Effortless Data Exploration with zenvisage: An Expressive and Interactive Visual Analytics System

Tarique Siddiqui[1] Albert Kim[2] John Lee[1] Karrie Karahalios[1] Aditya Parameswaran[1]
[1]University of Illinois (UIUC) [2]MIT
{tsiddiq2,lee98,kkarahal,adityagp}@illinois.edu alkim@csail.mit.edu

ABSTRACT

Data visualization is by far the most commonly used mechanism to explore and extract insights from datasets, especially by novice data scientists. And yet, current visual analytics tools are rather limited in their ability to operate on collections of visualizations—by composing, filtering, comparing, and sorting them—to find those that depict desired trends or patterns. The process of visual data exploration remains a tedious process of *trial-and-error*. We propose zenvisage, a visual analytics platform for effortlessly finding desired visual patterns from large datasets. We introduce zenvisage's general purpose visual exploration language, ZQL ("zee-quel") for specifying the desired visual patterns, drawing from use-cases in a variety of domains, including biology, mechanical engineering, climate science, and commerce. We formalize the expressiveness of ZQL via a visual exploration algebra—an algebra on collections of visualizations—and demonstrate that ZQL is as expressive as that algebra. zenvisage exposes an interactive front-end that supports the issuing of ZQL queries, and also supports interactions that are "short-cuts" to certain commonly used ZQL queries. To execute these queries, zenvisage uses a novel ZQL graph-based query optimizer that leverages a suite of optimizations tailored to the goal of processing collections of visualizations in certain pre-defined ways. Lastly, a user survey and study demonstrates that data scientists are able to effectively use zenvisage to eliminate error-prone and tedious exploration and directly identify desired visualizations.

1. INTRODUCTION

Interactive visualization tools, such as Tableau [4] and Spotfire [3], have paved the way for the democratization of data exploration and data science. These tools have witnessed an ever-expanding user base—as a concrete example, Tableau's revenues last year were in the hundreds of millions of US Dollars and is expected to reach tens of billions soon [7]. Using such tools, or even tools like Microsoft Excel, the standard data analysis recipe is as follows: the data scientists load a dataset into the tool, select visualizations to examine, study the results, and then repeat the process until they find ones that match their desired pattern or need. Thus, using this repeated process of manual examination, or *trial-and-error*, data scientists are able to formulate and test hypothesis, and derive insights. The key premise of this work is that to find desired patterns in datasets, ***manual examination of each visualization in a collection is simply unsustainable***, especially on large, complex datasets. Even on moderately sized datasets, a data scientist may need to examine as many as tens of thousands of visualizations, all to test a single hypothesis, a severe impediment to data exploration.

To illustrate, we describe the challenges of several collaborator groups who have been hobbled by the ineffectiveness of current data exploration tools:

Case Study 1: Engineering Data Analysis. Battery scientists at Carnegie Mellon University perform visual exploration of datasets of solvent properties to design better batteries. A specific task may involve finding solvents with desired behavior: e.g., those whose solvation energy of Li^+ vs. the boiling point is a *roughly increasing trend*. To do this using current tools, these scientists manually examine the plot of Li^+ solvation energy vs. boiling point for each of the thousands of solvents, to find those that match the desired pattern of a roughly increasing trend.

Case Study 2: Advertising Data Analysis. Advertisers at ad analytics firm Turn, Inc., often examine their portfolio of advertisements to see if their campaigns are performing as expected. For instance, an advertiser may be interested in seeing if there are any keywords that are *behaving unusually* with respect to other keywords in Asia—for example, maybe most keywords have a specific trend for click-through rates (CTR) over time, while a small number of them have a different trend. To do this using the current tools available at Turn, the advertiser needs to manually examine the plots of CTR over time for each keyword (thousands of such plots), and remember what are the typical trends.

Case Study 3: Genomic Data Analysis. Clinical researchers at the NIH-funded genomics center at UIUC and Mayo Clinic are interested in studying data from clinical trials. One such task involves finding pairs of genes that *visually explain the differences* in clinical trial outcomes (positive vs. negative)—visualized via a scatterplot with the x and y axes each referring to a gene, and each outcome depicted as a point in the scatterplot—with the positive outcomes depicted in one color, and the negative ones as another. Current tools require the researchers to generate and manually evaluate tens of thousands of scatter plots of pairs of genes for whether the outcomes can be clearly distinguished in the scatter plot.

Thus, in these examples, the recurring theme is the manual examination of a large number of generated visualizations for a specific visual pattern. Indeed, we have found that in these scenarios, as well as others that we have encountered via other collaborators—in *climate science, server monitoring, and mobile app analysis*—data exploration can be a tedious and time-consuming process with current visualization tools.

Key Insight. The goal of this paper is to develop zenvisage, a visual analytics system that can ***automate the search for desired visual patterns***. Our key insight in developing zenvisage is that the data exploration needs in all of these scenarios can be captured within a common set of operations on collections of visualizations. These operations include: *composing* collections of visualizations, *filtering* visualizations, based on some conditions, *comparing* visualizations, and *sorting* them based on some condition. The conditions include similarity or dissimilarity to a specific pattern, "typical" or anomalous behavior, or the ability to provide explanatory

or discriminatory power. These operations and conditions form the kernel of a new data exploration language, ZQL ("zee-quel"), that forms the foundation upon which zenvisage is built.

Key Challenges. We encountered many challenges in building the zenvisage visual analytics platform, a substantial advance over the manually-intensive visualization tools like Tableau and Spotfire; these tools enable the examination of *one visualization at a time*, without the ability to automatically identify relevant visualizations from a collection of visualizations.

First, there were many challenges in developing ZQL, the underlying query language for zenvisage. Unlike relational query languages that operate directly on data, ZQL operates on collections of visualizations, which are themselves aggregate queries on data. Thus, in a sense ZQL is a query language that operates on other queries a a first class citizen. This leads to a number of challenges that are not addressed in a relational query language context. For example, we had to develop a natural way to users to specify a collection of visualizations to operate on, without having to explicitly list them; even though the criteria on which the visualizations were compared varied widely, we had to develop a small number of general mechanisms that capture all of these criteria; often, the visualizations that we operated on had to be modified in various ways—e.g., we might be interested in visualizing the sales of a product whose profits have been dropping—composing these visualizations from existing ones is not straightforward; and lastly, drilling down into specific visualizations from a collection also required special care. Our ZQL language is a synthesis of desiderata after discussions with data scientists from a variety of domains, and has been under development for the past two years. To further show that ZQL is *complete* under a new visual exploration algebra that we develop, involved additional challenges.

Second, in terms of front-end development, zenvisage, being an interactive analytics tool, needs to support the ability for users to interactively specify ZQL queries—specifically, interactive shortcuts for commonly used ZQL queries, as well as the ability to pose extended ZQL queries for more complex needs. Identifying common interaction "idioms" for these needs took many months.

Third, an important challenge in building zenvisage is the backend that supports the execution of ZQL. A single ZQL query can lead to the generation of 10000s of visualizations—executing each one independently as an aggregate query, would take several hours, rendering the tool somewhat useless. zenvisage's query optimizer operates as a wrapper over any traditional relational database system. This query optimizer compiles ZQL queries down to a directed acyclic graph of operations on collections of visualizations, followed with the optimizer using a combination of intelligent speculation and combination, to issue queries to the underlying database. We also demonstrate that the underlying problem is NP-HARD via the PARTITION PROBLEM. Our query optimizer leads to substantial improvements over the naive schemes adopted within relational database systems for multi-query optimization.

Related Work. There are a number of tools one could use for interactive analysis; here, we briefly describe why those tools are inadequate for the important need of automating the search for desired visual insights. We describe related work in detail in Section 7.

To start, visualization tools like Tableau and Spotfire only generate and provide one visualization at a time, while zenvisage analyzes collections of visualizations at a time, and identifies relevant ones from that collection—making it substantially more powerful.

While we do use relational database systems as a computation layer, it is cumbersome to near-impossible to express these user needs in SQL. As an example, finding visualizations of solvents for whom a given property follows a roughly increasing trend is impossible to write within native SQL, and would require custom UDFs—these UDFs would need to be hand-written for every ZQL query. For the small space of queries where it is possible to write the queries within SQL these queries require non-standard constructs, and are both complex and cumbersome to write, even for expert SQL users, and are optimized very poorly (see Section 7).

Statistical, data mining, and machine learning certainly provide functionality beyond zenvisage in supporting prediction and statistics; these functionalities are exposed as "one-click" algorithms that can be applied on data. However, no functionality is provided for searching for desired patterns; no querying functionality beyond the one-click algorithms, and no optimization. To use such tools for ZQL, many lines of code and hand-optimization is needed.

Outline. We first describe our query language for zenvisage, ZQL (Section 2), and the graph-based query translator and optimizer for ZQL (Section 3). We then describe our initial prototype of zenvisage (Section 4). We describe our performance experiments (Section 5), and present a user survey and study focused on evaluating the effectiveness and usability of zenvisage (Section 6). In our extended technical report [2], we provide additional details that we weren't able to fit into the paper. In particular, we formalize the notion of a *visual exploration algebra*, an analog of relational algebra, describing a core set of capabilities for any language that supports visual data exploration, and demonstrate that ZQL is *complete* in that it subsumes these capabilities.

2. QUERY LANGUAGE

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

ZQL draws heavy inspiration from the Query by Example (QBE) language [33] and uses a similar table-based specification interface. Although ZQL components are not fundamentally tied to the tabular interface, we found that our end-users feel more at home with it; many of them are non-programmers who are used to spreadsheet tools like Microsoft Excel. Users may either directly write ZQL, or they may use the zenvisage front-end, which supports interactions that are transformed internally into ZQL.

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

2.1 Formalization

For describing ZQL, we assume that we are operating on a single relation or a star schema where the attributes are unique (barring key-foreign key joins), allowing ZQL to seamlessly support natural joins. In general, ZQL could be applied to arbitrary collections of relations by letting the user precede an attribute A with the relation name R, e.g., $R.A$. For ease of exposition, we focus on the single relation case.

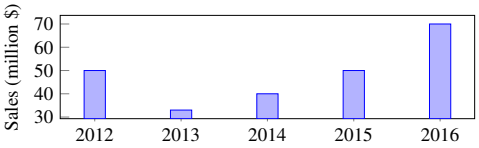

Figure 1: Sales over year visualization for the product chair.

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

Table 1: Query for the bar chart of sales over year for the product chair.

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

Table 2: Query for the bar chart of sales over year for each product.

2.1.1 Overview

The concept of visualizations. We start by defining the notion of a visualization. We use a sample visualization in Figure 1 to guide our discussion, Of course, different visual analysis tasks may require different types of visualizations (instead of bar charts, we may want scatter plots or trend lines), but across all types a visualization is defined by the following five main components: (i) the x-axis attribute, (ii) the y-axis attribute, (iii) the subset of data used, (iv) the type of visualization (e.g., bar chart, scatter plot), and (v) the binning and aggregation functions for the x- and y- axes.

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

Table 1 gives an example of a valid ZQL query that uses these columns to specify a bar chart visualization of overall sales over the years for the product chair (i.e., the visualization in Figure 1)—ignore the Name column for now. The details for each of these columns are presented subsequently. In short, the x axis (X) is the attribute year, the y axis (Y) is the attribute sales, and the subset of data (Z) is the product chair, while the type of visualization is a bar chart (bar), and the binning and aggregation functions indicate that the y axis is an aggregate (agg) — the sum of sales.

In addition to specifying a single visualization, users may often want to retrieve multiple visualizations. ZQL supports this in two ways. Users may use multiple rows, and specify one visualization per row. The user may also specify a *collection* of visualizations in a single row by iterating over a collection of values for one of the X, Y, Z, and Viz columns. Table 2 gives an example of how one may iterate over all products (using the notation * to indicate that the attribute product can take on all values), returning a separate sales bar chart for each product.

High-level structure of ZQL. Starting from these two examples, we can now move onto the general structure of ZQL queries. Overall, each ZQL query consists of multiple rows, where each row operates on collections of visualizations. Each row contains three sets of columns, as depicted in Table 3: (i) the first column corresponds to an identifier for a visualization collection, (ii) the second set of columns defines a visualization collection, while (iii) the last column corresponds to some operation on the visualization collection. All columns can be left empty if needed (in such cases, to save space, for convenience, we do not display these columns in our paper). For example, the last column may be empty if no operation is to be performed, like it was in Table 1 and 2. We have already discussed (ii); now we will briefly discuss (i) and (iii), corresponding to ***Name*** and ***Process*** respectively.

Identifiers and operations in ZQL. The ***Process*** column allows the user to operate on the defined collections of visualizations, applying high-level filtering, sorting, and comparison. The ***Name*** column provides a way to label and combine specified collections of visualizations, so users may refer to them in the Process column.

Name	X	Y	Z	Viz	Process
Identifier	Visualization Collection				Operation

Table 3: ZQL query structure.

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

Table 4: Query for the sales and profit bar charts for the product chair (missing values are the same as that in Table 1)

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

Table 5: Query for the sales and profit bar charts over years and months for chairs (missing values are the same as in Table 1).

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

Table 6: Query which returns the overall sales bar chart for the chairs in US (all missing values are the same as that in Table 1).

Thus, by repeatedly using the X, Y, Z, and Viz columns to compose visualizations and the Process column to process those visualizations, the user is able derive the exact set of visualizations she is looking for. Note that the result of a ZQL query is the *data* used to generate visualizations. The zenvisage front-end then uses this data to render the visualizations for the user to peruse.

2.1.2 X, Y, and Z

The X and Y columns specify the attributes used for the x- and y- axes. For example, Table 1 dictates that the returned visualization should have 'year' for its x-axis and 'sales' for its y-axis. As mentioned, the user may also specify a collection of values for the X and Y columns if they wish to refer to a collection of visualizations in one ZQL row. Table 4 refers the collection of both sales-over-years and profit-over-years bar charts for the chair—the missing values in this query ("...") are the same as Table 1. As we can see, a collection is constructed using {}. If the user wishes to denote all possible values, the shorthand * symbol may be used, as is shown by Table 2. In the case that multiple columns contain collections, a Cartesian product is performed, and visualizations for every combination of values is returned. For example, Table 5 would return the collection of visualizations with specifications: {(X: 'year', Y: 'sales'), (X: 'year', Y: 'profit'), (X: 'month', Y: 'sales'), (X: 'month', Y: 'profit')}.

With the Z column, the user can select which subset of the data they wish to construct their visualizations from. ZQL uses the ⟨*attribute*⟩.⟨*attribute-value*⟩ notation to denote the selection of data. Consequently, the query in Table 1 declares that the user wishes to retrieve the sales bar chart only for the chair product. Collections are allowed for both the attribute and the attribute value in the Z column. Table 2 shows an example of using the * shorthand to specify a collection of bar charts, one for each product. A Z column which has a collection over attributes might look like: {'location', 'product'}.* (i.e., a visualization for every product and a visualization for every location). In addition, the Z column allows users to specify predicate constraints using syntax like 'weight'.[? < 10]; this specifies all items whose weight is less than 10 lbs. To evaluate, the ? is replaced with the attribute and the resulting expression is passed to SQL's WHERE clause.

ZQL supports multiple constraints on different attributes through the use of multiple Z columns. In addition to the basic Z column, the user may choose to add Z2, Z3, ... columns depending on how many constraints she requires. Table 6 gives an example of a query which looks at sales plots for chairs only in the US. Note that Z columns are combined using conjunctive semantics.

2.1.3 Viz

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

Table 7: Query which returns the heat map of sales vs. weights across all transactions.

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

Table 8: Query which returns the sales for chairs or profits for US visualizations for all items less than 10 lbs.

The Viz column decides the visualization type, binning, and aggregation functions for the row. Elements in this column have the format: ⟨*type*⟩.⟨*bin+aggr*⟩. All examples so far have been bar charts with no binning and SUM aggregation for the y-axis, but other variants are supported. The visualization types are derived from the Grammar of Graphics [32] specification language, so all plots from the geometric transformation layer of ggplot [31] (the tool that implements Grammar of Graphics) are supported. For instance, scatter plots are requested with point and heat maps with bin2d. As for binning, binning based on bin width (bin) and number of bins (nbin) are supported for numerical attributes—we may want to use binning, for example, when we are plotting the total number of products whose prices lie within 0-10, 10-20, and so on.

Finally, ZQL supports all the basic SQL aggregation functions such as AVG, COUNT, and MAX. Table 7 is an example of a query which uses a different visualization type, heat map, and creates 20 bins for both x- and y- axes.

The Viz column allows users powerful control over the structure of the rendered visualization. However, there has been work from the visualization community which automatically tries to determine the most appropriate visualization type, binning, and aggregation for a dataset based on the x- and y- axis attributes [17, 21]. Thus, we can frequently leave the Viz column blank and zenvisage will use these rules of thumb to automatically decide the appropriate setting for us. With this in mind, we omit the Viz column from the remaining examples with the assumption that zenvisage will determine the "best" visualization structure for us.

2.1.4 Name

Together, the values in the X, Y, Z, and Viz columns of each row specify a collection of visualizations. The Name column allows us to label these collections so that they can be referred to be in the Process column. For example, f1 is the label or identifier given to the collection of sales bar charts in Table 2. The * in front of f1 signifies that the the collection is an output collection; that is, ZQL should return this collection of visualizations to the user.

However, not all rows need to have a * associated with their Name identifier. A user may define intermediate collections of visualizations if she wishes to further process them in the Process column before returning the final results. In the case of Table 8, f1 and f2 are examples of intermediate collections.

Also in Table 8, we have an example of how the Name column allows us to perform high-level set-like operations to combine visualization collections directly. For example, f3 <- f1 + f2 assigns f3 to the collection which includes all visualizations in f1 and f2 (similar to set union). This can be useful if the user wishes to combine variations of values without considering the full Cartesian product. Our example in Table 8, the user is able to combine the sales for chairs plots with the profits for the US plots without also having to consider the sales for the US plots or the profits for chairs plots; she would have had to do so if she had used the specification: (Y: {'sales', 'profit'}, Z: {'product'.'chair', 'location'.'US'}).

An interesting aspect of Table 8 is that the X and Y columns of the third row are devoid of values, and the Z column refer to the seemingly unrelated weight attribute. The values in the X, Y, Z, and Viz columns all help to specify a particular collection of visualizations from a larger collection. When this collection is defined via the Name column, we no longer need to fill in the values for X, Y, Z, or Viz, except to select from the collection—here, ZQL only selects the items which satisfy the constraint, weight < 10.

2.1.5 Process

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

Iterations over collections. Since collections may be composed of varying values from multiple columns, iterating over the collections is not straight-forward. Consider Table 9—the goal is to return profit by year visualizations for the top-10 products whose profit by year visualizations look the most different from the sales by year visualizations. While we will describe this query in detail below, at a high level the first row assembles the visualizations for profit over year for all products (f1), the second row assembles the visualizations for sales over year for all products (f2), followed by operating (via the Process column) on these two collections by finding the top-10 products who sales over year is most different from profit over year, while the third row displays the profit over year for those top-10 products. A array-based representation of the visualization collections f1 and f2, would look like the following:

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

We would like to iterate over the products—the Z dimension values—of both f1 and f2 to make our comparisons. Furthermore, we must iterate over the products in the *same order* for both f1 and f2 to ensure that a product's profit visualization correctly matches with its sales visualization. Using a single index for this would be complicated and need to take into account the sizes of each of the columns. Instead, ZQL opts for a more powerful *dimension-based* iteration, which assigns each column (or dimension) a separate iterator called an *axis variable*. This dimension-based iteration is a powerful idea that extends to any number of dimensions. As shown in Table 9, axis variables are defined and assigned using the syntax: ⟨*variable*⟩<- ⟨*collection*⟩; axis variable v1 is assigned to the Z dimension of f1 and iterates over all product values. For cases in which multiple collections must traverse over a dimension in the same order, an axis variable must be shared across those collections for that dimension; in Table 9, f1 and f2 share v1 for their Z dimension, since we want to iterate over the products in lockstep.

Operations on collections. With the axis variables defined, the user can then formulate the high-level operations on collections of visualizations as an optimization function which maximizes/minimizes for their desired pattern. Given that $\texttt{argmax}_x[k=10]\ g(x)$ returns the top-10 x values which maximizes the function $g(x)$, and $D(x,y)$ returns the "distance" between x and y, now consider the expression in the Process column for Table 9. Colloquially, the expression says to find the top-10 v1 values whose $D(f1,f2)$ values are the largest. The $f1$ and $f2$ in $D(f1,f2)$ refer to the collections

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

Table 9: Query which returns the top 10 profit visualizations for products which are most different from their sales visualizations.

of visualizations in the first and second row and are bound to the current value of the iteration for v1. In other words, for each product v1' in v1, retrieve the visualizations f1[z: v1'] from collection f1 and f2[z: v1'] from collection f2 and calculate the "distance" between these visualizations; then, retrieve the 10 v1' values for which this distance is the largest—these are the products, and assign v2 to this collection. Subsequently, we can access this set of products in Z column of the third line of Table 9.

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

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

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

where $\langle axvar\rangle$ refers to the axis variables, and $\langle nmvar\rangle$ refers to collections of visualizations. $\langle argopt\rangle$ may be one of argmax, argmin, or argany, which returns the values which have the largest, smallest, and any expressions. The $\langle limiter\rangle$ limits the number of results: $k=\mathbb{N}$ returns only the top-k values; $t>\mathbb{R}$ returns only values who are larger than a threshold value t (may also be smaller, greater than equal, etc.); $p=\mathbb{R}$ returns the top p-percentile values. T and D are two simple *functional primitives* supported by ZQL that can be applied to visualizations to find desired patterns:

- $[T(f)\rightarrow\mathbb{R}]$: T is a function which takes a visualization f and returns a real number measuring some visual property of the trend of f. One such property is "growth", which returns a positive number if the overall trend is "upwards" and a negative number otherwise; an example implementation might be to measure the slope of a linear fit to the given input visualization f. Other properties could measure the skewness, or the number of peaks, or noisiness of visualizations.
- $[D(f,f')\rightarrow\mathbb{R}]$: D is a function which takes two visualizations f and f' and measures the distance (or dissimilarity) between these visualizations. Examples of distance functions may include pointwise distance functions like Euclidean distance, Earth Mover's Distance, or the Kullback-Leibler Divergence. The distance D could also be measured using the difference in the number of peaks, or slopes, or some other property.

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

Concrete examples. With just dimension-based iteration, the optimization structure of the Process column, and the functional primitives T and D, we found that we were able to support the majority of the visual analysis tasks required by our users. Common patterns include filtering based on overall trend (Table 10), searching for the most similar visualization (Table 11), and determining outlier visualizations (Table 12). Table 13 features a realistic query inspired by one of our case studies. The overall goal of the query is to find the products which have positive sales and profits trends in locations and categories which have overall negative trends; the user may want to look at this set of products to see what makes them so special. Rows 1 and 2 specify the sales and profit visualizations for all locations and categories respectively, and the processes for these rows filter down to the locations and categories which have negative trends. Then rows 3 and 4 specify the sales and profit visualizations for products in these locations and categories, and the processes filter the visualizations down to the ones that have positive trends. Finally, row 5 takes the list of output products from the processes in rows 3 and 4 and takes the intersection of the two returning the sales and profits visualizations for these products.

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

2.2 Discussion of Capabilities and Limitations

Although ZQL can capture a wide range of visual exploration queries, it is not limitless. Here, we give a brief description of what ZQL can do. A more formal quantification can be found in [2].

ZQL's primary goal is to support queries over visualizations—which are themselves aggregate group-by queries on data. Using these queries, ZQL can compose a collection of visualizations, filter them in various ways, compare them against benchmarks or against each other, and sort the results. The functions T and D, while intuitive, support the ability to perform a range of computations on visualization collections—for example, any filter predicate on a single visualization, checking for a specific visual property, can be captured under T. Then, via the dimension-based iterators, ZQL supports the ability to chain these queries with each other and compose new visualization collections. These simple set of operations offer unprecedented power in being able to sift through visualizations to identify desired trends.

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

3. QUERY EXECUTION

In zenvisage, ZQL queries are automatically parsed and executed by the back-end. The ZQL compiler translates ZQL queries into a combination of SQL queries to fetch the visualization collections and processing tasks to operate on them. We present a basic

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

Table 10: Query which returns the sales visualizations for all products which have a negative trend.

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

Table 11: Query which returns the sales visualizations for the 10 products whose sales visualizations are the most similar to the sales visualization for the chair.

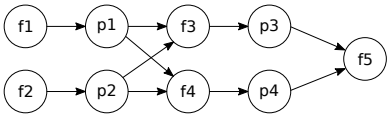

Figure 2: The query plan for the query presented in Table 13.

graph-based translation for ZQL and then provide several optimizations to the graph which reduce the overall runtime considerably.

3.1 Basic Translation

Every valid ZQL query can be transformed into a query plan in the form of a directed acyclic graph (DAG). The DAG contains *c*-nodes (or *collection nodes*) to represent the collections of visualizations in the ZQL query and *p*-nodes (or *process nodes*) to represent the optimizations (or *processes*) in the Process column. Directed edges are drawn between nodes that have a dependency relationship. Using this query plan, the ZQL engine can determine at each step which visualization collection to fetch from the database or which process to execute. The full steps to build a query plan for any ZQL query is as follows: (i) Create a *c*-node or *collection node* for every collection of visualizations (including singleton collections). (ii) Create a *p*-node or *processor node* for every optimization (or *process*) in the Process column. (iii) For each *c*-node, if any of its axis variables are derived as a result of a process, connect a directed edge from the corresponding *p*-node. (iv) For each *p*-node, connect a directed edge from the *c*-node of each collection which appears in the process. Following these steps, we can translate our realistic query example in Table 13 to its query plan presented in Figure 2. Here, the *c*-nodes are annotated with f#, and the *p*-nodes are annotated with p# (the *i*th *p*-node refers to the process in the *i*th row of the table). Further, f1 is a root node with no dependencies since it does not depend on any process, whereas f5 depends on the results of both p3 and p4 and have edges coming from both of them. Once the query plan has been constructed, the ZQL engine can execute it using the simple algorithm presented in in Algorithm 1.

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

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

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

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

is issued to the database, where X is the X column attribute, Y is the Y column attribute, A(Y) is the aggregation function on Y (specified in the Viz column), Z, Z2, ... are the attributes/dimensions we are iterating over in the Z columns, while C(X, Y, Z, Z2, ...) refers to any additional constraints specified in the Z columns. The ORDER BY is inserted to ensure that all rows corresponding to a visualization are grouped together, in order. As an example, the SQL query for the *c*-node for f1 in Table 12 would have the form:

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

If a collection is specified for the y-axis, each attribute in the collection is appended to the SELECT clause. If a collection is specified for the x-axis, a separate query must be issued for every X attribute in the collection. The results of the SQL query are then packed into a *m*-dimensional array (each dimension in the array corresponding to a dimension in the collection) and labeled with its f# tag.

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

$$\langle argopt\rangle_{v0}[k=k'] \left[\langle op_1\rangle_{v1}\left[\langle op_2\rangle_{v2}\cdots\left[\langle op_m\rangle_{vm}T(f1)\right]\right]\right] \quad (1)$$

where $\langle argopt\rangle$ is one of argmax or argmin, and $\langle op\rangle$ refers to one of $(\max|\min|\sum|\prod)$. Given this, the pseudocode which optimizes this process can automatically be generated based on the actual values of $\langle argopt\rangle$, $\langle op\rangle$, and the number of operations. In short, for each $\langle op\rangle$ or dimension traversed over, the ZQL engine generates a new nested for loop. Within each for loop, we iterate over all values of that dimension, evaluate the inner expression, and then eventually apply the overall operation (e.g., max, $\sum$).

3.2 Optimizations

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

3.2.1 Parallelization

One natural way to optimize the graph-based query plan is to take advantage of the multi-query optimization (MQO) [27] present in databases and issue in parallel the SQL queries for independent *c*-nodes—the *c*-nodes for which there is no dependency between them. With MQO, the database can receive multiple SQL queries at the same time and share the scans for those queries, thereby reducing the number of times the data needs to be read from disk.

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

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

Table 12: Query which returns the sales visualizations for the 10 products whose sales visualizations are the most different from the others.

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

Table 13: Query which returns the profit and sales visualizations for products which have positive trends in profit and sales in locations and categories which have overall negative trends.

3.2.2 Speculation

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

For example, in the query in Table 13, f3 depends on the results of p1 and p2 since it has constraints based on v2 and v4; specifically v2 and v4 should be locations and categories for which f1 and f2 have a negative trend. However, we note that v2 and v4 are derived as a result of v1 and v3, specified to take on all locations and categories in rows 1 and 2. So, a superset of f3, the set of profit over year visualizations for various products for all locations and categories (as opposed to just those that satisfy p1 and p2), could be retrieved pre-emptively. Later, when the ZQL engine executes p1 and p2, this superset can be filtered down correctly.

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

3.2.3 Query Combination

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

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

we can produce a combined Q_3 which has the form:

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

where C1 = C1(X1, Y1, Z1) and C2 is defined similarly. From the combined query Q_3, it is possible to regenerate the data which would have been retrieved using queries Q_1 and Q_2 by aggregating over the non-related groups for each query. For Q_1, we would select the data for which C1 holds, and for each (X1, Z1) pair, we would aggregate over the X2, Z2, and C2 groups.

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

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

As we show in the technical report [2], this optimization problem is NP-HARD via a reduction from the PARTITION PROBLEM.

Wrinkle and Solution. However, a wrinkle to the above formulation is that it assumes no two SQL queries share a group-by attribute. If two queries have a shared group-by attribute, it may be more beneficial to combine those two, since the number of group-by values does not go up on combining them. Overall, we developed the metric $EFGV$ or the effective increase in the number of group-by values to determine the utility of combining query Q' to query Q: $EFGV_Q(Q') = \prod_{g\in G(Q')} \#(g)^{[[g\notin G(Q)]]}$ where $G(Q)$ is the set of group-by values in Q, $\#(g)$ calculates the number of unique group-by values in g, and $[[g \notin G(Q)]]$ returns 1 if $g \notin G(Q)$ and 0 otherwise. In other words, this calculates the product of group-by values of the attributes which are in Q' but not Q. Using the $EFGV$ metric, we then apply a variant of agglomerative clustering [10] to decide the best choice of queries to combine. As we show in the experiments section, this technique leads to very good performance.

3.2.4 Cache-Aware Execution

Although the previous optimizations were all I/O-based optimizations for ZQL, there are cases in which optimizing the execution of *p*-nodes is important as well. In particular, when a process has multiple nested for loops, the cost of the *p*-node may start to dominate the overall runtime. To address this problem, we adapt techniques developed in high-performance computing—specifically, cache-based optimizations similar to those used in matrix multiplication [13]. With cache-aware execution, the ZQL engine partitions the iterated values in the for loops into blocks of data which fit into the L3 cache. Then, the ZQL engine reorders the order of iteration in the for loops to maximize the time that each block of data remains in the L3 cache. This allows the system to minimize the amount of data the cache needs to eject and thus the amount of data that needs to be copied from main memory to

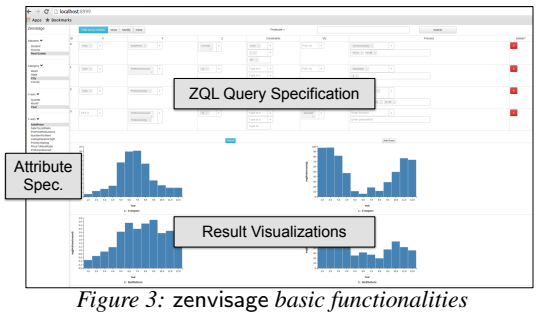

Figure 3: zenvisage *basic functionalities*

the cache, minimizing the time taken by the *p*-nodes.

4. zenvisage SYSTEM DESCRIPTION

We now give a brief description of the zenvisage system.

Front-end. The zenvisage front-end is designed as a lightweight web-based client application. It provides a GUI to compose ZQL queries, and displays the resulting visualizations using Vega-lite [17]. A screenshot of zenvisage in action is shown in Figure 3. A list of attributes, divided into qualitative and quantitative, is provided on the left; a table to enter ZQL queries, with auto-completion, is on top, and the resulting visualizations are rendered at the bottom. Users also have the option of hiding the ZQL specification table and instead using a simpler drop-down menu-based interface complemented by a sketching canvas. The sketching canvas allows users to draw their desired trend that can then be used to search for similar trends. The menu-based interface makes it easy for users to perform some of the more common visual exploration queries, such as searching for representative or outlier visualizations. Furthermore, the user may drag-and-drop visualizations from the results onto the sketching canvas, enabling further interaction with the results.

Back-end. The zenvisage front-end issues ZQL queries to the back-end over a REST protocol. The back-end (written in node.js) receives the queries and forwards them to the ZQL engine (written in Java), which is responsible for parsing, compiling, and optimizing the queries as in Section 3. SQL queries issued by the ZQL engine are submitted to one of our back-end databases (which currently include PostgreSQL and Vertica), and the resultant visualization data is returned back to the front-end encoded in JSON.

5. EXPERIMENTAL STUDY

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

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

Optimizations. The four versions of the ZQL engine we use are: (i) NO-OPT: The basic translation from Section 3. (ii) PARALLEL: Concurrent SQL queries for independent nodes from Section 3.2.1. (iii) SPECULATE: Speculates and pre-emptively issues SQL queries from Section 3.2.2. (iv) SMARTFUSE: Query combination with speculation from Section 3.2.3. In our experiments, we consider NO-OPT and the MQO-dependent PARALLEL to be our baselines, while SPECULATE and SMARTFUSE were considered to be completely novel optimizations. For certain experiments later on, we also evaluate the performance of the caching optimizations from Section 3.2.4 on SMARTFUSE.

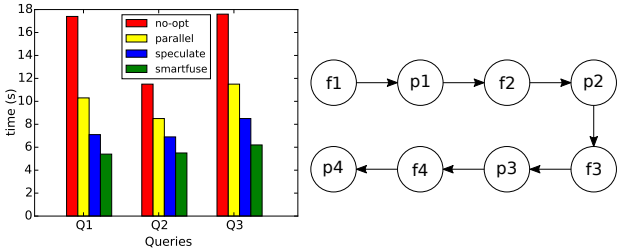

Figure 4: Runtimes for queries on real dataset (left) and single chain synthetic query (right)

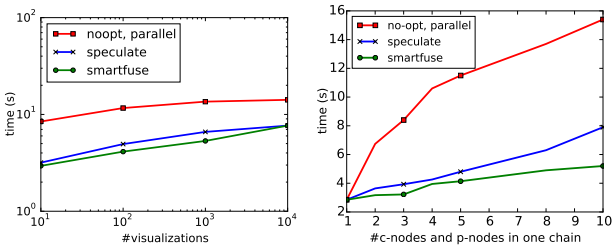

Figure 5: Effect of number of visualizations (left) and length of the chain (right) on the overall runtimes.

5.1 Realistic Queries

For our realistic queries, we used 20M rows of a real 1.5GB airline dataset [1] which contained the details of flights within the USA from 1987-2008, with 11 attributes. On this dataset, we performed 3 realistic ZQL queries inspired by the case studies in our introduction. Descriptions of the queries can be found in Table 14.

Figure 4 (left) depicts the runtime performance of the three realistic ZQL queries, for each of the optimizations. For all queries, each level of optimization provided a substantial speedup in execution time compared to the previous level. *Simply by going from* NO-OPT *to* PARALLEL*, we see a 45% reduction in runtime. From* PARALLEL *to* SPECULATE *and* SPECULATE *to* SMARTFUSE*, we see 15-20% reductions in runtime*. A large reason for why the optimizations were so effective was because ZQL runtimes are heavily dominated by the execution time of the issued SQL queries. In fact, we found that for these three queries, 94-98% of the overall runtime could be contributed to the SQL execution time. We can see from Table 14, SMARTFUSE always managed to lower the number of SQL queries to 1 or 2 after our optimizations, thereby heavily impacting the overall runtime performance of these queries.

5.2 Varying Characteristics of ZQL Queries

We were interested in evaluating the efficacy of our optimizations with respect to four different characteristics of a ZQL query: (i) the number of visualizations explored, (ii) the complexity of the ZQL query, (iii) the level of interconnectivity within the ZQL query, and (iv) the complexity of the processes. To control for all variables except these characteristics, we used a synthetic chain-based ZQL query to conduct these experiments. Every row of the chain-based ZQL query specified a collection of visualizations based on the results of the process from the previous row, and every process was applied on the collection of visualizations from the same row. Therefore, when we created the query plan for this ZQL query, it had the chain-like structure depicted by Figure 4 (right). Using the chain-based ZQL query, we could then (i) vary the number of visualizations explored, (ii) use the length of the

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

Table 14: Realistic queries for the airline dataset with the # of c-nodes, # of p-nodes, # of T functions calculated, # of D functions calculated, # of visualizations explored, # of SQL *queries issued with* NO-OPT*, and # of* SQL *queries issued with* SMARTFUSE *per query.*

chain as a measure of complexity, (iii) introduce additional independent chains to decrease interconnectivity, and (iv) increase the number of loops in a p-node to control the complexity of processes.

To study these characteristics, we used a synthetic dataset with 10M rows and 15 attributes (10 dimensional and 5 measure) with cardinalities of dimensional attributes varying from 10 to 10000. By default, we set the input number of visualizations per chain to be 100, with 10 values for the X attribute, number of c-nodes per chain as 5, the process as T (with a single for loop) with a selectivity of .50, and number of chains as 1.

Impact of number of visualizations. Figure 5 (left) shows the performance of NO-OPT, SPECULATE, and SMARTFUSE on our chain-based ZQL query as we increased the number of visualizations that the query operated on. The number of visualizations was increased by specifying larger collections of Z column values in the first c-node. We chose to omit PARALLEL here since it performs identically to NO-OPT. With the increase in visualizations, the overall response time increased for all versions because the amount of processing per SQL query increased. SMARTFUSE showed better performance than SPECULATE up to 10k visualizations due to reduction in the total number of SQL queries issued. However, at 10k visualization, we reached the threshold of the number of unique group-by values per combined query (100k for PostgreSQL), so it was less optimal to merge queries. At that point, SMARTFUSE behaved similarly to SPECULATE.

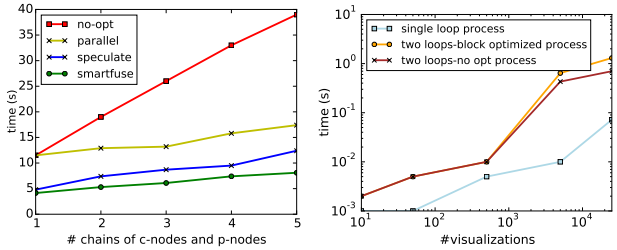

Figure 6: Effect of number of independent chains (left) and the number of loops in a p-node (right) on the overall runtimes.

Impact of the length of the chain. We varied the length of the chain in the query plan (or the number of rows in the ZQL query) to simulate a change in the complexity of the ZQL query and plotted the results in Figure 5 (right). As the number of nodes in the query plan grew, the overall runtimes for the different optimizations also grew. However, while the runtimes for both NO-OPT and SPECULATE grew at least linearly, the runtime for SMARTFUSE grew sublinearly due to its query combining optimization. While the runtime for NO-OPT was much greater than for SPECULATE, since the overall runtime is linearly dependent on the number of SQL queries run in parallel, we see a linear growth for SPECULATE.

Impact of the number of chains. We increased the number of independent chains from 1 to 5 to observe the effect on runtimes of our optimizations; the results are presented in Figure 6 (left). While NO-OPT grew linearly as expected, all PARALLEL, SPECULATE, and SMARTFUSE were close to constant with respect to the number of independent chains. We found that while the overall runtime for concurrent SQL queries did grow linearly with the number of SQL queries issued, they grew much slower compared to issuing those queries sequentially, thus leading to an almost flat line in comparison to NO-OPT.

Impact of process complexity. We increased the complexity of processes by increasing the number of loops in the first p-node from 1 to 2. For the single loop, the p-node filtered based on a positive trend via T, while for the double loop, the p-node found the outlier visualizations. Then, we varied the number of visualizations to see how that affected the overall runtimes. Figure 6 (right) shows the results. For this experiment, we compared regular SMARTFUSE with cache-aware SMARTFUSE to see how much of a cache-aware execution made. We observed that there was not much difference between cache-aware SMARTFUSE and regular SMARTFUSE below 5k visualizations when all data could fit in cache. After 5k visualizations, not all the visualizations could be fit into the cache the same time, and thus the cache-aware execution of the p-node had an improvement of 30-50% as the number of visualizations increased from 5k to 25k. This improvement, while substantial, is only a minor change in the overall runtime.

5.3 Interactivity

The previous figures showed that the overall execution times of ZQL queries took several seconds, even with SMARTFUSE, thus perhaps indicating ZQL is not fit for interactive use with large datasets. However, we found that this was primarily due to the disk-based I/O bottleneck of SQL queries. In Figure 7 (left), we show the SMARTFUSE runtimes of the 3 realistic queries from before on varying size subsets of the airline dataset, with the time that it takes to do a single group-by scan of the dataset. As we can see, the runtimes of the queries and scan time are *virtually the same*, indicating that SMARTFUSE comes very close to the optimal I/O runtime (i.e., a "fundamental limit" for the system).

To further test our hypothesis, we ran our ZQL engine with Vertica with a large working memory size to cache the data in memory to avoid expensive disk I/O. The results, presented in Figure 7 (right), showed that there was a 50× speedup in using Vertica over PostgreSQL with these settings. Even with a large dataset of 1.5GB, we were able to achieve sub-second response times for many queries. Furthermore, for the dataset with 120M records (11GB, so only 70% could be cached), we were able to to reduce the overall response times from 100s of seconds to less than 10 seconds. Thus, once again zenvisage returns results in a small multiple of the time it takes to execute a single group-by query.

Overall, SMARTFUSE will be interactive on moderate sized datasets on PostgreSQL, or on large datasets that can be cached in memory and operated on using a columnar database—which is standard practice adopted by visual analytics tools [29]. Improving on interactivity is impossible due to fundamental limits to the system; in the future, we plan to explore returning approximate answers using samples, since even reading the entire dataset is prohibitive.

6. USER STUDY

We conducted a user study to evaluate the utility of zenvisage for data exploration versus two types of systems—first, visualiza-

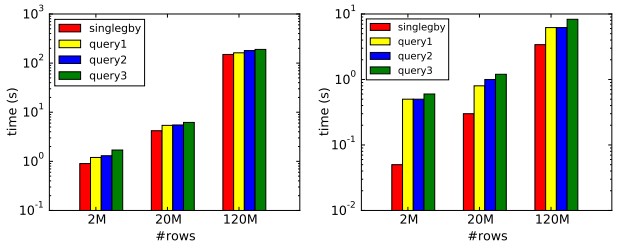

Figure 7: SMARTFUSE *on PostgreSQL (left) and Vertica (right)*

tion tools, similar to Tableau, and second, general database and data mining tools, which also support interactive analytics to a certain extent. In preparation for the user study, we conducted interviews with data analysts to identify the typical exploration tasks and tools used in their present workflow. Using these interviews, we identified a set of tasks to be used in the user study for zenvisage. We describe these interviews first, followed by the user study details.

6.1 Analyst Interviews and Task Selection

We hired seven data analysts via Upwork [5], a freelancing platform—we found these analysts by searching for freelancers who had the keywords `analyst` or `tableau` in their profile. We conducted one hour interviews with them to understand how they perform data exploration tasks. The interviewees had 3—10 years of prior experience, and told about every step of their workflow; from receiving the dataset to presenting the analysis to clients. The rough workflow of all interviewees identified was the following: first, data cleaning is performed; subsequently, the analysts perform data exploration; then, the analysts develop presentations using their findings. We then drilled down onto the data exploration step.

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

Then, to identify the common tasks used in data exploration, we used a taxonomy of abstract exploration tasks proposed by Amar et al. [9]. Amar et al. developed their taxonomy through summarizing the analytical questions that arose during the analysis of five different datasets, independent of the capabilities of existing tools or interfaces. The exploration tasks in Amar et al. include: filtering (f), sorting (s), determining range (r), characterizing distribution (d), finding anomalies (a), clustering (c), correlating attributes (co), retrieving value (v), computing derived value (dv), and finding extrema (e). When we asked the data analysts which tasks they use in their workflow, the responses were consistent in that all of them use all of these tasks, except for three exceptions—c, reported by four participants, and e, d, reported by six participants.

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

6.2 User Study Methodology

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

Participants. We recruited 12 graduate students as participants with varying degrees of expertise in data analytics. In short, half of them used databases; eight of them used Matlab, R, Python or Java; eight of them used spreadsheet software; and four of them used Tableau. Data for other not as popular tools are not reported.

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

Dataset. We used a housing dataset from Zillow.com [6], consisting of housing sales data for different cities, counties, and states from 2004-15, with over 245K rows, and 15 attributes. We selected this dataset since participants could relate to the dataset and understand the usefulness of the tasks.

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

Study Protocol. The user study was conducted using a within-subjects study design [11], forming three phases. First, participants described their previous experience with data analytics tools. Next, participants performed exploration tasks using zenvisage (Tool A) and the baseline tool (Tool B), with the orders randomized to reduce order effects. Participants were provided a 15-minute tutorial-cum-practice session per tool to get familiarized before performing the tasks. Finally, participants completed a survey that both measured their satisfaction levels and preferences, along with open-ended questions on the strengths and weaknesses of zenvisage and the baseline, when compared to other analytics tools they may have used. After the study, we reached out to participants with backgrounds in data mining and programming, and asked if they could complete a follow-up interview where they use their favorite analytics tool for performing one of the tasks, via email.

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

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

6.3 Key Findings

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

Finding 1: zenvisage ***enables faster and more accurate exploration than existing visualization tools.*** Since all of our tasks involved generating multiple visualizations and comparing them to find desired ones, participants were not only able to complete the tasks *faster*—μ=115s, σ=51.6 for zenvisage vs. μ=172.5s, σ=50.5 for the baseline—but also more *accurately*—μ=96.3%, σ=5.82 for zenvisage vs. μ=69.9%, σ=13.3 for the baseline. The baseline requires considerable manual exploration to complete the same task, explaining the high task completion times; in addition, participants frequently compromised by selecting suboptimal answers before browsing the entire list of results for better ones, explaining the low accuracy. On the other hand, zenvisage is able to automate the task of finding desired visualizations, considerably reducing manual effort. Also of note is the fact that the accuracy with zenvisage is close to 100%—indicating that a short 15 minute tutorial on ZQL was enough to equip users with the knowledge they needed to address the tasks—and that too, within 2 minutes (on average).

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

Finding 2: zenvisage ***complements existing database and data mining systems, and programming languages.*** When explicitly asking participants about comparing zenvisage with the tools they use on a regular basis for data analysis, all participants acknowledged that zenvisage adds value in data exploration not encompassed by their tools. ZQL augmented with inputs from the sketching canvas proved to be extremely effective. For example P8 stated: *"you can just [edit] and draw to find out similar patterns. You'll need to do a lot more through Matlab to do the same thing."* Another experienced participant mentioned the benefits of not needing to know much programming to accomplish certain tasks: *"The obvious good thing is that you can do complicated queries, and you don't have to write SQL queries... I can imagine a non-cs student [doing] this."* (P9). When asked about the specific tools they would use to solve the user study tasks, all participants reported a programming language like Matlab or Python. This is despite half of the participants reporting using a relational database regularly, and a smaller number of participants (2) reporting using a data mining tool regularly. Additionally, multiple participants even with extensive programming experience reported that zenvisage would take less time and fewer lines of code for certain data exploration tasks. (Indeed, we found that all participants were able to complete the user study tasks in under 2 minutes.) In follow-up email interviews, we asked a few participants to respond with code from their favorite data analytics tool for the user study tasks. Two participants responded — one with Matlab code, one with Python code. Both these code snippets were much longer than ZQL: as a concrete example, the participant accomplished the same task with 38 lines of Python code compared to 4 lines of ZQL. While comparing code may not be fair, the roughly order of magnitude difference demonstrates the power of zenvisage over existing systems.

```
with ranking    as (
with distances as (
with distance_ product_year  as (
with aggregate_ product_year as (
  select   product, year, avg(profit) as avg_profit
  from   table group by  product, year) )
  select  s. product as source, d. product as destination, s.year,
        power(s.avg_profit - d.avg_profit,2) as distance_year
  from   aggregate_ product_year s, aggregate_ product_year d
  where  s. product!=d. product and s.year=d.year )
  select  source, destination, sum(distance_year) as distance
  from   distance_ product_year   groupby  source, destination )
  select  source, destination, distance,
        rank() over (partition by source order by distance asc)
        rank from distances )
  select  source, destination, distance
  from   ranking   where  rank < 10;
```

Table 15: Verbose SQL *query*

Finding 3: zenvisage ***can be improved.*** Participants outlined some areas for improvement: some requested drag-and-drop interactions to support additional operations, such as outlier finding; others wanted a more polished interface; and some desired bookmarking and search history capabilities.

7. RELATED WORK

We now discuss related prior work in a number of areas. We begin with analytics tools — visualization tools, statistical packages and programming libraries, and relational databases. Then, we talk about other tools that overlap somewhat with zenvisage.

Visual Analytics Tools. Visualization tools, such as ShowMe, Spotfire, and Tableau [28, 22, 8], along with similar tools from the database community [12, 19, 20, 18] have recently gained in popularity, catering to data scientists who lack programming skills. Using these tools, these scientists can select and view one visualization at a time. However, these tools do not operate on collections of visualizations at a time—and thus they are much less powerful and the optimization challenges are minimal. zenvisage, on the other hand, supports queries over collections of visualizations, returning results not much slower than the time to execute a single query (See Section 5). Since these systems operate one visualization at a time, users are also not able to directly identify desired patterns or needs.

Statistical Packages and Programming Libraries: Statistical tools (e.g., KNIME, RapidMiner, SAS, SPSS) support the easy application of data mining and statistical primitives—including prediction algorithms and statistical tests. While these tools support the selection of a prediction algorithm (e.g., decision trees) to apply, and the appropriate parameters, they offer no querying capabilities, and as a result do not need extensive optimization. As a result, these tools cannot support user needs like those describe in the examples in the introduction. Similarly, programming libraries such as Weka [15] and Scikit-learn [24] embed machine learning within programs. However, manually translating the user desired patterns into code that uses these libraries will require substantial user effort and hand-optimization. In addition, writing new code and hand-optimization will need to be performed every time the exploration needs change. Additionally, for both statistical tools and programming libraries, there is a need for programming ability and understanding of machine learning and statistics to be useful—something we cannot expect all data scientists to possess.

Relational Databases. Relational databases can certainly support interactive analytics via SQL. In zenvisage, we use relational databases as a backend computational component, augmented with an engine that uses SMARTFUSE to optimize accesses to the database, along with efficient processing code. Thus, one can certainly express

some ZQL queries by writing multiple SQL queries (via procedural SQL), using complex constructs only found in some databases, such as common table expressions (CTE) and window functions. As we saw in Section 6, these SQL queries are very cumbersome to write, and are not known to most users of databases—during our user study, we found that all participants who had experience with SQL were not aware of these constructs; in fact, they responded that they did not know of any way of issuing ZQL queries in SQL, preferring instead to express these needs in Python. In Table 15, we list the verbose SQL query that computes the following: for each product, find 10 other products that have most similar profit over year trends. The equivalent ZQL query takes two lines. And we were able to write the SQL query only because the function D is Euclidean distance: for other functions, we are unable to come up with appropriate SQL rewritings. On the other hand, for ZQL, it is effortless to change the function by selecting it from a drop-down menu. Beyond being cumbersome to write, the constructs required lead to severe performance penalties on most databases—for instance, PostgreSQL materializes intermediate results when executing queries with CTEs. To illustrate, we took the SQL query in Table 15, and compared its execution with the execution of the equivalent ZQL. As depicted in Figure 8, the time taken by PostgreSQL increases sharply as the number of visualizations increases, taking up to 10X more time as compared to ZQL query executor. This indicates that zenvisage is still important even for the restricted cases where we are able to correctly write the queries in SQL.

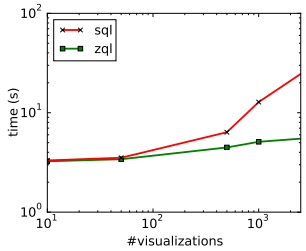

Figure 8: ZQL *vs* SQL*: we want to find top 10 similar products for every product on varying the number of products from 10—5000.*

OLAP Browsing. There has been some work on interactive browsing of data cubes [25, 26]. The work focuses on suggestions for raw aggregates to examine that are informative given past browsing, or those that show a generalization or explanation of a specific cell—an easier problem meriting simpler techniques—not addressing the full exploration capabilities provided by ZQL.

Data Mining Languages: There has been some limited work in data mining query languages, all from the early 90s, on association rule mining (DMQL [14], MSQL [16]), or on storing and retrieving models on data (OLE DB [23]), as opposed to a general-purpose visual data exploration language aimed at identifying visual trends.

Visualization Suggestion Tools: There has been some recent work on building systems that suggest visualizations. Voyager [17] recommends visualizations based on aesthetic properties of the visualizations, as opposed to queries. SeeDB [30] recommends visualizations that best display the difference between two sets of data. SeeDB and Voyager can be seen to be special cases of zenvisage. The optimization techniques outlined are a substantial generalization of the techniques described in SeeDB; while the techniques in SeeDB are special-cased to one setting (a simple comparison), here, our goal is to support and optimize all ZQL queries.

8. CONCLUSION

We propose zenvisage, a visual analytics tool for effortlessly identifying desired visual patterns from large datasets. We described the formal syntax of the query language ZQL, motivated by many real-world use-cases, and demonstrated that ZQL is visual exploration algebra-complete (See [2]) zenvisage enables users to effectively and accurately perform visual exploration tasks, as shown by our user study, and complements other tools. In addition, we show that our optimizations for ZQL execution lead to considerable improvements over leveraging the parallelism inherent in databases. Our work is a promising first step towards substantially simplifying and improving the process of interactive data exploration for novice and expert analysts alike.

9. REFERENCES

[1] Airline dataset (http://stat-computing.org/dataexpo/2009/the-data.html). [Online; accessed 30-Oct-2015].

[2] Effortless data exploration with zenvisage: An expressive and interactive visual analytics system. *Technical Report*. http://data-people.cs.illinois.edu/zenvisage.pdf.

[3] Spotfire, http://spotfire.com. [Online; accessed 17-Aug-2015].

[4] Tableau public (www.tableaupublic.com/). [Online; accessed 3-March-2014].

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

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

[7] Tableau q2 earnings: Impressive growth in customer base and revenues. http://www.forbes.com/sites/greatspeculations/2015/07/31/tableau-q2-earnings-impressive-growth-in-customer-base-and-revenues.

[8] C. Ahlberg. Spotfire: An information exploration environment. *SIGMOD Rec.*, 25(4):25–29, Dec. 1996.

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

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

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

[12] H. Gonzalez et al. Google fusion tables: web-centered data management and collaboration. In *SIGMOD Conference*, pages 1061–1066, 2010.

[13] K. Goto and R. A. Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.

[14] J. Han et al. Dmql: A data mining query language for relational databases. In *Proc. 1996 SiGMOD*, volume 96, pages 27–34, 1996.

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

[16] T. Imielinski and A. Virmani. A query language for database mining. *Data Mining and Knowledge Discovery*, 3(4):373–408, 2000.

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

[18] S. Kandel et al. Profiler: integrated statistical analysis and visualization for data quality assessment. In *AVI*, pages 547–554, 2012.

[19] A. Key, B. Howe, D. Perry, and C. Aragon. Vizdeck: Self-organizing dashboards for visual analytics. SIGMOD '12, pages 681–684, 2012.

[20] M. Livny et al. Devise: Integrated querying and visualization of large datasets. In *SIGMOD Conference*, pages 301–312, 1997.

[21] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, Apr. 1986.

[22] J. D. Mackinlay et al. Show me: Automatic presentation for visual analysis. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1137–1144, 2007.

[23] A. Netz et al. Integrating data mining with sql databases: Ole db for data mining. In *ICDE'01*, pages 379–387. IEEE, 2001.

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

[25] S. Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, pages 42–53, 1999.

[26] G. Sathe and S. Sarawagi. Intelligent rollups in multidimensional olap data. In *VLDB*, pages 531–540, 2001.

[27] T. K. Sellis. Multiple-query optimization. *ACM TODS*, 13(1):23–52, 1988.

[28] C. Stolte et al. Polaris: a system for query, analysis, and visualization of multidimensional databases. *Commun. ACM*, 51(11):75–84, 2008.

[29] P. Terlecki et al. On improving user response times in tableau. In *SIGMOD*, pages 1695–1706. ACM, 2015.

[30] M. Vartak et al. Seedb: Efficient data-driven visualization recommendations to support visual analytics. *VLDB*, 8(13), Sept. 2015.

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

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

[33] M. M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.