ABSTRACT

Data visualization is by far the most commonly used mechanism to explore data, especially by novice data analysts and data scientists. And yet, current visual analytics tools are rather limited in their ability to guide data scientists to interesting or desired visualizations: the process of visual data exploration remains cumbersome and time-consuming. We propose zenvisage, a platform for effortlessly visualizing interesting patterns, trends, or insights from large datasets. We describe zenvisage’s general purpose visual query language, ZQL (“zee-quel”) for specifying the desired visual trend, pattern, or insight — ZQL draws from use-cases in a variety of domains, including biology, mechanical engineering, commerce, and server monitoring. While analysts are free to use ZQL directly, we also expose ZQL via a visual specification interface, which we also describe in this paper. We then describe our architecture and optimizations, as well as preliminary experiments in supporting and optimizing for ZQL queries in our initial zenvisage prototype.

1. INTRODUCTION

The rising popularity of visual analytics tools have paved the way for the democratization of data exploration and data science. Increasingly, amateur data scientists from a variety of domains and sectors now have the ability to analyze and derive insights from datasets of increasing size and complexity. The standard recipe for data science then goes as follows: the data scientist loads the dataset into a visual analytics tool like Tableau [4] or Spotfire [3], or even a domain-specific data exploration tool, they create or select visualizations, and then examine whether those visualizations capture desired patterns or insights. If these visualizations do not meet the desired requirements, the process is repeated by examining a whole range of additional visualizations until they find the visualizations that do. This data exploration process is often cumbersome and time-consuming, since the number of visualizations that the data scientists need to examine to derive desired insights grows rapidly with size of the dataset: the number of records as well as the number of attributes across different relations. To illustrate, consider the following real case studies that demonstrate the limitations of current data exploration tools:

Case Study 1: Advertising Analytics. Advertisers—from small businesses to large corporations—at search company X are often interested in examining their portfolio of sponsored search and display ads in order to see if their advertising 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 the Asia-Pacific region. To do this using the current visual analytics tools available at X, the advertiser needs to generate the plot of click-through rates (CTR) over time for each keyword, and then examine each one of them in turn.

Case Study 2: Genomic Data Analysis. Clinical researchers at NIH’s BD2K (Big Data 2 Knowledge) Center at the University of X are interested to quickly gain insights about gene-gene and protein-protein relationships in the context of various clinical trials. For instance, one common task that the clinical researchers would like to perform is to compare two classes of genes and see what factors help them visually understand the differences between these classes. (For example, one class could be those genes positively correlated with cancer, while the rest are not.) To do this using the current visual analytics tools available at the center, these researchers would have to generate scatterplots corresponding to all pairs of factors, and examine each one in turn to see if the two classes of genes are well-separated in these scatterplots.

Case Study 3: Engineering Data Analysis. Battery scientists at X University are interested in performing visual exploration of a dataset containing electrolytes and their properties at various scales— molecular, meso, and continuum scales. For instance, one task that scientists often do is find classes of solvents that have desired behavior: for instance, solvents whose solvation energy of Li$^+$ vs. vs the boiling point is an increasing trend. To do this using current tools, these scientists would need to generate these plots for all solvent classes and manually examine each one.

Case Study 4: Server Monitoring Analysis. The server monitoring team at company X has noticed a spike in the per-query response time for Image Search in Russia around August 15, after which the response time has flattened out. The team would like to identify if there are other attributes that have a similar behavior with per-query response time, which may indicate the reason for the spike and subsequent flattening. To do this, the server monitoring team needs to generate visualizations for different metrics as a function of the date, and see if any of them has similar behavior to the response time for Image Search. Given that the number of metrics is likely in the hundreds or thousands, this could take a very long time.

Case Study 5: Mobile App Analysis. The complaints team at the mobile platform team of company X have noticed that a certain mobile app has received many complaints. They would like to figure out what is different about this app relative to others. To do this, they would need to plot various metrics for this app to figure out why it is behaving anomalously. For instance, they may look at network traffic generated by this app over time, or at the distribution of energy consumption across different users. In all of these cases, the team would need to generate several visualizations man-

---

$^1$While we have anonymized the identities of these teams and individuals for double-blind reasons, we must emphasize that these are very real teams; our research is being performed in active collaboration with these teams.
ually and browse through all of them in the hope of finding what could be the issues with the app.

In all of the above scenarios, there is the following recurring theme: *generate a large number of visualizations and evaluate each one for a desired visual property.*

Instead, our goal in this paper is to build zenvisage, a visual analytics system that can “fast-forward” to the desired insights, thereby minimizing significant burden on the part of the data scientists or analysts in scenarios like the ones described above.

Given the wealth of data analytics tools available, one may ask why a new tool is needed. With these tools, selecting the “right” view on the data that reveals the “desired” insight still remains laborious and time-consuming. The onus is on the user to manually specify the view they want to see, and then repeat this until they get the desired view. In particular, existing tools are inadequate, including:

- **Relational databases:** Databases are powerful and efficient, but the rigidity of the syntax limits the users ability to express queries like “show me visualizations of keywords where CTR over time in Asia is behaving unusually”.

- **Data mining tools:** Data mining tools are hard to expect users to use, since it involves extensive programming, as well as an understanding of which data mining tool applies to what purpose. Certainly, expressing each query will require extensive programming and manual optimization (not desirable for ad-hoc querying).

- **Visual analytics tools:** Visual analytics tools like Tableau and Spotfire have made it much easier for business analysts to analyze data; that said, the user needs to exactly specify what they want to visualize. If the visualization does not yield the desired insight, then the user must try again, now with a different visualization. One can view zenvisage as a generalization of standard visualization specification tools like Tableau; capturing all the Tableau functionality, while providing the means to skip ahead to the desired insights.

We describe related work in more detail in Section 6.

In this paper, we describe the specification for our query language for zenvisage, ZQL, that is an extension of the Polaris table algebra [35], which is the algebra underlying Tableau. We describe how ZQL is powerful enough to capture the use cases described above as well as many many other use cases (Section 2). Our primary contribution in this paper is ZQL, which resulted from a synthesis of desiderata after discussing with analytics teams from a variety of domains (described above). We develop a number of simple optimizations that we can use to simplify the execution of ZQL queries by minimizing the number of SQL queries that are issued (Section 3). We also describe our initial prototype of zenvisage which implements a subset of ZQL, the end-user interface, as well as the underlying systems architecture (Section 4). We describe our initial performance experiments (Section 5).

### 2. QUERY LANGUAGE

Our system zenvisage’s query language, ZQL, provides users a flexible and intuitive mechanism to specify desired insights from visualizations. The user may either directly write ZQL queries, or they may use the zenvisage front-end, which transforms all requests to ZQL queries internally. Our design of ZQL builds on work on visualization specification for visual analytics tools and platforms, in particular from Polaris [35], and Grammar of Graphics [40]. (Indeed, zenvisage is intended to be a generalization of Polaris/Tableau [35], and hence must encompass Polaris functionality as well as additional functionality for searching for desired trends, patterns, and insights.) In addition, ZQL also draws heavy inspiration from the Query by Example (QBE) Language [42] and uses a similar a similar table-based interface.

Our goal for ZQL was to ensure that users would be able to effortlessly express complex requirements using a small number of ZQL lines. Furthermore, the language itself should be robust and general enough to capture the wide range of possible visual queries. As we will see later, despite the generality of the language, we have built an automatic parser and optimizer that can apply to any ZQL query and transforms it into a collection of SQL queries, along with post-processing that is run on the results of the SQL queries: this means that zenvisage can use as a backend any traditional relational database. To illustrate the power and the generality of the language, we now illustrate a few examples of ZQL queries, before we dive into the ZQL formalism. To make it easy to follow without much background, we use a product sales-based dataset throughout this paper in our query examples—we will reveal attributes of this dataset as we go along.

**Query 1: Depict a collection of visualizations.** Table 1 depicts a very simple ZQL query. This ZQL query retrieves the data for each product’s total sales over years bar chart visualization for products sold in the US. As the reader can probably guess, the ‘year’ and ‘sales’ in the X and Y columns dictate the x- and y- axes of the visualizations, and the mlocation=’US’ in the Constraints column constrains the data to items sold in the US. Then, for the Z column, we use the variable v1 to iterate over ‘product.*’, the set of all possible product values. The bar.(y=agg(’sum’)) denotes that the visualization be a bar chart where the y-values are aggregated using the SUM function grouped by both the x- and z- axes. The Process column is typically used to filter, sort, or compare visualizations, but in this case, since we want the full set of visualizations (one for every product), we leave the the Process column blank. This generates a separate sales vs. year plot for each product, giving a collection of resulting visualizations. This collection of visualizations is referred to using the variable f1, with the * indicating that these visualizations are to be output to the user. (Note that both variables v1 and f1 are redundant in this current query, but will come in handy for other more complex queries.) Naturally, if the number of products is large, this could lead to a large number of visualizations being displayed, and so may not be desirable for the user to peruse. Later, we will describe mechanisms to constrain the space of visualizations that are displayed to be those that satisfy a user need. The idea that we can represent a set of visualizations with just one line is a powerful one, and it is part of what makes ZQL such an expressive language.

**Query 2: Find the product which has the most similar sales trend as the user-drawn input trend line.** Table 2 provides an example which integrates ZQL with user-drawn trend lines. Using zenvisage’s front-end, the user can draw a trend line2, which ZQL can use as an input and compare against other visualizations from the database. In Table 2, we use - in the -f1 to denote that it corresponds to a visualization provided by the user. After the user input line, we see a second line which looks similar to the example in the first query; f2 iterates over the set of sales over year visualizations for each product. With the Process column, we can compare the visualizations f1 and f2 with some distance metric D for every product value in v1. argmin looks through the comparisons and selects the one product which minimizes the distance. Finally, f3

---

2zenvisage provides many options for user input, including directly drawing a visualization using the zenvisage front-end, providing a set of data values, or specifying a list of constraints of what the visualization could be.
outputs the sales over year visualization for that product. Thus, with Table 2, we have managed to perform a search for visualizations against a user-drawn input. For this specific query, we depict zenvisage in action in Figure 3 in the System Architecture Section (Section 4).

**Query 3: Find and visualize profit for products that are doing well in the US but badly in the UK.** Finally, we look at an even more complex example in Table 3. This query captures the scenario of wishing to find the profit over years visualizations for products that have positive sales over years trends for the US but have negative sales over years trends for the UK. This is a visual query many business users would like to be able to perform. However, the way users would currently achieve this is by manually visualizing the sales over years for both the US and the UK for every item and remembering which item had the most discrepancy. With ZQL, the visual query can be expressed with three lines. Line one retrieves the set of sales over years visualizations for each product sold in the US and filters it to only include the ones in which the overall trend over years is positive. Line two retrieves the set of sales over years visualizations for each product sold in the UK and filters it to only include the ones in which the overall trend over years is negative. The third row combines the results of the two by taking the intersection of the sets of products. The type of visualizations for products sold in the US and filters it to only include the ones in which the overall trend over years is positive and the type of visualizations for products sold in the UK and filters it to only include the ones in which the overall trend over years is negative. The 10 visualizations which form the representative set of the visualizations in each row are then returned to the user. In the rest of this section, we go more in depth into the ZQL language and provide a formal specification for ZQL. We then provide several more real-world examples of how a user would use ZQL to find the visualizations she is interested in.

### 2.1 Formalization

We now formally describe the ZQL syntax. Throughout, we assume that we’re operating on a single relation or a star schema where the attributes are uniquely defined (barring key-foreign key joins). In general, ZQL could be applied to arbitrary collections of relations by letting the user precede an attribute name with the relation name, e.g., \textit{RA}. But for ease of exposition, we focus on the single relation case.

#### 2.1.1 Overview

As described earlier, a ZQL query is composed using a table, much like a QBE query. Unlike QBE, the columns of a ZQL query do not refer to attributes of the table being operated on; instead, they are predefined, and have fixed semantic meanings. In particular, at a high level, the columns are:

- Name: providing an identifier for a set or collection of visualizations, and allowing us to indicate if a specific set of visualizations are to be output;
- X, Y: specifying the X and Y axes of the collections of visualizations, restricted to sets of attributes;
- Z (optional): specifying the “slice” (or subset) of data that we’re varying, restricted to sets of attributes along with values for those attributes;
- Constraints (optional): specifying optional constraints applied to the data prior to any visualizations or collections of visualizations being generated.
- Viz (optional): specifying the mechanism of visualization, e.g., a bar chart, scatterplot, as well as the associated transformation, or aggregation, e.g., the X axis is binned in groups of 20, while the Y axis attribute is aggregated using SUM. If this is not specified, standard rules of thumb are used to determine the appropriate visualization [41, 27];
- Process (optional): specifying the “optimization” operation performed on a collection of visualizations, typically intended towards identifying desired visualizations.

A ZQL query may have any number of rows, and conceptually each row represents a set of visualizations the user is interested in. The user can then process and filter these rows until she is left with only the output visualizations she is interested in. The result of a ZQL query is the data used to generate visualizations. The zenvisage front-end then generates the visualizations for the user to peruse.

### 2.1.2 X and Y Columns

As mentioned, each row can be thought of as a set of visualizations, and the X and Y columns represent the x- and y- axes for those visualizations. In Table 1, the first row’s visualizations will have ‘time’ as their x-axis and ‘profit’ as their y-axis.

The only permissible entries for the X or Y column are either a single attribute from the table, or a set of attributes, along with a variable to iterate over them. The exact semantics of variables and sets are discussed later in Section 2.1.7, but essentially, a column is allowed to take on any attribute from the set. For example, in Table 4, the y-axis is allowed to take on either ‘profit’ or ‘sales’ as its attribute. Because the y-axis value can be taken from a set of attributes, the resulting output of this query is actually the set of visualizations whose x-axis is the ‘time’ and the y-axis is one of ‘profit’ and ‘sales’ for the ‘stapler’. This becomes a powerful notion later in the process column where we try to iterate over a set of visualizations and identify desired ones to return to the user or select for downstream processing.
2.1.3 Z Column

The Z column is used to either focus our attention on specific slice (or subset) of the dataset or iterate over a set of slices for one or more attributes. To specify a set of slices, the Z column must specify (a) one or more attributes, just like the X and Y column, and (b) one or more attribute values for each of those attributes — which allows us to slice the data in some way. For both (a) and (b) the Z column could specify a single entry, or a variable associated with a set of entries. Table 5 gives an example of using the Z column to visualize the sales over time data specifically with regards to the 'chair' and 'desk' products, one per line. Note that the attribute name and the attribute value are separated using a period.

Table 5: A ZQL query for a set of visualizations which returns the sales over year visualization for chairs and the sales over time visualization for desks.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>year</td>
<td>y1 &lt;- ('profit', 'sales')</td>
<td>product='stapler'</td>
</tr>
<tr>
<td>v2</td>
<td>year</td>
<td>y1 &lt;- ('profit', 'sales')</td>
<td>product='chair'</td>
</tr>
<tr>
<td>v3</td>
<td>year</td>
<td>y1 &lt;- ('profit', 'sales')</td>
<td>product='desk'</td>
</tr>
</tbody>
</table>

Table 6: A ZQL query for a set of visualizations which returns the set of sales over year visualizations for each product.

Furthermore, the Z column can be left blank if the user does not wish to slice the data in any way.

Additional Details. ZQL also allows the iteration over attributes in the Z column as shown in Table 7. The result of this query is the set of all sales over time visualizations for every possible slice in every dimension except ‘time’ and ‘sales’. Since both attribute and attribute value can vary in this case, we need separate variables for each component, and the full attribute name, value pair (z1.v1) must be specified. Note that the resulting set of visualizations comes from the Cartesian product of possible attribute and attribute value pairs. The first * symbol refers to all possible attributes, while the second * symbol refers to all possible attribute values given an attribute. If the user wishes to specify specific subsets of attribute values for attributes, she must name them individually. An example of this is given in Table 8 where the z1 and v1 iterate over the pairs {('product', 'chair'), ('product', 'desk'), ('location', 'US')}, and v2 over {('product', 'stapler')}.

Table 7: A ZQL query which returns the set of sales over year visualizations for each attribute that is not time or sales.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>year</td>
<td>y1 &lt;- ('profit', 'sales')</td>
<td>product='stapler'</td>
</tr>
<tr>
<td>z1.v1</td>
<td>v2</td>
<td>v1 &lt;- (*, ('year', 'sales')) *</td>
<td></td>
</tr>
</tbody>
</table>

2.1.4 Constraints Column

The constraints column is used to specify further constraints on the set of data used to generate the set of visualizations. Conceptually, we can view the constraints column as being applied to the dataset first, following which a collection of visualizations are generated on the constrained dataset.

Table 8: A ZQL query which returns a set of sales over year visualizations; one for chairs, one for desks, and one for items sold in the US. (Name is not displayed above.)

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>'year'</td>
<td>'sales'</td>
<td>z1.v1 &lt;- ('product', 'chair', 'desk')</td>
</tr>
</tbody>
</table>

Table 9: A ZQL query which returns the set of sales over year visualizations for each product in USA and Canada (Name is not displayed above.)

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>'year'</td>
<td>'sales'</td>
<td>v1 &lt;- 'product' *</td>
</tr>
</tbody>
</table>

2.1.5 Viz Column

Given the data from the X, Y, Z, and Constraints columns, the Viz column determines how the data is shaped or modified before returning it as a visualization to the user. There are two aspects to the Viz column: first, the visualization type (e.g., bar chart, scatter plot), and the summarization type (e.g., binning or grouping in some way, aggregating in some way). These two aspects are akin to the geometric and statistical transformation layers from the Grammar of Graphics, a language for specifying visualizations, and the inspiration behind ggplot [38].

In ZQL, the Viz column specifies the visualization type and the summarization using a period delimiter. Functions are used to represent both the visualization type and the summarization. To give a concrete example, a query such as Table 10 might be used to generate a total sales by weight bar chart, where the weights are grouped in bins of size 20. Here, the user specifies that she wants a bar chart, with bar, and specifies the type of summarization in the accompanying tuple: x=bin(20) denotes that x-axis should be binned into bins of size 20, and y=agg('sum') runs the SUM aggregation on the y-values when grouped by both the bins of the x-axis and the values in the z-axis, (if any are specified).

Although we can explicitly specify the Viz column, often we can leave the Viz column blank. In such cases, we would apply well-known rules of thumb for what types of visualization and summarization would be appropriate given specific X and Y axes. Work on recommending appropriate visualization types dates back to the 80s [27], that both Polaris/Tableau [35], and recent work builds...
on [41, 21], determining the best visualization type by examining the schema and statistical properties. In many of our examples of ZQL queries, we omit the Viz column for this reason.

2.1.6 Name Column

For any row of a ZQL query, the combination of X, Y, Z, Constraints, and Viz columns together represent the visual component of that row. A visual component represents a set of visualizations. The Name column allows us to provide a name to this visual component by binding a variable to it. These variables can be used in the Process column to subselect the desired visualizations from the set of visualizations, as we will see subsequently.

In the ZQL query given by Table 11, we see that the names for the visual components of the rows are named f1, f2, and f3 in order of the rows. For the first row, the visual component is a single visualization, since there are no sets, and f1 binds to the single visualization. For the second row, we see that the visual component is over the set of visualizations with varying Z column values, and f2 binds to the variable which iterates over this set. Note that f1 and f3 in Table 11 are prefaced by a * symbol. This symbol indicates that the visual component of the row is designated to be part of the output. As can be seen in the example, multiple rows can be part of the output, and in fact, even a row could correspond to multiple visualizations. Visualizations corresponding to all the rows marked with * are processed by the zenvisage frontend, which displays each of them.

2.1.7 Sets and Variables

Before we move on to the Process column, we must give the full formal semantics for sets and variables in ZQL. As we have alluded many times, sets in ZQL must always be accompanied by a variable which iterates over that set. This requirement was made to ensure that ZQL traverses over sets of visualizations in a consistent order when making comparisons. We use Table 13 to highlight the importance of this issue. Table 13 shows a query that iterates over the set of products, and for each of these products, compares the sales over years visualization with the profits over years visualization. Without a variable enforcing a consistent iteration order over the set of products, it is possible that the set of visualizations could be traversed in unintended ways. For example, the sales vs. year plot for chairs from the first set could be compared with the profit vs. year plot for desks in the second set, which is not what the user intended. By reusing v1 in both the first and second rows, the user can force the zenvisage back-end to step through the sets with the same product selected in each iteration.

Operations on Sets. Constant sets in ZQL must use \{\} to denote that the enclosed elements are part of a set. As a special case, the user may also use * to represent the set of all possible values. The union of sets can be taken using the \| sign; set difference can be taken using the \( sign, and the intersection can be taken with &. Ordering. All sets in zenvisage are ordered, but when defined using the \{\}, the ordering is defined arbitrarily. Only after a set has been passed through a Process column’s ordering mechanism can the user depend on the ordering of the set. Ordering mechanisms are discussed further in Section 2.1.8. However, even for arbitrarily ordered sets, if a variable iterator is defined for that set and reused, ZQL guarantees at least a consistent ordering in traversal across that set. This is what allows the sets in Table 13 to be traversed in the intended order.

Axis Variables. In ZQL, there are two types of variables: axis variables and name variables. Axis variables are the common variables used in any of the columns except the Name column. The declaration has the form: \(\text{variable name} \leftarrow \langle\text{set}\rangle\). The variable then can be used as an iterator in the Process column or reused in a different table cell to denote that the set be traversed in the same way for that cell. It is possible to declare multiple axis variables at once, as we have seen from the Z and Process columns: (e.g., \(z.v \leftarrow *, *,\)).

Sometimes, it is necessary to retrieve the set that an axis variable iterates over. In ZQL, the \(\text{range}\) notation allows the user to expand variables to their corresponding sets and apply arbitrary zenvisage set operations on them. For example, \(v4 \leftarrow (v2.range | v3.range)\) declares and binds v4 to the union of the sets iterated by v2 and v3.

Axis variables can be used freely in any column except the Constraints column. In the Constraints column, only the expanded set form of a variable may be used. Table 14 demonstrates how to use an axis variable in the Constraints. Column. In this example, the user is trying to plot the overall profits over years for the top 10 products which have had the most growth in sales over the years. The user finds these top 10 products in the first row and declares the variable v2 to iterate over that set. Afterwards, she uses the constraint product \(<\langle v2.range\rangle\) to get the overall profits across all 10 of these products in the second row.

Name Variables. Name variables (e.g., f1, f2) are declared only in the Name column and used only in the Process column. Named variables are iterators over the set of visualizations represented by the visual component. If the visual component represents only a single visualization, the name variable is set to that specific visualization. Name variables are bound to the axis variables present in their row. In Table 14, f1 is bound v1, so if v1 progresses to the next product in the set f1 also progresses to the next visualization in the set. This is what allows the Process column to iterate over the axis variable (v1), and still compare using the name variable (f1).

If multiple axis variables are present in the visual component, the name variable iterates over set of visualizations produced by the Cartesian product of the axis variables. If the axis variables have been declared independently of each other, the ordering of the Cartesian product is unspecified. However, if the variables were declared together as is the case with x2 and y2 in Table ??, the ordering is the same as the set in the declaration.

2.1.8 Process Column

Once the visual component for a row has been named, the user may use the Process column to sort, filter, and compare the visual component with previously defined visual components to isolate the set of visualizations she is interested in. While the user is free to define her own functions for the Process column, we have come up with a core set of primitives based on our case studies which we believe can handle the vast majority of the common exploration operations. Each non-empty process column entry is defined to be a task. Implicitly, a task may be impacted by one or more rows of a ZQL query (which would be input to the process optimization), and may also impact one or more rows (which will use the outputs of the process optimization), as we will see below.

Functional Primitives. First, we introduce three simple functions that can be applied to visualizations: T, D, and R. zenvisage will use default settings for each of these functions, but the user is free to specify their own variants for each of these functions that are more suited to their application.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
<th>Viz</th>
</tr>
</thead>
<tbody>
<tr>
<td>#f1</td>
<td>'weight'</td>
<td>'sales'</td>
<td>bar.(x=bin(20), y=agg('sum'))</td>
</tr>
</tbody>
</table>
Table 11: A ZQL query retrieving the sales over year visualization for the top 10 products whose sales over year visualization looks the most similar to that of the stapler.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Constraints</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>'year'</td>
<td>'sales'</td>
<td>v1 &lt;- 'product':* ('* - 'stapler')</td>
<td></td>
<td>v2 &lt;- argmin_{i \in</td>
</tr>
<tr>
<td>x2</td>
<td>'year'</td>
<td>'sales'</td>
<td>v1 &lt;- 'product':* ('* - 'stapler')</td>
<td></td>
<td>v3 &lt;- argmax_{i \in</td>
</tr>
<tr>
<td>x3</td>
<td>'year'</td>
<td>'sales'</td>
<td>v2</td>
<td></td>
<td>v2 &lt;- argmax_{i \in</td>
</tr>
</tbody>
</table>

Table 12: A ZQL query retrieving two different visualisations (among different combinations of x and y) of stapler which are most similar to each other.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Constraints</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>'year'</td>
<td>'sales'</td>
<td>v1 &lt;- 'product':* ('* - 'stapler')</td>
<td></td>
<td>v2 &lt;- argmin_{i \in</td>
</tr>
<tr>
<td>x2</td>
<td>'year'</td>
<td>'sales'</td>
<td>v1 &lt;- 'product':* ('* - 'stapler')</td>
<td></td>
<td>v3 &lt;- argmax_{i \in</td>
</tr>
<tr>
<td>x3</td>
<td>'year'</td>
<td>'sales'</td>
<td>v2</td>
<td></td>
<td>v2 &lt;- argmax_{i \in</td>
</tr>
</tbody>
</table>

Table 13: A ZQL query which returns the set sales over years and the profit over years visualizations for the top 10 products for which these two visualizations are the most dissimilar.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Constraints</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>'year'</td>
<td>'sales'</td>
<td>v1 &lt;- 'product':* ('* - 'stapler')</td>
<td></td>
<td>v2 &lt;- argmin_{i \in</td>
</tr>
<tr>
<td>x2</td>
<td>'year'</td>
<td>'sales'</td>
<td>v1 &lt;- 'product':* ('* - 'stapler')</td>
<td></td>
<td>v3 &lt;- argmax_{i \in</td>
</tr>
<tr>
<td>x3</td>
<td>'year'</td>
<td>'sales'</td>
<td>v2</td>
<td></td>
<td>v2 &lt;- argmax_{i \in</td>
</tr>
</tbody>
</table>

Table 14: A ZQL query which plots the profit over years for the top 10 products which have the highest sloping trend lines for sales over the years.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Constraints</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>'year'</td>
<td>'sales'</td>
<td>v1 &lt;- 'product':* ('* - 'stapler')</td>
<td></td>
<td>v2 &lt;- argmin_{i \in</td>
</tr>
<tr>
<td>x2</td>
<td>'year'</td>
<td>'sales'</td>
<td>v1 &lt;- 'product':* ('* - 'stapler')</td>
<td></td>
<td>v3 &lt;- argmax_{i \in</td>
</tr>
<tr>
<td>x3</td>
<td>'year'</td>
<td>'sales'</td>
<td>v2</td>
<td></td>
<td>v2 &lt;- argmax_{i \in</td>
</tr>
</tbody>
</table>

Table 15: A ZQL query which returns 10 sales over years visualizations for products which are outliers compared to the rest.

• $T(f)$ measures the overall trend of visualization $f$. It is positive if the overall trend indicates “growth” and negative if the overall trend goes “down”. There are obviously many ways that such a function can be implemented, but one example implementation might be to measure the slope of a linear fit to the given input visualization $f$.

• $D(f, f')$ measures the distance between the two visualizations $f$ and $f'$. For example, this might mean calculating the Earth Mover’s Distance or the Kullback-Leibler Divergence [39] between the induced probability distributions.

• $R(k, v, f)$ computes the set of $k$-representative visualizations given an axis variable, $v$, and an iterator over the set of visualizations, $f$. Different users might have different notions of what representative means, but one example would be to run $k$-means clustering on the given set of visualizations and return the $k$ centroid visualizations.

In addition to taking in a single axis variable, $v$, the user can also take in a tuple of axis variables, $v$. This is the set of axis variable values which produced the representative visualizations.

**Optimization.** Given these functional primitives, ZQL also provides some default sorting and filtering mechanisms: argmin, argmax, and argany. Although argmin and argmax are usually used to find the best value for which the objective function is optimized, ZQL typically returns the top-$k$ values, sorted in order. argany is used to return any $k$ values. In addition to the top-$k$, the user might also like to specify that she wants every value for which a certain threshold is met, and ZQL is able to support this as well.

Specifically, the expression $v2 \leftarrow \text{argmax}_{i \in |\mathcal{T}!1|} [k \in (0,1)] D(f1, f2)$ returns the top 10 $v1$ values for which $D(f1, f2)$ are maximized, sorts those in decreasing order of the distance, and declares the variable $v2$ to iterate over that set. This could be useful, for instance, to find the 10 visualizations with the most variation on some attributes. The expression $v2 \leftarrow \text{argmin}_{i \in |\mathcal{T}!1|} [k \in (0,1)] D(f1, f2)$ returns the $v1$ values for which the objective function $D(f1, f2)$ is below the threshold $0$, sorts the values in increasing order of the objective function, and declares $v2$ to iterate over that set. If a filtering option ($k,t$) is not specified, the mechanism simply sorts the values, so $v2 \leftarrow \text{argmin}_{i \in |\mathcal{T}!1|} T(f1)$ would bind $v2$ to iterate over the values of $v1$ sorted in increasing order of $T(f1)$. $v2 \leftarrow \text{argmin}_{i \in |\mathcal{T}!1|} [k \in (0,1)] D(f1, f2)$ would set $v2$ to iterate over the values of $v1$ for which the $T(f1)$ is greater than 0.

Note that the mechanisms may also take in multiple axis variables, as we saw from Table ?? . The mechanism iterates over the Cartesian product of its input variables. If a mechanism is given $k$ axis variables to iterate over, the resulting set of values must also be bound to $k$ declared variables. In the case of Table ??, since there were two variables to iterate over, $x1$ and $y1$, there are two output variables, $x2$ and $y2$. The order of the variables is important as the values of the $i$th input variable are set to the $j$th output variable.

**User Exploration Tasks.** By building on these functions and mechanisms, the user can perform most common tasks. This includes the similarity/dissimilarity search performed in Table 13, where the user intends to identify products that are similar; the comparative search performed in Table ??, where the user intends to identify attributes on which two slices of data are dissimilar, or even the outlier search query shown in Table 15, to identify the products that are outliers on the sales over year visualizations. Note that in Table 15, we use two levels of iteration.

### 2.2 Examples and Intuitions

To demonstrate the full expressive power of ZQL, we present three realistic, complex example queries. We show that even with complicated scenarios, the user is able to capture the insights she wants with a few meaningful lines of ZQL.

**Query 1.** The stapler has been one of the most profitable products in the last years for GlobalMart. The Vice President is interested in learning about other products which have had similar profit trends. She wishes to see some representative sales over the years visualizations for these products.

Table 16 shows what the query that the Vice President would write for this scenario. She first filters down to the top 100 products which have the most similar to profit over year visualizations to that of the stapler’s using the argmin in the second row. Then, from the resulting set of products, $v2$, she picks the 10 most representative set of sales over visualizations using $R$, and displays those
visualizations in the next line with \( \Phi \). Although the Vice President does not specify the exact distance metric for \( D \) or specify the exact algorithm for \( R \), she knows zensigave will select the most reasonable default based on the data.

**Query 2.** The Vice President, to her surprise, sees that there a few products whose sales has gone up over the last year, yet their profit has declined. She also notices some product’s sales have gone down, yet their profit has increased. To investigate, the Vice President would like to know about the top 10 products who have the most discrepancy in their sales and profit trends, and she would like to visualize those trends.

This scenario can be addressed with the query in Table 17. The Vice President names the set of visualizations for profit over month \( f_1 \) and the sales over month visualizations \( f_2 \). She then compares the visualizations in the two set using the \( \text{argmax} \) and retrieves the top 10 products whose visualizations are the most different. For these visualizations, she plots both the sales and profit over months; \( y_1 \leftarrow \{ \text{"sales"}, \text{"profit"} \} \) is a shortcut to avoid having to separate rows for sales and profit. Note that the Vice President was careful to constrain QQL to only look at the data from 2015.

**Query 3.** Finally, the Vice President would like to know more about the differences between a product whose sales numbers do not change over the year and a product that has the largest growth in the number of sales. To address this question, she writes the query in Table 18. The first \( R \) function call returns the one product whose sales over year visualization is most representative for all products; \( v_2 \) is set to the product that has the most average number of sales. The task in the second row selects the product \( v_3 \) which has the greatest upward trending slope \( T \) for sales. Finally, the Vice President tries to finds the \( y \)-axes which distinguish the two products the most, and visualizes them. Although we know \( v_2 \) and \( v_3 \) only contain one value, they are still sets, so \( \text{argmax} \) must iterate over them and output corresponding values \( v_4 \) and \( v_5 \).

### 3. QUERY EXECUTION

In zensigave, ZQL queries are automatically parsed and executed by the zensigave backend. Assuming the dataset is stored in a database, the ZQL compiler is capable of translating any ZQL query into a collection of SQL queries and accompanying post-processing computations that must be performed over the results of the SQL queries. Next, we first describe how a naive zensigave compiler would translate ZQL queries. Then, we provide optimizations over this naive zensigave compiler to reduce the number of SQL queries that must be issued.

#### 3.1 Naive Translation

A ZQL query can be broken up into two parts: the visual components and the the Process column. In the ZQL compiler, the visual components determine the set of data zensigave needs to retrieve from the database, and the Process column translates into the post-processing computation to be done on the returned data.

zensigave’s naive ZQL compiler performs the following operations for each row of a ZQL query. For each row, at a high level, the ZQL compiler issues a SQL query corresponding to each visualization in the set of visualizations specified in the visual component. Notice that this approach is akin to what a visual analyst manually generating each visualization of interest and perusing it would do; here, we are ignoring the human perception cost. More formally, for each row (i.e., each name variable), the ZQL compiler loops over all combinations of values for each of the \( n \) axis variables in the visual component corresponding to that row, and for each combination, issues a SQL query, the results of which are stored in the corresponding location in an \( n \) dimensional array. (Essentially, this corresponds to a nested for loop \( n \) levels deep.) Each generated SQL query has the form:

```
SELECT X, Y
WHERE Z = V and (CONSTRAINTS)
ORDER BY X
```

If the summarization is specified, additional clauses and keywords such as \text{GROUP BY}s and aggregations may be added. The results of each visualization are stored in an \( n \)-dimensional array at the current index. If \( n = 0 \), the name variable points directly to the data.

Once the name variable array has been filled in with results from the SQL queries, the ZQL compiler generates the post-processing code from the task in the Process column. At a high level, ZQL loops through all the visualizations, and applies the objective function on each one. In particular, for each input axis variable in the mechanism, a for loop is created and nested. The input axis variables are then used to step through the arrays specified by the name variable, and the objective function is called at each iteration. Instances of name variables are updated with the correct index based on the axis variables it depends on. The functions themselves are considered black boxes to the ZQL compiler and are unaltered.

As an example, we provide the pseudocode of the compiled version of the ZQL query expressed in Table 19 with Listing 1 in the appendix.

### 3.2 Optimizations

While the translation process of the naive ZQL compiler is simple and easy to follow, there is plenty of room for optimization. In particular, we look at three levels of optimization which focus on batching SQL queries. We found that the time it took to submit and wait for SQL queries was a bottleneck, and reducing the overall number of SQL queries can reduce performance significantly. While none of these optimizations may seem particularly surprising, it required a substantial amount of effort to identify how these optimizations may be applied automatically to any ZQL query.

#### 3.2.1 Intra-Line Optimization

The first level of optimization batches the SQL queries for a row into one query. For example, we see that for the first row Table 19, that a separate query is being made for every product in Listing 1 in the appendix. Instead, we can retrieve the data for all products in one SQL query:

```
SELECT year, SUM(sales), product
WHERE product IN P and location='US'
GROUP BY product, year
ORDER BY product, year
```

If the axis variable is in either the X or Y columns, we retrieve the data for the entire set of attributes the axis variable iterates
over. More concretely, if we have axis variable \( y_1 \) \( \in \{ \text{"sales"}, \text{"profit"} \} \), our SQL query would look like:

```sql
SELECT year, SUM(sales), SUM(profit), product
WHERE product IN P and location='US'
GROUP BY product, year
ORDER BY product, year
```

This optimization cuts down the number of queries by the sizes of sets the axis variables range over, and therefore will lead to substantial performance benefits, as we will see in the experiments.

A side effect of batching queries is that the compiled code must now have an extra phase to extract the data for different visualizations from the combined results. However, since the ZQL compiler includes an ORDER BY clause, the overhead of this phase is minimal.

### 3.2.2 Intra-Task Optimization

In addition to combining SQL queries within a row, SQL queries may also be batched across ZQL rows as well. However, in general, it may not be possible to compose queries across rows into a single SQL query, since different rows may access completely different attributes. Instead, our optimization is to batch multiple SQL queries from different rows into a single request to the database, effectively pipelining the data retrieval.

However, tasks in ZQL frequently filter and limit the space of visualizations being looked at. Moreover, visualizations from subsequent rows may depend on the output values of previous tasks, so it is not possible to batch queries across tasks. Therefore, we batch into a single request all SQL queries for task-less rows leading up to a row with a task. In Table 20, this optimization would batch rows 1 and 2 together and rows 3 and 4 together.

### 3.2.3 Inter-Task Optimization

Our final optimization is more sophisticated than the previous methods. As mentioned previously, it is generally not possible to batch SQL queries across tasks. However, sometimes a visual component defined after a task may be independent of the task. More formally, if visual component \( V \) is defined in a row later than the row task \( T \) is defined in, and no column of \( V \) depends on the output of \( T \), then \( V \) is independent of \( T \). The advantage of independence is that now we can batch \( V \) into an earlier request because we do not need to wait for the results of \( T \). More concretely, in Table 19, we can batch the SQL queries for the first and second rows into a single request since the visual component in the second row is independent of the task in the first row.

To determine all cases for which this independence can be taken advantage of, the ZQL compiler builds a query tree for the ZQL queries it parses. All axis variables, name variables, and tasks of a ZQL query are nodes in its query tree. Name variables become the parents of the axis variables in its visual component. Tasks become the parents of the visualizations it operates over. Axis variables become the parents over the nodes which are used in its declaration, which may be either tasks nodes or other axis variable nodes. The query tree for Table 19 is given in Figure 1. Here, the children point to their parents, and the tasks in rows 1 and 2 are labeled \( t_1 \) and \( t_2 \) respectively. As we can clearly see from the tree, the visual component for \( f_2 \) is independent of \( t_1 \).

The visualization execution engine uses the query tree to determine which SQL queries to batch in which step. At a high level, all SQL queries for name variable nodes whose children have all been satisfied or completed can be batched into a single request.

More specifically, the execution engine starts out by coloring all leaf nodes in the query tree. Then, the engine repeats the following until the entire query tree has been colored:

- Batch the SQL queries for name variable nodes, who have all their children colored, into a single request and submit to the database.
Once a response is received for the request, color all name variables nodes whose SQL queries were just submitted.

Until no longer possible, color all axis variable and task nodes whose children are all colored. If a task is about to be colored, run the task first. This execution plan ensures the maximal amount of batching is done while still respecting dependencies. Parts of the execution plan may also be parallelized (such as simultaneously running two independent, computation-expensive tasks like representative sets), to improve overall throughput.

While this optimization may seem like overkill for the short ZQL examples presented in this paper, nothing prevents the user from combining many non-related visual queries into a single ZQL query with many output rows. Particularly for the case of exploratory visual analysis, the user may want to submit one ZQL query which contains many different tasks to explore interesting trends, representative sets, and outliers in an unknown dataset.

4. zensage SYSTEM ARCHITECTURE

In this section, we give an overview of the system architecture of zensage, including the front-end and back-end, also displayed in Figure 2. We will describe all the components in the figure in turn.

4.1 Front-End

The zensage front-end is designed as a lightweight web-based client application. It performs two major functions: First, it provides the analyst an intuitive graphical interface to compose ZQL queries for exploring trends and insights in data. Second, it takes the results of these queries from the back-end and encodes them into the most effective visualizations, taking into consideration data properties and perceptual principles.

The zensage system is intended for both novice and expert users, and thus the interface is designed for both usage styles. Figure 3 shows a screenshot of our current front-end implementation. The interface is divided into three major components: the building blocks panel on the left, the main panel in the center, and the recommendation panel on the right.

The main panel is used for query building and output visualization. The query builder component consists of two parts: the drawing box and the ZQL custom query builder. Users can drag and drop any attribute from from the building blocks panel on to the x-, y- or z- axis placeholders on the drawing box. They can either directly draw the trend line, box chart, or scatterplot they are looking for from scratch or drag and drop trends from other already rendered visualizations to modify it. In addition to the drawing, the user must also specify the intended insights and trends that she is looking for. Some common data exploration queries for identifying insights and trends, such as similarity search or representative search (see Section 2.1.8), are built in to the system and exposed on the building blocks panel. (So for these data exploration queries, the user does not even need to compose ZQL queries; simply clicking the right button will do.) Novice and expert users alike can use the drawing box to easily explore the data using common exploration tasks. The ZQL front-end internally translates the selections in the drawing into a ZQL query and submits it to the back-end for execution.

**Custom Query Builder.** If the user would like the full expressive power of ZQL, the user may choose to use the front-end’s custom query builder, which allows users to directly specify their query in the ZQL query format, depicted in Figure 4. The builder contains all the columns necessary for writing a ZQL query. Users write their query in a row wise manner. For each row, they can either drag and drop the attributes from the building blocks panel on to a cell in the ZQL table. If a row requires a user-drawn input, the user may select the row and use the drawing box to draw the trend she is looking for. Iterators and outputs of previous rows may also be reused by dragging and dropping them to the current cell. Users also have the option to run and see the output of a specific row row by clicking on the “Submit Active Row” button.

**Recommendation Panel.** In addition to returning results for every query that the user submits, zensage runs a host of parallel queries to find the most interesting trends for that subset of data the user is currently viewing and presents them in the right panel. These visualizations can be considered interesting visualization recommendations, as opposed to direct answers to ZQL queries. The exact method for how zensage defines an interesting trend is discussed further in the Recommendation Service in the next section.

**Result Visualizer.** The zensage front-end’s visualizer makes use of Vega-lite [41] grammar and the Vega visualization specification for mapping the query results to effective visualizations. The zensage front-end determines the default visual encodings, such as size and position, and uses the visualizer to map the output data according to the Vega-lite grammar.

4.2 Back-End

The zensage front-end interacts with the back-end via a REST
protocol. The back-end is implemented in Java and uses node.js for the web server. The back-end is comprised of the following components: the ZQL Engine, Execution Engine, and Recommendation Engine.

ZQL Engine. ZQL Engine is responsible for parsing, compiling, and optimizing given ZQL queries. See Section 3 for extensive discussion on the inner workings of this component.

Execution Engine. The Execution Engine takes the compiled output of the ZQL Engine and issues SQL queries to the database back-end while handing off post-processing on the resulting data to the task processor. We currently support two database back-ends: PostgreSQL and our own Roaring Bitmap Database.

PostgreSQL. Our current implementation uses PostgreSQL as a database back-end. However, since our ZQL Engine outputs standard SQL queries, any other relational database could be used.

Roaring Bitmap Database. Since the queries in our system are mostly ad-hoc and unpredictable, we cannot pre-compute and store query results in advance. Nor can we apply conventional indexes like B-tree as they result in high memory consumption and computation overhead for interactive use. To address these problems, we have developed a new storage model which uses Roaring Bitmaps [10] as its principal data storage format. By exploiting bit-level parallelism, Bitmaps can significantly accelerate queries involving arbitrary and complex selection predicates and aggregation. Further, Roaring Bitmap is an improvement over conventional Bitmaps and has 4-10X faster and higher compression rates. In our storage model, we follow a column oriented storage model. Columns which are not indexed are stored as an array on disk, and columns which are indexed have a Roaring Bitmap in memory for every distinct value of that column. As a default policy, we create Roaring Bitmaps for all categorical columns in each measure column un-indexed. Because of the fast bit-level parallelism and filtering capability, we find that the performance of the Roaring Bitmap Database had speedups of 30% – 50% for for queries with 10% selectivity.

Task Processor. As discussed in Section 3, the Task Processor performs the post-processing computation on the results of the SQL queries. For final output data, the processor also serializes the data into JSON before handing it back to the front-end.

Recommendation Service. In addition to query results, the zenvisage back-end also maintains a set of interesting recommended visualizations to help user further understand the trends in the data. Currently we define interesting trends as those which reflect the most diversity in the data related to the user query. We identify diverse trends using a set of heuristics. For instance, if the user is looking for a specific product that matches her drawn profit over year trends, the Recommendation Service also returns profit over trends for other products, which are diverse. In order to find the diverse trends, we run the $k$-means clustering algorithm to find a set of $k$ diverse clusters in the data. By default, zenvisage sets $k$ as 5, but the user has the options to change this value. While our current implementation is primitive, we plan to explore alternative schemes for recommendations in future work.

5. EXPERIMENTAL STUDY

We evaluate several properties of our zenvisage prototype. First, we evaluate the impact of the optimizations described in Section 3. Second, we evaluate the performance of the task processor for three common tasks often used in ZQL queries. Finally, we compare the two databases we use in the backend and present the performance numbers. The results of this experiment can be found in Appendix C.

5.1 Effect of Query Optimizations

In these experiments, we measure the improvement in performance by applying the three optimizations discussed in Section 3. We run the two ZQL queries mentioned in Section 3 (Tables 19 and 20) on the synthetic dataset and two additional queries (Tables 21 and 22) on the real airline dataset. The two additional queries respectively express a user need to find either specific patterns (find airports with increasing delay), or anomalous patterns (find airports where the discrepancies between two visualizations is maximized).

Figures 5 and 6 show the total runtime including the SQL execution time and the computation time and the number of SQL requests made for Tables 19 and 20 on the synthetic dataset, and Figures 7 and 8 show the total runtime and number of SQL requests for Tables 21 and 22 on the real dataset. NoOpT is the runtime using the naive ZQL compiler. Intra-Task times are not shown for Figures 19 and 21 because they provide no opportunities for the optimization.

Results. For these queries, we found that the SQL execution time dominated the overall runtime. Even when we increased the number of products in $P$ and airports in $\Omega$ and $\Omega_A$ (see the query), the post-processing time (<100ms) was negligible to the query execution time (>1s).

Therefore, we see that our optimizations which reduce the number of SQL requests made to the database provide significant speed-ups in overall runtime. Specifically, the intra-line optimization has the most effect because it batches the most number of SQL queries into one. In the case of Table 19, there were 20 different products in $P$, and the intra-line optimization combined these 20 separate SQL queries into a single one. The other optimizations also provide benefits, but because the number of rows in these examples are so few, the improvements are marginal. We expect these optimizations to be more noticeable in large ZQL queries with many unrelated tasks.

5.2 Performance of the Task Processors

Figure 5: Runtimes for the ZQL query in Table 19.

Figure 6: Runtimes for the ZQL query in Table 20.

We performed our experiments on a machine with 20 cores of Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz. Our entire codebase was developed in Java. For the first two experiments, we use PostgreSQL as the database.

Datasets. For our experiments, we use the following datasets:
- A synthetic sales dataset with 10M rows and with the following 8 attributes: product, size, weight, city, country, category, month, year, profit, and revenue.
- A real census-income dataset [2] consisting of 300,000 rows and 40 attributes.

Appendix C.
Table 21: A ZQL query which returns the departure delay over year and weather delay over year visualizations for airports in OA ((JFK,SFO...) where the average departure or weather delay has been increasing over the years.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Constraints</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>'year'</td>
<td>'DepDelay'</td>
<td>v1 &lt; OA</td>
<td>v2 argmax_{t&gt;0} f1(t)</td>
<td>v3 IN argmax_{t&gt;0} f1(t)</td>
</tr>
<tr>
<td>f2</td>
<td>'year'</td>
<td>'ArrDelay'</td>
<td>v1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f3</td>
<td>'Month'</td>
<td>'ArrDelay', 'WeatherDelay'</td>
<td>v2</td>
<td>Month=&quot;06&quot;</td>
<td>v2 argmax_{t&gt;0} D(f1,f2)</td>
</tr>
</tbody>
</table>

Table 22: A ZQL query which returns the arrival delay over year and weather delay over year visualizations for airports DA ((JFK,SFO...) where the average arrival delay between 'June' and 'Dec' differs the most.

![Figure 7: Runtimes for the ZQL query in Table 21.](image)

In this experiment, we want to evaluate how quickly the task processors return interesting visualizations relative to the total time taken. We identify limitations or bottlenecks in our current zenvisage prototype, if any. We evaluate the total elapsed time, the total computation time, and the SQL query execution time for three types of ZQL queries:

![Figure 8: Runtimes for the ZQL query in Table 22.](image)

**Similarity Search Query.** Here, we evaluate a query similar to the one in Table 11, where we want to find a Z attribute value for which a given visualization is most similar to one drawn by the user or selected up front. We use \( \ell_2 \) as a distance metric to evaluate similarity between two visualizations.

**Representative Search Query.** Here, we evaluate a query similar to the line corresponding to f1 in Table 15, where we want to find \( k \) Z attribute values for which the corresponding X vs. Y visualization is representative of the rest. For the function \( R \), we use \( k \)-means clustering on the set of visualizations using the same distance metric \( D \), and then return the cluster centroids as our representative visualizations.

**Outlier Search Query.** Here, we evaluate a query similar to the one in Table 15, where we want to find \( k \) Z attribute values for which the corresponding X vs. Y visualization is anomalous relative to the rest. For this task, we apply the representative search task, and then return the \( k \) visualizations for which the minimum distance \( D \) to the representative trends is maximized.

![Figure 9: Performance on real world data](image)

To evaluate the task processors in evaluating these ZQL queries, we measure the performance as a function of the number of groups, where groups is simply the product of the number of distinct values of the X attribute and the number of distinct values of the Z attribute—notice that these are precisely the number of distinct groups we would have in the GROUP BY query we would issue to the back-end database for each of these queries using summarization. Recall that each distinct value of the Z attribute leads to an additional visualization under consideration for that task. In case of the synthetic dataset, the number of groups are varied by changing the number of unique values of the given Z attribute, while the size of the dataset is kept fixed at 10M.

**Results.** Figure 10 depicts the results on all three metrics that we evaluate (in milliseconds). As seen in Figure 10(a), the overall processing time increases as we increase the number of groups. This is not surprising since there is an increase in both computation as well as query execution time. The query execution time for all the methods is similar as they all need to fetch the same amount of data from the table, but the slight increase as the number of groups increases results from the increase in the number of groups in GROUP BY of the associated SQL query. The computation time varies in proportion to the number of pairs of visualizations a given task processor compares, and therefore increases as we increase the number of groups (and therefore visualizations, as seen in Figure 10(b)). Furthermore, similarity search has the least cost of computation while outlier search has the maximum as it internally uses both representative and similarity methods. For a small number of groups, the query execution dominates the overall time. As the number of groups increases, the computation cost increases much faster than the query execution time especially in the case of representative and outlier search. Figure 9 shows the overall performance of the three tasks on real datasets, wherein the same relative behavior holds between the three task processors and therefore queries. In case of real datasets, since the number of groups is small(between 100 to 300 for both the datasets), the overall time is dominated by the query execution time(>95%).

6. RELATED WORK
Our work is related to work on visual analytics tools, anomaly discovery, and time series searching, visualization recommendation tools, as well as data mining query languages.

**Visual Analytics Tools:** A number of interactive data analytics tools have been developed, such as ShowMe, Polaris, Spotfire, and Tableau [36, 35, 28, 7]. The onus is on the user to specify the visualization to be generated, and repeat the process until the desired visualization is identified. Similar visualization specification tools have also been introduced by database community [14, 25, 26, 21]. There has been some work on browsing data cubes in OLAP—finding explanations, getting suggestions for cubes to visit, or identifying generalizations starting from a single cube, e.g., [33, 34]. While we may be able to reuse the metrics from that line of work, the same techniques will not directly apply to the identification of trends, patterns, and anomalies for interactive visualizations. There has been some work on sketch-based interfaces for visualization [17, 37, 18, 8]; however, none of this work addresses the deep scalability issues at play if these techniques are to be applied on large datasets. There is limited work on visual query interfaces [29, 19, 42, 5], but the goal there is to make it easier to pose SQL queries.

**Anomaly Discovery:** Anomaly detection has been a well-studied topic [12, 6, 32]. Our goal in that zervisage is expected to be interactive, especially on large datasets; most work in anomaly detection focuses on accuracy at the cost of latency and is typically a batch operation. In our case, since interactivity is of the essence, and requests can come at any time, the emphasis is on scalable-on-the-fly data processing aspects.

**Time Series Similarity and Indexing:** There is a lot of work on indexing and retrieval of time series data, for example, [24, 15, 11, 23, 9, 13, 22]: for the attributes that are queried frequently, we plan to reuse these techniques for similarity search. For other attributes, indexing and maintaining all trends is impossible, since the number of trends grows exponentially with the number of indexed attributes.

**Visualization Recommendation Tools:** There has been some recent work on building systems that recommend interesting visualizations to analysts. Voyager [41] recommends visualizations based on aesthetic properties of the target visualizations. Unlike zervisage, Voyager does not take into account a user query or a user need to make recommendations. SeeDB [31] recommends visualizations that best display the difference between two sets of data. Both SeeDB and Voyager can be seen to be special cases of zervisage, since zervisage can be viewed to be a user-driven visualization recommendation engine as well. Furthermore, the optimization techniques we outline here are a generalization of the techniques described in SeeDB; while the techniques described in SeeDB are special-cased to the query considered (a simple comparative query), here, our goal is to support and optimize all ZQL queries.

**Data Mining Query Languages:** There has been some limited work in the space of data mining query languages. This line of work, which was conducted in the early 90s, focuses either on association rule mining (languages such as DMQL [16] and MSQL [20]), or on storing and retrieving models on data (languages such as OLE DB [30]), as opposed to a visual specification of the kind of insight the user is looking for.

7. **CONCLUSION**

We propose zervisage, a tool for effortless specification and visualization of interesting patterns and insights from large datasets. The bulk of our paper focused on the formal specification of ZQL, our query language; we described how ZQL captures a range of interesting and useful scenarios, and is powerful enough to capture all of the typical data exploration queries users wish to issue. We then described three simple optimizations for translating ZQL queries into SQL queries, and demonstrated how these optimizations lead to several orders of performance improvement on real and synthetic datasets. Our ZQL prototype is fully functional and allows both novice and expert users to issue ZQL queries. While our work is a promising first step towards simplifying and improving visual data exploration, much more work remains to be done in further improving optimization, as well as adapting the tool to work even better for specific domains. Learning from user feedback and providing better recommendations are also interesting questions that are worth pursuing.

8. **REFERENCES**


[38] H. Wickham. ggplot: An implementation of the grammar of graphics. R package version 0.4.0, 2006.


APPENDIX

A. QUERY LANGUAGE FORMALIZATION: ADDITIONAL DETAILS

In this section, we present some additional details on our formalization that we did not cover in the main body of the paper.

A.1 Additional Details on X and Y Columns

In addition to using a single attribute for an X or Y column, ZQL also allows the use of the Polaris table algebra [35] in the X and Y columns to to arbitrarily compose multiple attributes into a single attribute; all three operators are supported: +, ×, /.

Table 23 shows an example of using the + operator to visualize both profits and sales on a single y-axis. Note that this is different from the example given in Table 4, which generates two visualizations, as opposed to a single visualization. An example using both table algebra and sets is given in Table 24, which uses the × operator to return the set of visualizations which measures the sales for the Cartesian product of ‘product’ and one of ‘county’, ‘state’, and ‘country’.

Table 23: A ZQL query for a visualization which depicts both profits and sales on the y-axis for products in the US.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>*lt</td>
<td>product</td>
<td>profit + sales</td>
<td>location='US'</td>
</tr>
</tbody>
</table>

Table 24: A ZQL query for the set of visualizations which measures the sales for one of ({‘product’, ‘county’}, {‘product’, ‘state’}, and {‘product’, ‘country’}).

A.2 Additional Details on the Constraints Column

Beyond the reasons described previously, there are a few additional ways the constraints column differs from the Z column. When ZQL returns the data which the zensight front-end uses to render actual visualizations with, the values for the Z columns are returned as part of the output. This is usually necessary as the Z column often iterates over a set of values, so the values for the Z column must be returned for proper identification as to which part of the returned data belongs to which slice. The results of the Constraints column, however, is not returned as an output of ZQL.

Secondly, because of the restrictions we have put on the Constraints column, we can allow the Constraints column to be much more complex than the Z columns. Earlier, we saw that we required extra Z columns to be able to deal with more than one attribute in the z-axis, but with the Constraints column, all of those...
constraints can be combined in one single column. The best way to think about the Constraints column is to imagine that the expression in the Constraints column will simply be added conjunctively to the WHERE clause of the translated SQL queries. In fact, the syntax for the Constraints column has been adjusted so that the expression can be taken from a ZQL table and directly added to SQL. For example, there is no need to quote attributes in the Constraints column. As a result, the set of possible values for the Constraints clause is roughly equal to the set of possible expressions for the WHERE clause in SQL. An example which makes extensive use of the Constraints column is given by Table 25.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>t1</em></td>
<td><em>time</em></td>
<td><em>sales</em></td>
<td>product='chair' AND zip LIKE '02\d{3}'</td>
</tr>
</tbody>
</table>

Table 25: A ZQL query for a sales over time visualization for chairs sold in the US in zip codes between 02000 and 02999.

### A.3 Additional Details on the Viz Column

Like other columns, the Viz column also supports sets of values and iteration over those sets. Table 26 refers to a ZQL query which iterates over various binning sizes for the x-axis. Table 27 iterates over the types of visualizations using t1. Note, unlike the Z column, we do not need to create a variable to iterate over the summarizes even though the visualization type changes, since the same summarization is valid for both types.

Although the bar chart and dot plot does not take in any parameters, other types of charts, such as the box plot, may take in additional parameters (e.g., to determine where the whisker should end); these additional parameters may be added as part of the sum-marization.

### A.4 Additional Details on the Process Column

Even with the primitives defined in the paper so far, sometimes the user would like to be able to write her own. ZQL supports user-defined functions that are executed by the zenvisage backend. User-defined functions may take in name and axis variables and perform whatever computation is necessary; zenvisage treats them as black boxes. However, users are encouraged to use the primitives defined by ZQL as they allow the zenvisage back-end more opportunities for optimization.

Furthermore, although visual components typically outnumber processes, there may occur cases in which the user would like to specify multiple processes in one line. To accomplish this, the user simply delimits each process with a comma and surrounds each declaration of variables with parentheses. Table 28 gives an example of this.

### B. QUERY EXECUTION: NAIVE TRANSLATION LISTING

Here, we list the pseudocode for the naive translation for query execution for reference.

```plaintext
# Row 1
v1_range = P,
f1 = make_array(1, (v1_range))
for v1 in [0 .. size(v1_range)]:
    f1[v1] = sql("SELECT year, SUM(sales)
        WHERE product='%s' and location='UK'
        GROUP BY year ORDER BY year"),
        v1_range[v1])
v2_range = []
for v1 in [0 .. size(v1_range)]:
    if T(f1[v1]) > 0:
        v2_range.append(v1_range[v1])

# Row 2
f2 = make_array(1, size(v1_range))
for v1 in [0 .. size(v1_range)]:
    f2[v1] = sql("SELECT year, SUM(sales)
        WHERE product='%s' and location='UK'
        GROUP BY year ORDER BY year"),
        v1_range[v1])
v3_range = []
for v1 in [0 .. size(v1_range)]:
    if T(f2[v1]) < 0:
        v3_range.append(v1_range[v1])

# Row 3
v4_range = union(v2_range, v3_range)
f3 = make_array(1, size(v4_range))
for v4 in [0 .. size(v4_range)]:
    f3[v4] = sql("SELECT year, SUM(sales)
        WHERE product='%s' and location='UK'
        GROUP BY year ORDER BY year"),
        v4_range[v4])
return f3
```

### C. EVALUATING BACK-END DATABASES

In this experiment, we compare the performance between our two database back-ends: Postgresql and the in-memory Roaring Bitmap Database. To make the comparison fair, we indexed both databases similarly: we indexed all categorical attributes in the bitmap database, while for Postgresql, we applied multi-column indexing on all categorical attributes.

We use a simple aggregate query of the following form to measure the performance of two database systems under two levels of query selectivity: 10% and 100%. This query is representative of a vast majority of SQL queries generated while processing ZQL queries:

```plaintext
SELECT X, F(Y), Z FROM table
WHERE P1=p1 AND P2=p2
GROUP BY X, Y, Z ORDER BY X, Y, Z
```

For each trial, we chose random categorical attributes for the values of X, Z. P1, P2, and a random measure attribute for Y. We also randomly determined the values for p1 and p2. In addition, we ran a version of the SQL query above without any predicates for the 100% selectivity trials.

We compare the performance of two databases with varying the number of groups and selectivity of the query.
Table 26: A ZQL query which returns the bar charts of overall sales for different weight classes, for varying weight class sizes.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
<th>Viz</th>
</tr>
</thead>
<tbody>
<tr>
<td>*f1</td>
<td>'weight'</td>
<td>'sales'</td>
<td>s1</td>
</tr>
</tbody>
</table>

Table 27: A ZQL query which returns the set of bar chart and dot plot of overall sales for different weight classes.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
<th>Viz</th>
</tr>
</thead>
<tbody>
<tr>
<td>*f1</td>
<td>'weight'</td>
<td>'sales'</td>
<td>s1</td>
</tr>
</tbody>
</table>

Table 28: A ZQL query which returns the sales over years visualizations for the product that looks most similar to the user-drawn input and most dissimilar to the user-drawn input.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>*f1</td>
<td>'year'</td>
<td>'sales'</td>
<td>v1 &lt;- 'product'.*</td>
<td>(v2 &lt;- argmax_v1[k=1]D(f1, f2)), (v3 &lt;- argmin_v1[k=1]D(f1, f2))</td>
</tr>
<tr>
<td>*f2</td>
<td>'year'</td>
<td>'sales'</td>
<td>v2</td>
<td></td>
</tr>
<tr>
<td>*f3</td>
<td>'year'</td>
<td>'sales'</td>
<td>v3</td>
<td></td>
</tr>
</tbody>
</table>

- **10% selectivity.** When the selectivity of the query is low, Roaring Bitmap’s fast bitwise operations helps in identifying the rows used in the aggregation. We see in Figure 11 (b) that this allows our Roaring Bitmap Database to perform 30-80% better than PostgreSQL, regardless of groups.

- **100% selectivity.** However, when the selectivity is 100%, the bitmap indexes buy us nothing as we need to look at every row in the dataase. Figure 11 (a) shows us that Roaring Bitmap Database still outperforms than PostgreSQL on a small number of groups. However, as the number of groups increases, the overhead from the hash-lookups required to select the groups becomes too great, and Roaring Bitmap Database ends up performing 30-50% worse than PostgreSQL.

We observe similar results on the real dataset as shown in Figure 11 (c).