Explanatory Lineage

Eugene Wu
MIT CSAIL
eugenewu@mit.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

ABSTRACT

As data analytics becomes mainstream, and the complexity of the underlying data and computation grows, end-users will increasingly rely on visualizations to present simplified statistics that summarizes interesting properties in the data. It is even more important to provide tools that help analysts understand the underlying reasons when they see anomalous results. We envision adding a new *explanatory* dimension into visualization libraries and creation tools that automatically give end-users the capability to mine and understand the reasons for outliers and trends that they see in the visualizations. In this paper, we propose an initial problem formulation targeted towards business dashboard applications, and propose a set of modifications to existing declarative visualization libraries that enables this ability in existing visualizations with minimal changes by the developer.

1. INTRODUCTION

Visualizations are arguably the most common way end-users consume datasets and have long been a key tool for understanding data due to their ability to convey key statistics at a glance. Services such as Google Analytics provide pre-built visualizations customized to a specific type of dataset. Business intelligence visualization tools, such as Qlik View and Tableau let end-users interactively explore and build custom dashboards through a visualization front-end that sends SQL queries to an OLAP-style data store. Finally, tools such as Tenzing [3] and Hive [9] are designed to make "big data" analysis accessible to financial and business analysts through a SQL-like interface. The results are manually rendered using a separate visualization tool.

We dub these systems *What Visualization Systems* because they easily answer "what" type questions by transforming and computing aggregate statistics. For example, "what are the total hat sales this month", or "how many users came to the website and never returned (bounce rate)?". They are typically structured as a visualization front end that issues queries to a SQL backend. Users may even be able to slice and dice the statistics to examine how they vary across dimensions.

However, these systems are fundamentally *report generation tools* that display summary statistics. For example, they are able to report that a website's bounce rate is high, but are unable to explain *why* it is high. However this is exactly what an analyst is interested in – the user subsets that are exhibiting high bounce rates so that the website can be optimized for those users. Such subsets may be defined by complex predicates, e.g., they may be users with certain version of Firefox that crashes rendering the page, or visitors from a certain subnet or ISP that is experiencing outages. *What Visualization Systems* currently do not exist, leaving users perform these kinds of inferences themselves.

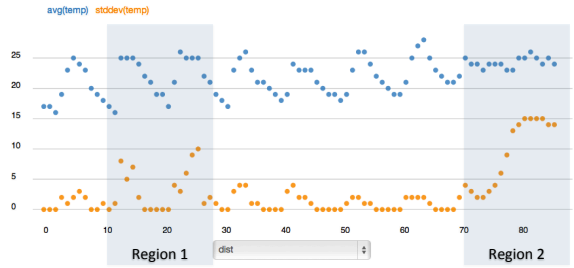

Figure 1: Mean and standard deviation of temperature readings from Intel sensor dataset

As a simple example, consider a deployment of sensors places throughout throughout a building. Each sensor records temperature, sound, humidity, light and other readings, and have associated metadata like a unique identifier and location coordinates. A typical dashboard may present the average and standard deviation temperature readings in 30 minute intervals (Figure 1 – this data was derived from the Intel sensor dataset [1]). An end-user may want to understand why the standard deviation fluctuates heavily (Region 1) and why the temperatures spikes (Region 2). It turns out that the former is due to sensors near windows that heat up under the sun around noon, while sensors running out of energy (indicated by low voltage) start producing erroneous readings in the latter. However, these facts are not obvious from the visualization and require manual inspection of the attributes of the readings that contribute to the outliers to determine what is going on. What is needed is a tool that can automate analyses such as determining that an outlier point is due to the location or voltage of the sensors that contributed to it.

Having a tool such as this would be useful in many other settings, including business intelligence and medical data analysis. As a real-word example (details anonymized), a doctor at a major hospital spent six months seeking to understand areas why the hospital was spending millions on a very small population of patients and if the hospital could reduce the costs without impacting mortality rates. He manually analyzed a number of dimensions (e.g., type of treatment, type of service) and isolated the costs to chemotherapy treatment. He later found that two doctors were responsible for

[1] `http://db.csail.mit.edu/labdata/labdata.html`

a large fraction of the costs because they over-prescribed unnecessary procedures. In this case, simply finding individually expensive treatments would be insufficient because the hospital is interested in descriptions of costs that are actionable.

In contrast to this manual, trial and error process, we envision a new type of data analysis tool, called a *Why Visualization System*, that interactively automates *why* analysis, in addition to supporting existing visualization interactions. Users use the system by highlighting arbitrary outlier elements in a visualization – such as a bar that is too high, or a set of anomalous points in a scatter plot – and specifying a small number of constraints (e.g., are the outliers too high or too low?). Our envisioned system then identifies the sets of input tuples that generated the suspicious outputs (e.g., the tuples in a `GROUP BY` group), and uses *sensitivity analysis* to further reduce the inputs to the subsets that most influence the output value in a way that makes it suspicious. Results are returned to the user as a set of predicates that describe each influential input subset and are automatically rendered in a plot.

It is important that such a system is tightly integrated into the visualization system. In most cases, the end-user that is consuming the visualization and identifying the anomalies is not the visualization creator, and may not have the access credentials nor knowledge to investigate the raw data. Integration enables any user in an organization to perform analyses that are otherwise restricted to a small group of analysts.

Building a *Why Visualization System* entails solving a number of key questions, including:

1. What are the useful classes of *why* queries?
2. What are efficient algorithms to execute these queries?
3. How can visualization systems and libraries support *why* analysis without burdening visualization developers?

We are currently implementing a prototype system called DBWipes to address the above questions. In Section 2, we describe the problem in the context of database lineage, and propose an extension called *explanatory lineage* to incorporate the notion of influential inputs. We introduce the problem statement and several promising algorithms for identifying the related properties of outliers in Section 3. Section 4 then illustrates an end-user walkthrough when using DBWipes and describes several methods to integrate with an existing declarative visualization engine or library in a way that minimizes the amount of programmer effort necessary to additionally support *why*-type lineage queries.

2. EXPLANATORY LINEAGE

This problem of *why visualizations* is closely related to the problem of *database lineage*, which involves tracking what set of input results contributed to an output result in a database query. This section briefly introduces database lineage and motivates the need for our extension called *explanatory lineage*, which augments lineage with the concept of input influence. We then describe two definitions of influence for different statistical aggregation operators.

2.1 Motivation

Lineage systems [10] record and query the history of data that flows through a workflow or database system. The runtime stores the parameters of every operator execution as well as the relationships between the operator's input and output data. A user can later specify a set of outputs and investigate a) the set of operators that were executed to generate the outputs and b) the inputs to any of the executed operators that were used to compute the output values. The former we call a coarse-grained lineage query whereas the latter is a fine-grained lineage query.

While these systems have traditionally focused on returning inputs at the dataset or file level (e.g., the output was generated by the the data files X and Y), relational and scientific database systems support lineage at the granularity of individual tuples or array elements. This support is ideal when operator fan-in (the number of inputs that generate an output) is low.

Unfortunately, visualization dashboards typically render aggregate statistics computed from large datasets. These high fanout aggregate functions are a key limitation of existing lineage systems, because they assume that every input tuple has equal weight on the result value. For example, consider a financial analyst that interacts with a visualization and discovers the output value of the query `SELECT sum(cost) FROM expenses` is too high and wants to understand which expenses are high, and why they are high. Existing lineage systems will return every tuple in the *expenses* relation because all of their values were used to compute the result. Although this is correct, it is not useful to the end user who is interested in the most influential expenses. Instead, because the largest expenses clearly influence the query result more than the cheaper expenses, it would be preferable to report a ranked list of expenses, and to perform some analysis of the common attributes of those expenses to help the analyst understand *why* those expenses are high.

This need motivated us to define *explanatory lineage*. In contrast to existing lineage queries, which return all inputs that computed user specified output values, explanatory lineage ranks the inputs by influence using sensitivity analysis and identifies clusters of influential inputs by their common attributes. The clusters are returned in the form of SQL-style predicates that succinctly describe the data in the clusters. The general problem is to rank the predicates that most influence the output, and return the top-k, and then cluster the top-k by their common properties. In some cases, doing this ranking is straightforward (e.g., for SUM queries), but can be hard for arbitrary user-defined aggregates. In addition the logic for clustering is is tricky, because it requires analyzing all attributes of the most influential tuples to cluster them together.

2.2 Notions of Influence

In order to approach this problem, we need to define a notion of influence. Our current investigation is in the context of *additive errors* – errors that can be resolved by deleting inputs – for common statistical functions. Alternative types of errors are *perturbation errors* that can be fixed by updating the inputs to a well defined value (e.g., set the costs in the highest quartile to the median). We have found that the type of explanatory queries that are meaningful to run, and consequently the specific notion of influence, depends on specific type of aggregate function. The rest of this section examples of queries for several common statistical functions.

Consider the *mean* statistical function. Each input contributes independently to the output, and the results are normalized such that the output value doesn't grow with the number of inputs. A lineage query may be "why is the mean temperature so high?", which defines the influence of a predicate as the expected change in the output if the inputs satisfying the predicate were deleted from the computation. Other statistical functions such as standard deviation and pearson's correlation share similar properties. call these queries heavy hitter queries, and the influence heavy hitter influence.

In contrast, for functions such as *count*, every input influences the output equally, and the influence of a set of inputs depends on the set size. Essentially, different input sets cannot be compared with one another, otherwise the most influential predicate would simply be TRUE. Such functions require a different notion of influence. One possible notion is to compare against user specified

baseline "normal" output values. For example, consider a chart that compares the number of homicides in each US state. A meaningful query may be "why is the homicide rate in California higher than the rates in Massachusettes or Hawaii, which both look normal?" . This notion of influence is defined by how much more deleting the inputs that satisfy the predicate change the "suspicious" output over the change of the "normal" output. The *sum* function has a similar notion. We call these queries comparison queries, and the influence comparison influence.

Extrema function such as *min* and *max* do not have meaningful definitions because the lineage is input is the same as the output. Thus, we do not consider such functions.

While we have described notions of influence for standard statistical functions, we are interested in understanding tractable notions for more complex aggregates such as SVMs, decision trees, and k-means. This is one of the key challenges in our research agenda.

3. IMPLEMENTATION

Section 2.2 introduced two explanatory lineage queries and notions of influence. In this section, we outline the general problem description and variants for the different notions of influence. We then sketch several algorithms that we are pursuing.

3.1 Problem Statement

We assume that a visualization has been rendered as the result of a `GROUP BY` SQL query over a table T with k attributes, $a_1, \cdots, a_k$. Additionally, assume that the query consists of a single statistical aggregate function, A. Suppose the user has selected an output result, o_{bad}, specified if the value is "too high" or "too low", and assigned weights to error functions as described below.

Let $D_{o_{bad}} \subseteq T$ be the set of input tuples that were used to compute o_{bad}.

Let P be the space of all distinct possible predicates over $D_{o_{bad}}$, where a predicate $p \in P$ is defined as the conjunction of range clauses over the continuous attributes and set containment clauses over the discrete attributes. Two predicates are distinct if they return different sets of inputs when applied to $D_{o_{bad}}$. Each column is present in at most one clause.

We would like to to find predicates that maximally reduce the value of o_{bad} (if the user flagged it as too high), or maximally increase it (if the user flagged it as too low.) In general, we assign a $score(p, D)$ to p over D, and find the top-k predicates $p*_1, \cdots, p*_k$ ordered by their $score$ value. This is difficult because it requires repeatedly evaluating the scoring function (and consequently, the aggregate function) over different subsets of D.

The scoring function depends on the aggregate and the type of user defined error. For example, explanatory lineage queries over *mean* and *standard deviation* type functions use:

$$score(p, D) = \begin{cases} o_{bad} - A(p(D)) & \text{output too high} \\ A(p(D)) - o_{bad} & \text{output too low} \end{cases}$$

In contrast, *count* type functions also depend on a baseline output result. For these functions, we assume that the user has selected a baseline output, o_{good}. These queries should return predicates that most influence $A(D_{o_{bad}})$ but not $A(D_{o_{good}})$. For example, the user may want to know the segment of users had the highest bounce rates that are different than the users from last week's web logs. In this case, she can define the baseline as data from last week.

We extend P to be the space of all distinct predicates over $D_{o_{bad}} \cup D_{o_{good}}$. We define a scoring function, $score(p, D, D')$, that also accepts the baseline dataset as an additional argument, D', and depends on $score(p, D)$ defined above. Finally, we introduce a penalty term, λ:

$$score(p, D, D') = \lambda * score(p, D) + (1 - \lambda) * score(p(D'), D')$$

3.2 Algorithms

Efficient algorithms for finding the optimal predicate ultimately depend on properties of the scoring function and the aggregate function. For example, whether the aggregate function is distributive, algebraic, or holistic. We now sketch several approaches that we are currently pursuing.

3.2.1 A Naive Algorithm

The naive approach is to exhaustively enumerate and evaluate all possible predicates. This approach first lists all distinct single-attribute clauses (clauses c_1 and c_2 are equal if and only if they result in identical results over D), then enumerates all non-empty conjections of clauses with different attributes. This algorithm is clearly exponential in the number of attributes. To bound the runtime, the user can specify a maximum number of clauses, n_{clause}, and limit the search to predicates will less than n_{clause} clauses.

3.2.2 Top Down Approach

We are currently investigating a top-down sampling based approach for aggregate functions (e.g., mean, standard deviation) whose inputs contribute independently to the output, and have well defined estimators. Since they have well defined estimators, we can also construct estimators of the influence.

The algorithm recursively bisects the attribute space to find partitions with low score values. Each partition is equilavent to a predicate. It evaluates a given partition by computing the mean and variance of the $score$ on a random sample of the tuples in the partition by using the estimators. If the expected error between the estimated score and the true value is below than a user-defined threshold, then the algorithm can stop evaluating the partition. Otherwise, the sample is used to decide the next dimension to split on. The algorithm can create partitions that are smaller than desired, so a final merging step is used to merge adjacent, influential, partitions. A benefit of this approach is that the scores of indiviual inputs can be memoized so that overlapping predicates can re-use previously computed scores.

A key insight is that the termination threshold does not need to be constant. Since the algorithm is searching for partitions with the minimum score value, partitions with high score can have a higher threshold. For example, if $score$ values range from 1 to 100, the user is interested in partitions where $score < 20$. Thus, if the mean $score$ is 90, then a variance of 10 may be acceptable, whereas the same variance is not acceptable when the mean $score$ is 10.

3.2.3 Bottom Up Approach

We are also considering several bottom-up approaches. In the first approach, we are attempting to reformulate the queries as a clustering problem, in order to re-use existing solutions. For example, consider a *COUNT(*)* query and an algorithm such as CLIQUE [1]. CLIQUE finds bounding boxes around maximally sized dense clusters and generates results in the form of predicates.

If we compute the union of the suspicious and baseline datasets, and assign them weights of 1 and -1, respectively, we can re-use the clustering algorithms by re-defining cluster size as the total weight of the cluster rather than the tuple count. The maximal cluster is naturally the predicate that most influences $D_{o_{bad}}$ over $D_{o_{good}}$.

An alternative approach can start with an initial set of predicates, and iteratively merge adjacent predicates while the resulting score

continues to improve. The key benefit of this approach is that it can relax the independence assumption used in the other approaches.

4. USER AND VISUALIZATION SUPPORT

Now that we've described the formal model of explanatory lineage, we turn to our envisioned interface for how a user would interact with a explanatory lineage system. In addition to illustrating the UI, our goal is to highlight places where the visualization developer must explicitly modify her code. Assuming that a visualization is written using a declarative framework that separates the mapping of data to visual elements from the actual rendering, we believe a majority of changes can be made to the visualization library without affecting the developer.

The rest of this section is described in the context of the javascript visualization library D3 [2], however the same ideas can be applied to any other declarative visualization framework.

D3 is a library that declaratively binds a set of data to a set web page (DOM) elements that are rendered by a web browser. Each DOM element is tied to a single data item, and additionally sets the DOM attributes based on functions over the data item. For example, consider the SQL query `SELECT day, count(*) AS bouncerate FROM web_log`. D3 can bind the result set to *cicle* DOM elements, which will create a *circle* element for each tuple in the result. The x, and y coordinates of the circles are defined as DOM attributes whose values depend on the result day and $bouncerate$ attributes, respectively.

```
d3('html').selectAll('circle')
.data(results)
  .enter().append('circle')
.attr('x', function(dataitem) {
 return dataitem.day;
})
.attr('y', function(dataitem) {
 return dataitem.bouncerate;
})
```

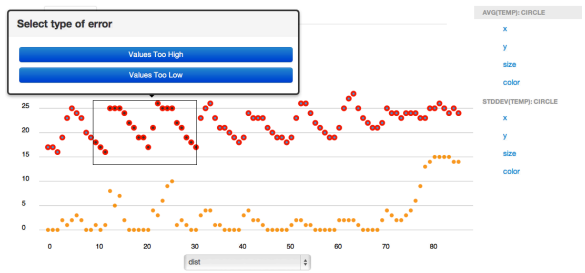

Figure 2: Figure 1 augmented with explanatory lineage.

DBWipes extends the library to track which DOM elements and attributes that are bound to data. At any time, the user can go into *why mode* to interactively select data that appears suspicious (Figure 2). In this mode, DBWipes displays a list of DOM element names (e.g., "Avg(temp): circle", "StdDev(temp): circle") and their associated attributes (e.g., size, x, y, color) . When a user hovers over an element name (e.g., "Avg(temp): circle") or attribute, the corresponding elements in the visualization are highlighted (e.g., red circles). Users select an individual or a group of rendered DOM elements (e.g., black bordered box) to specify the tuples that are supicious, and click on the attribute to specify the suspicious value. The library can automatically add the appropriate event handlers and highlight values. The developer adds functions to render a button to enter *why mode*, and list the element and attribute names.

DBWipes then creates a lineage UI that elicits additional information from the user, such as if the suspicious values are too high or low, normal output values, and error function weights. The developer only needs to add a container to contain the lineage UI.

The result predicates are rendered as a list. When the user clicks on a predicate, the original query is filtered by the predicate, and the subset of data is used to re-render the existing visualization. Thus, the user can toggle between the orginal query and the result predicate to visually inspect the predicate's influence.

An added benefit of explanatory lineage is that it indirectly performs dimensionality reduction. A key difficulty when visualizing high dimensional datasets is that visualizations are fundamentally 2-3 dimensional, and picking the optimal dimensions to plot is very difficult. Typical approaches use feature selection techniques [7] or PCA. On the other hand, the result predicates typically have a small number of clauses, which define the precise dimensions that distinguish the good and bad inputs. DBWipes can automatically create visualizations comparing the input points along the dimensions referenced in the predicates.

In each of the above steps, the minimum amount of work a visualization designer needs to perform is to define containers to render dialogs, buttons, and automatically generated plots.

5. CONCLUSION

The ultimate goal of data management is to make data easier to understand, whether that means making the interaction loop shorter (core database performance), seamlessly combining data from disparate sources (data integration), or making data more comprehensible (data visualization). Researchers have done a tremendous job helping bring data management to end-users through visual query languages and interfaces [8], intelligent cleaning tools [6, 5], and popularizing data-driven mashups [4]. Yet the core way that people consume and understand data, data visualizations, has stagnated to the "what" interfaces that simply summarize. We hope that explanatory lineage can add a new dimension of "why" to visualizations that will further encourage end-users to explore and better understand their data.

6. REFERENCES

[1] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, pages 94–105, New York, NY, USA, 1998. ACM.

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

[3] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing a sql implementation on the mapreduce framework. pages 1318–1327, 2011.

[4] H. Gonzalez, A. Halevy, C. S. Jensen, A. Langen, J. Madhavan, R. Shapley, and W. Shen. Google fusion tables: data management, integration and collaboration in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 175–180, New York, NY, USA, 2010. ACM.

[5] D. Huynh and S. Mazzocchi. Google refine. `http://code.google.com/p/google-refine/`.

[6] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, pages 3363–3372, 2011.

[7] Y. Saeys, I. n. Inza, and P. Larrañaga. A review of feature selection techniques in bioinformatics. *Bioinformatics*, 23(19):2507–2517, Oct. 2007.

[8] 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.

[9] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive- a warehousing solution over a map-reduce framework. In *VLDB*, pages 1626–1629, 2009.

[10] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. Technical Report 2004-40, 2004.