Explaining Database Anomalies Via Queries

Xiaolan Wang
UMass
email

Alexandra Meliou
UMass
Address

Eugene Wu
Columbia
Address

ABSTRACT

Abstract

1. INTRODUCTION

Sketch of arguments

While exploring data, its natural to come across surprising or unexpected data. For example, visual data analysis explores the current state of the database and users may be surprised by outliers in a visualization. Similarly, enterprise customers (e.g., billing) may find outliers in their monthly bills and be surprised by the amount they are asked to pay.

When presented with these surprises, users want to better understand the reasons behind the anomalies. A recent wave of research focuses on deriving predicate-based explanations for outliers for statistical aggregation queries. For example, if the user wants to understand why the total sales in the past few months have gone up, these systems can general explanations such as "most related to customers in California between the ages of 12 to 18." However, these approaches simply generate predicates that describe *current state* of the database, and do not resolve *how* the anomalous data came to be.

Specifically, the user may also be interested to understand which past database modification was responsible for these explanations. Describe why this makes sense to want. In this form of the problem, we are interested in historical database queries whose modifications, when propogated to the current database state,

In this paper, we approach anomaly explanation from the persepctive of the query log and seek to both *identify* historical database modification queries that most likely caused user complaints in the current state of the database, and suggest replacement queries that will resolve these complaints. We call this problem the *Query-based Complaint-Satisfaction Problem*.

Given a database query log and a set of *complaints* (e.g., tuple 1's attribute B should be 20% lower) about records in the current state of the database, we seek to identify the subset of queries in the log that, by modifying their parameters and propogating the new effects of the queries, will best resolve the complaints.

One way to solve this problem is to try modifying the most recent query until it fixes the complaints. If not, then try the second most recent query. The problem with this approach is the number of possible modifications is unbounded.

Our contributions include

1. Developing and formalizing the problem of Query-oriented explanation in contrast to data-oriented explanation
2. Prove that the general problem is impossible.
3. Designing alogirthems to solve the problem for complete complaint sets
4. Extending the algorithms to support incomplete complaint sets
5. Extending to support multiple queries

2. USE CASE

3. ARCHITECTURE

4. USE CASE

4.1 Telco

Call centers get customer complaints about billing all the time. For example, there was a change in the tax law, but the transactions that updated the taxes didn't change, so they would get lots of customer complaints. Each customer agent usually fixes each problem individually, however, it's a system problem to all of Kansas that usually doesn't get caught until late in the game. In this case it's a faulty update to a numerical value whose effects compound over time. There is external domain knowledge to isolate the culprits to tax-related transactions, and sensitivity analysis can be useful.

4.2 Insurance

A colleague recently switched to medicare but because of a problem with the transactions to switch him from his previous to new insurance, it's caused him 6 months of headaches. 3 months ago he learned that a bunch of other people have been complaining about the same issues regarding the switch. In this case it's a categorical change (from one insurance to another) that has more obvious impact.

5. PROBLEM DESCRIPTION

First, some notation used in the rest of the paper.

Our goal is to take the user's complaints about the current state of the database, and identify transformations over the database query log that, when applied to the database, "fixes" these complaints. Although the general formulation of the problem that we will introduce is intractable, we will reduce the scope of the problem to a useful and tractable version. To begin, we will introduce the notation that is used in the rest of the paper.

Def 1 (Database, State, Query): Let a database be defined as the result of applying a sequence of queries in the query log $Q_{seq} = \{Q_1, ..., Q_n\}$ to an initial database D_0. Applying the query log to the initial database results in n intermediate database states $\{D_i =$

$Q_i(D_{i-1}) | i \in [1, n]\}$. where $D_n = Q_{seq}(D_0) = Q_n(\ldots Q_1(D_0))$ is the current state of the database. We assume that a subset of the query log has been replaced with *corrupted queries* such that D_n and Q_{seq} differ from the true database state D_n^* and query log Q_{seq}^*.

In our current problem, we only deal with non aggregation and non-join queries. In addition, for ease of exposition, we assume that the database contains a single table T (T^* in the true database) containing m numerical attributes $a_1, \ldots, a_m$, where the primary key is a_1. In Section **??**, we will describe how our techniques extend to multi-table databases. Define the scope of queries supported

Def 2 (Complaints and Complaint Sets): The tuple-wise difference between two databases D_1 and D_2 can be viewed as a patch P_{D_1,D_2} that contains *difference pairs* $(t_i \in T, t_i^* \in T^*)$. Similar to source code patches, patches can be applied to a database $P(D_n) = D_n^*$. Each pair in a patch describes one of three error types that can be present in D_n:

- *ADD*: t_i^* should be added to the database (null, t_i^*)
- *DELETE*: t_i should be removed from the database (t_i, null)
- *WRONG*: t_i should be modified into t_i^*

The user provides a *complaint set* C that specify the perceived incorrect tuples in D_n. C is simply a patch. We define a complete complaint set when $C = P_{D_n, D_n^*}$. In contrast, an *incomplete complaint set* may contain complaints that are false negatives (insertions that should be in C but are not), false positives (deletions that are not present), and errors (proposed value of a t' is incorrect).

The accuracy of an incomplete complaint set can be measured by the ratio $acc_C = \frac{|C \cap \Delta|}{|\Delta|}$.

$$\begin{tikzcd}
 & \underbrace{\{Q_1 \cdots Q_n\}} & \\
 & Q_{seq} \arrow[r] \arrow[dd, "T"] & D_n \arrow[dd, "C"] \\
D_0 \arrow[ur] \arrow[dr] & & \\
 & Q_{seq}^* \arrow[r] & D_n^*
\end{tikzcd}$$

Figure 1: Graphical depiction of the general XXX problem.

5.1 Naive Formulation

The most general version of the problem (depicted in Figure 1) is to find a sequence of transformations T that insert, delete, and/or modify queries in Q_{seq} such that the resulting sequence, $Q'_{seq}{}^{=T(Q_{seq})}$, resolves the user's complaint set.

However this problem is ill-defined because there exist an unbounded set of transformations that can resolve the user's complaint set. A naive solution is to append to the query log a statement that deletes all the records in the database, followed by a query that insert all of the correct records. Unfortunately this naive solution does not help explain the complaints in any way!

5.2 Constraints

For this reason, we constrain the set of possible transformations $\mathcal{T}$ to the following:

- delete query
- modify insert statement constants
- modify constants in WHERE clause

Our transformations don't include adding new queries, synthesizing arbitrary queries, or modifying the number of clauses in a WHERE condition. We apply these restrictions because we believe it is more likely for the user to mis-type a constant value as opposed to having an error in the query structure.

Futhermore we define a distance metric between two query logs in order to evaluate the qulatiy of a transformation. define $\mathcal{T}$ here.

5.3 Problem Statements

In this paper, we present three variants of this problem.

PROBLEM 1 (PROB-COMPLETE). *Given $C = P_{D_n, D_n^*}$, Q_{seq}, and the sequence of database states $D_0, \ldots, D_n$, identify a sequence of transformations T such that:*

- $T(Q_{seq})(D_0) = C(D_n)$
- $|T| = 1$
- *T metric is minimized*

This variation of the problme relaxes the constraint that the complaint set must be complete, and allows for both false positives as well as false negatives. The goal is the same, however the constraints are relaxed:

PROBLEM 2 (PROB-INCOMPLETE). *Given C where $acc_C < 1$, Q_{seq}, and the sequence of database states $D_0, \ldots, D_n$, identify a sequence of transformations T such that:*

- $T(Q_{seq})(D_0) = D_n^*$
- *T metric is minimized.*
- $|T| = 1$

Finally, we extend the problem to allow transformations with one or more operations.

PROBLEM 3 (PROB-MULTIQ). *Given C where $acc_C < 1$, Q_{seq}, and the sequence of database states $D_0, \ldots, D_n$, identify a sequence of transformations T such that:*

- $T(Q_{seq})(D_0) = D_n^*$
- *T metric is minimized.*

5.4 A Naive Approach

describe naive approach and why it fails

6. BASIC SOLUTION

In this section we will introduce a simple approach to solve Prob-Complete.

Given $D'_n{}^{=C(D_n)}$, able to roll back to fixed intermediate state D'_i.

6.1 Single-Query Case

In this section, we walk through the core techniques in the case of a query sequence containing a single query. Walk through solution for each error type for each query type.

Inserts and deletes are straight forward, however modifying WHERE clause is harder because why?. To deal with this challenge, we tried three algorithmic approaches.

Constraint-solver (CPLEX) Describe how to encode problem as CPLEX

Bounding Box

Decision Tree Describe how to encode as decision tree, what algorithm, and how to interpret learned tree.

Also, how to encode constraint that structure needs to be the same.

6.1.1 *Comparing these approaches*

To understand the tradeoffs between these approachse, we ran a simple experiment for a single query.

Here are the results.

CPLEX takes the longest, however produces exact results. However, it fails to produce any results if there are conflicting complaints. Decision tree is fastest, however the quality of the results are poor. Bounding box is similar to CPLEX in quality, however it generates results even if there are conflicting complaints.

7. PROB-COMPLETE

Given the above, the algorithm for solving the complete complaint set problem is straightforward (Algorithm **??**). We can simply try each query and return the one for which the best solution is found.

8. INCOMPLETE COMPLAINTS

Incomplete complaints are a challenge because the exesting algorithms fail. We ran a simple experiment where the complaint set contains only M% of the true complaint set, and has N randomly generated erroneous complaints. Figure **??** shows that none of the algorithms work.

Describe assumptions for why a cleaning-based approach makes sense.

We use a cleaning-based method to deal with false positives, and some magic to deal with false negatives.

8.1 False Positives

Describe density, bi-partite graph, consistency, tuples affected scores. Which ones work well, which ones don't.

8.2 False Negatives

9. MULTI-QUERY RESOLUTION

We use a dynamic programming-based algorithm to support multiple queries.

10. IMPLEMENTATION

THESYSTEM is implemented as a Java-based middleware in front of PostgreSQL.

Talk about tricks to encode problem intto CPLEX/decision tree?

11. EXPERIMENTS

11.1 Setup

11.1.1 *Dataset and Workload*

11.1.2 *Generating Complaints and Complaint Sets*

11.2 Complete

11.3 False Positive

11.4 False Negative

11.5 Multi-Query

12. DISCUSSION

Systems soluttions would be interesting

13. RELATED WORK

Scorpion explanation uses data to synthesize predicates that explain.

Why not work.

View construction and query by example.