline visualizations, one corresponding to each gene, with the x-axis as days, and the y-axis as the expression values. For example, when influenced by an external factor, a gene can get induced (up-regulated), or repressed (down-regulated), or can have both patterns within a certain time window. Based on their domain understanding, researchers first hypothesize the expected change in expression that an affected gene should depict. They, then, generate thousands of visualizations, one for each gene, and manually inspect them for the hypothesized patterns.

We have seen similar examples across a plethora of other domains such as astronomy, material science, and public health, where analysts manually peruse thousands of visualizations to search for each insight. In most of these scenarios, the common theme is the manual examination of a large number of generated visualizations to match a specific visual pattern. As depicted in Figure 1, there are two challenges to this visualization search problem. First, it is hard for users to specify the search space of visualizations they are interested in, which forces them to manually generate a large collection of visualizations. The space

Figure 1. Two components of the visualization search problem.



The original version of this paper is entitled "From Sketching to Natural Language: Expressive Visual Querying for Accelerating Insight" and was published in *ACM SIGMOD Record 50*, 1 (Mar. 2021), 51–58.

of visualizations is determined by the number of possible attributes for X and Y axes, aggregation functions, and possible subsets of data (denoted by the symbol Z in Figure 1a). This space grows exponentially as the size and the number of attributes in the data increase. The second component deals with *visualization matching*. Given a specific pattern of interest, users are typically interested in a subset of visualizations that closely match this pattern. Unfortunately, existing visualization tools are not *expressive* enough to capture either of the two components.

Our first attempt to address these challenges resulted in a visual data exploration system, Zenvisage.[8, 17, 18] Zenvisage takes as input a high-level specification of what the user wants and automatically identifies the relevant visualizations. It supports an interactive interface that allows users to quickly search for simple patterns *via* sketching. For expressing more complex search enumeration and matching, Zenvisage supports ZQL—an expressive visualization exploration language that lets users operate over a collection of visualizations using a core set of primitives (e.g., comparison, filtering, sorting) based on visual patterns. With ZQL, users can express a complex visualization searching task using two or three lines.

While Zenvisage is an useful first step in solving the visualization search problem, the underlying challenge of visualization matching remains unsolved. In particular, Zenvisage uses standard similarity measures (e.g., Euclidean distance) for matching, thereby lacking sufficient *flexibility* to support search when the desired pattern is under-specified or approximate, for example, finding products whose sales is decreasing over some 3-month window, without specifying when, or those whose sales have many increasing and decreasing portions, without specifying when these portions occur, their magnitude, or their width. We note that such pattern-matching tasks are hard to express in most of the visual querying systems.

To support more flexible querying needs, we developed ShapeSearch,[19, 20] a pattern searching system that supports multiple mechanisms for helping users express and search for desired visual patterns. ShapeSearch incorporates an expressive shape query algebra consisting of shape-based primitives and operators for expressing a large variety of patterns in trendlines. We developed this algebra after discussions with domain experts, including those from astronomy and genomics, as well as studying a large corpus of pattern queries collected *via* Mechanical Turk.

ShapeSearch supports multiple specification mechanisms that are internally translated to a shape query algebra representation: ShapeSearch supports a *natural language interface*, coupled with a sophisticated parser and translator for translating them into the algebra. ShapeSearch also supports a *sketching interface* for simpler patterns and returns visualizations that precisely match the drawn trends. To support more complex needs, the system provides a *visual regular expression* language for issuing queries that cannot be easily expressed *via* natural language or sketching. The three interfaces can be used simultaneously and interchangeably, as user needs and pattern complexities evolve.

Finally, for ensuring interactive response times on *ad hoc* queries, ShapeSearch leverages a pattern-matching engine that relies on minimal preprocessing or indexing. Directly generating and processing a large collection of visualizations, where each visualization has thousands of values, can lead to a long response time. Instead, ShapeSearch uses *perceptually-aware* pattern scoring mechanisms and *query-aware* optimizations—which help prune a large number of visualizations and/or parts of visualizations, for effective and efficient pattern matching.

Outline. The rest of our paper is organized as follows. We first discuss our experiences from our prior work on Zenvisage that motivated us to develop ShapeSearch, describing a simple interactive interface and ZQL (Section 2). We then give an overview of ShapeSearch, discussing how it addresses the limitations of Zenvisage (Section 3). Next, we dive into the details of shape algebra that makes the core of ShapeSearch (Section 4). We then describe efficient algorithms for executing shape queries (Section 5). We discuss how we support natural language queries in ShapeSearch (Section 6). Finally, we discuss future directions to further improve the usability and performance of both Zenvisage and ShapeSearch (Section 7).

2. ZENVISAGE

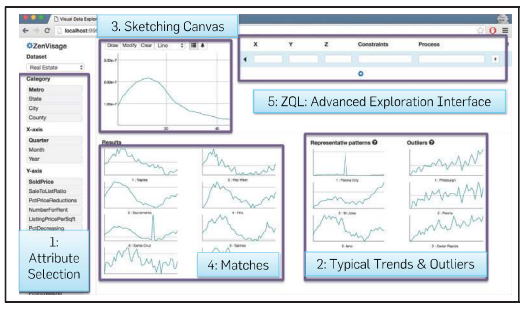
Zenvisage is a visual analytics system that supports an interactive interface for searching for visualization with simple patterns, along with an expressive query language for more complex queries. We briefly discuss each of these modes and then describe the findings from our user evaluation.

2.1. Interactive search interface

Figure 2 shows the interactive search interface of Zenvisage loaded with a real estate dataset.

Attribute selection. The first step is attribute selection (Box 1). Here, the user can specify the desired X-axis attribute, and the desired Y-axis attribute for the visualization(s) that the user is interested in. In this case, the user has specified the X axis as quarters (in other words, time) and the Y axis as the real estate sold price. Additionally, the user specifies the category: this is a variable indexing the space of candidate visualizations the user is operating over. Here, the selected category is "metro"—indicating a metro area or township. We depicted the category as "Z" in Figure 1a.

Figure 2. *Zenvisage* interactive visual query interface.



Summarization of typical and outlier trends. As soon as the user selects the X, Y, and category, immediately, Zenvisage populates Box 2 with typical or representative trends across categories, and outliers. In this case, there are three typical trends that were found across different metros (i.e., categories): one corresponding to a spike in the middle (Panama City), one to a gradual increasing trend (San Jose), and one to a trend that increased and then decreased (Reno)—most of the other trends were found to be similar to one of these three. The outlier visualizations (Pittsburgh, Peoria, Cedar Rapids) have a large number of seemingly random spikes.

Drawing or drag-and-drop canvas. Then, in Box 3, the editable canvas, the user can either draw a shape that they are looking for, or alternatively drag and drop one of the displayed visualizations into the canvas. In this manner, the user indicates that they would like to see a similarity search starting from the shape or pattern that they have drawn or dragged onto the canvas. The user is also free to edit the drawn pattern. In this figure, the user has drawn a trend, which is gradually increasing up and then gradually decreasing after that.

Similarity search results. As soon as the user completes an interaction in Box 3, Box 4 is populated with results corresponding to visualizations (on varying the category) that are most similar to the trend in Box 3, ordered by similarity. The system allows users to choose between three different similarity metrics. Currently, the three metrics Zenvisage provides are Euclidean Distance, DTW, and Segmentation.[18]

Overall, this interactive search interface satisfies simple pattern search needs *via* sketching and drag-and-drop, and provides context *via* representative and outlier patterns. However, it offers limited expressiveness when it comes to more complex data exploration needs. For instance, it is difficult to search for visualizations across a wide range of X and Y attributes (recall that before sketching, we need to set the X and Y axis to specific attributes) or compare two visualizations without using the drawing canvas (e.g., finding 2 products that have similar revenue and profit trends over years). Furthermore, one cannot specify multi-step queries involving search for multiple patterns simultaneously, for example, finding products with increasing sales trend in Europe but decreasing sales trend in the US. For supporting these more complex needs, we introduced a second mode, called ZQL, short for Zenvisage Query Language, that users can specify in Box 5 in Figure 4.

2.2. ZQL: A visualization querying language

ZQL is a high-level language that automates the manual data exploration process by allowing users to specify their desired visualization objective in a few lines. Instead of providing the low-level data retrieval and manipulation operations, users operate at the level of *sets of visualizations, and compare, sort, filter, and transform* visualizations as well as attributes—eventually visualized on either the X or Y axis, or used to select the set of data that is visualized.

We describe the capabilities of ZQL *via* two examples (depicted *via* Tables 1 and 2). Consider the first example where we want to find the states where the soldprice trend is most similar to the soldpricepersqft (i.e., sold price per square foot) trend. Table 1 depicts a 3-line ZQL query for this task. We first compose two collections of visualizations. The first row composes the first collection with X = year, Y = avg(soldprice), and Z = state.*, consisting of one visualization for each possible state. The Z column corresponds to the Category header in the previous section, indicating the space of visualizations over which the user is operating—in this case, the Z column is fairly simple, there is a single visualization, corresponding to each state. Similarly, the second row composes the second collection with X and Z column stay similar and Y is set to avg(soldpricepersqft).

Table 1. A *ZQL* query retrieving visualizations for a state where the soldprice over year trends are most dissimilar to the soldpricepersqft trend.

Name	X	Y	Z	Process
f1	year	soldprice	z1 <– 'state'.'*'	
f2	year	soldpricepersqft	z1	z2 <– $argmax_{z1}$ $[k = 1]D(f1, f2)$
*f3	year	{soldprice, soldpricepersqft}	z2	

Table 2. A *ZQL* query retrieving two different visualizations (among different combinations of x and y) for states of CA and NY that are the most dissimilar.

Name	X	Y	Z	Process
f1	x1 <– *	y1 <– *	'state'.'CA'	
f2	x1	y1	'state'.'NY'	x2,y2 <– $argmax_{x1,y1}$ $[k = 1]D(f1, f2)$
*f3	x2	y2	'state'.['NY', 'CA']	

Once we have composed the two visualization collections (referred *via* f1 and f2), the Process column is used to compare, sort, and filter the visualizations between the collections. In this example, we iterate the visualizations for each state (notice the variable z1) in f1 and f2 and compare them using a functional primitive D, computing distance, *via* D(f1, f2). Then, argmin is a sort-filter primitive that sorts the states based on distance scores and selects the top 1 state with minimum scores. Finally, in row 3, we output the overall sales over year visualizations for the selected products as bar charts. The * in *f3 indicates that these visualizations are to be output to the user.

As another example, say we are interested in finding a pair of X- and Y-axes where the visualizations for two specific states "NY" and "CA" differ the most. For doing this, we write a ZQL query depicted in Table 2. In the first line, we fetch all visualizations for the states "NY" that can be formed by having different combinations of X- and Y-axes. Similarly in the second row, we retrieve all possible visualizations for the product "stapler." In the process column, we iterate over the possible pairs of X- and Y-axis values, compare the corresponding visualizations in f1 and f2, and finally select the pair of X and Y axis values where the two products differ the most. In the last two rows, we output these visualizations.

Overall, ZQL can capture a wide range of visual exploration queries, including drill-downs and filtering based on specific

patterns. We formally describe the expressive power of ZQL using a visual exploration algebra in our full paper.[17]

2.3. Takeaways from Zenvisage

Our findings from user studies[17] and case studies with collaborators in domains such as genomics, astronomy, and battery science[7, 8] demonstrate that Zenvisage enables faster and more accurate exploration compared with existing visualization tools such as Tableau, which require considerable manual exploration for finding visualizations with specific patterns. Users who had worked with MATLAB, Python, and R said that ZQL can lead to faster initial exploration of data without requiring to write a lot of code. Those having experience with SQL found ZQL a lot less complicated, less verbose, and faster when it comes to comparing subsets of data.[17] Similarly, our collaborating researchers have used Zenvisage for various findings, including the fact that a dip in a light curve was caused by malfunctioning equipment (for astronomy), the fact that a relationship between two specific physical properties of electrolytes was independent of a third one (for battery science), and for reproducing of characteristic gene expression profiles from a recent paper (for genetics).[8]

While Zenvisage offers a promising first step to the problem of painful manual exploration of visualizations, the underlying challenge of visualization search is far from solved. We discovered two main challenges. One pertains to the usability of ZQL. In order to leverage ZQL, domain experts need to learn a new querying language, a major hindrance to its broader adoption. Domain experts with prior experience with computational notebooks often expressed a need for transitioning between writing code and using ZQL abstractions. Additionally, instead of writing their queries in one step, users often intended to construct them in an incremental manner using prior queries as context. In Section 7, we discuss these issues and potential solutions in more detail, highlighting another system LUX[9] that partially addresses these issues.

The second challenge with Zenvisage deals with how visualizations are matched. For the rest of the paper, we focus on this challenge and present a new system ShapeSearch to address it.

The problem of flexible shape matching. Zenvisage as well as other visual querying tools[2, 10, 21] offer limited flexibility in terms of how a visualization is matched. For instance, visualization search often involves pattern matching where the desired pattern of interest is underspecified and approximate, for example, finding stocks whose prices are decreasing for some time, followed by a sharp rise, with the position and intensity of movements being left unspecified, or when the desired shape is complex, for example, finding gene expression profiles where there is an unspecified number of peaks and valleys followed by a flattening out. We highlight the key characteristics of such pattern-matching tasks below.

Fuzzy matching. Domain experts (i) typically search for patterns that are *approximate* and are often not interested in the specific details or local fluctuations as much as the overall shape, and (ii) they often *do not* specify or even know the exact location of the occurrence of patterns. For example, biologists routinely look for structural changes in gene expression, for example, rising and falling at different times (Figure 3a), characterizing internal biological processes such as the cell cycle or circadian rhythms, or external perturbation, such as the influence of a drug or presence of a disease.

Searching multiple simple patterns. We notice that domain experts often describe complex patterns using a *combination of multiple simple ones*. Each individual pattern is typically described using words such as "increasing," "stable," "falling" that are easy to state in natural language but hard to specify using existing query languages. Moreover, pattern-matching tasks often go beyond finding a sequence of patterns, requiring arbitrary combinations, for example, disjunction, conjunction, or quantification, with varying location or width constraints. Examples include finding stocks with at least 2 peaks within a span of 6 months, for example, the so-called "double/triple top" patterns that indicate future downtrends.[5]

Ad hoc and interactive querying. Pattern-based queries are often defined *on-the-fly* during analysis, based on other patterns observed. For instance, biologists often search for a pattern in a group of genes similar to a pattern recently discovered in another group.[8] Similarly, astronomers monitor the shape of the luminosity trends of stars over time to search for and characterize new planetary objects (Figure 3c). For example, a dip in brightness often indicates a planetary object passing between the star and the telescope.

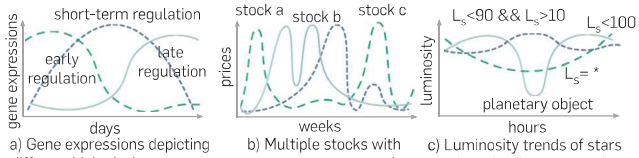
To support these characteristics, we developed ShapeSearch, described next.

3. SHAPESEARCH

ShapeSearch provides powerful yet flexible mechanisms for users to search for trendline visualizations with a desired shape. In this section, we first present an overview of ShapeSearch along with user experience.

ShapeSearch supports an interactive interface for composing shape queries. Figure 4 depicts this interface, with an example query on genomics data discussed in the introduction. Here, the user is interested in searching for genes that get suppressed due to the influence of a drug, depicted by a specific shape in their gene expression—first rising, then going down, and finally rising again—with three patterns: up, down, and up, in a sequence. To search for this shape, the user first loads the dataset[1] *via* form-based options on the left (Figure 4, Box 1), and then selects the space of visualizations to explore by setting the *x*-axis as time, the *y*-axis as expression values, and the category as gene. Each value of the category attribute results in a candidate visualization with the given *x*- and *y*-axis. Thus, the category attribute

Figure 3. Shapes characterizing real-world phenomena.



defines the space of visualizations over which we match the shape. ShapeSearch supports three mechanisms for shape specification—natural language, regular expressions (regex for short), and sketching on a canvas:

Sketching on Canvas. By drawing the desired shape as a sketch on the canvas (Figure 4, Box 2a), the user can search for visualizations that are *precisely* similar (using a distance measure such as Euclidean distance or dynamic time warping). After the user finishes sketching, ShapeSearch outputs visualizations that are similar to the drawn sketch in the results panel (Figure 4, Box 4).

Natural language (NL). For searching for visualizations that approximately match patterns, users can use natural language. For instance, as in Figure 4 Box 2b, the desired shape in the aforementioned genomics example can be expressed as *"show me genes that are rising, then going down, and then increasing."* Similarly, scientists analyzing cosmological data can easily search for supernovae (bright stellar explosions) using *"find objects with a sharp peak in luminosity."* We describe in Siddiqui et al.[20] how ShapeSearch translates natural language queries to a structured internal representation.

Regular expression (regex). For queries that involve complex combinations of patterns that are difficult to express using natural language or sketch, the user can issue a regular expression-like query that directly maps to the structured internal representation, consisting of ShapeSearch primitives and operations, described in detail in Section 4.

During exploration, users can choose specification mechanisms interchangeably based on the complexity of the query. For both NL and regex, ShapeSearch additionally supports an auto-complete functionality to guide users toward their target query. We use the term *user query* to refer to the submitted query using any of the specification mechanisms.

The ShapeSearch back-end parses and translates the user query into a ShapeQuery, a structured internal representation of the query consisting of operators and primitives supported in our algebra (Section 4). The back-end supports an ambiguity resolver that uses a set of rules for automatically resolving syntactic and semantic ambiguities, as well as forwards the parsed query to the user for further corrections and validation (Figure 4, Box 3). The validated query is finally optimized and executed by the execution engine (Section 4.3), and the top visualizations that best match the ShapeQuery are presented to the user in the result panel (Figure 4, Box 4). Next, we discuss a ShapeQuery algebra that makes the core of ShapeSearch.

4. SHAPE ALGEBRA

ShapeQueries help express a large variety of patterns over trendlines with a minimal set of primitives and operators. A ShapeQuery represents a *shape* as a combination of multiple *simple patterns*. A simple pattern can either be precise with specific location constraints, for example, matching $y = x$ between $x = 2$ to $x = 6$, or fuzzy, for example, roughly increasing, where the notion of the pattern is approximate and its location unspecified. Each simple pattern along with its precise or imprecise constraints is called a Shape-Segment. Complex shapes, for example, rising and then falling, are formed by combining multiple ShapeSegments using one or more *operators*. One can search for multiple patterns in a sequence (CONCAT, $\otimes$) or matching the same subregion of the trendline (AND, $\odot$), or one of many patterns matching a subregion (OR, $\oplus$), described later.

As an example, "rising from $x = 2$ to $x = 5$ and then falling" can be translated into a ShapeQuery `[x.s=2,x.e=5, p=up]` $\otimes$ `[p=down]` consisting of two ShapeSegments separated by a $\otimes$ operator. The first ShapeSegment captures "rising from $x = 2$ to $x = 5$"; the second expresses a "falling" pattern. Since the second must "follow" the first, the two ShapeSegments are combined using the CONCAT operator, denoted by $\otimes$. We now describe the shape primitives and operators that constitute the ShapeQuery algebra. Table 3 lists these primitives and operators.

4.1. Shape primitives and operators

A ShapeSegment is described using two high-level primitives: LOCATION and PATTERN. The LOCATION values can be skipped in order to match the PATTERN anywhere in the

Figure 4. *ShapeSearch* interface, consisting of six components. (1) Data upload, attribute selection, and applying filter constraints, (2) query specification: (2a) sketching canvas, (2b) natural language query interface, and (2c) regular expression interface, (3) correction panel, and (4) results panel.

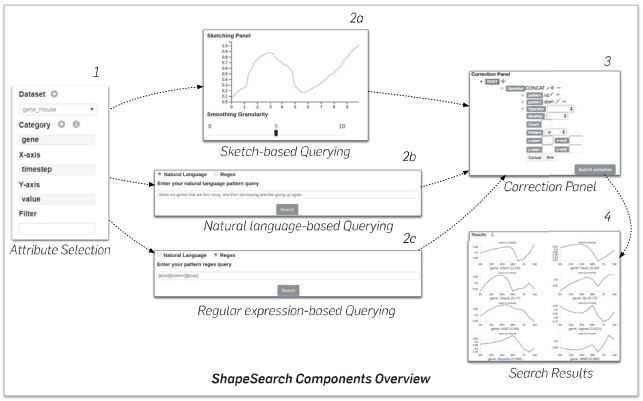


Table 3. Primitives and operators in ShapeQuery.

Symbol	Name	Type
x.s	START X VALUE	Location sub-primitive
y.s	START Y VALUE	Location sub-primitive
x.e	END X VALUE	Location sub-primitive
y.e	END Y VALUE	Location sub-primitive
v	SKETCH	Location sub-primitive
.	ITERATOR	Location sub-primitive
p	PATTERN	Primitive
$	POSITION	Pattern sub-primitive
m	MODIFIER	Primitive
>	MORE	Modifier value
>2	ATLEAST 2X	Modifier value
=	SIMILAR	Modifier value
$\otimes$	CONCAT	Operator
$\odot$	AND	Operator
$\oplus$	OR	Operator
!	OPPOSITE	Operator

trendline. Similarly, users can input the exact trendline to match or the endpoints of the ShapeSegments to match without specifying the PATTERN.

Specifying LOCATION. LOCATION defines the endpoints of the subregion of the trendline between which a pattern is matched: starting X/Y coordinate (`x.s`/`y.s`), ending X/Y coordinate (`x.e`/`y.e`). For example, [`x.s=2,x.e=10, y.s=10, y.e=100`] is a simple Shape-Query to find trendlines whose trend between $x = 2$ to $x = 10$ is similar to the line segment from (2, 10) to (10, 100). Users can also draw a sketch to find trendlines similar to the sketch, a functionality supported in other tools alluded to in the introduction.[2, 17] Shape-Search translates the pixel values of the user-drawn sketch to the domain values of the X and Y attributes, and adds the transformed vector of (x, y) values as a vector v in the ShapeQuery.

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

Combining PATTERNs. ShapeQuery supports three operators to combine ShapeSegments:

- CONCAT ($\otimes$) specifies a sequence of two or more ShapeSegments. For example, using [`p=up`]$\otimes$[`p=down`] one can search for genes that are first rising, and then falling. Note that $\otimes$ is one of the most frequently used operations, and we sometimes omit $\otimes$ between ShapeSegments, for example, [`p=up`] [`p=down`], to make it succinct to describe.
- AND ($\odot$) simultaneously matches multiple patterns in the same subregion of the trendline. Unlike CONCAT, all of the patterns must be present in the same subregion. For example, one can look for genes whose expression values rise twice but do not fall more than once within the same subregion.
- OR ($\oplus$) searches for one among many patterns in the same subregion of the trendline, picking the one that matches the subregion best. For example, one can search for genes whose expressions are either *up*- or *down*-regulated.

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

Expressing complex patterns. The aforementioned basic primitives and operators are powerful enough to express more complex ShapeSearch use cases. For instances, users may want to find specific shapes of specific width irrespective of their start location, for example, searching for cities with the steepest rise in temperature over a width of 3 months. To express such queries, ShapeSearch supports the ITERATOR (.), for example, [`x.s=.,x.e=x.s+3, p=up`] that iterates over all points in the trendline, setting each point as the start x position, with the x end position set to 3 units ahead. Internally, for a trendline of length n, this query can be rewritten as an OR operation over $(n - 3 + 1)$ ShapeSegments, where, for the ith Shape-Segment, `x.s=i` and `x.e=i+3`.

Similarly, one can search for trendlines where a pattern occurs a specific number of times using quantifiers, denoted by `q`. For example, [`p=up,q={1,2}`] can be used to search for trendlines where there is an increasing pattern at least once and at most twice. Quantifiers can be internally rewritten using an OR of one or more CONCAT operations. For example, the above query is rewritten as ([`p=*`]$\otimes$[`p=up`]$\otimes$[`p=*`])$\oplus$([`p=*`]$\otimes$[`p=up`]$\otimes$[`p=*`]$\otimes$[`p=up`]$\otimes$[`p=*`]). We discuss more complex patterns that can be expressed using ShapeQuery in our extended version.[20]

4.2. Semantics

A ShapeQuery Q operates on one trendline, V_i, at a time, and returns a real number, called *score*, between –1 and +1. It operates on V_i with the help of ShapeSegments $(S_1, S_2, ..., S_n)$ and operators $(O_1, O_2, ..., O_m)$. Each ShapeSegment S_i operates on $V_i^{p,q}$, a subregion of V_i starting at $p = x.s$ and ending at $q = x.e$ and returns a $score_i \in [-1, 1]$ using scoring functions we describe subsequently. One or more ShapeSegments are combined using operators such as $\otimes$, $\odot$, $\oplus$. Formally, an operator O_i takes as input the scores $score_1$, $score_2$, ..., $score_n$ from its n input ShapeSegments and outputs another $score_i$ using scoring functions that capture the behavior of the operators.

For both efficiency and effectiveness, ShapeSearch approximates each subregion with a line, using the slope to quantify how closely it captures any given ShapeSegment. As depicted in Table 4, ShapeSearch uses different scoring functions for each pattern primitive that transforms the slope to a value in [–1, 1] using a tan^{-1} function. For example, for an *up* pattern, the function returns a score between [0, 1] for a trendline with a slope from 0° to 90°, and a score of [–1, 0] for a slope of less than 0° (opposite of *up*).

For execution, ShapeSearch takes the entire trendline, the Abstract Tree Representation (AST) of ShapeQuery, and the list of scoring functions *ScrFunc* as in Tables 4 and 5 as inputs. If the root node of the ShapeQuery tree is a ShapeSegment, ShapeSearch directly computes the score of ShapeSegment on the specified part of the trendline. If the root node is $\odot$ or $\oplus$, ShapeSearch invokes each of the operands (i.e., child subtrees) to compute their scores on the subregion independently, combining the scores as per operator's functions. However, if the root node is a CONCAT with k operands, that is, child subtrees, ShapeSearch segments L into all possible

k subregions: $L_1, L_2, \ldots, L_k$ and then, for each segmentation, invokes the ith operand on ith segment. Finally, the maximum score across all segmentations is output.

4.3. Fuzzy ShapeQueries

A common subclass of ShapeQueries are *fuzzy* ShapeQueries, consisting of at least one ShapeSegment with missing or multiple possible values for *x.s* or *x.e*. Thus, for fuzzy ShapeQueries, a naive approach is to try all possible values of p and q, selecting the subregion that leads to the best score. This becomes prohibitively expensive as the number of points in the trendline increases. For a CONCAT with k operands, the exhaustive approach creates $n^{(k-1)}$ segmentations, where n is the number of points in the trendline.

The dynamic programming algorithm. We observe that for the CONCAT operation, the scoring of an operand on a given subregion is independent of scoring of other operands on other subregions.[20] We use this idea to develop a faster dynamic programming algorithm (DP) for scoring CONCAT operations over Shape-Segments. Formally, let $OPT(1, t, (1:j-1))$ be the best score corresponding to the optimal segmentation over the subregion between $x = 1$ and $x = t$ for the first $j-1$ operands, and $SC(t+1, i, j)$ be the score of the jth operand over the subregion between $x = t + 1$ and $x = i$. Then, the optimal segmentation $OPT(1, i, (1:j))$ for the first j operands over $x = 1$ and $x = i$ can be computed using the following recursion:

$$OPT(1, i, (1, j)) = \max_{t} \left\{ \frac{(j-1) \times OPT(1, t, (1:j-1)) + SC(t+1, i, j))}{j} \right\}$$

Unfortunately, even though the DP algorithm is orders of magnitude faster than the exhaustive approach, for trendlines with a large number of points, even a ShapeQuery with a single CONCAT operation can be slow, because of its quadratic complexity. We, next, discuss optimizations to further decrease the runtime of CONCAT operation on ShapeSegments.

A pattern-aware bottom-up approach. The DP-based optimal approach scores all possible subregions for each operand in the CONCAT operation. We observe that a more efficient approach could be to select end points to be those where the slope (or pattern) changes drastically. We first illustrate our intuition and then describe an algorithm that performs segmentation in a pattern-aware manner.

Intuition. As depicted in Figure 5, consider two subregions A on the left and B on the right for the trendline L. Say the trendline in subregion A is inverted V-shaped, that is, increasing until a point P and then decreasing. Now, for all possible segmentations where [p=up]'s subregion lies completely in A, there are the following possibilities for x.e of [p=up]: 1) [p=up]'s x.e point is before P. 2) [p=up]'s x.e point is after P. 3) [p=up]'s x.e point is at P.

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

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

Assumption 4.1 (Closure). *If a point is not locally optimal for any of the sub-expressions in the* CONCAT *operation (that is, a* CONCAT *on a sub-sequence of the operands), it cannot be* `x.s` *or* `x.e` *of a* ShapeSegment *in the optimal segmentation.*

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

At a high level, the algorithm starts by dividing the trendline into smaller contiguous subregions. Next, it selects locally optimal points (LOPs) over small subregions, followed by a bottom-up merging step that uses LOPs over small subregions to find LOPs over larger subregions. We discuss both the steps below.

Selection of LOPs. A point P is an LOP in a subregion for the sub-expression S_i if it is either the x.e of the first

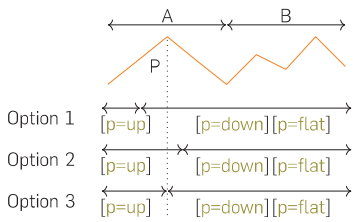
Table 4. Pattern scores.

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

Table 5. Operator scores.

Operator	Score
CONCAT	$\sum_{i=1}^{k} score_i / k$
AND	$min(score_1, \ldots, score_k)$
OR	$max(score_1, \ldots, score_k)$

Figure 5. Pattern-aware selection of LOPs.



ShapeSegment *or* x.s of the last ShapeSegment of S_i. For instance, in the above example, the LOP (P) in subregion A is the x.e value of [p=up] in the optimal segmentation of [p=up] ⊗ [p=down] in A. Since a CONCAT operation with k operands can have (k^2) sub-sequences, there can be a maximum of $2.k^2$ LOPs in A.

Merging. Next, we incrementally merge nodes in a bottom-up fashion to select LOPs over larger subregions. For example, in Figure 6, node 4 depicts the subsequences formed by combining subsequences from nodes 1 and 2, and node 5 depicts the subsequences formed by combining subsequences from nodes 3 and 4. When multiple subsequences in the children nodes generate the same subsequence in the parent node, we select the one with maximum score after concatenation (i.e., the one with the most optimal segmentation), thereby pruning out LOPs corresponding to non-selected subsequences. For example, at node 5, a⊗b can be computed from 1) a from node 3 and b from node 4, 2) a⊗b from node 3 and b from node 4, and 3) a from node 3 and a⊗b from node 4. Among these three concatenations, we pick the one that gives the maximum score. This merging process is repeated at each intermediate node. Finally, at the root node, we select the points that result in the maximum score for the entire sequence of operands. More details can be found in ShapeSearch extended version.[15]

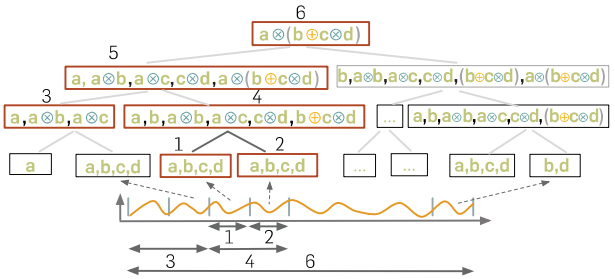
Given the closure assumption, we prove in ShapeSearch extended version[15] that the merging process leads to optimal segmentation and that the bottom-up algorithm with k CONCAT operands is optimal with a time complexity of $O(nk^4)$, that is, linear in the number of points in the trendlines.

5. NATURAL LANGUAGE TRANSLATION

A key component of ShapeSearch is the natural language parser that allows users to express ShapeQueries using natural language. We provide a brief overview of the three key steps involved in parsing and refer readers to Siddiqui et al.[15] for more details. We use the following natural language phrase for illustration: "show me the trendlines that are increasing from 2 to 5 and then decreasing."

Step 1. Primitives and operators recognition. Given a natural language query, the first step is to map words to their corresponding shape primitives and operators. For example, the above query is tagged as "show (noise) me (noise) the (noise) trendlines (noise) that (noise) are (noise) increasing (p) from (noise) 2 (x.s) to (noise) 5 (x.e) and then (⊗) decreasing (p)." In order to do so, we learn a linear-chain conditional-random field model (CRF)[6] and train it on the same 250 natural language queries we collected *via* Mechanical Turk (described in Shapesearch extended version[15]) for understanding query characteristics. For each word, we use its part-of-speech (POS) tags along with word-level context as features.

Figure 6. Bottom-up scoring of ShapeQuery.



Step 2. Identifying pattern value. For each of the words predicted of type p, for example, increasing and decreasing in the above query, we additionally map them to the corresponding semantic pattern supported in ShapeSearch, for example, "increasing" is mapped to p=up. For this mapping, ShapeSearch computes the similarity between the specified word and synonyms of the supported patterns, first using edit distance and then using wordnet. The semantic pattern with the highest similarity between any of its synonyms and the specified word is selected.

Step 3. ShapeQuery generation and ambiguity resolution. Next, we group primitives and operators into a ShapeQuery, first grouping all primitives between two operators into a single Shape-Segment. For the example query, the primitives are grouped as follows: [increasing (p=up), 2 (x.s), 5 (x.e)] and then (⊗) [decreasing (p=down)]. In some cases, this leads to incorrect grouping of primitives, for example, two patterns in the same ShapeSegment. There could further be semantic ambiguity because of wrong entity tagging, for example, decreasing (p=up) from 5 (y.s) to 10 (y.e) where x.s and x.e values are wrongly tagged as y.s and y.e, respectively. ShapeSearch uses rule-based transformations that try to reorder and change the types of entities to get a correct and meaningful ShapeQuery.[20]

The parsed ShapeQuery is sent to the front end (Box 4 in Figure 4) for users to edit or further refine it if needed. The validated query is then executed to generate the matching trendlines.

6. OPEN CHALLENGES

We now discuss open research directions for improving the usability and performance of both Zenvisage and ShapeSearch.

6.1. Search enumeration + shape matching

In ShapeSearch, users currently need to specify the X and Y attributes before issuing ShapeQueries. However, in certain scenarios, users may not know the X and Y attributes in advance or may want to search for the same shape over different combinations of attributes. Additionally, users may want to issue a multistep query involving multiple shapes at the same time, finding states with decreasing listing price trends but increasing soldprice trends of houses. To support such complex data exploration needs, we envision integrating ZQL with ShapeQuery. One simple option is support ShapeQuery as a functional primitive as part of the process column in ZQL. For instance, Table 6 depicts an integrated query for the above example for finding states with decreasing listing prices trends but increasing soldprice trends. Combining ZQL and ShapeQuery also adds to expressiveness and efficiency of ZQL—functional primitives are currently treated as black boxes and thus not optimized in Zenvisage. By adding support for Shape-Query, Zenvisage can

Table 6. Example of a *ZQL* query using *ShapeQuery* as a functional primtive within Process column. The query finds states with decreasing listprice but increasing sold price over trends of houses.

Name	X	Y	Z	Process
f1	year	listprice	z1 <– 'state'.*	z2 <– $argany_{z1}$ [p=down](*f1*)
f2	year	soldprice	z1	z3 <– $argany_{z1}$ [p=up](*f2*)
*f3	year	{listprice, soldprice}	z2 && z3	

leverage the shape matching algorithms discussed earlier for efficient processing of visualizations.

6.2. In-database support for fuzzy matching

ShapeSearch performs shape matching outside relational databases; consequently as the size of the dataset increases, the data transfer and serialization/deserialization overheads tend to dominate, resulting in an increase in latency. On the other hand, recognizing patterns in a sequence of rows in relational databases has been widely desired but only supported by a few vendors. For instance, Oracle Database 12c supports a MATCH RECOGNIZE[11] clause for pattern matching in native SQL. SQL-TS (Simple Query Language for Time Series)[14] is another proposal on SQL extensions for pattern queries. Nevertheless, none of these extensions support fuzzy matching capabilities; instead, they require users to define the patterns (e.g., up, down) using values of matching columns—making the specification quite tedious and verbose.

In order to support fuzzy shape queries, we envision developing new database extensions that take as input a ShapeQuery as part of the SQL query and leverage shape matching algorithms for efficiently executing the ShapeQuery within the database kernel. For instance, the following query depicts potential extensions for supporting ShapeQuery within the SQL syntax.

```
SELECT *
FROM Ticker T,
(MATCH BY symbol ON price
USING PATTERN [p=*]⊗p=down]⊗[p=up]⊗[p=*] AS
  score
ORDER BY score DESC
LIMIT 1) S
WHERE T.symbol = S.symbol
```

Given a table Ticker, the aforesaid query finds a stock symbol (specified *via* MATCH BY clause) with the closest matching V-shaped trend on the values of column price (specified *via* ON clause) and outputs its corresponding tuples.

6.3. Mixing code and interaction

To accelerate data exploration, an important next step is to integrate visualization search abstractions supported *via* ZQL and ShapeQuery with existing data science libraries such as Pandas. This will allow users to seamlessly transition between writing code (for example, for data cleaning, and transformation); getting recommendations *via* search specifications, and performing interactions on visualizations—all in one place. As a step in this direction, LUX[9], a recent Python library, combines partial user specifications with best practices from visual data analysis to recommend interesting visualizations for guiding users toward next steps. It further displays visualizations as a widget *in situ* within a Jupyter notebook to support easier transitions between code and interaction. Providing these capabilities "for free" for users has led to a lot of adoption in the open source, with over 50,000 downloads.[4] While specification in LUX is inspired from ZQL, adding natural language or regex-based pattern searching functionalities, as supported in ShapeQuery, can further enhance the power of such libraries.

7. RELATED WORK

Our work draws on prior work in visual querying, as well as symbolic pattern mining. Visual querying tools[13, 17, 21] help users search for visualizations with a desired shape by taking as input a sketch of that shape. Most of these tools perform precise point-wise matching using measures such as Euclidean distance or DTW. A few tools such as TimeSearcher[2] let users apply soft or hard constraints on the *x* and *y* range values *via* boxes or query envelopes, but do not support mechanisms for specifying shape primitives beyond location constraints. ShapeSearch introduces a novel algebra that improves extensibility by acting as a common "substrate" for various input mechanisms, along with an optimization engine that efficiently matches patterns against a large collection of trendlines.

Symbolic sequence matching papers approach the problem of pattern matching by employing offline computation to chunk trendlines into fixed-length blocks, encoding each block with a symbol that describes the pattern in that block.[3, 12, 16] Since these work operates on pre-chunked-and-labeled trendlines, the problem is one of matching regular expressions against string sequences (one per pre-labeled trendline). Most of these papers only return a Boolean score for whether the pattern matches the string sequence. Moreover, since the trendlines are pre-labeled and indexed, they do not support on-the-fly pattern matching where the same trendline can change shapes based on filters or aggregation constraints. ShapeSearch, on the other hand, adopts a more online query-aware ranking of trendlines without requiring precomputation and is thus more suited for *ad hoc* data exploration scenarios.

8. CONCLUSION

In this work, we described ShapeSearch, a pattern-matching system that complements our prior system Zenvisage by providing expressive and flexible mechanisms for domain experts to effortlessly and efficiently search for trendline visualizations. We described the ShapeQuery algebra that forms the core of ShapeSearch and helps express a large variety of patterns with a minimal set of primitives and operators. The algebra is backed by a shape matching engine that enables on-the-fly and scalable pattern matching. Overall, together with Zenvisage, ShapeSearch offers a promising first step toward substantially simplifying and improving the process of interactive data exploration for novice and expert analysts alike.

References

1. Bult, C.J., Eppig, J.T., Kadin, J.A., Richardson, J.E., Blake, J.A., M.G.D. Group. The mouse genome database (mgd): Mouse biology and model systems. *Nucl. Acids Res 36*, suppl_1 (2008), D724–D728.
2. Buono, P., Aris, A., Plaisant, C., Khella, A., Shneiderman, B. Interactive pattern search in time series. In *Visualization and Data Analysis 2005*. Volume 5669, International Society for Optics and Photonics, 2005, 175–187.
3. Faloutsos, C., Ranganathan, M., Manolopoulos, Y. *Fast subsequence matching in time-series databases*. Volume 23. ACM, 1994.
4. https://github.com/lux-org/lux.
5. Investopedia. https://www.investopedia.com/terms/t/tripletop.asp.
6. Lafferty, J., McCallum, A., Pereira, F.C. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.
7. Lee, D., Lee, J., Siddiqui, T., Kim, J., Karahalios, K., Parameswaran, A. Accelerating scientific data exploration via visual query systems. arXiv:1710.00763 (2017).
8. Lee, D., Lee, J., Siddiqui, T., Kim, J., Karahalios, K., Parameswaran, A. You can't always sketch what you want: Understanding sensemaking in visual query systems. *IEEE Trans. Vis. Comput. Graph* (2019).
9. Lee, D., Tang, D., Agarwal, K., Boonmark, T., Chen, C., Kang, J., Mukhopadhyay, U., Song, J., Yong, M., Hearst, M.A., et al. Lux: Always-on visualization recommendations for exploratory data science. arXiv:2105.00121 (Accepted at VLDB 2021).
10. Mannino, M., Abouzied, A. Expressive time series querying with hand-drawn scale-free sketches. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (2018), ACM, 388.
11. Match recognize. https://bit.ly/3bWwUhs. Online [accessed 17-Aug-2015].
12. Psaila, R.A.G., Wimmers, E.L. Mohamed &It. Querying shapes of histories. *Very Large Data Bases*. IEEE, Zurich, Switzerland, 1995.
13. Ryall, K., Lesh, N., Lanning, T., Leigh, D., Miyashita, H., Makino, S. Querylines: Approximate query for visual browsing. In *CHI'05 Extended Abstracts on Human Factors in Computing Systems* (2005), ACM, 1765–1768.
14. Sadri, R., Zaniolo, C., Zarkesh, A., Adibi, J. Expressing and optimizing sequence queries in database systems. *ACM TODS 29*, 2 (2004), 282–318.
15. Shapesearch extended version. https://arxiv.org/abs/1811.07977.
16. Shatkay, H., Zdonik, S.B. Approximate queries and representations for large data sequences. In *Data Engineering, 1996: Proceedings of the Twelfth International Conference on* (1996), IEEE, 536–545.
17. Siddiqui, T., Kim, A., Lee, J., Karahalios, K., Parameswaran, A. Effortless data exploration with zenvisage: An expressive and interactive visual analytics system. *Proc. VLDB Endow 10*, 4 (2016), 457–468.
18. Siddiqui, T., Lee, J., Kim, A., Xue, E., Yu, X., Zou, S., Guo, L., Liu, C., Wang, C., Karahalios, K. et al. Fast-forwarding to desired visualizations with zenvisage. In *CIDR*, 2017.
19. Siddiqui, T., Luh, P., Wang, Z., Karahalios, K., Parameswaran, A. Shapesearch: Flexible pattern-based querying of trend line visualizations. *Proc. VLDB Endow. 11*, 12 (2018), 1962–1965.
20. Siddiqui, T., Luh, P., Wang, Z., Karahalios, K., Parameswaran, A. Shapesearch: A flexible and efficient system for shape-based exploration of trendlines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), 51–65.
21. Wattenberg, M. Sketching a graph to query a time-series database. In *CHI'01 Extended Abstracts on Human factors in Computing Systems* (2001). ACM, 381–382.

Tarique Siddiqui(tasidd@microsoft.com), Microsoft Research, Redmond, WA, USA.

Paul Luh, Zesheng Wang, Karrie Karahalios([luh2, zwang180, kkarahal]@illinois.edu), University of Illinois, Urbana-Champaign, IL, USA.

Aditya G. Parameswaran(adityagp@berkeley.edu), UC Berkeley, CA, USA.

Watch the authors discuss this work in the exclusive *Communications* video. https://cacm.acm.org/videos/visual-querying