TOWARDS A SCRIPTING LANGUAGE FOR VISUAL DATA EXPLORATION

BY

EDWARD XUE

THESIS

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

Urbana, Illinois

Adviser:

Assistant Professor Aditya Parameswaran

ABSTRACT

Visualization of data is becoming increasingly important for data analysis; numerous insights can be derived from visualizations at a glance. While the information provided through visualization is powerful, the visual analysis process remains largely manual, consuming valuable man-hours. Zenvisage automates the search for desired visual patterns, speeding up visual exploration. It does so through a query language, ZQL, that encapsulates the key data analysis operations such as comparison, sorting, filtering, and composition. However, Zenvisage is somewhat lacking in support for expert data analysts, who are comfortable with programming in existing data analytics tools and may not want to switch to a new system. In order to bring our tool to such users, we propose the Zenvisage Scripting Language, which the power of provides the same expressive power as ZQL but also integrates directly into an expert user's programming language (in our case, we use ggplot and R as a case study). This would allow users to easily adapt their workflows to support visual exploration-based insights.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to express my gratitude to Professor Aditya Parameswaran, my advisor, for all his guidance throughout the years. I started research with Aditya during the end of my junior year as an undergraduate, so I've had quite a few years to work with him. My research abilities have grown remarkably, and I would have never expected to learn so much without my awesome and patient advisor.

I would also like to thank Tarique, who has helped me from the beginning as well, making sure I knew the ins and outs of Zenvisage, a project that he was instrumental in creating. He helped mentor me with the Zenvisage project.

I would like to thank Professor Karrie Karahalios, who has contributed a lot to our front-end design.

I would like to thank Doris, Jaywoo, Chaoran, Zhiwei, John, Lijin, and Xiaofu, all great people that I've had the pleasure to work with on Zenvisage.

Finally, I would like to thank my family. My mom, Jane, for always caring and worrying about me. My dad, Richard, for always supporting and driving me, and my brothers, Alex and Chris, for being fun to be with and being great brothers in general.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

We are currently in a data-rich era, and the availability of data has enabled us to gain a better understanding of our surroundings. As a result, visual analytics tools have risen in popularity, since visualizations are an invaluable means to understand and explore the trends, patterns, and hypotheses within a dataset. However, the process of generating insights from visualizations is still a tedious manual process that will continue to get more cumbersome as our datasets get larger and wider. Most visual analytics tools such as Tableau, Spotfire, ggplot and matplotlib [1] [2] [3] [4] focus on generating one visualization at a time, which the analyst must then look at one by one, and manually ascertain patterns, trends, and insights. This leads to a visual data analytics pipeline with a bottleneck based on the number of visualizations that can be processed by an analyst in order to test a single hypothesis. What if we could automate this exploration pipeline for desired visual patterns, eliminating this bottleneck?

This pipeline, and data exploration in general, has several key operations: composing and filtering visualization collections, comparing them, and sorting visualizations, in order to identify some desired visual pattern. These are the key operations that Zenvisage [5], a novel visual analytics system that we have developed, incorporates in order to partially automate the time-consuming steps of the pipeline. With the bottleneck removed, analysts are fast-forwarded to their desired results, without having to manually examine hundreds or thousands of visualizations. Importantly, non-programmers benefit from tools like Zenvisage that can automate this pipeline and eliminate the need for code.

Zenvisage allows easy exploration of data through a sketching tool that allows users to simply draw a trend on a canvas of what they are expecting or would be surprised to see in their dataset, as shown in figure 1.1. Our system would then fire the necessary computation to process the sketch query and identify matches. The frontend gives several options for an interactive experience. First, the user uses panel A in figure 1.1 to select the dataset and relevant attributes. Once these attributes are selected, representative patterns are generated in panel B to kickstart an analyst's exploration. A user can drag these patterns to the sketching canvas for more detailed analysis. The frontend also includes many customizable features for more detailed analysis, such as using different comparison measures like DTW [6] or Earth Mover's distance [7], or interactive smoothing, and dynamic class creation in Panel E. Dynamic class creation allows users to compare several subsets of data at once, based on different filtering criteria. These features provide the user more easy ways to explore their data.

Figure 1.1: Zenvisage interactive visual query interface

For more in-depth exploration, Zenvisage provides the Zenvisage Query Language (ZQL) [8], for multi-step exploratory queries. A user can click on the ZQL Table button to show the ZQL panel as seen in figure 1.2, replacing the options panel. ZQL operates on visualization collections, meaning it is a query language operating at a higher level of abstraction than those that operate directly on data (such as SQL), as visualizations themselves can be considered queries. Our user study shows that even individuals who have never programmed before were able to use ZQL after a small training period of ten to fifteen minutes [8].

How does ZQL help a user who is looking for insights? For example, consider an economist who wishes to study a real estate dataset [9] to see if we are heading towards another housing

	X	Y	Z	Constraint
f1	x1<-{'year','month'}	y1<-{'soldprice','listingprice'}	z1<-'state','CA'	
f2	x1	y1	z2<-'state','NY'	
x2,y2<-argmin_{x1,y1}[k=1]DEuclidean(f1,f2)				
*f3	x2	y2	'state',{'CA','NY'}	

Add row Add process Submit

Figure 1.2: Zenvisage ZQL Interface

bubble in the US. First, the economist might want to see if the average sold price of houses is increasing over time. Currently, she would need to manually generate a sold price over time visualization one at a time for each state. Then, she would need to inspect these 50 visualizations one at a time, a tedious and cumbersome process. With ZQL, this user can write a few lines in our query language specifying what she is looking for—increasing patterns—and get the visualizations of states where the sold price over time is increasing, automating the manual data exploration process. In addition, the economist may choose to use the frontend interface instead to sketch an increasing pattern. The backend would execute the corresponding ZQL query, and return the visualizations of states where the sold price over time is increasing.

While our system provides ample support for the novice user, it is somewhat lacking for expert users. ZQL allows for complex queries, but is cumbersome to use. The query format is very rigid, and makes custom user function support difficult. Saving queries for future use is hard. Once Zenvisage becomes easily usable in other languages, analysts can incorporate it within their own workflows. Analysts can use Zenvisage to help explore their datasets efficiently to generate insights, then take those insights directly and integrate them within their workflow in their programming language, making the analysis experience more fluid. This is the basic motivation towards creating a general grammar for a data exploration scripting language, which can then be adapted to many languages. The outline of this thesis is the following:

- Chapter 2: We will first cover related work in visual analytics, databases, interactive processing and in automatic data visualization.
- Chapter 3: Before we dive into the details of the scripting language, we will describe the basics of the Zenvisage query language to help contextualize the motivation for a scripting language.
- Chapter 4: Then, we will describe in detail our ZQL query parser and executor.
- Chapter 5: Next, we will motivate the Zenvisage Scripting Language (ZSL), and formalize the grammar.
- Chapter 6: Finally, we will conduct a case study with R and ggplot [3] to showcase the power of the ZSL integration.

CHAPTER 2: RELATED WORK

Zenvisage draws from work in a number of areas, including analytics tools, databases, and interactive processing. Finally, there is recent work that has similar goals in automating the data analytics process.

Visual Analytics. There are a number of visual analytics tools available. Tableau [1] is one of the more popular ones. We draw inspiration from both Tableau and Polaris [10]. Other visualization tools include Spotfire [2] and visualization systems from the database community such as Google Fusion Table [11], and Profiler [12]. Most of these tools, like Tableau, provide one visualization at a time. While the options provided for analysis on each visualization is extensive, Zenvisage focuses on analyzing collections of visualizations at a time, reducing the need for manual trial-and-error. This makes our tool novel and more powerful for visual data exploration.

Databases. Relational Databases such as Postgres [13] can support interactive analysis through SQL. For novice data analysts, SQL may be too complex, and they would rather use visual analytics tools instead. For more advanced analysts, they can write complex SQL queries, and even achieve the same expressibility of some ZQL queries. These SQL queries, however, are cumbersome to write. On the other hand, Zenvisage operates on sets of visualizations, where each visualization itself operates on sets of data. This makes ZQL a higher level language compared to SQL and makes it easier to write visual analytics style queries.

Scalable Interactive Processing. In online analytical processing, concepts such as data cubes are prominent to achieve scalability. We can view a visual collection as a data cube. There is research towards scaling data cubes for big data [14], as well as work on interactive browsing of data cubes [15] [16]. However, these techniques do not encompass the full expressiveness of ZQL.

Automatic Data Visualization. Recent research considers automatic data visualization [17]. This work focuses on rating visualizations based on various criteria. All of their operations are encapsulated within Zenvisage, through our functional primitives that we describe in Section 3.2. SeeDB [18] supports a subset of Zenvisage's capabilities. ShowMe [19] and Voyager [20] support the recommendation of visualization mechanisms for the same data subset without selecting diffierent views of the data.

CHAPTER 3: ZENVISAGE QUERY LANGUAGE

Before we discuss the merits of a scripting language for Zenvisage, we must cover the basics of the Zenvisage Query Language or ZQL. Our scripting language relies on much of the innovation of ZQL for visual data exploration. ZQL provides users a powerful way to operate on collections of visualizations, allowing them to query for desired visual insights. The language treats visualizations as first-class citizens, allowing operations such as composition, sorting, filtering, and comparison. To facilitate querying, ZQL provides a table-like interface similar to Query by Example [21]. This gives non-expert users a more familiar and intuitive interface. Our user study shows that even users with minimal programming experience can use ZQL proficiently [8].

Each ZQL query consists of multiple rows, where each row operates on collections of visualizations. Each row consists of a column for the identifier, columns specifying the visualization collection, and finally a column specifying the processing operation.

A more detailed description of the language, as well as a formalization of the expressiveness of ZQL through a visual exploration algebra is provided in the Zenvisage paper [8].

3.1 VISUALIZATION COLLECTION

A visualization can be thought of as points plotted on an chart with an x and y axis. A Visualization Collection (VC) then is a set of these visualizations. In ZQL, the four columns of X, Y, Z, and Viz specify the VC in a given ZQL query row. The Name column is used to refer to the VC in subsequent process columns.

X and Y Columns. The X and Y columns of a ZQL row specify the attributes used for the x and y axes of the visualizations. For example, in Table 3.1, the first row returns a VC where each visualization has 'year' for its x-axis and 'sales' for its y-axis. A user can also specify a collection of values in each column. For instance, Table 3.2 has 'sales' and 'profit' for its Y Column. This means the VC will have visualizations for sales-over-years and profit-over-years. A user can specify all the possible values a column can take by specifying *. If multiple columns have collections, then a Cartesian product is preformed and a visualization is created for every combination.

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

Table 3.1: Query for the bar chart of sales over year for the product chair.

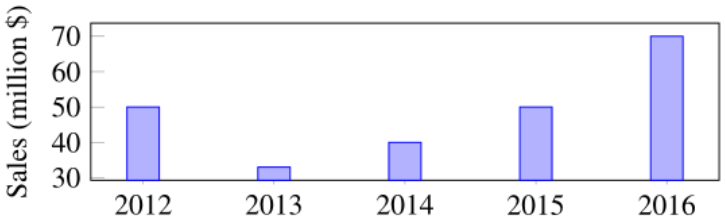

Figure 3.1: Bar Chart for query in Table 3.1

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

Table 3.2: Query for the sales and profit bar charts for the product chair

Z Column. The Z column subsets or slices the data. The notation is slightly different than the one for the X and Y columns. Here, the Z Column is written as $\langle attribute \rangle.\langle attribute\text{-}value \rangle$. Table 3.1 has a Z Column written as $'product'.'chair'$, so combined, the X, Y and Z columns retrieve the sales-over-year chart for only the product, chair. If * is used in place of 'chair', the Z column would be populated with all products, so a visualization for each product would be generated. ZQL can support multiple constraints on different attributes through additional Z columns that the user can specify. For example, if we add a Z column to table 1 with $'location'.'US'$, the sales-over-year chart for the product chair would only contain data for sales in the United States.

Viz Column. The Viz Column provides a plethora of visualization options for output. It describes the visualization type (such as bar chart or line chart), binning, and aggregation functions for the visualizations of the row. The format is $\langle type \rangle.\langle bin + aggr \rangle$. For example, Table 3.1 is a bar chart that aggregates the y values using SUM. The resultant visualization can be seen in figure 3.1. The visualization types are derived from the Grammer of Graphics [22]. For aggregation, ZQL supports the basic SQL aggregation functions. There is thus a lot of customization for output presentation, but also important aggregation and binning mechanisms that factor into the VC operations during the Process execution. For tables that follow we will omit the viz column.

3.2 PROCESS

The heart of ZQL and Zenvisage lies in the process column. This column specifies the parameters for operation, and must be flexible enough to support all the key data analytics operations somewhat declaratively. With the Process execution, users can filter visualizations based on a pattern, search for similar looking visualizations, discover representative

visualizations, and outliers. In order to compare visualizations from one set or another, or to find patterns in our VCs, there must be some kind of iteration order. Depending on the task, this order may change, thus requiring a mechanism designed for VC iteration.

Iteration over Visualization Collections. Recall what a VC is: a collection of visualizations, where each visualization is a collection of data points plotted on an XY chart. The points are specifically the y-axis attribute values for a given x-axis attribute value. Each visualization represents an attribute value of the Z attribute. For simple VCs, one simple iteration order is to go through each of the Z Column's attribute values: this is iterating over each visualization in a collection. This gives the evaluation order that is used in Table 3.3. The first row creates the VC for profit-over-years for every product. The second row creates the VC for sales-over-years for every product. The process column in the second row operates on these two collections by comparing each product's profit-over-year visualization with its sales-over-year visualization, in order to find the top 10 products whose sales-over-year pattern is most different from their profit-over-year pattern. The third row visualizes this result. This is an interesting insight and would lead to further analysis. Why does product X have increasing profits over the years, while declining sales? The user might now want to look for one or more external factors. Perhaps the product assembly process was refined, dropping the production costs.

However, we cannot just always use the Z column for iteration. Take for example Table 3.4. The first row has two X attributes, and two Y attributes. As stated earlier, ZQL will create the cartesian product. Thus, the first row creates four visualizations for its VC:

$$\{year, soldprice\}, \{year, listingprice\}, \{month, soldprice\}, \{month, listingprice\}$$

as the X and Y axis, respectively, which each visualization with the state of California as the Z axis. The second row one again creates four visualizations, put with Z axis of New York for each visualization. Iterating over Z would mean nothing gets compared! We need to iterate over the pair of $\{x1, y1\}$, which will compare each pair in the first VC with its corresponding one in the second VC.

Thus, ZQL opts for the more powerful *dimension-based* iteration. Users can choose which, and how many, dimensions are needed for iteration. Each of the columns gets assigned a separate iterator called an axis variable.

Axis Variable. Axis Variables are created with this syntax:

$$\langle variable \rangle \leftarrow \langle collection \rangle$$

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

Table 3.3: Query which retrieves the top 10 profit visualizations for products which are most different from their sales visualizations

Name	X	Y	Z	Process
f1	x1 ← {'year','month'}	y1 ← {'soldprice','listingprice'}	z1 ← 'state'.'CA'	
f2	x1	y1	z2 ← 'state'.'NY'	x2,y2 ← $argmax_{x1,y1}[k = 1]D(f1, f2)$
*f3	x2	y2	'state'.{'CA','NY'}	

Table 3.4: Query which returns the X,Y pair where California and New York are most different

$\langle v1 \rangle \leftarrow \langle' product'.* \rangle$ sets the axis variable $v1$ with a collection of the Z dimension, in this case all the products in the dataset. The process column returns a list of axis variables. Table 3.3 returns one axis variable, while the process column in Table 3.4 returns two axis variables. This Axis Variable is an important concept that we will come back to with the scripting language.

Operations on Visualization Collections. Once the process knows the iteration order, it can operate on the VCs. A process column knows which axis variables it needs to iterate over through the subscript of the formula. For example, in Table 3.3 the process column is written as

$$v2 \leftarrow argmax_v 1[k = 10]D(f1, f2)$$

so the axis variable for iteration is $v1$, and the output axis variable is $v2$. $D(f1, f2)$ returns the "distance" between visualizations in $f1$ and $f2$, following the iteration order. More specifically, for Table 3.3, for every product in $v1' \in v1$, retrieve the visualization(s) $f1[z : v1']$ and visualization(s) in $f2[z : v1']$ and calculate the "distance" between these pairs. Then, $argmax_v 1[k = 10]$ retrieves the 10 $v1'$ values where the distance between the pairs are largest. Thus $v2$ is a subset of $v1$, containing the top 10 products whose profits-over-year visualization is most different than its sales-over-year visualization.

Formal Structure. The general basic structure of the Process Column is:

$$\begin{aligned} &\langle argopt\rangle_{\langle axvar\rangle}[\langle limiter\rangle]\langle expr\rangle \quad \text{where} \\ \langle expr\rangle &\rightarrow (\max|\min|\sum|\prod)_{\langle axvar\rangle}\langle expr\rangle \\ &\rightarrow \langle expr\rangle(+|-|\times|\div)\langle expr\rangle \\ &\rightarrow T(\langle vc\rangle) \\ &\rightarrow D(\langle vc\rangle, \langle vc\rangle) \\ \langle argopt\rangle &\rightarrow (\text{argmax}|\text{argmin}|\text{argany}) \\ \langle limiter\rangle &\rightarrow (k = \mathbb{N}|t < \mathbb{R}|p = \mathbb{R}) \end{aligned} \tag{3.1}$$

where $\langle axvar\rangle$ refers to the axis variables, $\langle vc\rangle$ refers to visualization collections.

Functional Primitives. A key part of this syntax are the T and D functions. These are two functional primitives ZQL incorporates to facilitate accelerated visual exploration.

- $[T(f1) \rightarrow \mathbb{R}]$: T is a function that takes each visualization $f1' \in f1$ and returns a real number measuring some visual property of the trend in f'. For example, a user might be looking for stocks that are growing: stocks whose price-over-time visualization shows "growth". T in this case can return a positive number based on how overall upward the pattern is. Other measures could be skewness, number of peaks, noisiness of visualization, or number of clusters.

- $[D(f1, f2) \rightarrow \mathbb{R}]$ D is a function that takes takes visualizations from $f1$ and $f2$ (in order based on the axis variable iterator) and measures distance (or dissimilarity) between them. Example distance functions include Euclidean distance, Earth Mover's Distance [7], or the Kullback-Liebler Divergence. D can also be measured by difference in number of peaks, or slopes, or some other property.

These functions are general, in the sense that ZQL supports different implementations of T and D and can also allow user functions with their own implementations. However, user function support in ZQL is more cumbersome than in the scripting language, as we will see.

We will be revisiting the formal structure of ZQL when creating the scripting language. The scripting language's process mechanism needs to be just as expressive. Before that, we will cover the ZQL execution engine, where the processing of visualizations actually takes place.

CHAPTER 4: QUERY PARSING AND EXECUTION

Once a user submits a query using our query language, it needs to be parsed and then executed by our visual exploration system. This chapter will describe in detail our implementation of ZQL execution.

4.1 QUERY PARSING

Our system parses the entire query, row by row. With the ZQL Table frontend interface, the ZQL query written by the user is first converted into a JSON object representing the ZQL query through regex matching. Then, this JSON format is sent to our Java execution engine where it is first validated, then converted into a corresponding Java object that represents a ZQL Query. This object is then parsed row by row to generate the query graph, which represents the execution plan. The front-end sketching and drag and drop features has an additional first step: the visualization on the sketch canvas is used to create a ZQL row representing that visualization. Then, additional ZQL rows are added based on what the user wants–for example, similarity search using Euclidean distance. Finally, the complete ZQL query is sent to the backend to process as described in the ZQL table case.

4.2 QUERY GRAPH

In our execution engine, we first convert the ZQL Java object into an equivalent graph form. The query graph represents the flow of data through the various visual analytics operations. This embodies the idea of dataflow programming. The graph is a directed acyclic graph (DAG) which provides a clear execution plan. In addition, this graph allows for various execution optimizations, such as parallelization.

The graph in figure 4.1 shows the previous example query in Table 3.3 in a graph form. The variables f1, f2, and f3 refer to the visual collections in row 1, 2, and 3, respectively. The variable p1 refers to the process in row 2. The Query Graph consists of two classes of nodes: Visualization Collection Nodes (VCNode) and Process nodes (PNode). Entry nodes are a subset of VCNodes, and output Nodes are subsets of VCNodes and PNodes. For each ZQL row, the parser creates a VCNode for the visual collection, and a PNode for the visual task (the process).

VCNode. A VCNode represents the VC of one row of the ZQL query, and thus holds the data. The input is the visual component metadata: the information to create the

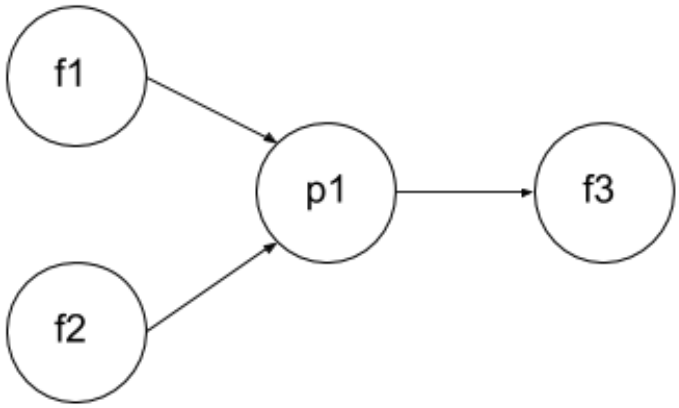

Figure 4.1: An example query plan in graph form.

visualization collection. The output is the VC data itself. This generic design allows us to support different visualization types, such as line charts or scatter plots.

PNode. A PNode represents the process column of one row of the ZQL query. Processes operates on VCs. For instance, the D functional primitive operates on two VCs, comparing the distances between visualizations in one with visualizations in the other. Thus, the PNode for this process would depend on two VCs represented by two VCNodes with arrows to the PNode. In addition, the Process node holds the parameters relevant for the operation that will be operated on, such as the count that limits the number of output axis variables. The output contains the resultant axis variables.

Entry Nodes. Entry nodes are VCNodes that do not depend on any other node: i.e., the query execution engine can start at the entry nodes. A valid query graph always has at least one entry node.

Output Nodes. Output nodes are nodes marked for output (by the user). A user might want the results of a PNodes (which would return a list of axis variables), or the results of a VCNode (which would return a VC).

The Graph creator takes the Java ZQL Table object and does two passes to transform it into the query graph. The first pass creates all the nodes. A VCNode is created for every collection of visualizations (including singleton collections), while a PNode is created for every operation (or process). Then a second pass adds all the dependencies. PNodes are dependent on the VCNodes that are in the process. For every such dependency, an input edge from those VCNodes to the PNode is created. VCNodes may be dependent on a PNode if any of its axis variables are derived from the result of a process. For every such dependency, an input edge to a VCNode from those PNodes is created.

4.3 QUERY EXECUTION

Once the query graph is created, it is passed to the executor. The general execution approach can be described as the following: first, search for a node to execute that either has no parents or one all of whose parents have been executed. Then, execute that corresponding node. Finally, repeat the approach until all nodes are executed. For our implementation, the query graph executor uses breadth-first search, starting by adding all entry nodes to the queue. Then, it executes nodes one node at a time, by removing the first node in the queue. During execution, a VCNode retrieves the data for the visualization collection, while a PNode executes the process.

VCNode Execution. In ZQL, the VCNode retrieves data from PostgreSQL. The appropriate SQL group-by queries are issued to the database to retrieve the data for multiple visualizations at once. For line charts, the query we use in the backend has the form:

```
SELECT X, agg(Y), Z1, Z2, ...
FROM dataset
WHERE C(X,Y,Z1,Z2...)
GROUP BY X,Y,Z,Z2...
ORDER BY X,Z,Z2...
```

X,Y are the column attributes, agg(Y) is some aggregate function on Y column, and Z1, Z2... are the attributes/dimensions we are iterating over, and c(X,Y,Z1,Z2...) are the optional constraints. Order by is used to ensure all rows in a visualization are grouped together and in order. An example query for the VCNode for f1 in Table 3.3 has the form of:

```
SELECT year, SUM(profit), product
FROM real_estate
GROUP BY year,product
ORDER BY year,product
```

Note that we do not need to order by the Z attributes. Instead, we can create a map structure that stores the X,Y points associated with each product in this example. As we iterate through the rows of year, SUM(profit), product, we add the ordered year, SUM(profit) pair into the corresponding Z attribute in our map. We lose the sort order for the product, but save on execution time.

Besides support for line charts, our system supports other visualization types including bar charts and scatter plots. For scatter plots, we do not aggregate on Y.

PNode Execution. The appropriate pseudocode is generated to execute the Process,

based on the structure of the expression. Expanded out, one example of our formal process structure looks as follows:

$$\langle argopt \rangle_{v0}[k = k'] \Big[\langle op1 \rangle_{v1} \Big[\langle op2 \rangle_{v2} ... \Big[\langle opm \rangle_{vm} T \langle f1 \rangle \Big] \Big] \Big] \tag{4.1}$$

where $\langle argopt \rangle$ is either argmin, argmax, or argany and $\langle op \rangle$ is one of $(min|max|sum|prod)$. For each dimension $\langle op \rangle$, ZQL generates a new nested for loop to process it, one value of that dimension at a time. Thus, every loop iterates one value of the time, evaluating the inner expression, then applying the overall operation.

If a node successfully executes, its children nodes are added to the queue. If the node does not successfully execute, it is due to it being blocked. Some dependencies have not been executed yet. For example, a PNode needs to wait for the VCNodes it depends on to finish executing first to load the data. So the node is added to the back of the queue again. Or, a VCNode needs to wait on a PNode to generate the axis variables it will use for one of its X,Y,Z columns, so it is added back to queue again. This structure allows for parallelization: each entry node can be executed in parallel, and certain subgraphs can be executed in parallel with other subgraphs that do not have unfinished dependencies. Since the graph is a DAG, the execution is guaranteed to finish.

Finally, the query executor is linked to our frontend. After a user finishes sketching or using our tabular interface in our frontend, our frontend fires a ZQL query to our backend. Our backend receives the ZQL query and the query exeucution engine described above handles the rest. Outputs are then sent back to the frontend for display. Now with the query language explained, we can move on to the scripting language, and explain why we would want two languages for one system even if it sounds redundant.

CHAPTER 5: ZENVISAGE SCRIPTING LANGUAGE

5.1 MOTIVATION

With the interactive sketch interface and the query language, our system provides novices with ample support and ability to explore their data visually. However, this is often not sufficient for expert users. *Expert users want more control over the internals of the systems and an integrated workflow that help streamline their analysis when using Visual Query Systems* [23].

Currently, data scientists using our system need to interact through a graphical interface. They can do basic exploration with the sketching and recommendation functionalities, or write more complex queries with ZQL through the table interface. But their workflows usually include data preprocessing before data exploration and deeper analysis after. After using Zenvisage to narrow down a broad search, more specialized tools are used. In our user study [23] we worked with data scientists working on astrology datasets as part of a multi-institutional project called the Dark Energy Survey (DES). The scientists use a multi-band telescope that takes images of 300 million galaxies over 525 nights to study dark energy [24]. One task is to find astrophysical transients, objects whose brightness changes dramatically as a function of time, such as supernovas or quasars. This can be visualized as a time series chart of brightness observations of objects (extracted from images of patches in the sky) over time, and Zenvisage allows these scientists to quickly find these potential astrophysical transients, narrowing down a huge set of visualizations automatically. The data scientists in this study would use Zenvisage for initial data exploration, then do more sophisticated downstream analysis and summary statistics to confirm the astrophysical transients. This creates a clunky workflow, where the expert user must switch context and transfer data between different tools every time they wanted to do analysis with Zenvisage followed by further analysis in a programming language like Python. To solve this issue, we need to remove the unnecessary and tiresome context shift. To do so, we need to integrate Zenvisage directly into a user's workflow—into their programming language of choice.

Thus, we need to adapt ZQL for use within programming languages. The original goal of our declarative query language was to provide users the full expressiveness of our visual analytic engine, allowing them to write complex queries and derive more meaningful insights. However, the tabular format of ZQL is not suitable for integration into a workflow. While it is possible to adapt the table-like structure into a programming language, it is less intuitive than a scripting language with the same expressiveness as ZQL that integrates much more

easily into the target language. This is a shift from the more visually appealing but simple system that the table interface provides to a more complex but flexible system that the scripting language provides, mainly through the use of functions and new classes. Thus, the Zenvisage Scripting Language (ZSL) is an adaptation of ZQL for use with programming languages, which brings about its own set of unique challenges. This will be discussed in the next section when we describe the grammar developed for ZSL.

Being able to use Zenvisage directly in a programming language such as Python through ZSL would allow users to take their exploratory work and transfer the information directly into other python libraries or tools for further analysis. For example, let's say a user wants to find states where the sold prices of houses have a downward trending pattern. If a user needed to do this manually, they would have to examine 50 different visualizations, one for each state, one at a time in order to pick out the ones that are trending down. ZSL will find the list of such states automatically, and now that information is ready to use. The user's workflow can now take this list of states with a downward trend and correlate this with location information in a tool such as GeoPandas [25] to narrow down to the specific troublesome counties. Thus, the entire workflow is in one general script that saves on context switch time.

Our goal is to create a scripting language that can be integrated into other programming languages. However, programming languages come in different styles and flavors, so it is important to create one generic overarching grammar to serve as the backbone or template.

5.2 ZENVISAGE SCRIPTING GRAMMAR

The grammar for Zenvisage should be generic and encapsulate the same expressiveness of ZQL (and thus the visual exploration algebra). It is important to establish the grammar upfront, leading to consistent language integrations. The syntax of ZSL implementations in languages such as R and Python will differ, but will be based on the same general grammar. This allows the implementations to take advantage of their respective language, while still being expressive and semi-portable. Chapter 6 will discuss our implementation of ZSL with R, and how ZSL takes unique advantage of that language. The grammer is written in extended BNF [26].

5.3 AXIS VARIABLE

The Axis Variable simply references a set of elements from the axis columns. Thus, the rule for it is fairly simple. However, we must also allow for the Z column syntax of $\langle attribute \rangle.attribute\text{-}value$ as well. Let $\langle string \rangle$ be the set of all strings (in the target language):

$\langle term \rangle$ ::= $\langle string \rangle$
| $\langle string \rangle$ '.' $\langle string \rangle$
| $\langle string \rangle$ '.' '{' $\langle string \rangle$ {',' $\langle string \rangle$} '}'

$\langle axvar \rangle$ ::= '{' $\langle term \rangle${ ',' $\langle term \rangle$} '}'

Since Axis Variables is a set of elements, we can define set operations on it, to support set union, set difference, and set intersection:

$\langle expr \rangle$ ::= $\langle axvar \rangle$ {('$\cup$' | '\' | '$\cap$') $\langle axvar \rangle$}

Finally, for assignment, we have:

$\langle assign \rangle$ ::= $\langle term \rangle$ '=' ($\langle axvar \rangle$ | $\langle expr \rangle$)

This allows us to write axis variables such as $x1 = \{month, year\}$, or $z1 = \{state.*\}$, or $z1 = \{state.NY, state.CA, state.NJ\}$, which can be condensed as $z1 = \{state.\{NY, CA, NJ\}\}$. Set operations are supported as well. Given $z1 = \{state.NY, state.CA, state.NJ\}$ and $z2 = \{state.NY, state.TX\}$, $z3 = z1 \cup z2 = \{state.NY, state.CA, state.NJ, state.TX\}$. $z4 = z1 \cap z2 = \{state.NY\}$ and $z5 = z1 \backslash z2 = \{state.CA, NJ\}$

5.4 VISUALIZATION COLLECTION

The next part is to express a VC. From ZQL, remember we need X, Y, and Z axis variables:

$\langle vc \rangle$ ::= '{' $\langle axvar \rangle$ ',' $\langle axvar \rangle$ ',' $\langle axvar \rangle$ { ',' $\langle axvar \rangle$ } { ',' $\langle string \rangle$ } '}'

The three axis variables are necessary. ZQL supports multiple Z columns, so axis variables can be added optionally. For example, $v1 = \{year, soldprice, product.*, location.US\}$ visualizes the soldprice-over-years trend for all products in the United States only. The remaining strings are for the optional visualization options.

Visual Collections are also sets, so we can define set operations on them as well:

$\langle var \rangle$ ::= ($\langle axvar \rangle$ | $\langle vc \rangle$)

$\langle expr \rangle$ ::= $\langle var \rangle$ {('$\cup$' | '\' | '$\cap$') $\langle var \rangle$}

And assignment becomes:

⟨*assign*⟩ ::= ⟨*term*⟩ '=' (⟨*var*⟩ | ⟨*expr*⟩)

For example, we can write $v1 = \{year, profit, product.*\}$, which represents the profit over year visualizations for all the products. This also supports multiple axis variables. As another example, consider $v2 = \{\{month, year\}, \{soldprice, listingprice\}, product.chair\}$, which consists of four visualizations: one for each X, Y pair for the product chair.

5.5 PROCESS

Recall the formal structure of a process function in ZQL. The Process function at a high level actually comprises two steps: first, it applies a visual operator on some number of VCs in a specific iteration order specified by the axis variables, then, it is filtered by the ⟨*argopt*⟩ operator with a limiter to return a subset of the input axis variables. In our query lanague, it is written in a single step, but for our scripting language, we split these into two separate parts. This allows assigning the partial result (the visual operator) to some variable, allowing multiple ⟨*argopt*⟩ filters to apply to the same visual operation. For instance, a user might want the top 10 performing stocks, as well as the bottom 10 performing stocks for further analysis. Splitting it into two function saves on redundant execution time.

⟨*list*⟩ ::= ⟨*term*⟩{ ',' ⟨*term*⟩ }

⟨*argopt*⟩ ::= ('`argmin`' | '`argmax`' | '`argany`')

⟨*limiter*⟩ ::= '`count(`' ⟨*integer*⟩ '`)`'
| '`cond(`' '`gt`' | '`geq`' | '`lt`' | '`leq`' '`)`'
| '`percent(`' ⟨*p*⟩ '`)`'

⟨*func*⟩ ::= ⟨*term*⟩'`(`' ⟨*axvar*⟩ '`,`' ⟨*func*⟩ '`)`' | ⟨*term*⟩'`(`'⟨*list*⟩'`)`'

⟨*process*⟩ ::= '`process(`' ⟨*argopt*⟩ '`,`' ⟨*axvar*⟩ { '`,`' ⟨*axvar*⟩ } '`,`' ⟨*limiter*⟩ '`,`' ⟨*func*⟩ '`)`'

⟨*assign*⟩ ::= ⟨*list*⟩ '`=`' ⟨*process*⟩

Where ⟨*p*⟩ is a decimal in the range $[0, 1]$ and gt, geq, lt, leq are greater than, greater than equal to, less than, and less than equal to, respectively. As an example, we could have:

$$v3 = process(argmax, v1, count(10), sum(v2, D(f1, f2))$$

In this example, for every $v1' \in v1$, we sum the distance between every $f[z : v1']$ and $f2[v : v2']$ for every $v2' \in v2$. Then, we return the top 10 $v1'$ with greatest distance.

5.6 COMPLETE GRAMMAR

The complete grammar combines all the pieces above, with a few additions like the actual visual output.

⟨*term*⟩ ::= ⟨*string*⟩
| ⟨*string*⟩ '`.`' ⟨*string*⟩
| ⟨*string*⟩ '`.`' '`{`' ⟨*string*⟩ {'`,`' ⟨*string*⟩} '`}`'

⟨*list*⟩ ::= ⟨*term*⟩{ '`,`' ⟨*term*⟩ }

⟨*argopt*⟩ ::= ('`argmin`' | '`argmax`' | '`argany`')

⟨*limiter*⟩ ::= '`count(`' ⟨*integer*⟩ '`)`'
| '`cond(`' '`gt`' | '`geq`' | '`lt`' | '`leq`' '`)`'
| '`percent(`' ⟨*p*⟩ '`)`'

⟨*axvar*⟩ ::= ⟨*term*⟩{ '`,`' ⟨*term*⟩ }

⟨*vc*⟩ ::= '`{`' ⟨*axvar*⟩ '`,`' ⟨*axvar*⟩ '`,`' ⟨*axvar*⟩ { '`,`' ⟨*axvar*⟩ } { '`,`' ⟨*string*⟩ } '`}`'

⟨*expr*⟩ ::= ⟨*var*⟩ {('∪' | '\ '| '∩') ⟨*var*⟩ }

⟨*func*⟩ ::= ⟨*term*⟩'`(`' ⟨*axvar*⟩ '`,`' ⟨*func*⟩ '`)`' | ⟨*term*⟩'`(`'⟨*list*⟩'`)`'

⟨*process*⟩ ::= '`process(`' ⟨*argopt*⟩ '`,`' ⟨*axvar*⟩ { '`,`' ⟨*axvar*⟩ } '`,`' ⟨*limiter*⟩ '`,`' ⟨*func*⟩ '`)`'

⟨*assign*⟩ ::= ⟨*list*⟩ '`=`' (⟨*axvar*⟩ | ⟨*expr*⟩ | ⟨*func*⟩ | ⟨*process*⟩)

⟨*visualize*⟩ ::= '`visualize(`' ⟨*vc*⟩ '`)`'

⟨*statement*⟩ ::= ⟨*assign*⟩ | ⟨*expr*⟩ | ⟨*process*⟩ | ⟨*visualize*⟩

⟨*program*⟩ ::= {⟨*statement*⟩ ⟨*line-break*⟩ }

⟨*visualize*⟩ is used to visualize a visualization component, usually using visualization libraries such as ggplot. ⟨*statement*⟩ represents what each line of ZSL can be, and ⟨*program*⟩ is the entire ZSL script, representing the expressiveness of the language. Note ⟨*list*⟩ and ⟨*axvar*⟩ have the same definition, as an axis variable really just is a list of elements. These are separated to emphasize the importance of the axis variable, only using ⟨*axvar*⟩ wherever axis variables would actually be used.

Return Values. ⟨*process*⟩ and ⟨*func*⟩ returns a list of axis variables. We can optionally also attach the real number output for each value in the axis variable (or for each pair of

axis variables for D functions) for the "growth" or "distance" number, so it can be used for further analysis as well.

Here is an example complete script:

$$\begin{aligned} v1 &= product.* \\ f1 &= \{year, sales, v1\} \\ f2 &= \{year, sales, v1\} \\ v2 &= process(argmax, v1, count(10), sum(v2, D(f1, f2)) \\ f3 &= \{year, sales, v2\} \end{aligned} \tag{5.1}$$

This script returns the sales visualization for the 10 products whose sales visualization are most different from the others, which is a way of finding the outliers.

Here is another example complete script that translate the ZQL query in Table 3.4:

$$\begin{aligned} x1 &= \{year, month\} \\ y1 &= \{soldprice, listingprice\} \\ f1 &= \{x1, y1, state.CA\} \\ f2 &= \{x1, y1, state.NY\} \\ x2, y2 &= process(argmax, x1, y1, count(1), D(f1, f2)) \\ f3 &= \{x2, y2, state.\{CA, NY\}\} \end{aligned} \tag{5.2}$$

This script returns the X, Y pair, for instance *year* and *listingprice*, where the state of California and New York are most different. The grammar has the same expressive power of ZQL and is more suitable for programming languages. In the next section, we will integrate ZSL with R and ggplot.

CHAPTER 6: ZENVISAGE AND R

R is a programming language that is well suited for statistical computation and thus a language many data analysts know and use. In addition, it has a great visualization tool through ggplot2 [3] which is very customizable. ggplot2 is an implementation of the Grammar of Graphics specified by Wilkinson [22]. This makes it a primary target language to integrate with ZSL. ZenvisageR is our implementation of ZSL in R, using ggplot2 as the visualization tool. The scripts do not exactly match the general grammar, but follows them more as a guideline and adapts it to the R language.

6.1 CASE STUDY: ASTRONOMY

Our case study begins with a real world use case of Zenvisage from our user study with a few data scientists doing research in the field of Astronomy (or astro for short) [23]. One data scientist wanted to find supernovae in an astro dataset. These stars are characterized by an existence of a peak in brightness above a certain amplitude with an appropriate width of the curve. Thus, for every object in the dataset, we want to find the objects whose flux-over-time visualizations have such a peak. The scientist can use Zenvisage to query for these peak patterns and generate a set of candidate objects. However, not all these candidate objects would be supernovae. Besides manual inspection, further analysis can be done to filter out the time series that were too faint to be supernovae. In the following section we will step through how a data scientist would find such supernovae using R, with ggplot2 for visualization and Zenvisage for visual data exploration.

6.2 ZENVISAGE AND R

Importing and Preprocessing. The initial steps correspond to including the relevant libraries, and loading the relevant dataset. For this example, we will assume the dataset is already preprocessed, and has at least the following columns: Object, Flux, Seconds. The object represents the potential astrophysical transients. The flux represent the measured brightness for an object, and the time represents the time of a row of measurements. Finally, we assume the dataset is stored in a .csv file called kepler.csv for simplicity of the example. The R code then to setup this dataset would look similar to this:

```
1 library(ggplot2)
2 library(zenvisage)
3 kepler <- read.csv("kepler.csv")
```

Visualization Collection. Now, we need to generate the visualization collection that we want to operate on. We want to get the flux-over-time visualizations in order to find peaks in them. The corresponding ZSL code is:

$$f1 = \{Seconds, Flux, Object.*\}$$

In R, we will use a ggplot2 object to represent a new Visualization Collection:

```
4 f1 <- ggplot(kepler, aes(x=Seconds, y=Flux, color=Object))
```

This serves the same purpose, representing the visualization collection columns through the aesthetics (aes() function). ggplot require data in the form of a dataframe, thus we pass in the kepler dataset. Then, the aesthetics set the x, y, and z attributes. Here, color sets the z attributes. Since the dataframe contains the entire dataset, it is akin to writing $Object.*$ in ZSL. If the data is too large, the relevant subset can be used instead, for example through a SQL query if the data is stored in a database.

Process and Visual Operators. The process and visual operator syntax remains largely similar to the general grammar, just adapted to suit the language syntax. It outputs a list of lists of axis variables. In order to find supernovae, we need to look for peaks, so we will call our vizPeaks functional primitive, which is one of the $T(f1) \rightarrow \mathbb{R}$ functional primitives.

```
5 temp <- vizPeaks(f1)
6 v1 <- process(argmax, kepler$Object, vizCount(50), temp)
7 f2 <- ggplot(subset(kepler, Object %in% v1))
```

The R script here uses the vizPeaks function. For a visualization it gives a real number represent how "peak-like" the visualization is. Through the process function, we can understand what the first two lines of code are doing: for every object in the kepler dataset, find the "peakiness" of its flux-over-time visualization. Then, return the 50 that show the peaks with greatest amplitude. Now we can get the subset dataframe corresponding to only these 50 objects, and create a new ggplot to represent this subset vc.

Visualizing. Integrating with R and ggplot2 allows easy visualization.

```
8 f2 <- f2 + aes(x=Seconds, y=Flux, color=Object)
9 f2 <- f2 + geom_line()
10 plot(f3 + facet_wrap(~Object)
```

The first line adds the axes for display purposes, and the second line adds the viz option to display a line chart. The final line plots every Object (in the subset data frame) as a separate plot in the same window. ggplot2 provides numerous visualization customization options, so the user can customize it to display exactly what he wants. Now, the scientist can manually inspect this subset to confirm or reject supernovae candidates.

Further Analysis. At this point, the scientist can add further analysis if needed, such as filtering out the visualizations that are not bright enough. The list of these 50 supernovae candidates are readily available for him to use directly in the language. In addition, the subset dataframe is easy to generate for further analysis as well. No context switch is necessary, all the code is in one script.

Advantages of Integration. There are several advantages to this integration.

- **Ease of Use** For any one familiar with the target language, R, and the ggplot2 package, they should be immediately familiar with ZenvisageR. Our data structure, the VC, is represented in R by a ggplot with the x, y, and z axes defined through aesthetics $aes(x, y, z)$. Then, our visual operators like vizCompare or vizPeaks are black boxes that take care of operation, either doing computation locally, or calling an external Zenvisage library written for optimization. For the end user, they does not have to worry about any of this.

- **In-Depth Analysis** R has many statistical and data analytic packages. We can use some filtering provided by R, before sending data to the process, for instance. And, once the process is done, we can send the results to another package for more detailed analysis.

6.3 DESIGN CONSIDERATION: PORTABILITY

There are valid arguments for making the library as portable as possible, as this is how most database management systems (DBMS) are built. A mySQL script or postgres script gets executed by their respective SQL engine. The entire flow is separate from the user's other tools, thus requiring database connectors to bring the relevant data into their workflow. By being separate, they are more portable: just the connectors need to be created to plug into specific languages. In addition, once users learn SQL, it should be easy for them to use it across multiple RDBMS implementations. Other advantages include in-house optimizations: better storage techniques for the data, and faster query retrieval.

The other option is to integrate Zenvisage directly with programming languages, so users can do visual exploration with just a few functions and new variables. Our package can

be fairly efficient if the important part of our functions, mainly the process, are coded in a language where we target our optimizations for, be it C/C++ or another language. Then our process function in another language like Python would call our process function in C++. All these calls are abstracted away from the user. This integration would allow users to pretty easily add our package and use our relevant functions as they please, without changing much of their existing workflow.

In addition, this allows us to be data storage agnostic. Depending on the size of their data, they may choose different RDBMS or NoSQL databases.

There are still definite efficiency issues, such as how a language handles a lot of data, but the most important goal of this is to allow more data analysts to get their hands on Zenvisage. Ease of use is the most important factor here.

6.4 FULL SCRIPT

```
1  library(ggplot2)
2  library(zenvisage)
3  kepler <- read.csv("kepler.csv")
4  f1 <- ggplot(kepler, aes(x=Seconds, y=Flux, color=Object))
5  temp <- vizPeaks(f1)
6  v1 <- process(argmax, kepler$Object, vizCount(50), temp)
7  f2 <- ggplot(subset(kepler, Object %in% v1))
8  f2 <- f2 + aes(x=Seconds, y=Flux, color=Object)
9  f2 <- f2 + geom_line()
10 plot(f3 + facet_wrap(~Object)
```

Generalization. This was just one example of ZSL integration in R, but ZSL supports more than just what is shown here. Let's take an example code that initializes a vc in ZSL:

$$f1 = \{\{*\}\{*\}, state.CA\}$$

. The $*$ for the X and Y axis variable means consider all X attributes and all Y attributes, respectively. Thus, $f1$ consists of every X, Y pair for the state of CA. This is difficult to directly describe in ggplot2 without manual reshaping of data. Users can use the package GGally, an extension of ggplot2 that can easily create an object that contains all the visualization pairs. The user would first subset the dataframe to contain the data for the state of California only, then use the function *ggpairs*() to generate the visualization pairs.

Finally, ZSL will accept this object into the process and continue with the workflow. Another solution is to add variables to the ggplot object for zenvisage use. For instance, if we write $f1\$x = c(\text{“}year\text{”}, \text{“}month\text{”})$, ZSL can interpret this as using *year* and *month* as the attributes for the X axis. For the multiple z column case, we can also add additional aesthetics such as $aes(z1, z2, ...)$, or add the extra variables directly to the ggplot object, such as $f1\$z1$ and $f1\$z2$.

The integration of ZSL and R will allow data scientists to have the control they want when using this powerful visual query system.

CHAPTER 7: FUTURE WORK

7.1 LANGUAGE SUPPORT

Our primary goal is to increase the adoption of our visual analytics tool to more data analysts. This means supporting more languages, especially those with powerful data analytical packages. Our targets include Python, Julia, and Scala.

7.2 SINGLE SOURCE SERVICE

ZSL is pluggable with any data source. However, another approach would have Zenvisage as a separate service, more like a database engine. Thus we can support large datasets and optimize them for visual analytics. We are exploring Thrift [27] for this.

7.3 OPTIMIZATIONS

The basic process functionality in the implemented example of ZenvisageR is fully expressive, but is rather slow. We decided it important to release the tool so data analysts can use it as soon as possible and add in optimizations for larger datasets later. Once datasets get really large, both memory and runtime complexity becomes important issues. Data would have be stored in various data sources, from RDBMS to more distributed storage solutions such as BigTable [28], HBase with Hadoop [29] or with sharding in MongoDB [30] . In addition, the main computation can be improved through parallelization, or using more cluster friendly computing such as Spark [31]. On the computation side, an external Zenvisage library focused on improving process times would be valuable too.

CHAPTER 8: CONCLUSION

We propose the Zenvisage Scripting Language (ZSL) to address the needs of data analyst experts, who find the Zenvisage tool inadequate. The Zenvisage Query Language gives these experts the power to express complex visual queries in a few lines of ZQL. The query executor makes the query language efficient on large datasets. However, in common data analysis workflows, ZQL has some issues related to the table format that make it ill-fitted for the job. Instead, ZSL allows users to integrate Zenvisage into their existing workflow, saving on context switch time and providing the expressive capabilities of ZQL naturally. The scripting language faces different challenges compared to the query language, thus requiring its own formal specification. Finally, we show an example of the scripting language in action, showing the benefits users get with ZenvisageR. Our work is a promising step towards a scripting language for visual data exploration.

REFERENCES

[1] "Tableau," 2018. [Online]. Available: https://www.tableau.com/

[2] C. Ahlberg, "Spotfire: an information exploration environment," *ACM SIGMOD Record*, vol. 25, no. 4, pp. 25–29, 1996.

[3] H. Wickham and W. Chang, "An implementation of the grammar of graphics," *WWW: http://ggplot2. org*, 2016.

[4] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

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

[6] M. Müller, "Dynamic time warping," *Information retrieval for music and motion*, pp. 69–84, 2007.

[7] Y. Rubner, C. Tomasi, and L. J. Guibas, "The earth mover's distance as a metric for image retrieval," *International journal of computer vision*, vol. 40, no. 2, pp. 99–121, 2000.

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

[9] "Zillow real estate data," 2018. [Online]. Available: https://www.zillow.com/research/data/

[10] C. Stolte, D. Tang, and P. Hanrahan, "Polaris: A system for query, analysis, and visualization of multidimensional relational databases," *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 1, pp. 52–65, 2002.

[11] H. Gonzalez, A. Y. Halevy, C. S. Jensen, A. Langen, J. Madhavan, R. Shapley, W. Shen, and J. Goldberg-Kidon, "Google fusion tables: web-centered data management and collaboration," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 1061–1066.

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

[13] M. Stonebraker and G. Kemnitz, "The postgres next generation database management system," *Communications of the ACM*, vol. 34, no. 10, pp. 78–92, 1991.

[14] Z. Wang, Y. Chu, K.-L. Tan, D. Agrawal, A. E. Abbadi, and X. Xu, "Scalable data cube analysis over big data," *arXiv preprint arXiv:1311.5663*, 2013.

[15] S. Sarawagi, "Explaining differences in multidimensional aggregates," in *VLDB*, vol. 99, 1999, pp. 7–10.

[16] G. Sathe and S. Sarawagi, "Intelligent rollups in multidimensional olap data," in *VLDB*, vol. 1, 2001, pp. 531–540.

[17] X. Qin, Y. Luo, N. Tang, and G. Li, "Deepeye: Visualizing your data by keyword search," *EDBT Vision*, 2018.

[18] M. Vartak, S. Madden, A. Parameswaran, and N. Polyzotis, "Seedb: supporting visual analytics with data-driven recommendations." VLDB, 2015.

[19] J. Mackinlay, P. Hanrahan, and C. Stolte, "Show me: Automatic presentation for visual analysis," *IEEE transactions on visualization and computer graphics*, vol. 13, no. 6, 2007.

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

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

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

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

[24] T. Abbott, F. Abdalla, A. Alarcon, J. Aleksić, S. Allam, S. Allen, A. Amara, J. Annis, J. Asorey, S. Avila et al., "Dark energy survey year 1 results: Cosmological constraints from galaxy clustering and weak lensing," *arXiv preprint arXiv:1708.01530*, 2017.

[25] "Geopandas," 2018. [Online]. Available: http://geopandas.org/

[26] R. S. Scowen, "Extended bnf-a generic base standard," Technical report, ISO/IEC 14977. http://www. cl. cam. ac. uk/mgk25/iso-14977. pdf, Tech. Rep., 1998.

[27] M. Slee, A. Agarwal, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," *Facebook White Paper*, vol. 5, no. 8, 2007.

[28] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[29] M. N. Vora, "Hadoop-hbase for large-scale data," in *Computer science and network technology (ICCSNT), 2011 international conference on*, vol. 1. IEEE, 2011, pp. 601–605.

[30] K. Banker, *MongoDB in action*. Manning Publications Co., 2011.

[31] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.