QFix: Demonstrating error diagnosis in query histories

Xiaolan Wang
School of Computer Science
University of Massachusetts
xlwang@cs.umass.edu

Alexandra Meliou
School of Computer Science
University of Massachusetts
ameli@cs.umass.edu

Eugene Wu
Computer Science
Columbia University
ewu@cs.columbia.edu

ABSTRACT
An increasing number of applications in all aspects of society rely on data. Despite the long line of research in data cleaning and repairs, data correctness has been an elusive goal. Errors in the data can be extremely disruptive, and are detrimental to the effectiveness and proper function of data-driven applications. Even when data is cleaned, new errors can be introduced by applications and users who interact with the data. Subsequent valid updates can obscure these errors and propagate them through the database causing more discrepancies. Any discovered errors tend to be corrected superficially, on a case-by-case basis, further obscuring the true underlying cause, and making detection of the remaining errors harder.

In this demo proposal, we outline the design of QFix, a query-centric framework that derives explanations and repairs for discrepancies in relational data based on potential errors in the queries that operated on the data. This is a marked departure from traditional data-centric techniques that directly fix the data. We then describe how users will use QFix to run a demonstration scenario. Participants will be able to select from a number of transactional benchmarks, introduce errors into the queries that are executed, and compare the fixes to the queries proposed by QFix as well as existing alternative algorithms such as decision trees.

1. INTRODUCTION

Data errors are a pervasive, expensive, and time consuming problem that afflicts the vast majority of data-driven applications. For example, errors in retail price data cost US consumers $2.5 billion each year [4]. In aggregate, studies estimate data errors to cost the US economy more than $600 billion per year [4]. Despite the costliness of data errors, studies have found that a significant fraction of time in data analysis — up to 80% [6] — is devoted to cleaning and wrangling the data into a structured and sufficiently clean form to use in downstream applications.

In response, both industry and academia have focused on data cleaning solutions to mitigate this problem. ETL-type systems [10,13] focus on cleansing the data before it is loaded into the database using a set of pre-defined rules; outlier and anomaly detection algorithms [1] are used to identify errors in the data after the data has been loaded; while recent approaches use downstream applications such as interactive visualizations [7,15], application queries [11], or data-products to facilitate error detection and correction algorithms. In each of these cases, the focus of error diagnosis and cleaning has been data-centric, in the sense that the process is meant to identify and directly fix data values.

Figure 1: Q1 updates the tax amount with 30% tax rate for high income employees using an incorrect predicate. The error is propagated by Q2 to the pay field in the database. Finally, a benign insert query Q3 inserts correct salary information. The final database state contains a mixture of incorrect and correct salary data.

These efforts have largely ignored an important source of errors — queries. Mistakes in queries can introduce errors that spread throughout the database due to subsequent, possibly correct updates. Consider the following salary management example:

Example 1 (Salary Update Error). A manager updates the employees’ financial records to set the tax rate to 30% for high income employees who earn more than $87500. She submits the update through a form in the salary accounting application, but incorrectly types $85700 for the income threshold. Later queries that insert new paychecks, compute tax calculations, and aggregate department salaries, end up propagating this error to other records in the database, resulting in incorrect paychecks and employee dissatisfaction.

By the time some errors in the database are identified, possibly by employees reporting incorrect pay stubs, it is difficult to (a) identify all other errors in the database, and (b) trace these errors back to the erroneous update. Such problems can occur in any data processing system with a dynamic database: errors can be introduced by interactive queries executed by a system administrator, web-based forms that construct queries based on user input, stored procedures that use human input to fill the parameter values, or even queries with automatically generated parameters if there is
a chance of errors in the code or data used for the parameter generation.

Although existing data-centric cleaning techniques may help identify and correct these reported errors directly, this is suboptimal because it treats the symptom — the errors in the current database state — rather than the anomalous queries that are the underlying cause. In practice, only a subset of the paystub errors will be reported, thus fixing the reported errors on a case-by-case basis will likely obscure the root problem, making it more difficult to find both the erroneous query and the other affected data. Furthermore, a data-centric approach must repair all errors — a non-repaired value such as the incorrect tax rate may continue to introduce errors in the database through future queries (e.g., inserting incorrectly computed paychecks based on the still-incorrect tax rate).

For these reasons, traditional data cleaning approaches may be helpful for finding errors in the data, but are not designed to diagnose causes of the errors when they are rooted in incorrect queries. There has been some work along a similar spirit, but not directly for this problem. For example, integrity constraint-based methods [5] reject some improper updates if the data will fall outside predefined ranges. Certificate-based verification [2] handles a broader class of erroneous queries, but relies on asking users queries prior to executing the update, which renders application-generated updates infeasible. Several techniques have proposed diagnoses that describe structural sources of errors either in the form of predicates most correlated with the errors [15] or common components of a workflow that caused the errors [14], but not at the level of query identification.

Ultimately, query-centric cleaning and repair is challenging because an error introduced by a query can be obscured by, or propagated throughout the database by subsequent queries. This alludes to several factors that make even identifying problematic queries difficult:

1. **Butterfly effect**: An error in even a single query can affect a large number of records, as documented in several real-world cases [6][12][16]. Even if a single record is incorrect, its value may be used as part of the WHERE or SET clauses of subsequent valid queries that introduce additional errors that are seemingly unrelated.

2. **Partial information**: In most practical settings, we cannot assume that we can identify all errors in the database — for example, not all employees will complain about their incorrect paystubs. More likely, we only have access to a subset of the data errors, and must use them to extrapolate the queries that affected a possibly larger set of data. A diagnostic tool that can reduce the entire transaction log to the most likely candidate queries and propose fixes is needed to make this process manageable.

3. **Multiple types of errors**: An erroneous query can cause multiple types of data errors. For example, a record that should not exist may have been accidentally inserted, or conversely a record that should exist was unintentionally deleted. Similarly, attribute values may be incorrectly updated, updated when they should not have been, or not updated when they should have. Any combination of these error types may be present in the current state of the database, and although they may not be obviously related to each other, they must be addressed in a holistic manner.

In this demo proposal, we outline the design of QFix, a framework that derives explanations and repairs for discrepancies in relational data based on potential errors in the queries that operated on the data, and describe how users will use QFix in a demonstration scenario. In contrast to existing approaches in data cleaning that aim to detect and correct errors in the data directly, the goal of QFix is to identify the problematic queries that introduced errors into the database. These diagnoses both explain how errors were introduced to a dataset, and also lead to the identification of additional discrepancies in the data that would have otherwise remained undetected. Participants will be able to select from a number of transactional benchmarks to generate a query workload, introduce errors into the queries that are executed, and compare the fixes to the queries proposed by QFix against existing alternative algorithms such as decision trees.

2. **THE QFix ARCHITECTURE**

Figure 2 shows QFix’s major components. QFix takes as input a query log containing UPDATE, INSERT and DELETE queries, the database, along with a set of identified data errors (called complaints). These complaints are pairs of tuple id of tuples that are wrong, along with an estimate of their correct values (e.g., 21500 as tuple t’s tax value in Example 1). QFix uses this information to trace the causes of the errors and output the most likely set of queries in the log (diagnoses), along with proposed repairs of these queries.

To achieve this, QFix first performs an optional outlier removal step to deal with potential false positives in the complaints. Then the MILP Encoding component transforms the query diagnosis problem into a Mixed Integer Linear Program (MILP) that is further optimized through slicing and incremental repair techniques, before being sent to an industrial MILP solver. The output of the solver constitutes solutions to the query diagnosis problem.

3. **PROBLEM AND SOLUTION SKETCH**

3.1 Problem Outline

We assume a query log \( Q \) containing a sequence of update, insert and delete queries \( q_1, \ldots, q_n \) that have been executed on an initial database state \( D_0 \). For simplicity, we assume that the database is a single relation with attributes \( A_1, \ldots, A_m \), though this single-relation restriction is not a requirement for QFix. Let \( D_i = q_i(\ldots q_1(D_0)) \) be the the database state output after executing queries \( q_i \) through \( q_1 \) on \( D_0 \). Thus, the final database state is simply the application of the
query log to the initial state: \( D_0 = Q(D_0) = q_0(\ldots q_1(D_0)) \). We assume that UPDATE and DELETE queries use predicates composed of conjunctive and disjunctive range clauses of the form \( \langle A_i \ op \ ? \rangle \), where \( \op \in \{=, >, <, \leq, \geq, \neq \} \) and \( ? \) is a constant value. We also assume that SET clauses are of the form \( \langle A_j = \? \rangle \) (relative update) or \( \langle A_j = \? \rangle \) (constant update).

Queries in \( Q \) are possibly erroneous and introduce errors in the data. We assume there is a true sequence of queries \( Q' = \{q_1, \ldots, q_n\} \) that generate a true sequence of database states \( \{D_0, D_1, \ldots, D_n\} \). The true query log and database states are unknown, and our goal is to find and correct errors in \( Q \) to retrieve the correct database state \( D_n^* \).

To do so, QFix takes as input a set of identified or user-reported data errors, called a complaint set and denoted as \( C \). A complaint \( c \in C \) corresponds to a tuple in \( D_n \) along with its true attribute value assignments. For example, \( C = \{c_1\} \), where \( c_1 = (t_3, \{\text{tax} = 21500\}, \{\text{income} = 86000\}, \{\text{pay} = 64500\}) \) forms the complaint set with incorrect tax and pay attribute for the query log \( Q \) in Example [1]. A complaint can also model addition or removal of tuples: \( c = (\perp, t^* \rangle \) means that \( t^* \) should be added to the database, while \( c = (t_i, \perp) \) denotes that \( t_i \) should be removed.

Our goal is to derive a diagnosis as a log repair \( Q' = \{q_1, \ldots, q'_n\} \) such that \( Q'(D_0) = D_n^* \). In this work, we focus on errors produced by incorrect parameters in queries, so our repairs focus on altering query constants rather than query structure. Therefore, each query \( q_i \in Q' \) has the same structure as \( q_i \) (e.g., the same number of predicates, the same variables and operators in the WHERE clause), but possibly different parameters. For example, a good log repair for the example of Figure [1] is \( Q' = \{q_1, q_2, q_3\} \), where \( q_1 \) = UPDATE Taxes SET \( \text{tax} = 0.3 \times \text{income} \) WHERE \( \text{income} > 85700 \), when applied to \( t_3^{old} \) to produce \( t_3^{new} \), is translated into a set of constraints as detailed below.

First, we can rewrite the query as a conditional statement:

\[
\begin{align*}
{t_3^{new}.\text{tax}} &= \begin{cases} 
0.3 \times t_3^{old}.\text{rate} & \text{if } p \\
{t_3^{old}.\text{tax}} & 1 - p
\end{cases}
\end{align*}
\]

where

\[
p = t_3^{old}.\text{income} > 85700
\]

This is equivalent to the following linearized form. In addition, we add constraints on the starting and ending value for the rate attribute: If \( t_3^{new}.\text{tax} \) was specified in \( C \), then the provided value is used instead of the value in the database:

\[
\begin{align*}
t_3^{new}.\text{tax} &= (0.3 \times t_3^{old}.\text{income}) \times p + t_3^{old}.\text{tax} \times (1 - p)
\end{align*}
\]

This solves the naive encoding procedure that solves the Optimal Diagnosis problem. However, the size of the resulting constraint problem increases at a rate of \( O(|D| \times |Q| \times \#\text{attributes}) \), rendering it infeasible for all but the smallest databases and query logs.

QFix uses four additional optimizations not presented in this paper to scale to large query log and database sizes. The first three are called Slicing optimizations that reduce each of the components in the problem size: Tuple-slicing, Query-slicing, Attribute-slicing. MILP solvers typically (though not guaranteed) take a much longer time as the size of the MILP increases, thus each of the slicing techniques reduces the problem and speeds up the solver time.

The final optimization serves to reduce the number of undetermined variables that the MILP solver must provide a solution for. The cost of the solver, in our experiments, increases exponentially with the number of undetermined variables. To this end, QFix uses an incremental algorithm that tries to fix the queries in the query log one at a time.

These optimizations enable QFix to propose a solution within several seconds for thousands of queries and tuples on common transaction benchmarks in OLTPBench [3].
To introduce errors, the interface allows users to select any workload generators from the benchmarks in OLTPBench [3]. Since most transactional benchmarks focus on point update queries, we additionally include a synthetic workload generator that includes range updates, as well as insert and delete queries. The text box on the right side allows users to additionally specify the number of queries to generate in the workload.

**Step 1 (Select Dataset):** Participants may first choose from a dropdown menu containing a number of transaction workload generators from the benchmarks in OLTPBench [3]. Once the workload generator is specified, the Query Log component of the interface renders a scrollable list containing all of the queries. Users can either select the system to inject errors randomly by clicking the “Random Error” button or manually add errors. To introduce errors, the interface allows users to select any editable query in the log and shows the selected query in an editable popup so that users can edit the queries. For example, in the figure, the user has edited query Q2 and reduced the threshold from income < 10,000 to income < 100. The interface renders a scrollable list containing all of the queries.

**Step 2 (Corrupt the Query Log):** Once the workload generator is specified, the Query Log component of the interface renders a scrollable list containing all of the queries. Users can either select the system to inject errors randomly by clicking the “Random Error” button or manually add errors. To introduce errors, the interface allows users to select any editable query in the log and shows the selected query in an editable popup so that users can edit the queries. For example, in the figure, the user has edited query Q2 and reduced the threshold from income < 10,000 to income < 100.

**Step 3 (Form a Complaint Set):** The modified query cause the state of the database at the end of the workload to differ from the result of the original workload. The candidate complaints table lists the tuples that are different and highlights the attribute values in those tuples as red text. For instance, t1.tax, t1.income and t1.pay are all errors introduced by the modified query. Users can select individual attribute values or entire tuples to add to the complaint set that is used as input to the QFix algorithms. When she is satisfied, the user clicks Run QFix to execute the QFix and alternative algorithms.

**Step 4 (View Log Repairs):** The result page lists the original query ID and text at the top. The ID is important because some proposed fixes may identify an incorrect query. Below the original query, the interface shows each of the proposed fixes as columns. For example, Figure 4 shows that both the QFix and alternative fixes identified the correct query Q5, however QFix only took 0.2 seconds to run, and correctly fixed Q5, whereas the alternative took 10 sec, incorrectly selected Q6, and proposed an incorrect fix.

**Step 5 (Validate Repairs):** The bottom tables show the effects of the fixes on the complaints from step 3. Correctly fixed attribute values are highlighted in blue, unfixed errors are shown as red text, while incorrectly fixed values are highlighted with a red background. Finally, it is possible for proposed fixes to introduce new errors, which are shown as entire rows that are highlighted with a red background.

**4. DEMONSTRATION OUTLINE**

The objective of this demonstration is to show how QFix can quickly and accurately detect and propose fixes to errors in a query log, and compare its results to alternatives that use existing techniques.

Figures 3 and 4 show screenshots of the initial and results pages. Each step is annotated with a circled number, which we detail below.

**Step 1 (Select Dataset):** Participants may first choose from a dropdown menu containing a number of transaction workload generators from the benchmarks in OLTPBench [3]. Since most transactional benchmarks focus on point update queries, we additionally include a synthetic workload generator that includes range updates, as well as insert and delete queries. The text box on the right side allows users to additionally specify the number of queries to generate in the workload.

**Step 2 (Corrupt the Query Log):** Once the workload generator is specified, the Query Log component of the interface renders a scrollable list containing all of the queries. Users can either select the system to inject errors randomly by clicking the “Random Error” button or manually add errors. To introduce errors, the interface allows users to select any editable query in the log and shows the selected query in an editable popup so that users can edit the queries. For example, in the figure, the user has edited query Q2 and reduced the threshold from income < 10,000 to income < 100.

**Step 3 (Form a Complaint Set):** The modified query cause the state of the database at the end of the workload to differ from the result of the original workload. The candidate complaints table lists the tuples that are different and highlights the attribute values in those tuples as red text. For instance, t1.tax, t1.income and t1.pay are all errors introduced by the modified query. Users can select individual attribute values or entire tuples to add to the complaint set that is used as input to the QFix algorithms. When she is satisfied, the user clicks Run QFix to execute the QFix and alternative algorithms.

**Step 4 (View Log Repairs):** The result page lists the original query ID and text at the top. The ID is important because some proposed fixes may identify an incorrect query. Below the original query, the interface shows each of the proposed fixes as columns. For example, Figure 4 shows that both the QFix and alternative fixes identified the correct query Q5, however QFix only took 0.2 seconds to run, and correctly fixed Q5, whereas the alternative took 10 sec, incorrectly selected Q6, and proposed an incorrect fix.