Explaining Database Anomalies Via Queries

Xiaolan Wang
University of Massachusetts
xlwang@cs.umass.edu

Alexandra Meliou
University of Massachusetts
ameli@cs.umass.edu

Eugene Wu
Columbia University
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, The goal is to provide diagnostic tools that can peer into past transactions.

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

Sometimes, data starts clean but gets tainted due to erroneous updates. Errors in queries need to be handled differently than errors in data, since they pose their own unique challenges:

Systemic errors. The errors created by bad queries are systemic. The link between the resulting errors is the query that created them; cleaning techniques should leverage this connection to diagnose and fix the problem.

Large impact. Erroneous queries cause errors in a large scale. This means that the potential impact of the error is high, which has been manifested in several real-world cases.

Obscurity. Detection of the resulting errors often leads to partial fixes that further complicate the eventual diagnosis and resolution of the problem. For example, a transaction implementing a change in the state tax law updated tax rates using the wrong rate, affecting a large number of consumers. This causes a large number of complaints to a call center, but each customer agent usually fixes each problem individually, which ends up obscuring the source of the problem.

Feasibility. Query errors can easily remain undetected for a significant amount of time, which, in turn, makes correcting the mistake a huge undertaking. Essentially, a very old transaction would need to be rolled back, which poses obvious hurdles.

The goal of this paper is to design effective query diagnosis techniques and identify possible fixes for query errors. We model the problem assuming a log of update workloads over a database, and a set of complaints that identify errors in the final database state. We organize our contributions as follows:

Diagnosis with complete information: We will first tackle the problem assuming that the complaint set is complete, i.e., all errors in the final data instance have been labeled. In Section **??**, we will present algorithms that identify the query responsible for all errors in the complaint set.

Diagnosis with incomplete information: We will extend our algorithms to account for incomplete information, which is a more realistic scenario: some complaints have identified errors in the final data instance, but the set is not complete (Section **??**).

Deriving fixes: We will derive potential fixes to correct the errors.

Evaluation: ...

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 $acc_C = \frac{|C \cap \Delta|}{|\Delta|}$.

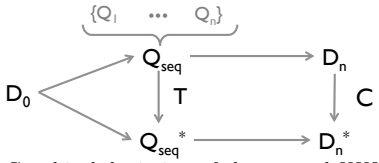

Figure 1: Graphical depiction of the general XXX problem.

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

- roll back complaints to penultimate state using algebraic expressions
- perturb each expression in query until the query result matches correct state
- if an expression cannot be found, iterate

Not clear how to roll back complaints

Ways to perturb query expressions is unbounded

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 Metrics

Metrics we use to measure how good a fix is.

Used in the optimization problem.

6.2 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, by simply limiting the possible attribute choices at each step in the learning algorithm.

6.2.1 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.

7.1 Provenance-based Log Filtering

Use provenance information (tuple-level or query level) to filter out queries that do not affect the complaint tuples at all.

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.

NP-completeness of density-based approach.

8.2 False Negatives

9. MULTI-QUERY RESOLUTION

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

10. ROLLING BACK THE LOG

background about why this is a hard problem

10.1 A Solver-based approach

Table $R(A, B)$ contains tuple $t_0 = (a_0, b_0)$. Query Q_1 is applied to the database: Q_1 = UPDATE R SET $A = u_1$ WHERE $B \leq u_2$. The parameters u_1 and u_2 are constants specified by the query, but we parameterize them to allow for their modification based on the complaints.

After the execution of Q_1, tuple t_0 has become $t_1 = (a_1, b_1)$. We will use attribute-level provenance to represent the resulting values a_1 and b_1. For what follows, I use the linearization logic from section 4.4 of the Tiresias paper.

Let X be a Boolean variable that is true if the WHERE clause of Q_1 affects t_0: $X = (b_0 \leq u_2)$. We will map this to integer variable x, following the logic of the Tiresias provenance representation: $x \otimes b_0 \leq x \otimes u_2$. We assign new, real variables $v_1 = x \otimes b_0$ and $v_2 = x \otimes u_2$.

To linearize the expressions of v_1 and v_2, we use the following constraints (same as in Tiresias, page 8):

$$v_1 \leq b_0 \qquad v_2 \leq u_2$$
$$v_1 \leq xM \qquad v_2 \leq xM$$
$$v_1 \geq b_0 - (1-x)M \qquad v_2 \geq u_2 - (1-x)M$$

Here, M denotes a very large number (upper bound for the u_i).

In t_1 the attribute a_1 can be expressed: $a_1 = x \otimes u_1 + (1 - x) \otimes a_0$, meaning that a_1 takes the value u_1 if $x = 1$, and the value a_0 if $x = 0$. We can now assign: $v_3 = x \otimes u_1$ and $v_4 = (1 - x) \otimes a_0$, and similarly linearize them:

$$v_3 \leq u_1 \qquad v_4 \leq a_0$$
$$v_3 \leq xM \qquad v_4 \leq (1-x)M$$
$$v_3 \geq u_1 - (1-x)M \qquad v_4 \geq a_0 - xM$$

So, now, we can express the transformation of t_0 through query Q_1 with a program with 1 integer variable (x) and 8 real variables (u_1, u_2, v_1, v_2, v_3, v_4, a_1, b_1). a_0 and b_0 are constants in this case, as they are the original values of t_0, but they could also be variables if t_0 was derived from another query.

Overall, we have a mixed integer program with the constraints:

$$b_1 = b_0$$
$$a_1 = v_3 + v_4$$
$$0 \leq x \leq 1$$
$$v_1 \leq b_0$$
$$v_1 \leq xM$$
$$v_1 \geq b_0 - (1-x)M$$
$$v_2 \leq u_2$$
$$v_2 \leq xM$$
$$v_2 \geq u_2 - (1-x)M$$
$$v_3 \leq u_1$$
$$v_3 \leq xM$$
$$v_3 \geq u_1 - (1-x)M$$
$$v_4 \leq a_0$$
$$v_4 \leq (1-x)M$$
$$v_4 \geq a_0 - xM$$

This is for the transformation of a single tuple (t_0) by a single query. Similar constraints would be derived for the other tuples, and other queries. In the end, based on the complaints, we can assign values to the corresponding variables (e.g., $a_1 = 5$), and solve the program for the parameters u_i, which will give us a correction. The optimization objective can be set so that the sum of the differences from the query parameters is minimized.

Since the provenance maintains the transformations, there wouldn't be a need for rolling back the database instance.

10.2 Hybrid algorithm

Describe Exhastive and greedy special cases.

Algorithm:

```
querylog = filter(querylog)
batchsize = 1
while i > 0
  batch = [qi to q{i-batchsize}]
  alexandras_solver(batch)
  i -= batchsize
```

11. IMPLEMENTATION

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

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

12. EXPERIMENTS

We have two goals for our evaluation: First, we study how effectively the techniques proposed in this paper are able to identify and fix a single corrupt query in various types of query logs. Second, we seek to understand the conditions in which our multi-query techniques are effective. Finally, we want to understand how different parameters of our rollback technique affects the quality of our proposed fixes.

To this end, we ran a combination of synthetically generated query logs, and well as logs generated using a subset of the TPC-C []tpcc transaction log that consists of inserts and updates. In each experiment, we corrupt the query log as described below, execute the original and corrupt logs on an initial (possibly empty) database, and compare the resulting database states to generate a true complaint set. We then add noise to the complaint set by adding false positive complaints, and removing true complaints to simulate false negatives. Finally, we execute THESYSTEM on the complaints and compare the fixed query log with the true query log, as well as the fixed and true final database states to measure perforance and accuracy metrics.

12.1 Experimental Setup

12.1.1 Metrics

- Percentage of complaints correctly modified
- Percentage of errors introduced
- Execution time
- Distance from corrected modification (str distance, clause distance, numerical distance)

12.1.2 Algorithms

- Naive
- Box
- CPLEX
- DT
- Box,Density

12.1.3 Comparison

- Query By Example algorithm
- Quoc's ConQueR

Conditions, given a database $\mathcal{D}$ and query log $qlog$:

- N_q**:** Vary number of queries in $qlog$.
- $N_{\mathcal{D}}$**:** Size of the database (number of tuples)

- N_{pred}: The number of predicates in each UPDATE query's WHERE condition.
- N_{attrs}: When corrupting the log, the number of attributes that are corrupted.
- Idx: The index of the query in the query log that was corrupted.
- p_I: Percentage of INSERT queries in the query log (as compared to UPDATEs).
- p_{pk}: Percentage of UPDATE queries with primary key filter clauses as compared to range clauses over non-primary key attributes.
- p_{FP}: Percentage of false positives in the complaint set.
- p_{FN}: Percentage of false negatives in the complaint set.

12.1.4 Dataset and Workload

Anant's workload?
TPC-C
Synthetic

We generate an initial database of $N_{\mathcal{D}}$ random tuples. The schema contains 5 attributes $a_1 \ldots a_5$ with a value within $[0, 100]$ and a primary key id. Generate N_q queries containing a mixture of insert queries randomly generated tuples and two types of update queries, PK and Range, that have the following respective forms, where the parameters ? are picked randomly:

```
UPDATE SET (a_i = ?),.. WHERE id = ?
UPDATE SET (a_i = ?),.. WHERE a_j in [?, ?+10] AND ...
```

We first consider three different homogenous query logs: *INSERT* only ($p_I = 1$), *PK* update only ($p_I = 0, p_{pk} = 1$), and *RANGE* update only ($p_I = 0, p_{pk} = 0$). These query logs help us understand THESYSTEM's performance characteristics for each query type individually. Finally, we investigate heterogenous mixtures of the three query types to simulate varying amounts of real settings.

Within each query log, we independently vary the log size, the database size, the complexity of the WHERE clause predicates, and location of the query, and the number of attributes that are corrupted (by replacing the constant with a random value within $[0, 100]$).

12.2 Single-Query Log

In the first set of experiments, we evaluate the simplest case where there is a single update query. In each experiment, we vary the DBSize, NClauses, as well as the number of clauses in the query that have been corrupted and report the metrics described above. We first compare the learning algorithms on a complete complaint set, then evaluate them using incomplete complaint sets with varying percentages of false positive and negative complaints.

12.2.1 Complete Complaints

Vary DBSize
Vary NClauses, corrupt 1 and 2 clauses

We found that CPLEX and BBOX identify the correct fix, however their running times are significantly higher than DTree. This is because CPLEX is an exact solution, as compared to DTree, whose poor early splitting decisions can adversely affect the final tree structure.

12.2.2 Incomplete Complaints

Vary DBSize
Vary NClauses, corrupt 1 and 2 clauses
Each line is plots has different perc FP

We first increased the number of false positives in the complaint set (no false negatives). Figure **??** shows how the fix quality and running time vary as the percentage of false positives increases. Compared variations of CPlex and Bounding box with varying density thresholds (?).

Each line varies perc FN

We then varied the number of false negatives while keeping the percentage of false positives fixed at 5%. Figure **??**

12.3 Increased Query Log Size

In the following set of experiments, we increase the number of queries in the log while varying ?. The number of corrupted queries is still one. In these experiments, we set the DBSize to 10000, the default NClauses to 4, and the number of corrupted clauses to 2. We first show results for varying the false positives and negatives in the complaint set and comparing the algorithms described in Section 8. We then evaluate the efficacy of of provenance-based query log filtering, which reduces the running time without affecting the result quality.

To generate the false-positives, we randomly sample without replacement from the tuples in the database that are not in the true complaint set.

12.4 False Positive

Vary false positives (1 graph)

12.5 False Negative

Vary false negatives (1 graph)

12.5.1 Filtering Queries

We also compared the provenance-based filtering techniques in the above experiments to measure their effectiveness at reducing the running time. We varied the complexity of the update WHERE clauses to control the amount that queries in the log overlap in their updates. The query log contained 50 update WHERE queries. The quality of the suggested fixes were the same, As the clauses became less complex, the likelihood of overlap increased, and increased the amount of queries that affected the complaint sets.

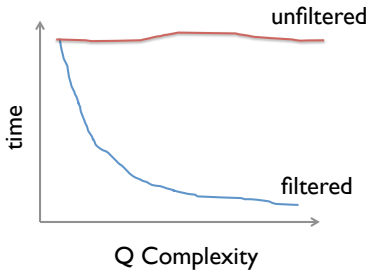

Figure 2: Varying query complexity.

12.6 Multi-Query

Using a previous experimental configuration, we varied the number of queries that are corrupted. Figures **??** show the quality and running times of the results for a query log of size 1000 and dbsize of 100k. As we can see, the cost increases quadratically with each corrupted query and the

accuracy of the proposed fixes increases marginally. This is because XXX.

We focus on two scenarios. **Try multiple corruptions that don't overlap (provenance-wise) with each other**. This is the condition of multiple silo'd that corrupt their own logs. Then **Try multiple corrupted queries where one query modifies the updated state of the other**. This shows that it is really hard and hopefully we get close?

To better understand the algorithm, we plot the quality metrics after each fixed query and measure how quickly THESYSTEM converges to the final result. This suggests that an incremental approach where the user can set a threshold to stop the algorithm may be effective.

12.7 Real Transactional Workload

We used the web application workload, and evaluated our alogirthms with artifically injected corruptions. We compared two types of corruptinos. In figure **??**, we randomly picked a single existing query and corrupted its value. If the query was an INSERT, we randomly pick a value and perturbed it. If the transaction was an UPDAET, we randomly varied the SET or WHERE clauses. We re-ran this 100 times and plot the average and standard deviation of the results.

13. DISCUSSION

Systems soluttions would be interesting

14. RELATED WORK

Scorpion explanation uses data to synthesize predicates that explain.

Why not work.

View construction and query by example.