

# Fast-Forwarding to Desired Visualizations with zenvisage

Tarique Siddiqui, John Lee, Albert Kim, Edward Xue, Chaoran Wang, Yuxuan Zou,  
Changfeng Liu, Lijin Guo, Xiaofu Yu, Karrie Karahalios, Aditya Parameswaran

University of Illinois, Urbana-Champaign (UIUC)

## ABSTRACT

Data exploration and analysis, especially for non-programmers, remains a tedious and frustrating process of trial-and-error. We demonstrate zenvisage, an interactive data exploration system tailored towards “fast-forwarding” to desired trends, patterns, or insights, without much effort from the user. zenvisage supports simple drag-and-drop and sketch-based interactions as specification mechanisms for the exploration need, as well as an intuitive data exploration language called ZQL for more complex needs. zenvisage is being developed in collaboration with ad analysts, battery scientists, and genomic data analysts, and will be demonstrated on their datasets.

## 1. INTRODUCTION

We are in the cusp of a data-enabled era, with virtually every sector of society—spanning business, government, science, medicine, and defense—having access to large volumes of data, and a pressing need for analyzing and extracting insights from it. Unfortunately, the domain experts in these sectors analyzing the data do not typically possess extensive programming experience [18]. As a result, these experts primarily rely on interactive visualization tools like Tableau [5] or Microsoft Excel. These commercial tools make it easy for such individuals to interactively specify a visualization of interest from a preset set of styles, and the tools generate and display the desired visualization.

However, these tools, while immensely popular and broadening the reach of data analysis—Excel has a user base in the billions [4], while Tableau is a publicly traded company with valuation in the billions [6]—still leave a lot to be desired. Specifically, these tools have little by way of *guiding their users to visualizations that capture desired trends or patterns*—the onus is on the user to step through a number of visualizations before they find these trends or patterns. We illustrate by means of an example.

**EXAMPLE 1.** *Consider an economist who wishes to study if we’re heading towards another housing bubble in the USA. To do so, she wants to explore a real estate dataset [7]. One specific question that this economist may be interested in is whether there are any towns for which the average sales prices has been roughly increasing over time. Presently, our economist would need to gener-*

*ate the sales prices over time, one visualization for each town, and manually step through each one to find those that match her desired pattern—a tedious and cumbersome process, given that there are 100s of towns. Next, say our economist has a hypothesis: she feels that the sales prices may be correlated with the reduced availability of houses in the areas where the sales prices have been going up. To verify this hypothesis, our economist will have to first find all the towns for which the sales prices is going up like before, following which she needs to individually generate the availability by time charts for each of these areas, and then verify if the availability is indeed going down for each one—an even more cumbersome process than the previous scenario, since she now needs to look at both sales over time and availability over time visualizations for all towns. Lastly, say our economist wants to explore the percentage of properties that are foreclosed across these towns—what are the typical patterns, and what are the outliers? Here, the economist will have to perform “manual data mining”—she will have to individually step through the visualization of foreclosure rates over time for each of these towns, and remember what she finds to be typical trends, and what are surprising or anomalous. Given a trend, it may be almost impossible for the economist to remember if she’s seen a similar trend before, if it’s actually anomalous.*

*In short, no matter which hypothesis she wants to test, or which pattern she wants to find, tedium and pain abounds, virtually preventing data exploration.*

In contrast, we have been developing an interactive data exploration tool called zenvisage (a portmanteau of zen and envisage, meaning to *effortlessly visualize*), targeted at easing the pain of data exploration in scenarios like the one described above. zenvisage uses two mechanisms to support effortless data exploration:

- **Built-in Interactions and Summarization:** zenvisage supports simple interactions that allow users to specify the desired patterns, following which zenvisage will automate the search for those patterns. In our example, finding towns where the sales price is going up is as simple as sketching an increasing curve on a canvas, following which zenvisage will automate the search for that curve among all the candidate visualizations. We show a screenshot of zenvisage in action for this query in Figure 1. zenvisage also supports other interactions, as we will describe subsequently. Additionally, at each step along the way, zenvisage shows a summary of the typical trends (also known as motifs in the literature [30]), and outliers (also seen in Figure 1), reducing the need in our example to remember whether a specific pattern was seen previously.
- **Sophisticated Query Language, ZQL:** For more complex patterns, like the second hypothesis in our example, zenvisage supports a query language called ZQL, drawing from prior work on Query-by-Example [39]. Via a user study, we have demon-

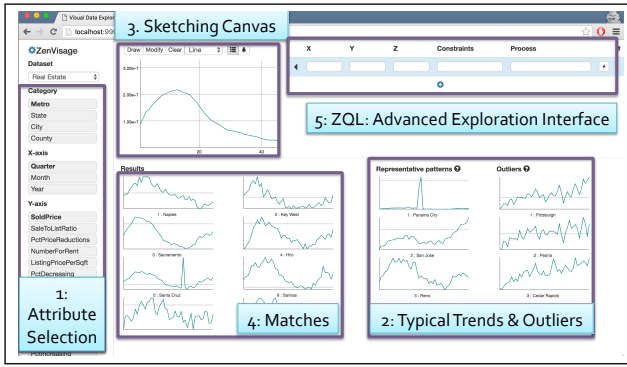


Figure 1: zenvisage interactive visual query interface

strated that even individuals who have never programmed before, are able to use ZQL effectively after a small training period of ten to fifteen minutes [2].

In our companion full paper [2], we describe the complete details of zenvisage, including the front-end and back-end architecture, the details of the query language along with its underlying exploration algebra, and query optimization. We also describe *concrete real-world use cases* via partners with whom we’re working to test out zenvisage, spanning ad analytics, battery science, and genomic data analysis. These real-world use cases inform and provide some of the datasets for our demonstration scenarios later on.

The outline for this paper is as follows: in Section 2, we describe the user experience of someone using zenvisage; in Section 3, we briefly explain the zenvisage query language, ZQL; in Section 4, we give a brief overview of the system architecture and query processing; in Section 5, we describe the goals of our demonstration scenarios; and in Section 6, we give an overview of the related work.

## 2. USER EXPERIENCE

Since zenvisage is meant to be an end-user-facing interactive data exploration tool, the user experience of the tool is hugely important in determining the utility and usability of the tool. Here, we describe the experience of an individual using zenvisage. In the next section, we dive into the details of the ZQL query language.

We once again return to our running example of the real estate data analysis scenario. In Figure 1, we show zenvisage loaded with the real estate dataset.

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

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

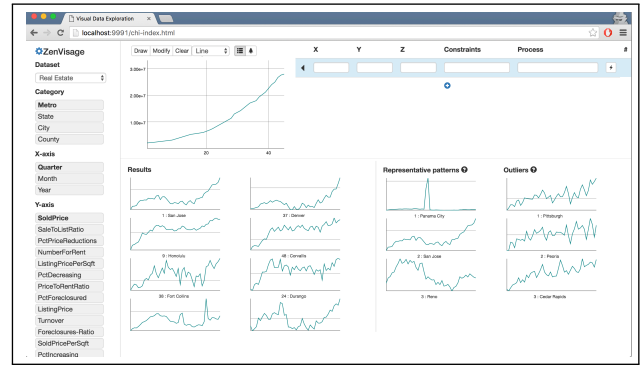


Figure 2: Finding cities with similar sales over year trends to a user-drawn trend

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

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

**Similarity Search Results.** As soon as the user completes an interaction in Box 3, Box 4 is populated with results corresponding to visualizations (on varying the category) that are most similar to the trend in Box 3, ordered by similarity. We describe how the similarity search results are computed in Section 5. As yet another example of similarity search, see Figure 2, where the user has drawn a gradually increasing trend in the canvas area (or dragged an existing visualization onto the area), and the results returned below, corresponding to San Jose, Denver, and Honolulu are matches of increasing trends.

**ZQL Specification Interface.** Lastly, the user can specify a multi-line ZQL query in Box 5, and once the user completes the action, this request triggers a recomputation and redisplaying of the results shown in Box 4.

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

## 3. ZQL QUERY LANGUAGE

We now briefly describe zenvisage’s query language, ZQL, forming the core of zenvisage and aimed at supporting general data exploration. ZQL draws from and extends existing languages for visualization specification and encoding such as Cleveland’s Grammar of Graphics [38] and the visualization algebra of Polaris, the basis for Tableau [35], by adding data exploration capabilities to automate the search for visualizations with specific patterns or insights. The specification format of ZQL is inspired by Query-by-Example (QBE) and similar to QBE, a ZQL query can be constructed using a tabular structure as depicted in Box 5 in Figure 1—for clarity, we provide three examples of ZQL queries explicitly laid out in Tables 1, 2, and 3, and we will explain these examples in detail in the following. Note that ZQL invocations can also be em-

bedded within code—there is no restriction that the language has to be only used or specified within the *zenvisage* frontend interface. Details about our formal syntax, the expressiveness and power, and completeness of ZQL can be found in our companion full paper [2].

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

As depicted in Table 1, a ZQL query consists of one or more rows, where each row has well-defined columns, namely Name, X, Y, Z, Viz, Constraints, and Process. These columns can be grouped into two components: the *visual component* consisting of the X, Y, Z, Viz, and Constraints columns, and the *task component* consisting of the Process column, while the Name column is an identifier for a line of ZQL. The goal of the visual component is to specify a set of visualizations. Task component operates on and subselects from these visualizations, applies filtering, sorting, or processing using a core set of data exploration primitives. The output of the task component can be further reused in the subsequent rows. We now explain the syntax and semantics of ZQL with the help of examples; for these examples, we operate on a fictitious product sales dataset consisting of a single table over which visualizations are specified. ZQL also operates over multiple tables, but we do not cover the general case in this short demonstration paper.

**Example 1.** In this example, we are interested in finding the sales over time over all regions for the products whose sales over time in the US is similar to the sales over time for staplers. The example is displayed in Table 1. In the first row, we find our first line of ZQL, with the Name identifier set to f1. This row retrieves the visualization corresponding to sum of sales by year for the product ‘stapler’. The X column (corresponding to the X axis of the visualization) is set to year, the Y column (corresponding to the Y axis) is set to sales, and the Z column is set to product.stapler, indicating that the attribute product has been set to the value ‘stapler’. 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 product stapler. Lastly, the Viz column is set to indicate that the displayed chart needs to be a bar chart with aggregation as the SUM performed on the Y axis. The Viz column thus specifies the visualization type and the aggregation method, additionally it can also apply binning and interpolation; this column draws from the Grammar of Graphics format [38]—this column can be omitted, and defaults will be used [35]. For this row, there are no Constraints or Process.

In the next row, with identifier f2, the X, Y, and Viz columns stay similar, while the Z column is set to product.\* indicating that the visual component for this row corresponds to a set of visualizations formed by iterating over various product categories, one for each product. The variable v1 is used to iterate over these categories. Additionally, there is an entry in the Constraint column, indicating that location has been set to ‘US’. Unlike the Z column, which is used to iterate through visualizations, the Constraints column is used for applying filters to the data prior to the visualizations being generated or specified. Since we only want to compare with product sales in the US, the location has been set to US. Thus, we operate over visualizations for various products for sales over time in the US.

The Process column in this row is used to compare, sort, and filter the visualizations retrieved in this row or previous rows. The

process column returns a subset of values for one or more variables that it operates over, essentially corresponding to visualizations that satisfy the desired properties. These variable values can be then used in visual component columns of subsequent rows for output visualization or further processing. The process column consists of the two main parts: a *functional primitive*, and a *sort-filter primitive*. The functional primitives assign a score to each visualization based on how well the visualization satisfies the condition laid out by the primitive. To handle the vast majority of visual data exploration use cases, we define three classes of functional primitives, differing in their inputs: *T* is a class of functional primitives that assign a score by measuring the prevalence of a particular pattern or trend within a single visualization—for example, monotonicity, repetitiveness, or number of peaks. In our system at the moment, we support monotonicity, but other primitives are easy to handle. *D* is a class of primitives that assign a score by comparing two visualizations: for example, one instantiation we support is distance computation—for which standard distance measure can be used (more details later). Lastly, *R* is a generic class of functional primitives that support arbitrary processing on visualizations, and assigns a score to each visualization. One concrete instantiation of *R* in *zenvisage* is for typical trends and outlier computation, for which standard clustering algorithms can be used (again, details later).

Then, the sort-filter primitive takes the output of a functional primitive, sorts them using *argmax*, *argmin* or *argany* (returns any visualization that satisfies some condition) and then filters them either based on top-k or a threshold based criteria. In the second row of Table 1, we compare the visualization for each product in f2 with the visualization of stapler (f1) using a functional primitive *D*, computing distance, via *D*(f1, f2). Then, *argmin* is a sort-filter primitive that sorts the products based on distance scores and selects the top 10 product with minimum scores. Finally, in row 3, we output the overall sales over year visualization for the selected products as bar-charts. The \* in \*f3 indicates that these visualizations are to be output to the user.

**Example 2.** In this example, we want to examine typical trends for profit over time over all regions for those products whose sales are increasing over time in the US, while decreasing over time in UK. The ZQL query is depicted in Table 2. (Note that we exclude the Viz and Constraints column if it is unused, in the former case, default settings are used.) In the first row, we first fetch the sales over time visualizations for all products in US and in the process column, we select those products that have an increasing trend with the help of the *T* functional primitive. Similarly, in the second row, we select the products that have decreasing sales over time trends in UK. In the third row, we first find the products whose visualizations appeared in both the first and the second rows (by applying the expression *v4 <- v2.range & v3.range*, where *v4* is the intersection of the elements in *v2* and *v3*), and generate their profit over time trends followed by using the *R* functional primitive to find five representative or typical trends all in the third row corresponding to f3. Finally in the last row, we output the profit over time line charts visualizations for these five representative products.

**Example 3.** In this example, we are interested in finding a pair of X and Y axes where the visualizations for two products ‘stapler’ and ‘chair’ differ the most. For doing this, we write a ZQL query depicted in Table 3. In the first line, we fetch all visualizations for the product ‘chair’ 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 ‘Chair’. In the process column, we iterate over the possible pairs of X and Y axes values, compare the corresponding visualizations in f1 and f2 and finally

Name	X	Y	Z	Constraints	Viz	Process
f1	'year'	'sales'	'product'. 'stapler'	location='US'	bar.(y=avg('sum'))	$v2 \leftarrow \text{argmin}_{v1}[k = 10]D(f1, f2)$
f2	'year'	'sales'	$v1 \leftarrow \text{'product'}.*$		bar.(y=avg('sum'))	
*f3	'year'	'sales'	v2		bar.(y=avg('sum'))	

Table 1: A ZQL query which returns top 10 products that have the most similar sales over year visualization to stapler.

Name	X	Y	Z	Constraints	Process
f1	'year'	'sales'	$v1 \leftarrow \text{'product'}.*$	location='US'	$v2 \leftarrow \text{argany}_{v1}[t > 0]T(f1)$
f2	'year'	'sales'	v1	location='UK'	$v3 \leftarrow \text{argany}_{v1}[t < 0]T(f2)$
f3	'year'	'profit'	$v4 \leftarrow (v2.\text{range} \ \& \ v3.\text{range})$		$v5 \leftarrow \text{argmax}_{v4}[k = 5]R(f3)$
*f4	'year'	'profit'	v5		

Table 2: A ZQL query which returns 5 representative profit over years visualizations among the products that have positive sales over years trends for the US but have negative sales over years trends for the UK.

Name	X	Y	Z	Process
f1	$x1 \leftarrow *$	$y1 \leftarrow *$	'product'. 'chair'	$x2, y2 \leftarrow \text{argmax}_{x1, y1}[k = 1]D(f1, f2)$
f2	x1	y1	'product'. 'desk'	
*f3	x2	y2	'product'. 'chair'	
*f4	x2	y2	'product'. 'desk'	

Table 3: A ZQL query retrieving two different visualisations (among different combinations of x and y) for chair and desk that are the most dissimilar.

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.

**Capabilities and Limitations.** While the examples above have indicated that ZQL is rather powerful, the reader may be wondering what it does not handle. Indeed, there are many types of data analysis tasks that ZQL is not meant for, including developing predictive models, or data cleaning. We expect that users are already operating on structured datasets, i.e., data cleaning is already performed, and are performing visual analysis and exploration as a precursor to developing predictive models. Indeed, the wide popularity of Tableau indicates that there is a need for this intermediate step. We characterize the space of data exploration operations that ZQL is capable of handling in our paper [2].

## 4. SYSTEM OVERVIEW

In this section, we provide an overview of the system architecture of zenvisage: we begin by describing the various components of zenvisage followed by a brief description of one of the most interesting components of zenvisage: the query processor and optimizer.

### 4.1 zenvisage components

zenvisage is fully functional, with our collaborators in battery science, ad analytics, and genomic data analysis either already using the tool, or fine-tuning the tool to their requirements. The source-code of our current implementation is also available to public<sup>1</sup>, with regular updates posted at zenvisage homepage<sup>2</sup>.

As depicted in Figure 3, zenvisage consists of two main components: a front-end and a back-end, both of which work independently of each other.

**Front-end.** The zenvisage front-end is implemented as a lightweight web-client application that runs completely within a user's browsers. As described in Section 2 and Section 3, the front-end provides a combination of intuitive drag-and-drop based operations as well as an advanced ZQL based exploration interface for users to search for visualization with desired insights. An important component of the interface is the drawing panel where the users draw trend lines, bar-charts, scatter-plots, or drag and drop an existing visualization and edit it. Dygraph [1] is an open-source charting library, that we use for the drawing panel as well as for visualizing the output. In addition to dygraph, the front-end uses javascript libraries such as bootstrap (getbootstrap.com) and angular (angularjs.org). All user inputs at the interface are internally translated

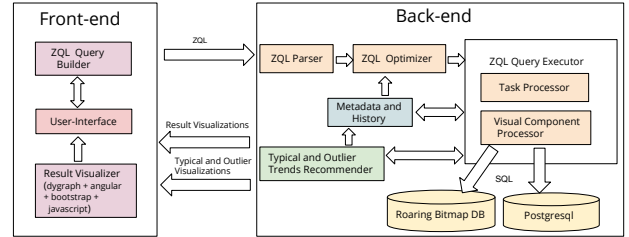


Figure 3: System Architecture

and composed into one or more ZQL queries by the query builder module at the front-end before being sent to the back-end for processing. The front-end talks to the back-end through a REST interface, and all of the data transfers happen in a JSON format. The results from the back-end are processed and rendered using the result visualizer module. By applying simple rules that we draw from prior work [20, 35], the result visualizer can also figure out effective visualization mappings and visual encodings for the results, if the user has not already specified these in the query.

**Back-end.** The zenvisage back-end is responsible for running all of the computations necessary for generating output visualizations that match user-specified insights. It is developed completely in Java and runs within an embedded Jetty web-server [3]. At a high level, the back-end consists of a ZQL compiler, consisting of a parser, an optimizer, and a query executor, and is capable of processing any ZQL query. We provide the details of query processing in the next Section 4.2. For storing and retrieving data, the back-end currently supports two types of databases: a roaring bitmap-based [9] in-memory database for small to medium-sized datasets, and a PostgreSQL relational database for extremely large datasets. In addition to ZQL query processing, the back-end also recommends typical trends and outliers for the attributes specified, independent of user queries. The generated visualizations are all sent to the front-end for rendering in a JSON format.

### 4.2 Query Processing

The ZQL query processor is responsible for compiling and executing ZQL queries. It consists of four sub-components: the parser, optimizer, visual component processor, and task component processor. The visual component processor and task component processor together make the ZQL query executor.

**Parsing.** The parser reads in the ZQL query is a textual format, parses the query and validates its structure, and checks the database catalog for the existence of the referenced columns and operators including the functional primitives. If everything succeeds, the

<sup>1</sup><https://github.com/zenvisage/>

<sup>2</sup><http://zenvisage.github.io>

parser creates a graph of computation from the ZQL rows—this graph is a directed acyclic graph that describes the steps of computation and the dependencies across them as expressed in the ZQL query. For each ZQL row, the parser creates two types of graph nodes: a node for the visual component, and one for the task component: we will simply call these the visual node, and the task node, respectively. The visual node corresponds to the X, Y, Z, Viz and Constraints columns; these columns specify the collection of visualizations to be retrieved. The task node consisting of the functional primitive and the sort-filter primitive, specifies the processing to be applied on the visualizations generated from the visual nodes.

**Optimizations.** At a high level, we have two types of optimizations on the parsed ZQL graph: inter-node and intra-node optimizations. Inter-node optimizations reduce the ZQL graph by merging multiple visual or task nodes. While merging multiple visual nodes, we try to minimize the number of SQL queries as well as the number of operations that need to be issued to the database for retrieving data. For instance, we can merge two visual nodes that have the same X axis value but different Y axis values. By doing so, we reduce the number of scans and group by operations applied to the same data. Similar to merging visual nodes, multiple task nodes can be merged if we can apply multiple forms of processing together on the same collection of visualizations. Inter-node optimizations also exploit *speculation*: where two nodes are combined even if the latter depends on the results of the former, as long as there is benefit to doing it jointly. Intra-node optimizations transform individual graph nodes by minimizing the number of visualizations or the number of possible values in a given visualization by applying data reduction techniques such as sampling, binning, and regression. By doing this, we minimize the time taken by the task processor for processing these visualizations. For instance, if we know the maximum number of the pixels that can be visualized for a scatterplot, we can apply the appropriate binning to both aggregate at a coarser granularity, and reduce the size of the intermediate JSON that needs to be sent to the front-end.

**Query Execution.** The query executor takes the transformed graph as an input; starting from the root nodes and following the outgoing edges, it executes one or more nodes in parallel. Based on the type of the node, it creates an instance of either a visual processor or a task processor. The visual processor translates a visual node to a SQL query and issues it to the underlying database. The generated SQL query has the following form: `SELECT X, Y FROM R WHERE Z=V AND (CONSTRAINTS) ORDER BY X`. The retrieved data is transformed into a set of visualizations by applying interpolation, regression, binning or aggregation. This set of visualizations is stored in an n-dimensional array where each location in the array contains one visualization. The result is either sent as an input to another processor for further processing, or is sent to the front-end for rendering. The task processor generates the post-processing code from the functional and the sort-filter primitives in the task node. It iterates through visualizations and for each visualization, the functional primitive is called to process it and give it a score. After scoring all the visualizations, the sort-filter primitive is used to sort and filter the visualizations based on their scores. The attribute values of the selected visualizations are then passed to subsequent nodes for further processing, or for generating output visualizations.

## 5. DEMONSTRATION SCENARIOS

The goals of our demonstration scenarios are to enable the conference attendees to (1) understand how *zenvisage*’s simple interactions can help facilitate the fast-forwarding to interesting insights;

(2) view how ZQL queries can support multi-step data exploration workflows; (3) appreciate the wide applicability of *zenvisage*, across a spectrum of use cases within a domain, and across domains; (4) see how *zenvisage* supports customizability for the basic interactions, and the impact of these customizations; and (5) take a bit of a peek under the covers to see how *zenvisage* parses and optimizes ZQL queries. Since we have already described (1) and (2) in Section 2, we focus on the remaining points in the present section. Note that given that we may not have a lot of time for our on-stage presentation, that presentation will necessarily be condensed and may not explore all aspects; however, during the demo session, attendees will be able to explore all aspects of our demonstration scenarios.

**Datasets.** Our primary focus will be on the real estate dataset [7], like in our example in the introduction. This real estate dataset is relatively small but quite intuitive, with 11K tuples and 12 attributes. In addition, we will use datasets from three real domains with a need for fast-forwarding to interesting insights: (1) a synthetic ad analytics dataset, modeled after the real datasets at Turn, Inc., for enabling ad analysts to ask questions like “which keyword has similar behavior in terms of click-through rates over time to a given keyword?”; (2) a physical dataset of electrolyte properties from battery scientists at Carnegie Mellon University, for enabling the rational design of Lithium-Ion batteries, asking questions like “are there any electrolytes for which the dependence between these two physical properties follows a hockey-stick shape?”; (3) a genomics dataset of gene-gene and protein interactions from an NIH-sponsored genomics center at Illinois, supporting questions like “are there features on which these two classes of genes can be effectively separated on a scatterplot?”. For all our datasets and usage scenarios, *zenvisage* will come pre-loaded with starting points for analysis—via canned queries that the domain experts found to be very useful for their objectives—with the participants able to change the queries if they so choose to. Our intended objective is to both convey some of the richness of exploration goals in these domains, and educate the participants about these domains.

**Customizability.** *zenvisage* supports the retrieval of visualizations similar to a given visualization, as well as typical and outlier visualizations. To do so, *zenvisage* needs distance measures to assess the distance between two visualizations, be it ordinal visualizations (like time charts), categorical visualizations (like bar charts or histograms), or non-aggregated visualizations (like scatterplots). For example, for ordinal visualizations, one standard distance measure is the Euclidean distance measure, which computes the sum of the element-wise square of the difference between corresponding values in two visualizations, followed by an overall square-root. Yet another distance measure is Dynamic Time Warping [32], a standard distance measure for time series analysis that is based on computing the least amount of effort to transform two visualizations by stretching and squishing them until they look like each other. We have also been developing other home-grown distance measures that assess the perceptual difference between the two visualizations, e.g., measures that weight different features on the visualizations based on their visual prominence. One aspect of our demonstration will be to allow participants to set the distance measure (once again hidden away under the gear symbol), allowing them to observe the impact of these measures on visual similarity. Similarly, the choice of the algorithm for typical patterns and outliers also has a huge impact on performance. Currently, we support variations of the k-means algorithm, as well as our perceptually-aware variants—once again, the attendees will be able to see the impact both in terms of performance and accuracy of these mechanisms.

**Under the Covers.** As described previously, zenvisage’s ZQL query optimizer operates on a graph of nodes corresponding to visual and task processors, with edges indicating the dependencies between them. The query optimizer rewrites or simplifies this graph using a combination of batching, parallelism, and speculation-based rules, applying a cost model that we have developed [2], to dictate if applying the rule helps reduce the query execution time. Moreover, the optimizer simplifies or transforms individual nodes in the graph by applying binning or interpolation. Subsequently this graph is executed as a sequence of SQL queries on a traditional relational database or on the in-memory roaring-bitmap-based database. To gain an appreciation for the query optimization approach, attendees will be able to view the graph representing the starting point of optimization, as well as the rewritten graph post application of the optimization rules.

## 6. RELATED WORK

zenvisage draws from work in several communities; detailed related work descriptions can be found in our companion full paper [2]—here, we briefly survey the most important related work.

From the visualization community, zenvisage draws from visual specification algebra developed by Polaris and Tableau [5, 35] and extends it to add support for exploration, aimed at reducing the need for manual trial-and-error. Visualization systems like SeeDB [36] or Profiler [21], and Voyager [20] provide restricted forms of visualization recommendation—the first two based on what is visually different, and the last based on aesthetics—without being full-fledged exploration tools. Similarly, from the data mining community, there has been a lot of work on time series data mining [25, 14, 10, 24, 8, 11, 23] including clustering and similarity search, however, this work has primarily focused on indexing for retrieval of, or clustering for a fixed set of time series as opposed to a full-fledged exploration tool that supports arbitrary exploration of attributes. Work by the visualization community on TimeSearcher [15] develops a front-end for time-series data mining while being restricted to a fixed set of time series, and only supporting a specific form of drill-down, as opposed to the many operations possible on zenvisage, plus a full-fledged query language. There are systems [29, 37, 31, 16] which let users search for visualizations through query by sketch on a single attribute type, zenvisage extends these work to multiple data types, multiple sets of visualizations, and multiple data sets, with necessary customization capabilities to the sketching interface that can adapt to various needs of analysts.

Work from the databases community on data cube exploration [33, 34] is also related; our focus is not recommendation of aggregates to explore and instead to support the search for patterns, trends, or insights via a data exploration language, and simple interaction primitives.

Our technical approach draws from principles in multi-query optimization (MQO) [12, 17, 22, 13], since our setting requires us to generate many SQL queries that need to be executed in parallel; however more fine-grained optimizations that do not apply in the general MQO setting apply here. There has been some work on generating visualizations on large datasets more rapidly preserving visual properties; we draw from that work to apply sampling to generate visualizations even faster [26, 19]. Unlike Immens [28] and Nanocubes [27], also tailored for large-scale visualization, we cannot precompute all aggregates upfront.

## Acknowledgements

We thank the anonymous reviewers for their valuable feedback. We acknowledge support from grant IIS-1513407 and IIS-1633755

awarded by the National Science Foundation, grant 1U54GM114838 awarded by NIGMS and 3U54EB020406-02S1 awarded by NIBIB through funds provided by the trans-NIH Big Data to Knowledge (BD2K) initiative ([www.bd2k.nih.gov](http://www.bd2k.nih.gov)), and funds from Adobe, Google, and the Siebel Energy Institute. The content is solely the responsibility of the authors and does not necessarily represent the official views of the funding agencies and organizations.

## 7. REFERENCES

- [1] Dygraph. <http://dygraphs.com/>.
- [2] Effortless data exploration with zenvisage: An expressive and interactive visual analytics system. *Technical Report*. <http://data-people.cs.illinois.edu/zenvisage.pdf>.
- [3] jetty. <http://www.eclipse.org/jetty/>.
- [4] Microsoft by the numbers. <https://news.microsoft.com/bythenumbers>.
- [5] Tableau public. [Online; accessed 3-March-2014].
- [6] Tableau valuation. <http://www.sramanamitra.com/2015/01/12/billion-dollar-unicorns-tableaus-valuation-increases-to-6-billion/>.
- [7] Zillow real estate data. [Online; accessed 1-Feb-2016].
- [8] K. Chakrabarti, E. Keogh, S. Mehrotra, and M. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Trans. Database Syst.*, 27(2):188–228, June 2002.
- [9] S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 2015.
- [10] K.-P. Chan and A.-C. Fu. Efficient time series matching by wavelets. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 126–133, Mar 1999.
- [11] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. *SIGMOD Rec.*, 23(2):419–429, May 1994.
- [12] G. Giannikis et al. Workload optimization using shareddb. In *SIGMOD*, pages 1045–1048. ACM, 2013.
- [13] G. Giannikis et al. Shared workload optimization. *Proceedings of the VLDB Endowment*, 7(6):429–440, 2014.
- [14] D. Gunopulos and G. Das. Time series similarity measures and time series indexing (abstract only). *SIGMOD Rec.*, 30(2):624–, May 2001.
- [15] H. Hochheiser and B. Shneiderman. Dynamic query tools for time series data sets: timebox widgets for interactive exploration. *Information Visualization*, 3(1):1–18, 2004.
- [16] C. Holz and S. Feiner. Relaxed selection techniques for querying time-series graphs. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 213–222. ACM, 2009.
- [17] I. Psaroudakis et al. Sharing data and work across concurrent analytical queries. *VLDB*, 6(9):637–648, 2013.
- [18] M. James, C. Michael, B. Brad, B. Jacques, D. Richard, R. Charles, and H. Angela. Big data: The next frontier for innovation, competition, and productivity. *The McKinsey Global Institute*, 2011.
- [19] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. M4: a visualization-oriented time series data aggregation. *Proceedings of the VLDB Endowment*, 7(10):797–808, 2014.
- [20] K. Wongsupphasawat et al. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE TVCG*, 2015.
- [21] S. Kandel et al. Profiler: integrated statistical analysis and visualization for data quality assessment. In *AVI*, pages 547–554, 2012.
- [22] A. Kementsietsidis et al. Scalable multi-query optimization for exploratory queries over federated scientific databases. *PVLDB*, 1(1):16–27, 2008.
- [23] E. Keogh. A decade of progress in indexing and mining large time series databases. In *Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB ’06*, pages 1268–1268. VLDB Endowment, 2006.
- [24] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and Information Systems*, 3(3):263–286, 2001.
- [25] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. *SIGMOD Rec.*, 30(2):151–162, May 2001.
- [26] A. Kim, E. Blais, A. G. Parameswaran, P. Indyk, S. Madden, and R. Rubinfeld. Rapid sampling for visualizations with ordering guarantees. *CoRR*, abs/1412.3040, 2014.
- [27] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2456–2465, 2013.
- [28] Z. Liu, B. Jiang, and J. Heer. immens: Real-time visual querying of big data. *Computer Graphics Forum (Proc. EuroVis)*, 32, 2013.
- [29] M. Mohebbi, D. Vanderkam, J. Kodysh, R. Schonberger, H. Choi, and S. Kumar. Google correlate whitepaper. 2011.
- [30] A. Mueen, E. Keogh, Q. Zhu, S. Cash, and B. Westover. Exact discovery of time series motifs. In *Proceedings of the 2009 SIAM International Conference on Data Mining*, pages 473–484. SIAM, 2009.

- [31] K. Ryall, N. Lesh, T. Lanning, D. Leigh, H. Miyashita, and S. Makino. Querylines: approximate query for visual browsing. In *CHI'05 Extended Abstracts on Human Factors in Computing Systems*, pages 1765–1768. ACM, 2005.
- [32] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE transactions on acoustics, speech, and signal processing*, 26(1):43–49, 1978.
- [33] S. Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, pages 42–53, 1999.
- [34] G. Sathe and S. Sarawagi. Intelligent rollups in multidimensional olap data. In *VLDB*, pages 531–540, 2001.
- [35] C. Stolte et al. Polaris: a system for query, analysis, and visualization of multidimensional databases. *Commun. ACM*, 51(11):75–84, 2008.
- [36] M. Vartak, S. Madden, A. G. Parameswaran, and N. Polyzotis. SeeDB: supporting visual analytics with data-driven recommendations. *VLDB 2015 (accepted)*, data-people.cs.illinois.edu/seedb-tr.pdf.
- [37] M. Wattenberg. Sketching a graph to query a time-series database. In *CHI'01 Extended Abstracts on Human factors in Computing Systems*, pages 381–382. ACM, 2001.
- [38] L. Wilkinson. *The grammar of graphics*. Springer Science & Business Media, 2006.
- [39] M. M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.