A Demonstration of DBWipes: Clean as You Query

Eugene Wu
MIT CSAIL
eugenewu@mit.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

ABSTRACT

As data analytics becomes mainstream, and the complexity of the underlying data and computation grows, it will be increasingly important to provide tools that help analysts understand the underlying reasons when they encounter errors in the result. While data provenance has been a large step in providing tools to help debug complex workflows, its current form has limited utility when debugging aggregation operators that compute a single output from a large collection of inputs. Traditional provenance will return the entire input collection, which has very low precision. In contrast, users are seeking *precise descriptions* of the inputs that caused the errors. We propose a *Ranked Provenance System*, which identifies subsets of inputs that influenced the output error, describes each subset with human readable predicates and orders them by contribution to the error. In this demonstration, we will present DBWipes, a novel data cleaning system that allows users to execute aggregate queries, and interactively detect, understand, and clean errors in the query results. Conference attendees will explore anomalies in campaign donations from the current US presidential election and in readings from a 54-node sensor deployment.

1. INTRODUCTION

As data analytics becomes mainstream, and the complexity of the underlying data and computation grows, it will be increasingly important to provide tools that help analysts understand the underlying reasons when they encounter strange query results. Data provenance [9, 7, 6] has been a large step in providing tools to help debug complex workflows. There are currently two broad classes of provenance: coarse-grained and fine-grained provenance. Coarse-grained provenance enables users to specify an output result and retrieve the graph of operators that were executed to generate the result; fine-grained provenance returns the inputs that were used to compute the output result.

Unfortunately, neither class of provenance is useful when debugging aggregate operators that ingest a large collection of inputs and produce a single output – a common operation performed during data analysis. Consider a sensor deployment that takes temperature, humidity and light readings. A user executes an aggregate query that computes the average temperature in 30 minute windows. When the user sees that the average temperature was 120 degrees, she will ask why the temperature was so high. If the user retrieves coarse-grained provenance, she will be presented with the query execution plan, which is uninformative because every input went through the same sequence of operators. On the other hand, if she retrieves fine-grained provenance, she will be presented with all of the sensor readings (easily several thousand) and be forced to manually inspect them. Neither approach helps the user precisely identify and understand the inputs that most likely caused the error.

Existing provenance systems exhibit two key limitations:

1) They do not provide a mechanism to rank inputs based on how much each input influences the output – all inputs are assigned equal importance. While it is possible to construct pre-defined ranking criteria for certain aggregate operators (e.g., for an average that is higher than expected, the inputs that bring the average down the most are the largest inputs), the user's notion of error is often different than the pre-defined criteria (e.g., the user may actually be concerned with a set of moderately high values that are clustered together). A provenance system needs a mechanism for users to easily specify their additional criteria.

2) Provenance systems return a set of tuples. While a collection of tuples can be used to train "blackbox" classifiers to identify similar tuples in the future, the user will ultimately want to understand the properties that describe the set of tuples, to understand where or why error arises. This necessitates a system that returns an "explanation" of the individual tuples.

With this in mind, we are developing a *Ranked Provenance System* that orders query inputs by how much each input influenced a set of output results based on a user-defined error metric. Our key insight is that users are often able and willing to provide additional information such as how the results are wrong and/or providing examples of suspicious inputs. Since our approach relies on user input to specify erroneous or outlier results, the system is tightly coupled with a visual interface that is designed to help users efficiently identify and specify these additional inputs.

In this demonstration, we will show how ranked provenance can be coupled with a visual querying interface to help users quickly go from noticing a suspicious query result to understanding the reasons for the result. We have created an end-to-end querying and data cleaning system called DBWipes that enables users to detect and remove data-related errors. Conference attendees will query two datasets – the 2012 Federal Election Commission (FEC) presidential contributions dataset[1], and the Intel sensor dataset[2]. The DBWipes dashboard can be used to visualize the query results, and includes interactive controls that allow attendees to identify and describe suspicious results. The dashboard presents a ranked

[1] `ftp://ftp.fec.gov/FEC/Presidential_Map/2012/P00000001/P00000001-ALL.zip`

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

list of predicates that compactly describe the set of suspicious input tuples. The system employs novel uses of subgroup discovery [4], and decision tree learning. Finally, the audience can clean the database by clicking on predicates to remove them from future queries. This interaction will highlight the value of tightly coupling a visual interface with a ranked provenance system.

2. SYSTEM OVERVIEW

In this section we provide an overview of the ranked provenance problem formulation and then report our current system architecture and implementation.

2.1 Problem Overview

For simplicity, consider a dataset D, and a query Q that contains a single aggregate operator, O, a group by operator, G, that generates g partitions, $G(D) = \{D_i \subseteq D | i \in [1, g]\}$. That is, D_i are the tuples in the i'th group. For example, in the temperature sensor query from the previous section, $O()$ is the $avg()$ operator; D is the table, $sensors$, that contains all sensor readings; G partitions D into sensor readings of 30 minute windows; and D_i is the set of tuples in the i'th window. Furthermore, let r_i be the result of $O(D_i)$ (e.g., average temperature for i'th sensor), and $R = Q(D) = \{r_i | i \in [1, g]\}$ be the set of all aggregate results.

When the user views the results, she will specify a subset, $S \subseteq R$, that are wrong (e.g., windows where the temperature was 120 degrees), and an error metric, $\epsilon(S)$, that is 0 when S is error-free and otherwise >0. For example, the following $\epsilon_{diff}(S)$ is defined as the maximum amount an element $s \in S$ exceeds a constant c. In the Intel sensor example, it computes the maximum difference between each user-specified outlier average temperature value in S and an expected indoor temperature, $temp_{expected}$. The user only needs to pick ϵ_{diff} from a collection a pre-defined error functions and specify $temp_{expected}$:

$$\epsilon_{diff}(S) = \max(0, \max_{s_i \in S}(s_i - c))$$

Let us first consider an idealistic optimization goal of a ranked provenance system before describing our current formulation.

Given D, O, S, and ϵ, we would like the ranked provenance system to produce a predicate, P, that is applied to D to produce the input tuples, $D*$, causing ϵ to be non-zero. That is, $D* = P(D)$ is the result of applying P to the dataset, such that $\epsilon(O(D - D*))$ is minimized. For example, P may be "(sensorid = 15 and time between 11am to 1pm)".

Unfortunately, the formulation above has several problems. First, it is not clear how to efficiently construct $D*$ given an arbitrary ϵ function. In the worst case, the system must evaluate ϵ on a number of datasets exponential in $|D|$. Second, ranking solely using ϵ does not address limitation 1 from the introduction – namely that the user has additional ranking criteria not captured by ϵ.

We address the above limitations by asking the user to provide an example set of inputs $D' \subseteq D$ that approximates $D*$. As an approximation, D' does not need to be complete nor fully accurate, it simply needs to highlight the type of inputs the user is interested in. D' will be used to bootstrap the process of finding $D*$ and evaluating candidate predicates P.

We now break the problem into three sub-problems:

1) Candidate $D*$ Enumeration: D' must first be cleaned and extended into a set of candidate $D*$ datasets. We first remove erroneous tuples by identifying a consistent subset of D', then extend it to enumerate a set of candidate datasets $D_1^c, \ldots, D_n^c$ that are self consistent and closer approximations of $D*$. ϵ is used to control the extension process.

2) Predicate Enumeration: DBWipes then generates a set of compact predicates, $P_1^i, \ldots, P_m^i$, that describe each candidate dataset, D_i^c.

3) Predicate Ranking: The third problem is to rank the predicates based on the size of the predicate., how well it describes the dataset D', and how much it minimizes ϵ.

2.2 System Architecture

DBWipes combines a visualization frontend and a provenance backend. The frontend provides a query and visualization interface to gather provenance query information and maximize the accuracy of the user selected D'. The backend then computes ranked provenance results and sends a ranked list of predicates for the frontend to display. Figure1 displays the tight interactive loop that ties the frontend (top) together with the backend (bottom) – the frontend diagram shows a flow chart of user actions, while the backend diagram depicts major system components and control flow between them.

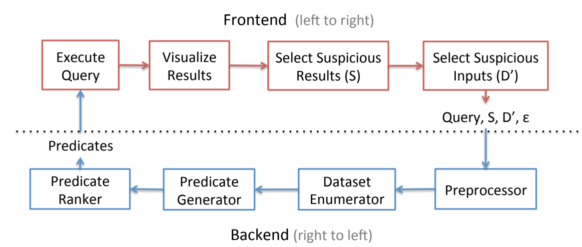

Figure 1: DBWipes Architecture Diagram. The top section shows the sequence of user actions to describe a provenance query, the bottom section describes the major system components and control flow.

2.2.1 Frontend Design

Figure 2 highlights the four main frontend components. The frontend interface is designed to simplify the process of specifying S, ϵ, and D'. The major components are:

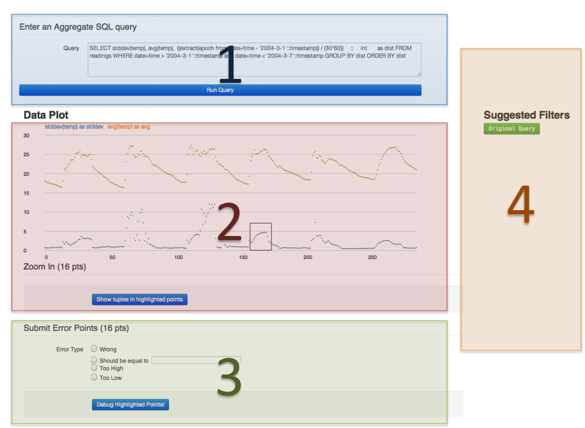

Figure 2: DBWipes Interface

1) Query Input Form: Users submit aggregate SQL queries using the web form (Figure 3). As users clean the database and select predicates to (see **Ranked Predicates**), the query is automatically updated.

2) Visualization and S, D Selection: Query results are automatically rendered as a scatterplot. When the query contains a single

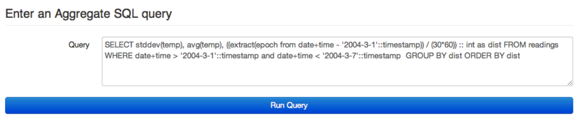

Figure 3: Query Input Form

group-by attribute, the group keys are plotted an the x-axis and the aggregate values on the y-axis. If the query contains a multi-attribute group-by, the user can pick two group-by attributes to plot against each other. We are currently investigating additional methods to visualize multi-attribute group-by results, such as plotting the two largest principal components against each other.

Figure 4 shows how users view and highlight suspicious query resultsand then zoom in to view the individual tuple values. The left graph plots the average and standard deviation of temperature (y-axis) in 30 minute windows (x-axis) from the Intel sensor dataset. The user specifies S by highlighting suspiciously high standard deviation values and then clicks "zoom in". She is then presented with the right half, which plots the temperature value of all the tuples in the highlighted groups. She finally specifies D' by highlighting outlier points with temperature values above 100 degrees.

3) Error Metric Form: The frontend dynamically offers the user a choice of predefined metric functions depending on the query results that are highlighted by the user. For example, if the user highlighted results generated by the avg function, DBWipes would generate such as "value is too high", and "should be equal to ____" (Figure 5).

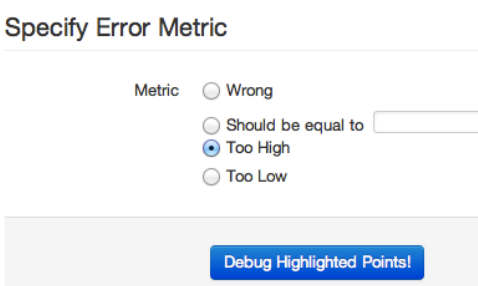

Figure 5: Error Forms

4) Ranked Predicates: After selecting tuples in D', and selecting an error metric, the system computes a ranked set of predicates (Figure 6) that compactly describe the properties of the selected points (we describe how this selection is done in the next section). These predicates a shown in on the right side of the dashboard. The user can click on a hypotheses to see the result of the original query on a version of the database that does not contain tuples satisfying the hypothesis. The visualization and query (Figure 3) automatically update so that the user can immediately explore new suspicious points after clicking the appropriate predicate.

2.2.2 *Backend Components*

The backend takes Q, ϵ, S and D' as input, and outputs a ranked list of predicates. First, the *Preprocessor* computes F, the set of input tuples that generated S; $F - D'$ is an approximate set of error-free input tuples. It then uses leave-one-out analysis to rank each tuple in F by how much it influences ϵ. The rest of the components correspond to the sub-problems described in Section2.1. The *Dataset Enumerator* cleans and extends D' to generate a set of candidate D^*s – $\{D^c_1, \ldots, D^c_n\}$. The *Predicate Enumerator* constructs multiple decision trees for each candidate D^c_i. The decisions trees are ranked by the *Predicate Ranker*, and finally converted into predicates to return to the frontend.

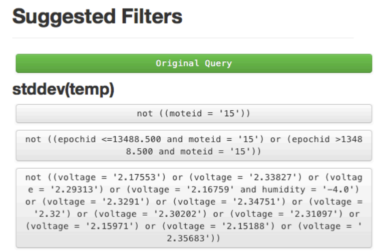

Figure 6: A ranked list of predicates that can be added to the Intel sensor query

The *Dataset Enumerator* cleans D' by identifying a self consistent subset. We are currently experimenting with clustering (e.g., K-means) and classification based techniques that train classifiers on D' and remove elements that are not consistent with the classifier. We then extend the cleaned D' using subgroup discovery algorithms to find groups of inputs that highly influence ϵ. Subgroup discovery [4] is a variant of decision tree classifiers that find descriptions of large subgroups that have the same class value in a dataset. Consider a database of patient health information (e.g., weight, age, and smoking attributes) and whether or not they are at high-risk for cancer (the class variable). Subgroup discovery will find that smokers over the age of 65, and heavy weight people are two significant subgroups of the high-risk patients. In our case, we want to find subgroups that contain both the cleaned D' and subsets of F that most strongly influence ϵ. The output of the component is a set of n candidate datasets $D^c_1, \ldots, D^c_n$

The *Predicate Enumerator* then builds a decision tree on each candidate dataset D^c_i by labeling D^c_i as the positive class and $F - D^c_i$ as negative. We currently use m standard splitting and pruning strategies (e.g., gini, gain ratio) to construct several trees, $T^i_1, \ldots, T^i_m$ from each dataset, D^c_i.

Finally, the *Predicate Ranker* computes a score for each tree, T^i_j, that increases with improvement in the error metric, and the accuracy of the tree at differentiating D^c_i from $F - D^c_i$, and decreases by the complexity (number of terms in) the predicate.

DBWipes currently supports the common PostgreSQL aggregates (e.g., avg, sum, min, max, and stddev) and several error functions (e.g., "higher/lower/not equal to expected value").

3. DEMONSTRATION DESCRIPTION

As described above, DBWipes combines a a web interface with a ranked provenance engine that lets the audience analyze imported data by running SQL aggregation queries.

3.1 Datasets

In the demonstration, users explore the underlying reasons for anomalies in two datasets: the 2012 FEC presidential campaign donations dataset and the Intel sensor dataset. We have found several interesting anomalies, and will provide a inital queries that users can use. We additionally encourage attendees to write their own ad-hoc queries and explore the datasets.

The FEC dataset contains several tables that contain donation and expenditure data in the 2012 presidential election. Each table contains information such as the presidential candidate (e.g., Obama, McCain), the donor's city, state, and occupation informa-

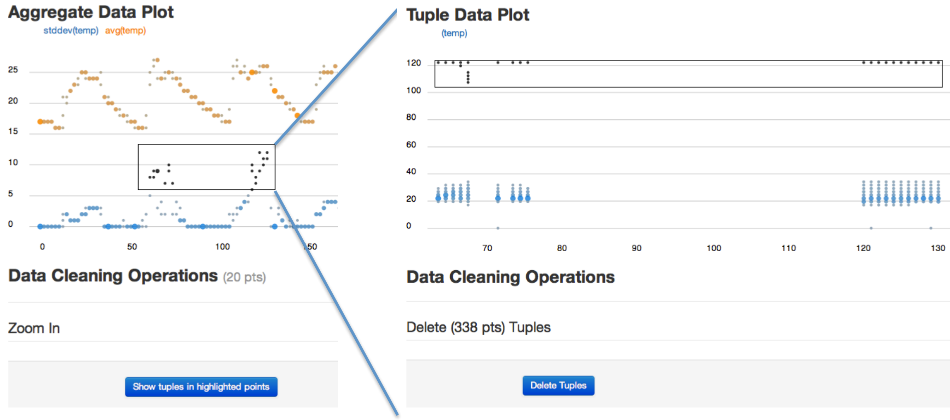

Figure 4: The user highlights several suspicious results in the left visualization and zooms in to view the raw tuple values.

tion, the donation amount and date, and a memo field that describes the type of contribution in more detail.

The Intel sensor dataset contains 2.3 million sensor readings collected from 54 sensors across one month. The sensors gather temperature, light, humidity, and voltage data about twice per minute.

3.2 Demo Walkthrough

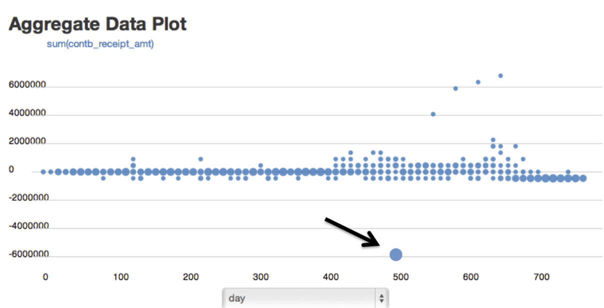

Figure 7: McCain's total recieved donations per day since 11/14/2006

We now describe a walkthrough of how a data journalist uses DBWipes – this is similar to how conference attendees will interact with the demo. Imagine a data journalist that is analyzing the 2008 presidential election, looking for the next big story. She downloads and imports a campaign contributions dataset from the FEC. After executing a few exploratory queries, she generates Figure 7, which plots the total amount of donations John McCain recieved per day. While each contribution spike correlates with a major campaign event, she finds a strange negative spike in McCain's contributions around day 500 into the campaign. At this point, instead of writing additional queries to manually view each donation around that day and attempt to construct an explanation, she highlights the suspicious data point and clicks "zoom". The plot is updated to show all of the individual donations on the days around day 500, and she sees several negative donations. She highlights them, picks the error metric "values are too low" and clicks "debug!". The system then returns several predicates, one of which includes several references to the memo attribute containing the string "REATTRIBUTION TO SPOUSE". When she clicks the predicate, a significant fraction of the negative value disappears. She later researches the term and finds that it is a technique to hide donations from high profile individuals (e.g., CEOs) to controversial/unpopular candidates by attributing the donation to the individual's spouse.

We will provide several queries that conference attendees can run to bootstrap their investigations on the FEC and Intel datasets.

4. RELATED WORK

The goals in DBWipes are most similar to that of Causal Relationships and View Conditioned Causality [5]. Meliou et al define a notion of provenance causality that enables them to rank inputs based on a relevance metric. An input X is *VC Causal* if there exist any additional inputs, Γ, such that altering $\{X\} \cup \Gamma$ will fix the outputs (in our case, minimize ϵ). X's *relevance* is then defined as $\frac{1}{1+min_{\Gamma}\Gamma}$ – this metric is an elegant way to rank individual inputs. Using this framework, they efficiently answer causality questions of boolean expressions using a SAT solver. In contrast, DBWipes answers ranked provenance queries over complex aggregate operators (instead of boolean expressions) and supports additional ranking metrics. Kanagal et al. [2] have very similar work that uses sensitivity analysis to rank probabilistic provenance.

DBWipes is also similar to interactive data cleaning systems [3, 8, 1], which support interactive data cleaning and transformations for unstructured data during data import, and provenance systems such as Trio [9], which compute the full set of inputs that generated an output result.

5. REFERENCES

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

[2] B. Kanagal, J. Li, and A. Deshpande. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *SIGMOD*, pages 841–852, 2011.

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

[4] N. Lavra, B. Kavek, P. Flach, and L. Todorovski. Subgroup discovery with cn2-sd. In *JMLR*, volume 5, pages 153–188, Feb. 2004.

[5] A. Meliou, W. Gatterbauer, J. Y. Halpern, C. Koch, K. F. Moore, and D. Suciu. Causality in databases. In *IEEE Data Eng. Bull.*, volume 33, pages 59–67, 2010.

[6] K.-K. Muniswamy-Reddy, J. Barillariy, U. Braun, D. A. Holland, D. Maclean, M. Seltzer, and S. D. Holland. Layering in provenance-aware storage systems. Technical Report 04-08, 2008.

[7] T. Oinn, M. Greenwood, M. Addis, N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: lessons in creating a workflow environment for the life sciences. In *Concurrency and Computation: Practice and Experience*, volume 18, pages 1067–1100, Aug. 2006.

[8] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.

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