Optimizing Databases for Efficient Interactive Visualization

Eugene Wu
sirrice@csail.mit.edu

Leilani Battle
leilani@csail.mit.edu

ABSTRACT

Existing end-to-end systems treat visualizers and DBMS's as separate, black-box components with minimal interaction. As a result, these components must be optimized separately, and the resulting systems are limited in their opportunities for overall performance improvements. We instead propose a declarative visualization language that completely encompasses how to process and visualize underlying data. Using this declarative language, we present anintegrated visualization system that utilizes structured information from the visualizer during query execution to provide significantly faster performance compared to alternative systems for visualization workloads.

1. INTRODUCTION

Current database management systems (DBMS's) are not optimized for visualization workloads. However, visualization is one of the most prevalent data analysis and presentation tasks for DBMS's. Critical information about how the front-end visualizer will utilize query results is hidden from the DBMS. For example, the axes and mark type (point, rectangle, etc.) of the output visualization are unknown to the DBMS. Thus the back-end loses opportunities to filter out data that is irrelevant to the end result, such as points that will result in over-plotting. Similarly, visualization systems are not optimized for use with DBMS's. The back-end DBMS holds valuable metadata that can help the visualizer infer output visualizations and estimate time and resource costs associated with computing visualizations. In addition, visualization systems often re-implement common database funcitonality to provide direct-manipulation interactions to the user. Treating the visualizer and DBMS as two separate black boxes leads to optimization in isolation, which ignores many opportunities for significant performance improvements.

Expressive visualizers provide rich visualization languages to perform more complicated visualization and processing tasks, but do not scale well due to lack of a performant back-end. ggplot2[1] is a powerful visualization package that provides a concise specification for visualizing data frames in R. D3[5] and Protovis[4] are Javascript libraries to map data points to aesthetic elements in interactive browser-based visualizations.

To make visualization scale, larger systems often choose a single visualization use case, such as drawing a particular kind of visualization or generating specific data representations, and heavily optimize the DBMS back-end accordingly. For example, the imMens[6] visualization sytem computes data cubes over the entire dataset to draw interactive aggregate visualizations fast. BlinkDB[2] builds stratified samples over datasets, and returns query results given error bounds and/or time bounds from the user. The ScalaR[3] visualization system takes the desired output size of the user's query and a reduction type as additional input, and produces reductions over the data on the fly. Tableau (originally Polaris[7]) builds interactive visualizations over large datasets quickly using a visualization language (VizQL) that maps database schemas directly to a core set of visualization templates.

The above systems are extremely efficient for their specific use cases, but are narrow in scope and require a significant amount of time and experience to extend. We instead observe that much of the functionality in the above systems can be automated in the DBMS. Each approach to efficiently processing the data can be distilled into a set of reusable optimization techniques available to all parts of the visualization pipeline. We propose designing a declarative language that specifies how to process and visualize the data, and maps directly into SQL. Using this declarative visualization language, we can build an extensible framework for incorperating visualization-specific predicates into query execution.

In this paper, we describe how various visualization and processing tasks can inform both query execution and how to build output visualizations. We present our visualization language for expressing how to visualize and process data. We describe the design of a fully-integrated visualization system utilizing our specification for database optimizations, and explain how query execution changes in the face of these updates. Lastly, we show how our system provides significant performance gains over existing visualization systems.

2. DECLARATIVE VISUALIZATION LANGUAGE

2.1 High-Level Components

This section describes the basic building blocks of our declarative visualization language.

Data Specifies the data set to be visualized.

Functions Simple data manipulation operations to be performed on a particular column of the data set. This includes computing the standard deviation of a column, or binning by a number columns and computing counts over the resulting bins.

Functions take tables as input and return tables as output. Here, tables refer to a set of columns. Functions can also be nested and combined to produce more complicated results. Visualization marks and aesthetic mappings can take as input the output of all functions.

Faceting Describes how to partition the dataset into multiple data sets to be visualized individually. Faceting is the first operation in our workflow and acts a global operator, as it affects all subsequent operations.

Traditional faceting allows the user to specify columns by which to separate the data into subsets for visualization. ggplot2 supports basic faceting. In addition to basic faceting, Tableau also supports faceting via range partitioning. Range partitioning allows the user to specify a set of columns and bin widths along these columns by which to partition the data into bins. A restricted faceting model allows current systems to make assumptions about the relationships between facets.

We relax the faceting model to allow for facets that are not directly related to each other. This gives users much more flexibility in how to specify facets. For example, we also allow users to specify a function by which to facet the data. Instead of specifying a single set of parameters, users can specify lists of parameters. The faceting function would be computed over the data set for all combinations of parameters specified in the lists, and each resulting data set visualized as a facet. An example would be specifying uniform sampling as the faceting function, and passing a list of sampling probabilities. In this example, a facet is produced for each sampling probability passed in the list.

Users can also provide a list of functions to facet on, along with their associated parameters. Users can utilize this technique to compare the visual effects of a variety functions. For example, how sampling affects the output when compared to other reduction techniques, such as aggregation.

Visualization Marks This specifies the kind of visual mark that will be used to denote data values in the visualization. Examples are points (scatterplots) and rectangles (bar charts/histograms).

Aesthetic Mappings These mappings dictate what data translates to aesthetic elements of the output visualization. Examples are mapping columns to axes or a color scheme. Function output can also be mapped to aesthetic elements.

Placement/Positioning This component controls the coordinate system used for mapping data to the pixel space, such as polar or cartesian coordinates. This part of the language also allows users to directly modify placement of visual marks the pixel space. An example of this would be adding jitter to points to avoid overplotting.

Placement/positioning directly affects the final layout of the output, and thus must also apply as a global operator over the visualization.

Interactivity This specifies interactive widgets to be applied to the visualization. Examples include pan-and-zoom functionality, as well as brushing and linking.

Current systems embed interactivity as part of each individual visualization type, or hardcode basic interactivity across all visualizations. We instead propose making interactivity a first-class citizen in our language specification, allowing interactive widgets to be added to each individual facet. This supports our flexible faceting scheme, where facets may not necessarily be directly related.

Users can then also specify the relationships between facets, which in turn dictates how they interact with each other.

Layering All components above, except faceting and placement/positioning, can be specified multiple times in separate *layers* of the visualization.

2.2 Using the Language

This section provides examples of our specification, and how they translate to visualizations.

2.3 Translating the Language into an Operator Model

3. POTENTIAL OPTIMIZATIONS

need content

4. CONCLUSIONS

need content

5. REFERENCES

[1] ggplot2. `ggplot2.org`.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. pages 29–42, New York, NY, USA, 2013. ACM.

[3] L. Battle, R. Chang, and M. Stonebraker. Dynamic reduction of query result sets for interactive visualization. 2013.

[4] M. Bostock and J. Heer. Protovis: A graphical toolkit for visualization. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2009.

[5] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[6] Z. Liu, B. Jiang, and J. Heer. immens: Real-time visual querying of big data. *Computer Graphics Forum (Proc. EuroVis)*, 32, 2013.

[7] C. Stolte and P. Hanrahan. Polaris: A system for query, analysis and visualization of multi-dimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8:52–65, 2002.