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 dataset 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 in 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 [5]. 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% [8] — is devoted to cleaning and wrangling [7] 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 database 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.

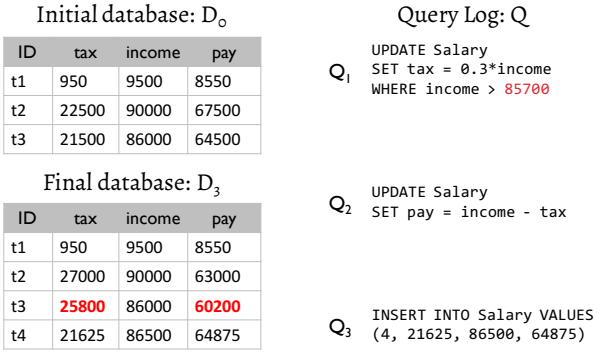

Figure 1: Q_1 updates the tax amount with 30% tax rate for high income employees using an incorrect predicate. The error is propogated by Q_2 to the *pay* field in the database. Finally, a benign insert query Q_3 inserts correct salary information. The final database state contains a mixture of incorrect 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. Figure 1 illustrates this example where Q_1, Q_2, and Q_3 are executed on an initial salary database D_0. The error in the predicate of Q_1 is propagated to other fields in the table, by other correct queries.*

By the time some errors in the database are identified, possibly by employees reporting incorrect paystubs, 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 adhoc 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 [9] 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

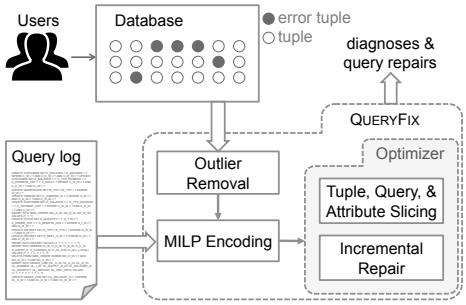

Figure 2: QFix architecture diagram

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_3'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 $A_i, \ldots, A_m$, 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_1 though q_i on D_0. Thus, the final database state is simply the application of the

query log to the initial state: $D_n = Q(D_0) = q_n(\ldots q_1(D_0))$. We assume that `UPDATE` and `DELETE` queries use predicates composed of conjunctive and disjunctive range clauses of the form $\langle A_j \; op \; ?\rangle$, where $op \in \{=, >, <, \leq, \geq, \neq\}$ and ? is a constant value. We also assume that `SET` clauses are of the form $\langle A_j = A_i + ?\rangle$ (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 $\mathcal{C}$. A complaint $c \in \mathcal{C}$ corresponds to a tuple in D_n along with its true attribute value assignments. For example, $\mathcal{C} = \{c_1\}$, where $c_1 = (t3, \{tax = 21500\}, \{income = 86000\}, \{pay = 64500\})$ 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: $c = (\perp, t^*)$ 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 tax = 0.3 * income WHERE income >= 87500`.

Problem definition

We now formalize the problem definition for diagnosing data errors using query logs. A diagnosis is a log repair Q' that resolves all complaints in the set $\mathcal{C}$ and leads to a correct database state D_n^*.

Definition 2 (Optimal diagnosis). *Given database states D_0 and D_n, a query log Q such that $Q(D_0) = D_n$, a set of complaints $\mathcal{C}$ on D_n, and a distance function d, the optimal diagnosis is a log repair Q', such that:*

- *$Q'(D_0) = D_n^*$, where D_n^* has no errors*
- *$d(\mathcal{Q}, \mathcal{Q}^*)$ is minimized*

Where $d(\mathcal{Q}, \mathcal{Q}^*)$ measures the changes made in the repaired query log. Informally, we seek the minimum changes to the log $\mathcal{Q}$ that would result in a clean database state D_n^*. Obviously, a challenge is that D_n^* is unknown, unless we know that the complaint set contains all of, and only, true complaints.

3.2 Solution Sketch

Our general strategy is to translate the starting and ending database states, D_0 and D_n, the query log Q, and complaint set $\mathcal{C}$ into a mixed-integer linear program (MILP) and solve the resulting program using a generic solver. Briefly, a linear program is a minimization problem consisting of a set of constraints in the form of linear equations, along with an objective function that is minimized; an MILP is a linear program where only a subset of the undetermined variables are required to be integers, while the rest are real valued.

To achieve this translation, we model each query as a linear equation that computes the output tuple values from the inputs or previous database state, and transforms the equation into a set of of linear constraints. In addition, the constant values in the queries are parameterized into a set of undetermined variables, while the database state is encoded as constraints on the initial and final tuple values. Finally, the undetermined variables are used to construct an objective function that prefers value assignments that minimize both the amount that the queries change and the number of non-complaint tuples that are affected.

Due to space constraints, we will walk through an example of how the query `UPDATE Taxes SET tax = 0.3 * income WHERE 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:

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

where

$$p = t_3^{old}.income > 85700$$

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

$$\begin{aligned} t_3^{new}.tax &= (0.3 * t_3^{old}.income) \times p + t_3^{old}.tax \times (1 - p) \\ p &= t_3^{old}.income > 85700 \\ t_3^{old}.tax &= 30 \\ t_3^{new}.tax &= 25 \end{aligned}$$

In the above formulation, all variables are determined and is trivially solvable. Instead, we replace the constaints 0.3 and 85700 with undetermined variables v_1 and v_2, so that solving the constraints will reassign those query constants to new values that result in the desired value for $t_3^{new}.rate$. Note that the initial range $[minval, maxval]$ of the undetermined variables could be defined based on the valid range of the corresponding attribute(s), or, as a heuristic, based on the empirical distribute of the attribute values in the database.

Extending this process to all tuples and all queries in the log describes 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 problem 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].

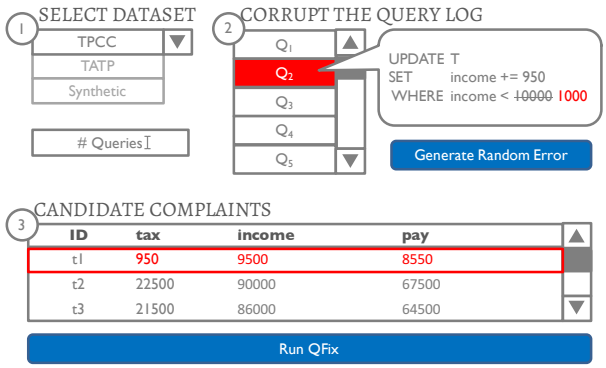

Figure 3: Users introduce errors to benchmark workloads.

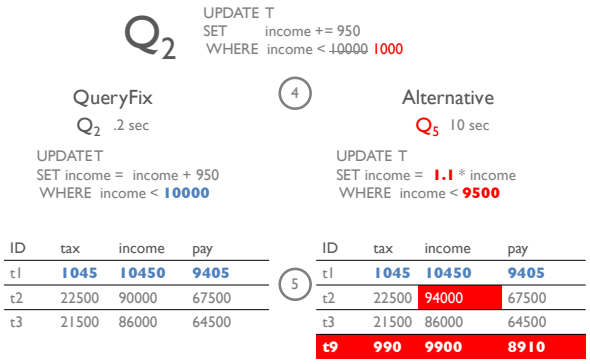

Figure 4: Comparisons between proposed fixes.

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 let 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 Q_2 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, $t_1.tax$, $t_1.income$ and $t_1.pay$ 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 Q_2, however QFix only took 0.2 seconds to run, and correctly fixed Q_2, whereas the alternative took 10 sec, incorrectly selected Q_5, 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.

5. REFERENCES

[1] V. Chandola, A. Banerjee, and V. Kumar. Outlier detection: A survey. *ACM Computing Surveys*, 2007.

[2] S. Chen, X. L. Dong, L. V. Lakshmanan, and D. Srivastava. We challenge you to certify your updates. In *SIGMOD*, 2011.

[3] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. In *VLDB*, 2013.

[4] W. W. Eckerson. Data quality and the bottom line. *TDWI Report, The Data Warehouse Institute*, 2002.

[5] W. Fan, F. Geerts, and X. Jia. A revival of integrity constraints for data cleaning. In *VLDB*, 2008.

[6] B. Grady. Oakland unified makes $7.6M accounting error in budget; asking schools not to count on it. In *Oakland*, 2013.

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

[8] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *IEEE Trans. Vis. Comput. Graph.*, 2012.

[9] N. Khoussainova, M. Balazinska, and D. Suciu. Towards correcting input data errors probabilistically using integrity constraints. In *MobiDE*, 2006.

[10] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu. Systemt: a system for declarative information extraction. *SIGMOD Record*, 2009.

[11] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, T. Kraska, T. Milo, and E. Wu. Sampleclean: Fast and reliable analytics on dirty data. 2015.

[12] M. Sakal and L. Raković. Errors in building and using electronic tables: Financial consequences and minimisation techniques. In *Strategic Management*, 2012.

[13] C. Thomsen and T. B. Pedersen. A survey of open source tools for business intelligence. In *Data Warehousing and Knowledge Discovery*. 2005.

[14] X. Wang, X. L. Dong, and A. Meliou. Data x-ray: A diagnostic tool for data errors. In *SIGMOD*, 2015.

[15] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. In *PVLDB*, 2013.

[16] J. Yates. Data entry error wipes out life insurance coverage. In *Chicago Tribune*, 2005.