QFix: Diagnosing Errors through 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

Data-driven applications rely on the correctness of their data to function properly and effectively. Errors in data can be incredibly costly and disruptive, leading to loss of revenue, incorrect conclusions, and misguided policy decisions. While data cleaning tools can purge datasets of many errors before the data is used, applications and users interacting with the data can introduce new errors. Subsequent valid updates can obscure these errors and propagate them through the dataset causing more discrepancies. Even when some of these discrepancies are discovered, they are often corrected superficially, on a case-by-case basis, further obscuring the true underlying cause, and making detection of the remaining errors harder.

In this paper, we propose QFix, a framework that derives explanations and repairs for discrepancies in relational data, by analyzing the effect of queries that operated on the data and identifying potential mistakes in those queries. QFix is flexible, handling scenarios where only a subset of the true discrepancies is known, and robust to different types of update workloads. We make four important contributions: (a) we formalize the problem of diagnosing the causes of data errors based on the queries that operated on and introduced errors to a dataset; (b) we develop exact methods for deriving diagnoses and fixes for identified errors using state-of-the-art tools; (c) we present several optimization techniques that improve our basic approach without compromising accuracy, and (d) we leverage a tradeoff between accuracy and performance to scale diagnosis to large datasets and query logs, while achieving near-optimal results. We demonstrate the effectiveness of QFix through extensive evaluation over benchmark and synthetic data.

1. INTRODUCTION

In spite of the growing importance of big data, sensors, and automated data collection, manual data entry continues to be a primary source of high-value data across organizations of all sizes, industries, and applications: sales representatives manage lead and sales data through SaaS applications [77]; human resources, accounting, and finance departments manage employee and corporate information through terminal or internal browser-based applications [61]; driver data is updated and managed by representatives throughout local DMV departments [7, 16]; consumer banking and investment data is managed through web or mobile-based applications [13,19]. In all of these examples, the database is updated by translating form-based human inputs into INSERT, DELETE or UPDATE query parameters that run over the backend database—in essence, these are instances of OLTP applications that translate human input into stored procedure parameters. Unfortunately, numerous studies [9, 53, 57], reports [31, 67, 81, 91] and citizen journalists [50] have consistently found evidence that human-generated data is both error-prone, and can significantly corrupt downstream data analyses [68]. Thus, even if systems assume that data import pipelines are error-free, queries of human-driven applications continue to be a significant source of data errors, and there is a pressing need for solutions to diagnose and repair these errors. Consider the following representative toy example that we will use throughout this paper:

Example 1 (Tax bracket adjustment). *Tax brackets, determining tax rates for different income levels, are often adjusted. Accounting firms implement these changes to their databases by appropriately updating the customer tax rates. Figure 1 shows a simplified tax rate adjustment scenario and highlights how a single error to the predicate in update query q_1 can introduce errors in the* `owed` *attribute; a benign query q_3 then propagates the error to affect the* `pay` *attribute.*

This type of data entry error can be found throughout information management systems. In 2012, there were nearly 90,000 local governments in the United States, each responsible for tracking taxation information such as the tax rate, regulatory penalties, and the amount each citizen owes. Government employees manage this information using form-based accounting systems [66] and are ultimately susceptible to form entry error. A cursory search of the news finds numerous reports of how data entry errors resulted in utility companies overcharging cities by millions of dollars [23], a state attorney prosecuting the wrong people [62], and the Boston Police paying tens of millions of dollars for a new records system to combat data entry errors that "riddled" the previous system, which contained "highly scrutinized stop-and-frisk information" [14]. In addition, numerous articles describe how clerical errors by local governments [4] and the CRA [49] affect citizen tax bills and returns calculations.

In real-world cases like these, data errors are typically identified and reported by individuals to departments that

SIGMOD'17, May 14 - 19, 2017, Chicago, Illinois, USA

ISBN 978-1-4503-4197-4/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3035918.3035925

Taxes: D_0

ID	income	owed	pay
t_1	$9500	$950	$8550
t_2	$90000	$22500	$67500
t_3	$86000	$21500	$64500
t_4	$86500	$21625	$64875

Query log: $\mathcal{Q}$

```
q1: UPDATE Taxes SET owed=income*0.3
    WHERE income>=85700
q2: INSERT INTO Taxes
    VALUES (85800, 21450, 64350)
q3: UPDATE Taxes SET pay=income-owed
```

Taxes: D_4

ID	income	owed	pay
t_1	$9500	$950	$8550
t_2	$90000	$27000	$63000
t_3	$86000	$25800	$60200
t_4	$86500	$25950	$60550
t_5	$85800	$21450	$64350

Figure 1: A recent change in tax rate brackets calls for a tax rate of 30% for those with income above $87500. The accounting department issues query q_1 to implement the new policy, but the predicate of the WHERE clause condition transposed two digits of the income value.

do not have the resources nor the capability to deeply investigate the reports. Instead, the standard course of action is to correct mistakes on a case-by-case basis for each complaint. As a result, unreported errors can remain in the database indefinitely, and their cause becomes harder to trace as further queries modify the database, propagate the errors, and obscure their root cause. There is a need for tools that can use the error reports to diagnose and identify the anomalous queries (root causes) in the database.

In this paper, we present QFix, a diagnosis and repair system for data errors caused by anomalous DML queries in OLTP applications. Given a set of reported errors (*complaints*) about records in the current database state, QFix analyzes the sequence of historical queries executed on the database, filters them to those that may have affected the erroneous records, and generates diagnoses by identifying the specific subset of queries that most likely introduced the errors. Alongside these diagnoses, QFix also proposes repairs for the erroneous queries; these repairs can correct the reported errors, as well as potentially identify and fix additional errors in the data that would have otherwise remained undetected. To derive these diagnoses and repairs, we must address three key characteristics, which make this problem both difficult to solve and unsuitable to existing techniques:

Obscurity. Handling data errors directly 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.

Large impact. Erroneous queries cause errors at a large scale. The potential impact of the errors is high, as manifested in several real-world cases [67, 76, 91]. Further, errors that remain undetected for a significant amount of time can instigate additional errors, even through valid updates. This increases both their impact, and their obscurity.

Systemic errors. The errors created by bad queries are *systemic*: they have common characteristics, as they share the same cause. The link between the resulting data errors is the query that created them; cleaning techniques should leverage this connection to diagnose and fix the problem. Diagnosing the cause of the errors will achieve systemic fixes that will correct all relevant errors, even if they have not been explicitly identified.

Traditional approaches to data errors take two main forms. The first uses a combination of detection algorithms (e.g., human reporting, outlier detection, constraint violations) to identify a candidate set of error values that are corrected through human-based [42, 47, 52] or semi-automated means (e.g., denial constraints [22], value imputation). Unfortunately, this has the following problems: (a) it targets the symptom (incorrect database state) rather than the underlying cause (incorrect queries), (b) it can be expensive to perform, (c) it may introduce errors if the automated corrections are not perfect [1], and (d) it may make it harder to identify other data affected by the bad query.

The second form attempts to prevent data errors by guarding against erroneous updates. For example, integrity constraints [54] reject some improper updates, but only if the data falls outside rigid, predefined ranges. In addition, data entry errors such as in the tax example will satisfy the integrity constraints and not be rejected, despite being incorrect. Certificate-based verification [20] is less rigid, but it is impractical and non-scalable as it requires users to answer challenge questions before allowing each update.

QFix is complementary to existing techniques: it does not prevent errors from entering the database, and its primary goal is not to identify errors in the data. Rather, given some reported data errors, QFix analyzes query histories to determine how the errors entered the database. Determining the root cause of data errors can in turn help identify additional data errors that are due to the same cause, and which would have potentially remained unidentified. Specifically, in this paper, we make the following contributions:

- We formalize the problem of *Query Explanation*: diagnosing a set of data errors using the log of update queries over the database. Given a set of *complaints* as representations of data discrepancies in the current database state, QFix determines how to resolve all of the complaints with the minimum number of changes to the query log (Section 2)
- We illustrate how existing synthesis, learning, and cleaning-oriented techniques have difficulty scaling beyond a query log containing a single query. We then introduce an exact error-diagnosis solution using a novel mixed integer linear programming (MILP) formulation that can be applied to a broad class of OLTP applications. This approach uses state-of-the-art solvers to identify optimal diagnoses that are guaranteed to resolve all complaints without introducing new errors to the data (Section 3).
- We present a suite of optimizations that reduce the problem size without affecting the quality of the proposed repairs. Further, we propose a pragmatic incremental algorithm tailored to cases when the user is looking for individual corrupt queries (in contrast to sets of corruptions), and show how these optimizations can scale to large datasets (100K records, Figure 10a) and query logs (up to 2K DML statements, Figure 6), and tolerate incomplete information such as unreported errors (Section 6.4).

Notation	Description
$\mathcal{Q}$	The sequence of executed update queries (log) $\mathcal{Q} = \{q_1, \ldots, q_n\}$
D_0	Initial database state at beginning of log
D_n	End database state (current) $D_n = \mathcal{Q}(D_0)$
D_i	Database state after query q_i: $D_i = q_i(\ldots q_1(D_0))$
$c : t \mapsto t^*$	Complaint: $\mathcal{T}_c(D) = (D_n \setminus \{t\}) \cup \{t^*\}$
$\mathcal{C}$	Complaint set $\mathcal{C} = \{c_1, \ldots, c_k\}$
$\mu_q(t)$	Modifier function of q (e.g., `SET` clause)
$\sigma_q(t)$	Conditional function of q (e.g., `WHERE` clause)
t_{new}	Tuple values introduced in an `INSERT` query
$\mathcal{Q}^*$	Log repair
$d(\mathcal{Q}, \mathcal{Q}^*)$	Distance functions between two query logs

Figure 2: Summary of notations used in the paper.

- We perform a thorough evaluation of the data and query log characteristics that influence QFix's trade-offs between performance and accuracy. We compare the baseline and optimized algorithms under a controlled, synthetic setting and demonstrate that our optimizations improve response times by up to 40× and exhibit superior accuracy. We also evaluate QFix on common OLTP benchmarks and show how QFix can propose fully accurate repairs within milliseconds on a scale 1 TPC-C workload with 2000 queries (Section 6.2).

2. MODELING ABSTRACTIONS

In this section, we introduce a running example inspired from the use-case of Example 1, and describe the model abstractions that we use to formalize the diagnosis problem.

EXAMPLE 2. *Figure 1 demonstrates an example tax bracket adjustment in the spirit of Example 1. The adjustment sets the tax rate to 30% for income levels above \$87,500, and is implemented by query q_1. A digit transposition mistake in the query, results in an incorrect owed amount for tuples t_3 and t_4. Query q_2, which inserts a tuple with slightly higher income than t_3 and t_4 and the correct information, obscures this mistake. This mistake is further propagated by query q_3, which calculates the pay check amount based on the corresponding income and owed.*

While traditional data cleaning techniques seek to identify and correct the erroneous values in the table *Taxes* directly, our goal is to diagnose the problem, and understand the reasons for these errors. In this case, the reason for the data errors is the incorrect predicate value in query q_1.

In this paper, we assume that we know *some* errors in the dataset, and that these errors were caused by erroneous updates. The errors may be obtained in different ways: traditional data cleaning tools may identify discrepancies in the data (e.g., a tuple with lower income has higher owed tax amount), or errors can be reported directly from users (e.g., customers reporting discrepancies to customer service). *Our goal is not to correct the errors directly in the data, but to analyze them as a "symptom" and provide a diagnosis.* The diagnosis can produce a targeted treatment: knowing how the errors were introduced guides the proper way to trace and resolve them.

2.1 Error Modeling

In our setting, the diagnoses are associated with errors in the queries that operated on the data. In Example 2, the errors in the dataset are due to the digit transposition mistake in the WHERE clause predicate of query q_1. Our goal is to infer the errors in a log of queries automatically, given a set of incorrect values in the data. We proceed to describe our modeling abstractions for data, queries, and errors, and how we use them to define the diagnosis problem.

Data and query models

Query log ($\mathcal{Q}$): We define a query log that update the database as an ordered sequence of `UPDATE`, `INSERT`, and `DELETE` queries $\mathcal{Q} = \{q_1, \ldots, q_n\}$, that have operated on a database D. In the rest of the paper, we use the term *update queries*, or just *queries*, to refer to any of the queries in (Q), including insertion and deletion queries.

Query (q_i): We model each query as a function over a database D, resulting in a new database D'. For `INSERT` queries, $D' = q(D) = D \cup \{t_{new}\}$. We model `UPDATE` and `DELETE` queries as follows:

$$D' = q(D) = \{\mu_q(t) \mid t \in D, \sigma_q(t)\} \cup \{t \mid t \in D, \neg\sigma_q(t)\}$$

In this definition, the modifier function $\mu_q(t)$ represents the query's update equations, and it transforms a tuple by either deleting it ($\mu_q(t) = \perp$) or changing the values of some of its attributes. The conditional function $\sigma_q(t)$ is a boolean function that represents the query's condition predicates. In the example of Figure 1:

$$\mu_{q_1}(t) = (t.income, t.income * 0.3, t.pay)$$
$$\sigma_{q_1}(t) = (t.income \geq 85700)$$
$$\mu_{q_3}(t) = (t.income, t.owed, t.income - t.owed)$$
$$\sigma_{q_2}(t) = \texttt{true}$$

Note that in this paper, we do not consider sub-queries or aggregation. We clarify all assumptions later in this section.

Database state (D_i): We use D_i to represent the state of a database D after the application of queries q_1 through q_i from the log $\mathcal{Q}$. D_0 represents the original database state, and D_n the final, or current, database state. Out of all the states, the system only maintains D_0 and D_n. In practice, D_0 can be a checkpoint: a state of the database that we assume is correct; we cannot diagnose errors before this state. The intermediate states can be derived by executing the log: $D_i = q_i(q_{i-1}(\ldots q_1(D_0)))$. We also write $D_n = \mathcal{Q}(D_0)$ to denote that the final database state D_n can be derived by applying the sequence of queries in the log to the original database state D_0.

True database state (D_i^*): Queries in $\mathcal{Q}$ are possibly erroneous, introducing errors in the data. There exists a sequence of *true* database states $\{D_0^*, D_1^* \ldots, D_n^*\}$, with $D_0^* = D_0$, representing the database states that would have occurred if there had been no errors in the queries. The true database states are unknown; our goal is to find and correct the errors in $\mathcal{Q}$ and retrieve the correct database state D_n^*.

For ease of exposition, in the remainder of the paper we assume that the database contains a single relation with attributes $A_1, \ldots, A_m$, but the single table is not a requirement in our framework.

Error models

Following the terminology in Example 1, we model a set of identified or user-reported data errors as *complaints*. A complaint corresponds to a particular tuple in the final database state D_n^*, and identifies that tuple's correct value assignment. We formally define complaints below:

DEFINITION 3 (COMPLAINT). *A complaint c is a mapping between two tuples: $c : t \mapsto t^*$, such that t and t^* have the same schema, $t, t^* \in D_n \cup \{\perp\}$, and $t \neq t^*$. A complaint defines a transformation $\mathcal{T}_c$ on the final database state D_n: $\mathcal{T}_c(D_n) = (D_n \setminus \{t\}) \cup \{t^*\}$, which replaces t in D_n with t^*.*

In the example of Figure 1, two complaints are reported on the final database state D_3: $c_1 : t_3 \mapsto t_3^*$ and $c_2 : t_4 \mapsto t_4^*$, where $t_3^* = (86000, 21500, 64500)$ and $t_4^* = (86500, 21625, 64875)$. For both these cases, each complaint denotes a **value correction** for a tuple in D_3. Complaints can also model the **addition** or **removal** of tuples: $c : \perp \mapsto t^*$ means that t^* should be added to the database, whereas $c : t \mapsto \perp$ means that t should be removed from the database.

Note that in this definition, a complaint needs to specify the exact values for all fields of a tuple. However, this is done for ease of exposition, and it is not an inherent restriction in our approach. The definitions and our algorithms can trivially generalize to handle complaints where only some of the correct tuple values are known.

Complaint set ($\mathcal{C}$): We use $\mathcal{C}$ to denote the set of all known complaints $\mathcal{C} = \{c_1, \ldots, c_k\}$, and we call it the *complaint set*. Each complaint in $\mathcal{C}$ represents a transformation (addition, deletion, or modification) of a tuple in D_n. We assume that the complaint set is consistent, i.e., there are no two complaints that propose different transformations to the same tuple $t \in D_n$. Applying all these transformations to D_n results in a new database instance $D'_n = \mathcal{T}_{c_1}(\mathcal{T}_{c_2}(\ldots \mathcal{T}_{c_k}(D_n)))$.[1] $\mathcal{C}$ is *complete* if it contains a complaint for each error in D_n. In that case, $D'_n = D_n^*$. In our work, we do not assume that the complaint set is complete, but, as is more common in practice, we only know a subset of the errors (incomplete complaint set). Further, we focus our analysis on *valid* complaints; we briefly discuss dealing with invalid complaints (complaints identifying a correct value as an error) in Section 5, but these techniques are beyond the scope of this paper.

Log repair ($\mathcal{Q}^*$): The goal of our framework is to derive a diagnosis as a log repair $\mathcal{Q}^* = \{q_1^*, \ldots, q_n^*\}$, such that $\mathcal{Q}^*(D_0) = D_n^*$. 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, for each query $q_i^* \in \mathcal{Q}^*$, q_i^* has the same structure as q_i (e.g., the same number of predicates and the same variables in the `WHERE` clause), but possibly different parameters. For example, a good log repair for the example of Figure 1 is $\mathcal{Q}^* = \{q_1^*, q_2, q_3\}$, where q_1^*=`UPDATE Taxes SET owed=income*0.3 WHERE income >= 87500`.

Problem definition

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

DEFINITION 4 (OPTIMAL DIAGNOSIS). *Given database states D_0 and D_n, a query log $\mathcal{Q}$ such that $\mathcal{Q}(D_0) = D_n$, a set of complaints $\mathcal{C}$ on D_n, and a distance function d, the optimal diagnosis is a log repair $\mathcal{Q}^*$, such that:*

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

[1]Since the complaint set is consistent, it is easy to see that the order of transformations is inconsequential.

More 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 is complete.

Problem scope and solution outline

In this work, we focus on handling data manipulation statements (`UPDATE`, `INSERT`, and `DELETE` queries) with simple, basic query structures without subqueries, aggregations, or joins. Expressions (in predicates and SET clauses) may be over linear combinations of constants and a single attribute, and we do not support arbitrary user defined functions. We find that the queries that we focus on are applicable to a broad range of user-facing web applications (e.g., conference attendee scheduling, voting, data entry) and OLTP benchmarks, and that complex query structures and operations are less common than in read-oriented analytical workloads.

Given a query log and a set of complaints, QFix proposes repairs as modifications of values in one or more queries. QFix does not modify the structure of queries, and makes the implicit assumption that the log starts either with an empty or a clean database. We demonstrate that QFix can solve problems with corruptions in multiple queries, but its scalability in this setting is limited (up to about 50 queries in the log). For cases where corruptions are restricted to a single query, QFix can scale to large data and log sizes.

In Section 3, we describe our basic method, which uses a constraint programming formulation that expresses this diagnosis problem as a mixed integer linear program (MILP). Section 4 presents several optimization techniques that extend the basic method, allowing QFix to (1) handle cases of incomplete information (incomplete complaint set), and (2) scale to large data and log sizes. Specifically, the fully optimized, incremental algorithm (Section 4.4), can handle query logs with hundreds of queries within minutes, while the performance of the basic approach collapses by 50 queries.

3. A MILP-BASED SOLUTION

The *Optimal Diagnosis* problem states that a log repair should resolve all complaints when re-executing the repaired log on the initial (possibly empty) database state. The key challenge is that solutions must be able to handle data dependencies between queries in the log (e.g., q_i reads what q_j wrote). Unfortunately, this challenge renders existing database techniques [18,89], as well as machine learning-based approaches, infeasible because they are designed to "repair" individual non-nested `SELECT` queries. Appendix A uses a decision tree-based approach to highlight why learning-based techniques perform poorly for even a single DML statement, and fail to work for more than one query.

To address these cross-query dependencies, we introduce a constraint-based approach to the *Optimal Diagnosis* problem. To do so, it maps the problem into a mixed-integer linear programming (MILP) problem by linearizing and parameterizing the corrupted query log over the tuples in the database. Briefly, a MILP problem involves assigning values to a set of undetermined variables such that they satisfy a set of linear equations and minimize an objective function—it is mixed because the variables may be integers or real numbers.

Our general strategy is to model each query as a linear equation that computes the output tuple values from the inputs and to transform 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 before and after the query is encoded as constraints on the initial and final tuple values. Finally, the objective function over the undetermined variables prefers assignments that minimize the amount that the queries change and the number of non-complaint tuples that are affected.

The rest of this section will introduce the properties of MILP solvers, describe how to encode a single query and single tuple attribute, then extend the encoding procedure to the entire database and query log. We finally define the objective function. Subsequent sections introduce optimizations and variations of the problem.

3.1 MILP Solvers

MILP problems are known to be NP-hard with respect to the number of constraints and undetermined variables, however numerous pruning and pre-processing optimizations and heuristics have made solvers very fast [5,29,69,74,78,82]. As a result, MILP solvers are both quite efficient and widely used in practice for applications such as trajectory planning [58,60,73], assembly and scheduling processes [36,79,80], and general decision support [39,59]. Modern solver performance is primarily sensitive to *the number of constraints* and *the number of undetermined variables* in the problem [8,40,63]. One of our key contributions in this paper is to use this observation to design a set of optimizations to dramatically reduce the size of the MILP problem—enough so that we can produce repairs for TPC-C workloads within one second.

3.2 Encoding a Single Query

MILP problems express constraints as a set of linear inequalities. Our task is to derive such a mathematical representation for each query in $\mathcal{Q}$. Starting with the functional representation of a query (Section 2.1), we describe how each query type, `UPDATE`, `INSERT`, and `DELETE`, can be transformed into a set of linear constraints over a tuple t and an attribute value A_j.

`UPDATE`: Recall from Section 2.1 that query q_i can be modeled as the combination of a modifier function $\mu_{q_i}(t)$ and conditional function $\sigma_{q_i}(t)$. First, we use binary variable $x_{q_i,t}$ to indicate whether query q_i produces an effect on tuple t.

$$x_{q_i,t} = e_{q_i,t} \otimes \sigma_{q_i}(t) \tag{1}$$

We use $e_{q_i,t}$ to support `DELETE` statements. $e_{q_i,t}$ is a binary indicator of t's existence in the database prior to q_i, and is by default set to 1 when there are no `DELETE` queries in the log. If a tuple exists, then $x_{q_i,t}$ depends on t satisfying the condition function σ_{q_i}. Otherwise, t has been deleted, $e_{q_i,t} = 0$, and $x_{q_i,t}$ will always be false. We describe how to set $e_{q_i,t}$ (Equation 7) when we introduce `DELETE` queries.

Next, we introduce real-valued variables for the attributes of t. We express the updated value of an attribute using semi-modules, borrowing from the models of provenance for aggregate operations [6]. A semi-module consists of a commutative semi-ring, whose elements are scalars, a commutative monoid whose elements are vectors, and a multiplication-by-scalars operation that takes a scalar x and a vector u and returns a vector $x \otimes u$. A similar formalism has been used in literature to model hypothetical data updates [65].

Given a query q_i and tuple t, we express the value of attribute A_j in the updated tuple t' as follows:

$$t'.A_j = x_{q_i,t} \otimes \mu_{q_i}(t).A_j + (1 - x_{q_i,t}) \otimes t.A_j \tag{2}$$

In this expression, the $\otimes$ operation corresponds to regular multiplication, but we maintain the $\otimes$ notation to indicate that it is a semi-module multiplication by scalars. This expression models the action of the update: If t satisfies the conditional function ($x_{q_i,t} = 1$), then $t'.A_j$ takes the value $\mu_{q_i}(t).A_j$; if t does not satisfy the conditional function ($x_{q_i,t} = 0$), then $t'.A_j$ takes the value $t.A_j$. In our running example, the rate value of a tuple t after query q_1 would be expressed as: $t'.owed = x_{q_1,t} \otimes (t.income * 0.3) + (1 - x_{q_1,t}) \otimes t.owed$. Equation (2) does not yet provide a linear representation of the corresponding constraint, as it contains multiplication of variables. To linearize this expression, we adapt a method from [65]: We introduce two variables $u.A_j$ and $v.A_j$ to represent the two terms of Equation (2): $u.A_j = x_{q_i,t} \otimes \mu_{q_i}(t.A_j)$ and $v.A_j = (1 - x_{q_i,t}) \otimes t.A_j$. Assuming a number M is a large enough value [10] that is outside of the domain of $t.A_j$, we get the following constraints:

$$\begin{array}{ll} u.A_j \leq \mu_{q_i}(t).A_j & v.A_j \leq t.A_j \\ u.A_j \leq x_{q_i,t} M & v.A_j \leq (1 - x_{q_i,t}) M \\ u.A_j \geq \mu_{q_i}(t).A_j - (1 - x_{q_i,t}) M & v.A_j \geq t.A_j - x_{q_i,t} M \end{array} \tag{3}$$

The set of conditions on $u.A_j$ ensure that $u.A_j = \mu_{q_i}(t).A_j$ if $x_{q_i,t} = 1$, and 0 otherwise. Similarly, the conditions on $v.A_j$ ensure that $v.A_j = t.A_j$ if $x_{q_i,t} = 0$, and 0 otherwise. Now, Equation (2) becomes linear:

$$t.A'_j = u.A_j + v.A_j \tag{4}$$

`INSERT`: An insert query adds a new tuple t_{new} to the database. If the query were corrupted, then the inserted values need repair. We use a binary variable $x_{q_i,t}$ to model whether the query impacts the value of tuple t. Each attribute of the newly inserted tuple ($t'.A_j$) may take one of two values: the value specified by the insertion query ($t_{new}.A_j$) if the query changes the value of the tuple t ($x_{q_i,t} = 1$), or an undetermined value ($v.A_j$) otherwise. Thus, similar with Equation (2), we write:

$$t'.A_j = x_{q_i,t} \otimes t_{new}.A_j + (1 - x_{q_i,t}) \otimes v.A_j \tag{5}$$

`DELETE`: A delete query removes a set of tuples from the database. Since the MILP problem doesn't have a way to express a non-existent value, we encode a deleted tuple by setting its attributes to a *"ghost"* value, M^-, outside of the attribute domain. Since M^- is outside of the attribute domain, any subsequent conditional functions will evaluate to false, so subsequent queries do not affect ghost tuples. There are nuances to how M^- is set. It needs to be sufficiently large, for the MILP problem to prioritize a modification to the `WHERE` clause of the `DELETE` query ($\sigma_{q_i}(t) = 0/1$), compared to a modification of the `SET` clause of an `UPDATE` query to the ghost value ($\mu_{q_i}(t.A_j) = M^-$). However, it should be $M^- \leq M$ to ensure the constraints remain feasible (Equation 3). Using M^- thus ensures that subsequent queries will treat the tuple as a *"ghost"* and ignore it.

$$t'.A_j = x_{q_i,t} \otimes M^- + (1 - x_{q_i,t}) \otimes t.A_j \tag{6}$$

The variable $x_{q_i,t}$ is set according to Equation (1); in Equation (1), $e_{q_i,t}$ is set to 0 if t was deleted in q_i ($t.A_j = M^-$) and the deletion is correct due to its presence in the complaint set ($t^*.A_j = M^-$). Otherwise, the tuple exists and $e_{q_i,t} = 1$.

$$e_{q_i,t} = \neg((t.A_j = M^-) \wedge (t^*.A_j = M^-)) \tag{7}$$

Algorithm 1: *Basic* : The MILP-based approach.

Require: $\mathcal{Q}, D_0, D_n, \mathcal{C}$
1: $milp\_cons \leftarrow \emptyset$
2: **for** each t in R **do**
3: **for** each q in $\mathcal{Q}$ **do**
4: $milp\_cons \leftarrow milp\_cons \cup Linearize(q, t)$
5: **end for**
6: $milp\_cons \leftarrow milp\_cons \cup AssignVals(D_0.t, D_n.t, \mathcal{C})$
7: **for** each i in $\{0, \ldots, N-1\}$ **do**
8: $milp\_cons \leftarrow milp\_cons \cup ConnectQueries(q_i, q_{i+1})$
9: **end for**
10: **end for**
11: $milp\_obj \leftarrow EncodeObjective(milp\_cons, \mathcal{Q})$
12: $solved\_vals \leftarrow MILPSolver(milp\_cons, milp\_obj)$
13: $\mathcal{Q}^* \leftarrow ConvertQLog(Q, solved\_vals)$
14: Return $\mathcal{Q}^*$

This expression is further linearized using the same method as Equation (3).

Putting it all together. The constraints defined in Equations (1)–(6) form the main structure of the MILP problem for a single attribute A_j of a single tuple t. To linearize a query q_i one needs to apply this procedure to all attributes and tuples. This process is denoted as $Linearize(q, t)$ in Algorithm 1. Our MILP formulation includes three types of variables: the binary variables $x_{q_i,t}$, the real-valued attribute values (e.g., $u.A_j$), and the real-valued constants in μ_{q_i} and σ_{q_i}. All these variables are undetermined and need to be assigned values by a MILP solver.

Next, we extend this encoding to the entire query log, and incorporate an objective function encouraging solutions that minimize the overall changes to the query log.

3.3 Encoding and Repairing the Query Log

We proceed to describe the procedure (Algorithm 1) that encodes the full query log into a MILP problem, and solves the MILP problem to derive $\mathcal{Q}^*$. The algorithm takes as input the query log $\mathcal{Q}$, the initial and final (dirty) database states $\mathcal{D}_{0,n}$, and the complaint set $\mathcal{C}$, and outputs a fixed query log $\mathcal{Q}^*$.

We first call *Linearize* on each tuple in $\mathcal{D}_0$ and each query in $\mathcal{Q}$, and add the result to a set of constraints *milp_cons*. The function *AssignVals* adds constraints to set the values of the inputs to q_0 and the outputs of q_n to their respective values in $\mathcal{D}_0$ and $\mathcal{T}_\mathcal{C}(\mathcal{D}_n)$. Additional constraints account for the fact that the output of query q_i is the input of q_{i+1} (*ConnectQueries*). This function simply equates t' from the linearized result for q_i to the t input for the linearized result of q_{i+1}.

Finally, *EncodeObjective* augments the program with an objective function that models the distance function between the original query log and the log repair ($d(\mathcal{Q}, \mathcal{Q}^*)$). In the following section we describe our model for the distance function, though other models are also possible. Once the MILP solver returns a variable assignment, *ConvertQLog* updates the constants in the query log based on this assignment, and constructs the fixed query log $\mathcal{Q}^*$.

3.4 The Objective Function

The optimal diagnosis problem (Definition 4) seeks a log repair $\mathcal{Q}^*$, such that the distance $d(\mathcal{Q}, \mathcal{Q}^*)$ is minimized. We follow similar intuition as other existing data repair problems [28] in our objective function. In this section, we describe our model for the objective function, which assumes numerical parameters and attributes. This assumption is not a restriction of the QFix framework. Handling other data types, such as categorical values, comes down to defining an appropriate distance function, which can then be directly incorporated into QFix.

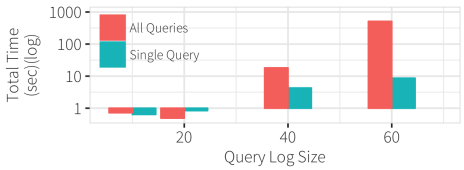


Figure 3: Log size vs. execution time for 1000 records. The basic approach failed to complete by the time limit of 1000sec for a log of 80 queries.

In our experiments, we use the normalized Manhattan distance (in linearized format in the MILP problem) between the parameters in $\mathcal{Q}$ and $\mathcal{Q}^*$. We use $q.param_i$ to denote the i^{th} parameter of query q, and $|q.param|$ to denote the total number of parameters in q:

$$d(\mathcal{Q}, \mathcal{Q}^*) = \sum_{i=1}^{n} \sum_{j=1}^{|q_i.param|} |q_i.param_j - q_i.param_j^*|$$

Different choices for the objective function are also possible. For example, one may prioritize the total number of changes incurred in the log, rather than the magnitude of these changes. However, a thorough investigation of different possible distance metrics is beyond the scope of our work.

4. OPTIMIZING THE BASIC APPROACH

A major drawback of our *basic* MILP transformation (Section 3) is that it exhaustively encodes the combination of all tuples in the database and all queries in the query log. In this approach, the number of constraints (as well as undetermined variables) grows quadratically with respect to the database and the query log. This increase has a large impact on the running time of the solver, since it needs to find a (near)-optimal assignment of all undetermined variables (exponential with the number of undetermined variables). This is depicted in Figure 3, which increases the query log size over a database of 1000 tuples. The red bars encode the problem using the *basic* algorithm that parameterizes all queries, while the blue bars show the potential gain of only parameterizing the oldest query that we assume is incorrect. Beyond 80 queries, *basic* fails to produce an answer within 1000 seconds. Although MILP solvers exhibit empirical performance variation, this experiment illustrates the performance limitation of the *basic* approach.

A second limitation of *basic* is its inability to handle errors in the complaint set. This is because the *basic* MILP formulation generates hard constraints for all of the database records, thus any error, whether a false negative missing complaint or a false positive incorrect complaint, must be correct. It may be impossible to find a repair that satisfies this condition and will lead to solver infeasibility errors.

The rest of this section describes three classes of *slicing* optimizations that reduce the number of tuples, queries, and attributes that are encoded in the MILP problem. The tuple-slicing technique additionally improves the repair accuracy when the complaint set is incomplete. We also propose an incremental algorithm that avoids the exponential increase in solver time by only parameterizing a small number of queries at a time—thus limiting the cost to the left side of Figure 3.

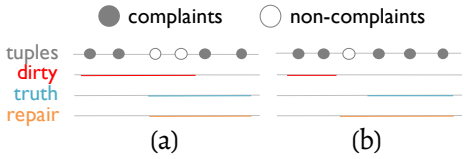


Figure 4: Graphical depiction of correct (a) and over-generalized (b) repairs. Solid and empty circles represent complaint and non-complaint tuples. Each thick line represents the interval of query q's range predicate. Dirty: incorrect interval in corrupted query; truth: correct interval in true query; repair: interval returned by the solver.

4.1 Tuple Slicing: Reducing Tuples

Our first optimization, *tuple-slicing*, applies a two step process to reduce the problem size without sacrificing accuracy: it first aggressively reduces the problem size by only encoding tuples in the complaint set and then refines the log repair through a second but much smaller MILP problem.

Step 1 (Initial Repair Step): The first step of *tuple slicing* aggressively reduces the problem size by only encoding those tuples in the complaint set $\mathcal{C}$ (Algorithm 1 line 2 is replaced with `for each t in` $\mathcal{C}$). Each tuple necessitates the linearization of the entire query log, thus, only encoding the complaint tuples minimizes the size of the problem with respect to the relevant tuples. This optimization is guaranteed to resolve $\mathcal{C}$, thus depending on the properties of the non-complaint records, it can generate correct repairs an order of magnitude faster without hurting the accuracy. In Figure 4(a), the solver will guarantee a repair interval that excludes the two left-most complaints, includes the two right-most complaints, and minimizes the difference between the dirty and repaired intervals (due to the objective function). This effectively pushes the repair's lower-bound towards that of the dirty interval. This is a case where such a solution is correct, because the dirty and truth intervals overlap. Recall that we do not have access to the truth interval, and our goal is to reproduce the truth interval given $\mathcal{C}$ (solid circles) and the corrupted query.

However, this approach can also cause the repair to be a *superset* of the truth interval, and affect tuples not part of the complaint set. Figure 4(b) highlights such a case where the dirty and truth intervals are non-overlapping, and the non-complaint record between them has been incorrectly included in the repair interval—*because the MILP problem did not include the non-complaint.*

In both of these cases, the objective function will ensure that the repair does not over-generalize the upper bound towards the right because that strictly increases the objective function. Therefore, our main concern is to refine the repair interval to exclude those non-complaint tuples in case (b). Note that in the case of incomplete complaint sets, the user may choose to not execute the refinement step if she believes that the non-complaint records are indeed in error.

Step 2 (Refinement Step): Although there are many possible mechanisms to refine the initial repair (e.g., incrementally shrinking the repaired interval until the non-complaint tuples are all excluded), the straightforward approaches are not effective when multiple corrupt queries have been repaired because they don't take the query interactions into account.

Instead, we solve this with a second, significantly smaller, MILP problem. Let $\mathcal{Q}^*_{rep}$ be the set of repaired queries from the initial MILP formulation with tuple slicing; $\mathcal{NC}$ be the set of non-complaint tuples now matching the repaired `WHERE` clauses, as in Figure 4(b); and $\mathcal{C}^+ = \mathcal{C} \cup \mathcal{NC}$. We create a new MILP using $\mathcal{C}^+$ as the complaint set. The key is to only parameterize the repaired clauses from Step 1 as constraints with undetermined variables. The variables for all other tuples and queries are fixed to their assigned values from Step 1. This *refines* the solutions from the previous step while incorporating knowledge about complaints in $\mathcal{NC}$, Finally, we use a new objective function to minimize the number of non-complaint tuples $t \in \mathcal{NC}$ that are matched by the solution.

In our experiments, we find that this second MILP iteration adds minimal overhead $(0.1 - 0.5\%)$ with respect to the initial MILP problem. *Tuple-slicing* is a **heuristic** method that decomposes a large MILP problem into two, typically much smaller, MILP problems. It is effective in practice and greatly helps improve QFix performance, especially when the ratio of the complaint set size and the database size is small. In general, this heuristic can result in incorrect repairs. However, if corruptions are restricted to a single query, the complaint set is complete, and incremental repair is employed (Section 4.4), we can guarantee that tuple slicing will not lead to loss of accuracy using a small modification: By *disallowing non-complaint tuples* in the refinement step (e.g., by restricting the value of the objective function in the refinement MILP to zero), the solver will be forced to pick the correct repair.

4.2 Query Slicing: Reducing Queries

In practice, many of the queries in the query log could not have affected the *complaint attributes* (defined below). For example, if q_{N-1} and q_N only read and wrote attribute A_1, then they could not have contributed to an error in A_2. However, if q_N wrote A_2, then either or both queries may have caused the error. In short, if we model a query as a set of attribute read and write operations, those not part of the causal read-write chain to the *complaint attributes* can be ignored. This is the foundation of our *query-slicing* optimization.

Definition 5 (Complaint Attributes $\mathcal{A}(\mathcal{C})$). *The set of attributes identified as incorrect in the complaint set.*

$$\mathcal{A}(\mathcal{C}) = \{A_i | t.A_i \neq t^*.A_i, c(t,t^*) \in \mathcal{C}\}$$

We proceed to define the impact that a query has directly (the set of attributes it modifies), and its full impact (the set of attributes it may affect through all subsequent queries).

Definition 6 (Query dependency & impact). *Query q_i has **direct impact**, $\mathcal{I}(q_i)$, which is the set of attributes updated in its modifier function μ_{q_i}, and **dependency**, $\mathcal{P}(q_i)$, which is the set of attributes involved in its condition function σ_{q_i}. We use $\mathcal{F}_j(q_i)$ to denote the impact of q_i on the output of q_j $(j \geq i)$:*

$$\mathcal{F}_j(q_i) = \begin{cases} \mathcal{F}_{j-1}(q_i) \cup \mathcal{I}(q_j), & \text{if } j > i \wedge \mathcal{F}_{j-1}(q_i) \cap \mathcal{P}(q_j) \neq \emptyset \\ \mathcal{F}_{j-1}(q_i), & \text{if } j > i \wedge \mathcal{F}_{j-1}(q_i) \cap \mathcal{P}(q_j) = \emptyset \\ \mathcal{I}(q_i), & \text{otherwise} \end{cases}$$

*Thus, query q_i's **full impact**—the set of attributes it may affect through all subsequent queries in the log—is its impact to the most recent query: $\mathcal{F}(q_i) = \mathcal{F}_n(q_i)$.*

Full impact is a form of forward provenance: it traces the query history toward more recent queries and extending the

Algorithm 2: *FullImpact* : Algorithm for finding $\mathcal{F}(q)$.

Require: $\mathcal{Q}$, q_i
$\mathcal{F}(q_i) \leftarrow \mathcal{I}(q_i)$
for each q_j in $q_{i+1}, ..., q_n \in \mathcal{Q}$ **do**
  if $\mathcal{F}(q_i) \cap \mathcal{P}(q_j) \neq \emptyset$ **then**
    $\mathcal{F}(q_i) \leftarrow \mathcal{F}(q_i) \cup \mathcal{I}(q_j)$
  end if
end for
Return $\mathcal{F}(q_i)$

Algorithm 3: Inc_k : The incremental algorithm.

Require: $Q, \mathcal{D}_j, \mathcal{D}_n, \mathcal{C}, k$
Sort Q from most to least recent
for each $q_i ... q_{i+k} \in Q$ **do**
  $\mathcal{Q}_{suffix} = \{q_j | j \geq i\}$
  $\mathcal{Q}^* \leftarrow Basic_{params}(\mathcal{Q}_{suffix}, \mathcal{D}_j, \mathcal{D}_n, \mathcal{C}, \{q_i, q_{i+k}\})$
  if $\mathcal{Q}^* \neq \emptyset$ **then**
    Return $\mathcal{Q}^*$
  end if
end for

impact of a query to include every attribute that may have been affected by it. For example, assume the following query log: $\{q_1$ writes A; q_2 reads A writes B, q_3 reads B writes $C\}$. The direct impact of q_1 is $\{A\}$. Through q_2, it extends to $\{A, B\}$, and it's full impact (after q_3) is $\{A, B, C\}$.

Based on the full impact of q, we can determine if it affects the complaints $\mathcal{C}$ and is a candidate for repair. Specifically, if $\mathcal{F}(q) \cap \mathcal{A}(C) = \emptyset$, then q does not contribute to the complaints and can be ignored in the repair process; otherwise, it is a candidate for repair. In the case of *single-query corruptions*, q is a candidate for repair only if $\mathcal{F}(q) \cap \mathcal{A}(C) = \mathcal{A}(C)$, which makes this optimization even more effective.

We use $Rel(\mathcal{Q})$ to denote the set of queries that are candidates for repair. Query slicing only linearizes the queries in $Rel(\mathcal{Q})$ which is a conservative estimate of all queries that may have affected the user complaints. Since this set is typically much smaller than the entire query log, this optimization leads to smaller problems than *basic* without any loss of accuracy. We formalize this result in Lemma 7.

4.3 Attribute Slicing: Reducing Attributes

In addition to removing irrelevant queries, we additionally avoid encoding irrelevant attributes. Given $Rel(\mathcal{Q})$, the relevant attributes can be defined as: $Rel(\mathcal{A}) = \cup_{q_i \in Rel\mathcal{Q}}(\mathcal{F}(q_i) \cup \mathcal{P}(q_i))$ We propose *attribute slicing* optimization that only encodes constraints for attributes in $Rel(\mathcal{A})$. We find that this type of slicing can be effective for wide tables along with queries that focus on a small subset of attributes.

Similar to query slicing, this optimization removes only non-relevant attributes through static analysis of the query log. Thus, it reduces the problem size without loss of accuracy. We formalize the soundness of the query and attribute slicing in the following lemma.

LEMMA 7. *If* $\mathbf{R}$ *is a set of repairs that QFix produces under no slicing optimizations, then QFix will produce the same repairs* $\mathbf{R}$ *under query and attribute slicing.*

4.4 Incremental Repairs

Even with the slicing optimizations, the number of undetermined variables can remain high, resulting in slow solver runtime. The red bars in Figure 3 showed the exponential cost of parameterizing the entire query log as compared to only solving for a single query (blue bars). These results suggest that it is *faster* to run many small MILP problems than a single large one, and motivates our incremental algorithm.

Our Inc_k approach (Algorithm 3) focuses on the case where there is a single corrupted query to repair. It does so by linearizing the full query log, including any slicing optimizations, but only parameterizing and repairing a batch of k consecutive queries at a time. This procedure first attempts to repair the k most recent queries, and continues to the next k queries if a repair was not generated. The algorithm internally calls a modified version of the *basic* approach that takes extra parameters $\{q_i, q_{i+k}\}$, only parameterizes those queries, and fixes the values of all other variables.

The incremental approach prioritizes repairs for complaints that are due to more recent corruptions. Given that the *basic* algorithm simply fails beyond a small log size, we believe this is a natural and pragmatic assumption to use, and results in a 10× scalability improvement. Our experiments further evaluate different batching levels k in the incremental algorithm and show that it is impractical from both a performance and accuracy to have $k > 1$.

5. NOISY COMPLAINT SETS

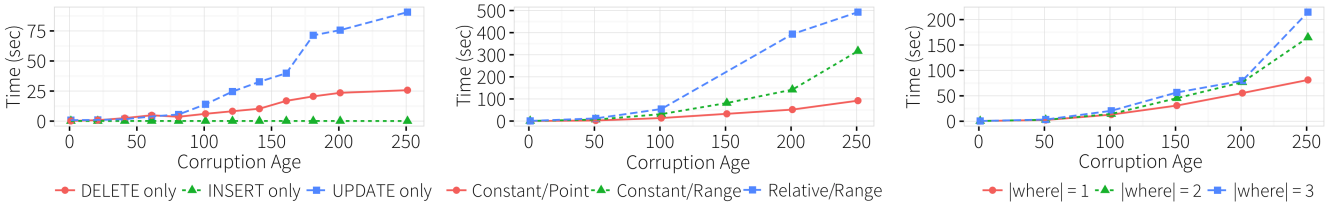
As described in the problem setup (Section 2.1), complaint sets may be imperfect. First, complaint sets are typically incomplete, missing errors that occur in D_n, but are not reported. In this case, the naive encoding of the query log and database (*basic*) will likely produce an infeasible MILP. In the running example of Figure 1, if the complaint set is incomplete and only contains a complaint on t_4, *basic* will interpret t_3 as a correct state and repairing the condition of q_1 to a value greater than \$86500 will appear to introduce a new error. The solver will declare the problem infeasible and will not return a solution.

However, the tuple slicing optimization (Section 4.1) implicitly corrects this problem: By only encoding the tuples in the incomplete complaint set, the encoded problem does not enforce constraints on the query's effect on other tuples in the database. This allows the result to generalize to tuples not in the complaint set. The second iteration of the MILP execution then uses a soft constraint on the number of non-complaint tuples that are affected by the repair in order to address the possibility of over-generalization.

Another possible inaccuracy in the complaint set is the presence of false positives: some complaints may be incorrectly reporting errors, or the target tuple t^* of a complaint may be incorrect. This type of noise in the complaint set can also lead to infeasibility. One can remove such erroneous complaints as a pre-processing step, using one of numerous outlier detection algorithms. While this is an interesting problem, it is orthogonal to the query repair problem that we are investigating in this work. Thus, in our experiments, we focus on incomplete complaint sets and assume that there are not erroneous complaints.

6. EXPERIMENTS

We now study the sensitivity of the basic, optimized and incremental variations of the QFix algorithm to changes in the database and query log characteristics. Due to the difficulty of collecting corrupt query logs from active deployments, we try to understand these trade-offs in controlled synthetic



(a) Performance for different query types. (b) Performance of diff. query clause types. (c) Query dimensionality vs time.

Figure 5: QFix can repair `INSERT` and `DELETE` workloads quickly; complex `UPDATE` queries are more expensive to repair.

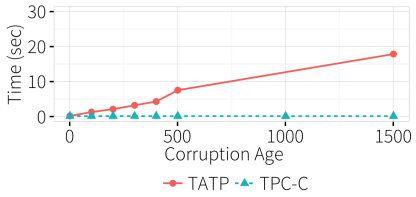


Figure 6: QFix quickly produces repairs for OLTP workloads.

scenarios, as well as for queries from two widely used OLTP benchmarks.

To this end, we first focus on the case of single query corruptions and evaluate the optimized QFix incremental algorithm on two OLTP benchmarks (Section 6.2) and find that QFix can return high quality repairs in interactive speeds. We then evaluate the variations of the incremental algorithm in a wide range of controlled database and query log settings (Section 6.3) Finally, we evaluate the incremental and basic algorithms on more challenging settings of incomplete complaint sets and multiple query corruptions and discuss why the latter setting is fundamentally difficult (Section 6.3.1). All experiments were run on 12x2.66 GHz machines with 16GB RAM running IBM CPLEX [26] as the MILP solver on CentOS release 6.6. In the following text, QFix refers to the incremental algorithm with the tuple-slicing optimization: this setting works well in most OLTP benchmark settings because the tables seldom have more than 50 attributes; Section 6.3.2 shows how attribute and query-slicing can further improve performance a wide (up to 500 attributes) tables. We refer the reader to Appendix A for a study of alternative machine learning-based repair algorithms.

6.1 Experimental Setup

For each of our experiments we generate and corrupt a query log. We execute the original and corrupt query logs on an initial (possibly empty) database, perform a tuple-wise comparison between the resulting database states to generate a true complaint set, and simulate incomplete complaint sets by removing a subset of the true complaints. Finally, we execute the algorithms and compare the repaired query log with the true query log, as well as the repaired and true final database states, to measure performance and accuracy metrics. Performance is measured as wall clock time between submitting a complaint set and the system terminating after retrieving a valid repair. Accuracy is reported as the repair's precision (percentage of repaired tuples that were correctly fixed), recall (the percentage of the full complaint set that was repaired), and F1 measure (the harmonic mean of precision and recall). These metrics measure whether the complaint tuples are repaired correctly, but it is possible that the repaired query differs from the true query. We separately evaluate whether QFix selects the right query to repair; these results are included in Appendix B. In summary, we find that QFix always fixes the right query when the complaint set is complete. However, the less complete the complaint set, and the older the corruption, the more likely it is that QFix will repair the wrong query. Our experiments report averages across 20 runs. We describe the experimental parameters in the context of the datasets and workloads below.

Synthetic: We generate an initial database of N_D random tuples. The schema contains a primary key *id* along with N_a attributes $a_1 \dots a_{N_a}$, whose values are integers picked from $[0, V_d]$ uniformly at random. We then generate a sequence of N_q queries. The default setting for these parameters are: $N_D = 1000, N_a = 10, V_d = 200, N_q = 300$.

`UPDATE` queries are defined by a SET clause that assigns an attribute a *Constant* or *Relative* value, and a WHERE clause can either be a *Point* predicate on a key, or a *Range* predicate on non-key attributes:

```
SET Clause:                    WHERE Clause:
 Constant: SET (a_i=?), ...     Point: WHERE a_j=? & ...
 Relative: SET (a_i=a_i+?)      Range: WHERE a_j in [?,?+r] & ...
```

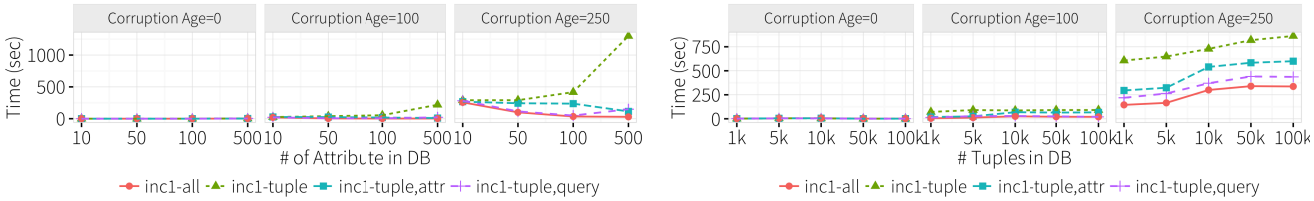
where `?`$\in [0, V_d]$ is random and `r` is the size of the range predicate. Query selectivity is by default 2% (`r`$= 4$). Note that a range predicate where `r = 0` is distinct from a *Point* predicate due to the non-key attribute. The WHERE clauses in `DELETE` queries are generated in an identical fashion, while `INSERT` queries insert values picked uniformly at random from V_d. By default, we generate `UPDATE` queries with non-key range predicates and constant set clauses.

Benchmarks: We use the TPC-C [25] and TATP [88] benchmarks. The former generates the *ORDER* table at scale 1 with one warehouse, and uses the queries that modify the *ORDER* table. We execute a log of 2000 queries over an initial table containing 6000 tuples. 1837 queries are `INSERT`s and the rest are `UPDATE`s. The latter TATP workload simulates the caller location system. We generate a database from *SUBSCRIBER* table with 5000 tuples and 2000 `UPDATE` queries. Both setups were generated using the OLTP-bench [32].

Corrupting Queries: We corrupt query q_i by replacing it with a randomly generated query of the same type based on the procedures described above. To standardize our procedures, we selected a fixed set of queries indexes based on their age with respect to the most recent query. For instance, an age of 50 means the corruption was 50 queries ago on q_{N_q-50}. We call this parameter the *Corruption Age*.

6.2 Benchmark Results

In this experiment, we vary the location of a single corrupt query in the TPC-C and TATP benchmark query logs and



(a) # of attributes vs time ($N_D = 100$). (b) Database size vs time ($N_a = 100$).

Figure 7: For datasets with many attributes or many records, the optimizations result in significant improvements.

report QFix's performance; in all runs, QFix achieves an F1 score of 1. Figure 6 shows that QFix can generate a repair for TPC-C and TATP within milliseconds and tens of seconds, respectively. The key reason is that in practice, each query affects a small set of records and results in a very small complaint set—$1-2$ on average. Tuple and query slicing are also able to aggressively reduce the total number of constraints to < 100 constraints on average.

QFix can repair TPC-C queries are predominantly `INSERT`s, which QFix can solve within milliseconds. In contrast, TATP only contains `UPDATE`s, which are harder to solve than `INSERT` queries and thus lead to higher execution time compared with TPC-C queries. Note that these experiments stripped out read-only queries from the workload, which account for 8 and 80% of the queries in TPC-C and TATP, respectively. Finally, QFix repairs Example 1 in Figure 1 within 35 milliseconds.

Takeaways: many workloads in practice are dominated by `INSERT` and point `UPDATE` queries (ignoring the dominant percentage of read-only queries). In these settings, QFix is very effective at reducing the number of constraints and can derive repairs with near-interactive latencies.

6.3 Sensitivity of the Incremental Algorithms

This subsection evaluates the efficacy of using each slicing optimization on the incremental algorithm by varying the characteristics of the database and query log. By default, the tuple-slicing optimization is always enabled because the algorithms are unable to scale beyond 50 queries without it (Figure 3). We report performance and omit accuracy numbers because the F1 for all settings is nearly 1.

6.3.1 Sensitivity to the Query Log

The following experiments evaluate QFix (incremental with all optimizations) under differing query log characteristics. We first vary the query type and find that `UPDATE` queries are the most expensive query type to repair. We then focus solely on `UPDATE`-only workloads and vary query complexity and predicate dimensionality. The database is set to the default settings ($N_a = 10$, $N_D = 1000$, $N_q = 300$) and we vary the location of the single corrupt query.

Query Type: This experiment compares QFix over `INSERT`, `DELETE`, or `UPDATE`-only query logs to test the effect of the query type. Figure 5a shows that while the cost of repairing `INSERT` workloads remains relatively constant, the costs for `DELETE`-only and `UPDATE`-only workloads increase as the corruption happens earlier in the query log—and a much faster rate for `UPDATE` queries. This is because `UPDATE` queries translate into more undetermined variables than `INSERT` or `DELETE` queries, and are significantly more expensive to repair. For this reason, our subsequent experiments focus specifically on the more challenging `UPDATE`-only workloads.

Query Clause Type: So far, we have focused on `UPDATE` queries with constant set clauses and range predicates (*Constant/Range*). Figure 5b compares this against *Constant/Point* and *Relative/Range* `UPDATE` query workloads. We found that point predicates are easier to solve than range predicates because 1) the latter doubles the number of undetermined variables as compared to point predicates and 2) point queries are on key attributes, which further reduces the MILP search space. In addition, constant set clauses are easier than relative set clauses because the former breaks the causal relationship between input and output records for the overwritten values. This both simplifies the difficulty of the constraint problem, and reduces the total number of constraints.

Predicate Dimensionality: Figure 5c varies the dimensionality of the update queries by increasing the number of predicates in the `WHERE` clause, while keeping the query cardinality constant (so the number of complaints is fixed). The cost increases with the dimensionality because each additional predicate is translated into a new set of constraints and undetermined variables, increasing the problem complexity.

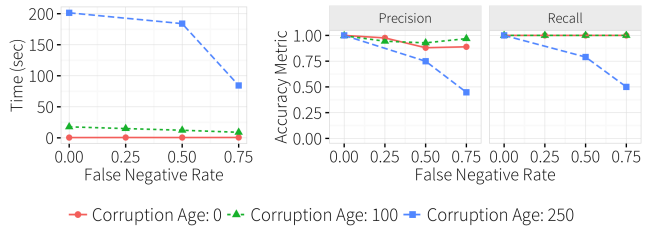
Takeaways: we find `UPDATE`-workloads are on average significantly harder than workloads with other types of queries, and that performance is closely related to the complexity and the dimensionality of queries. In the challenging setting of range `UPDATE`-only workloads, QFix find a repair within seconds or minutes for 200 *queries—particularly if the corruption is recent.*

6.3.2 Sensitivity to Database Properties

The following two experiments compare different combinations of the slicing optimizations *tuple/query/attr* under varying database size and schema size settings. The query log contains the default $N_q = 300$ *Constant/Range* `UPDATE` queries. Each facet (subplot) in Figure 7 represents the location of the corruption as $q_{N_q-250}, q_{N_q-100}, q_{N_q-0}$.

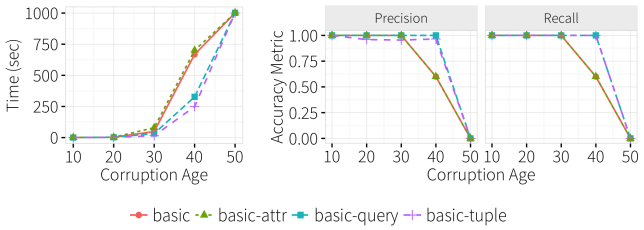
# of Attributes: We first vary the number of attributes ($N_a \in [10, 500]$) under a fixed database size $N_D = 100$. As shown in Figure 7a, when the number of attribute in a table is small (e.g., $N_a = 10$) or when the corruption is recent (e.g., $q_{200,300}$), then all optimizations appear identical. However, increasing the number of attribute exhibits a larger benefit for query and attribute slicing (up to 6.8× reduction compared to tuple-slicing). When the table is wide ($N_a = 500$), applying all optimizations ($inc_1 - all$) is 40× faster than tuple-slicing alone.

Database Size: We vary the database size ($N_D \in [1k, 100k]$) with a large number of attributes ($N_a = 100$). We fix the number of complaints by decreasing the query selectivity in proportion to N_D's increase—the specific mechanism to do so did not affect the findings. Figure 7b shows that the costs are relatively flat until the corruption occurs in an old query



(a) False negatives vs time. (b) False negatives vs accuracy.

Figure 8: Incomplete complaint sets improve repair speed due to less complaints, but degrade repair quality for older corruption (with higher *Corruption Age*).



(a) Multi-corrupt. vs time. (b) Multi-corrupt. vs accuracy.

Figure 9: Our analysis highlights limitations of *basic*, the value of tuple-slicing, and the high cost of `UPDATE` queries.

(*Corruption Age* = 250). In addition, we find that the cost is highly correlated with the number of candidate queries and attributes that are encoded in the MILP problem. The increase in cost despite tuple-slicing is due to the increasing number of candidate queries and attributes in the system; we believe this increasing trend limits the solver's ability to prune constraints that correspond to queries and attributes that clearly will not affect the complaint set—an implicit form of query and attribute slicing. Ultimately, combining all three optimizations outperforms tuple-slicing by $2-3\times$.

Takeaways: we find that repair performance is sensitive to the number of attributes and the number of tuples in the database, particularly when the corruption is old. Tuple slicing is essential to solve general problems, while attribute and query slicing show significant gain for datasets with a large number of attributes.

Performance Limitations: QFix fundamentally relies on MILP solvers to produce repairs. This comes with two scalability limitations evident in QFix: (1) current solver limitations have trouble scaling to very large problem sizes, and (2) generating very large problems is memory-intensive with respect to the data and log size. We use almost all memory on the experiment machines with 100 *attributes,* 100*k tuples, and* 250 *queries. Techniques to address or side-step these limitations will be valuable in future work.*

6.4 More Challenging Repair Settings

In this section, we further study the performance of QFix in solving hard problems—when the set of complaints is incomplete, and when there is more than one corruption that led to the complaints[2].

Incomplete Complaint Set: The first experiment (Figures 8a and 8b) varies the false negative rate in incomplete complaint sets. We increase the rate from 0 (0% missing in the complaint set) to 0.75 (75% are missing). We find that reducing the number of reported complaints lowers the runtime; however, we observe a small reduction in repair quality (precision and recall in Figure 8b) for recent corruptions and a significant drop for older ones. This is expected: the less information we have on the problem (fewer complaints), the lower our ability to fix it correctly. In the extreme case where very few complaints are available, the problem can be under-specified, and the true repair harder to identify.

[2] Note that even when there are multiple corruptions in the log, the incremental algorithm may still be applicable if only one is responsible for the set of complaints. We leave this study to future work.

Multiple Corrupt Queries: This experiment studies how the *basic* algorithm, along with the slicing optimizations, are able to repair complaints resulting from multiple corrupt queries. We use the default settings, vary the number of corruptions using the following procedure: we corrupt every tenth query in the log starting from oldest query q_1, and vary the `UPDATE`-only query log size in increments of 10 within $[10, 50]$ inclusive. For example, when the $N_q = 30$, we corrupt queries $q_{1,11,21}$.

We find that all variants of *basic* have difficulty scaling beyond 30 queries (consistent with the case with one corrupt query in Figure 3), although tuple and query slicing modestly improve repair performance and quality. In addition, the number of corruptions and queries greatly affect the scalability (Figure 9a) and the accuracy (Figure 9b) of the algorithms. Specifically, as the corruptions increase, the number of possible assignments of the MILP parameters increases exponentially and the solver often takes longer than our experimental time limit of 1000 seconds and returns an empty result. We find that problem infeasibility is the predominant explanation for why the accuracy degrades past 30 queries. For example, with 40 queries (4 corruptions), *basic* takes nearly 750s; however if we ignore the infeasible executions, the average is 300 seconds and the precision and recall are greater than 0.94. Unfortunately, with 50 queries (5 corruptions), all runs are infeasible and exceed the time limit.

Takeaways: QFix solves recent errors (e.g., errors in most recent 100 queries) efficiently and effectively even with very incomplete complaint information. Also, basic*, even with slicing optimizations, has severe scalability limitations due to the large number of undetermined variables—this is unsurprising as MILP constraint solving is an NP-hard problem. This result highlights the value of the incremental algorithm optimization.*

7. RELATED WORK

QFix tackles the problem of diagnosis and repair in relational query histories (query logs). However, since QFix does not modify the query structure, one could frame this as a data diagnosis and repair problem: Queries are represented in a relation R_q where each tuple is a record of a query and its parameters, and the log is modeled as a single nested `SELECT` query over R_q and D_0. The result of this query is a view representing the final database state D_n and complaints are errors annotated on this view. Prior work has focused on the problem of identifying the source tuples responsible for errors in the output. However, the existing techniques are not effective for this problem: Some of the existing work focuses on aggregate queries [75, 89] and there is no direct

mapping between an aggregate function and individual error tuples. Descriptive and prescriptive cleaning [17] uses the lineage of output errors to identify the input tuples that are connected to most errors. This intuition is not suitable for our problem setting, because there is no causal relationship between the coverage of an update and the likelihood of error (a query that updates the entire database is no more likely to be incorrect than a query that updates a single tuple). Finally, causality techniques [64] are inefficient and cannot scale because the problem size grows exponentially. Moreover, they do not produce repairs.

QFix does not aim to correct errors in the data directly, but rather to find the underlying reason for reported errors in the queries that operated on the data. This is in contrast to traditional data cleaning [28, 35, 51, 71, 72] which focuses on identifying and correcting data "in-place." Identifying and correcting errors is an important, and rightfully well-studied problem. Existing literature has supplied a variety of tools to capture errors originating from data integration [2], recognizing the same entities in data [46, 56], identifying true facts among conflicting data [33, 92, 93], and language support for cleaning [38]. All of these techniques are complementary to our work. Their goal is to identify which data is correct and which data is incorrect, but they don't look for the sources of these errors in the processes or queries that generate and modify this data. In fact the output of such methods can be used to augment the complaint sets used by QFix, which focuses on identifying errors in the queries that produced the data, rather than the data itself.

An aspect of data cleaning focuses on providing repairs for the identified errors [22]. Tools in this domain have targeted different methods and interactions for providing fixes, ranging from rules [12, 24] and functional dependencies [22, 35], to interactive methods that solicit user feedback [72, 90]. As with the other data cleaning approaches, all these techniques again operate on the data directly. In contrast, QFix analyzes errors in the data to diagnose and repair errors in queries that operated on the data. Thus, QFix leverages the fact that some data errors are systemic, linked to erroneous updates. Diagnosing the cause of the errors, will achieve systematic fixes that will correct all relevant errors, even if they have not been explicitly identified.

Closer to exploring systemic reasons and patterns for errors are systems such as Data Auditor [43, 44] and Data X-Ray [87]. Both tools tools identify features, which can be selected from the tuple attributes, that best summarize or describe groups of tuples (or specifically errors). While these tools can generate feature sets or patterns of attributes that characterize errors, these are not linked to the queries, but are again characterizations over the data itself. Such techniques can be tremendously useful if the processes that generate or modify the data are unknown or black-box computations. In these cases, Data Auditor and Data X-Ray can provide high-level clues for potential problems in the data derivation process. However, both approaches are oblivious to the actual queries that operated on the data, and they do not provide particular fixes. Ontology-based why-not explanations [83] is similar to Data X-Ray, but only relevant to absent tuples (deletions), and does not consider the query history.

The topic of query revisions has been studied in the context of why-not explanations [18]. These scenarios investigate the absence of answers in a query result, and often attempt to modify the query to change its outcome. Skyline refinement [85] focuses specifically on refinements that use skyline queries, while semi-automatic SQL debugging [86] revises a given query to make it return specified tuple groups in its output. Furthermore, Query-by-example [94] and query correction [3] are similar problems that generate or modify queries based on user interaction such as desired result records. All these approaches are limited to selection predicates of `SELECT` queries, and they only typically consider one query at a time. In contrast, QFix handles update workloads, processes large query histories, and can model several steps in the dataset's evolution. A lot of explanation work [11, 30, 34, 41, 55, 84] targets specific application domains, limiting its applicability to our setting.

Finally, as QFix traces errors in the queries that manipulate data, it has connections to the field of *data and workflow provenance*. Our algorithms build on several formalisms introduced by work in this domain. These formalisms express why a particular data item appears in a query result, or how that query result was produced in relation to input data [15, 21, 27, 45].

8. SUMMARY AND DISCUSSION

The general problem of data errors is complex, and exacerbated by its highly contextual nature. We believe that an approach to explain and repair such data errors, based on operations performed by the application or user, is a promising step towards incorporating contextual hints into the analysis process.

Towards this goal, QFix is the first framework to diagnose and repair errors in the queries that operate on the data. Datasets are typically dynamic: even if a dataset starts clean, updates may introduce new errors. QFix analyzes OLTP query logs to trace reported errors to the queries that introduced them. This in turn helps identify additional errors in the data that may have been missed and gone unreported.

We proposed `basic` which uses non-trivial transformation rules to encode the data and query log as a MILP problem. We further presented two types of optimizations: (1) slicing-based optimizations that reduce the problem size and often improve, rather than compromise accuracy, and (2) an incremental approach that analyzes one query at a time. Our experiments show that the latter significantly increases the scalability and latency of repairing single-query corruptions—at interactive speeds for OLTP benchmarks such as TPC-C—without significant reduction in accuracy.

To the best of our knowledge, QFix is the first formalization and solution to the diagnosis and repair of errors using past executed queries. Obviously, correcting such errors in practice poses additional challenges. The initial version of QFix described in this paper focuses on a constrained problem consisting of simple (no subqueries, complex expressions, UDFs, aggregations, nor joins) single-query transactions with clauses composed of linear functions, and complaint sets without false positives. In future work, we hope to extend our techniques to relax these limitations towards more complex query structures and towards support for CRUD-type web application logic. In addition, we plan to investigate additional methods of scaling the constraint analysis, as well as techniques that can adapt the benefits of single-query analysis to errors in multiple queries.

Acknowledgements. This material is based upon work supported by the National Science Foundation under grants IIS-1421322 and IIS-1453543.

9. REFERENCES

[1] Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang. Detecting data errors: Where are we and what needs to be done? *PVLDB*, 9(12):993–1004, 2016.

[2] S. Abiteboul, S. Cluet, T. Milo, P. Mogilevsky, J. Siméon, and S. Zohar. Tools for Data Translation and Integration. *IEEE Data Eng. Bull.*, 22(1):3–8, 1999.

[3] A. Abouzied, J. Hellerstein, and A. Silberschatz. DataPlay: interactive tweaking and example-driven correction of graphical database queries. In *UIST*, pages 207–218, 2012.

[4] Accounting mistake could mean new tax bills for twin falls county. `http://magicvalley.com/news/local/accounting-mistake-could-mean-new-tax-bills-for-twin-falls/article_24904ffe-003d-51d0-b536-60d9210e2116.html`.

[5] T. Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[6] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *PVLDB*, 5(4):346–357, 2011.

[7] Annual report of the california DUI management information system. `https://www.dmv.ca.gov/portal/wcm/connect/ea06d0a4-a73f-4b2d-b6f1-257029275629/S5-246.pdf?CACHEID=ea06d0a4-a73f-4b2d-b6f1-257029275629`, 2014.

[8] A. Atamtürk and M. W. Savelsbergh. Integer-programming software systems. *Annals of Operations Research*, 140(1):67–124, 2005.

[9] K. A. Barchard and L. A. Pace. Preventing human error: The impact of data entry methods on data accuracy and statistical results. *Computers in Human Behavior*, 27(5):1834 – 1839, 2011.

[10] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali. *Linear Programming and Network Flows*. John Wiley & Sons, 2004.

[11] G. Bender, L. Kot, and J. Gehrke. Explainable security for relational databases. In *SIGMOD*, pages 1411–1422, 2014.

[12] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the Repairs of Functional Dependency Violations Under Hard Constraints. *PVLDB*, 3(1-2):197–207, 2010.

[13] Betterment. `https://www.betterment.com/`.

[14] Boston police to update traffic stop database. `http://www.govtech.com/public-safety/Boston-Police-to-Update-Traffic-Stop-Database.html`.

[15] P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[16] California department of motor vehicles. `https://www.dmv.ca.gov`.

[17] A. Chalamalla, I. F. Ilyas, M. Ouzzani, and P. Papotti. Descriptive and prescriptive data cleaning. In *SIGMOD*, pages 445–456, 2014.

[18] A. Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.

[19] Chase online banking. `https://www.chase.com/online/digital/online-banking.html`.

[20] S. Chen, X. L. Dong, L. V. Lakshmanan, and D. Srivastava. We Challenge You to Certify Your Updates. In *SIGMOD*, pages 481–492, 2011.

[21] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[22] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, pages 458–469, 2013.

[23] City of buffalo overcharged $1 million by power company. `http://www.buffalonews.com/business/city-of-buffalo-overcharged-1-million-by-power-company-20160209`.

[24] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, pages 315–326, 2007.

[25] T. T. P. Council. Benchmark C: Standard specification (revision 5.11.0), 2010.

[26] I. CPLEX. High-performance software for mathematical programming and optimization, 2005.

[27] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *TODS*, 25(2):179–227, 2000.

[28] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: A Commodity Data Cleaning System. In *SIGMOD*, pages 541–552, 2013.

[29] E. Danna, E. Rothberg, and C. Le Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming*, 102(1):71–90, 2005.

[30] M. Das, S. Amer-Yahia, G. Das, and C. Yu. MRI: Meaningful Interpretations of Collaborative Ratings. *PVLDB*, 4(11):1063–1074, 2011.

[31] Data entry is a top cause of medication errors. `http://www.amednews.com/article/20050124/profession/301249959/4`.

[32] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB*, 7(4):277–288, 2013.

[33] X. L. Dong and F. Naumann. Data fusion–Resolving data conflicts for integration. *PVLDB*, 2(2):1654–1655, 2009.

[34] D. Fabbri and K. LeFevre. Explanation-based auditing. *PVLDB*, 5(1):1–12, 2011.

[35] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(2):6:1–6:48, 2008.

[36] C. A. Floudas and X. Lin. Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications. *Annals of Operations Research*, 139(1):131–162, 2005.

[37] M. Galar, A. Fernandez, E. Barrenechea, H. Bustince, and F. Herrera. A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(4):463–484, 2012.

[38] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. AJAX: An Extensible Data Cleaning Tool. In *SIGMOD*, 2000.

[39] J. P. Garcia-Sabater, J. Maheut, and J. J. Garcia-Sabater. A decision support system for aggregate production planning based on MILP: A case study from the automotive industry. In *CIE*, pages 366–371, 2009.

[40] J. L. Gearhart, K. L. Adair, R. J. Detry, J. D. Durfee, K. A. Jones, and N. Martin. Comparison of open-source linear programming solvers. Technical report, Sandia National Laboratories, 2013.

[41] K. E. Gebaly, P. Agrawal, L. Golab, F. Korn, and D. Srivastava. Interpretable and Informative Explanations of Outcomes. *PVLDB*, 8(1):61–72, 2014.

[42] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: Hands-off Crowdsourcing for Entity Matching. In *SIGMOD*, pages 601–612, 2014.

[43] L. Golab, H. Karloff, F. Korn, D. Srivastava, and B. Yu. On Generating Near-optimal Tableaux for Conditional Functional Dependencies. *PVLDB*, 1(1):376–390, 2008.

[44] L. Golab, H. J. Karloff, F. Korn, and D. Srivastava. Data Auditor: Exploring Data Quality and Semantics using Pattern Tableaux. *PVLDB*, 3(2):1641–1644, 2010.

[45] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[46] A. Gruenheid, X. L. Dong, and D. Srivastava. Incremental Record Linkage. *PVLDB*, 7(9):697–708, 2014.

[47] D. Haas, J. Wang, E. Wu, and M. J. Franklin. CLAMShell: Speeding Up Crowds for Low-latency Data Labeling. *PVLDB*, 9(4):372–383, 2015.

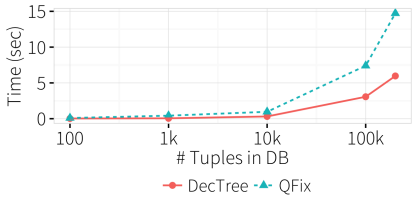
[48] H. He and E. A. Garcia. Learning from Imbalanced Data. *TKDE*, 21(9):1263–1284, 2009.

[49] How to object to a CRA notice of assessment. http://www.newsoptimist.ca/opinion/columnists/how-to-object-to-a-cra-notice-of-assessment-1.2280680.
[50] I Quant NY. http://iquantny.tumblr.com/.
[51] D. V. Kalashnikov and S. Mehrotra. Domain-independent Data Cleaning via Analysis of Entity-relationship Graph. *TODS*, 31(2):716–767, 2006.
[52] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *CHI*, pages 3363–3372, 2011.
[53] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. In *TVCG*, pages 2917–2926, 2012.
[54] N. Khoussainova, M. Balazinska, and D. Suciu. Towards Correcting Input Data Errors Probabilistically Using Integrity Constraints. In *MobiDE*, pages 43–50, 2006.
[55] N. Khoussainova, M. Balazinska, and D. Suciu. PerfXplain: debugging MapReduce job performance. *PVLDB*, 5(7):598–609, 2012.
[56] N. Koudas, S. Sarawagi, and D. Srivastava. Record Linkage: Similarity Measures and Algorithms. In *SIGMOD*, pages 802–803, 2006.
[57] S. Krishnan, D. Haas, M. J. Franklin, and E. Wu. Towards reliable interactive data cleaning: A user survey and recommendations. In *HILDA*, pages 9:1–9:5, 2016.
[58] Y. Kuwata and J. P. How. Receding horizon implementation of MILP for vehicle guidance. In *ACC*, pages 2684–2685, 2005.
[59] E. Lopez-Milan and L. M. Pla-Aragones. A decision support system to manage the supply chain of sugar cane. *Annals of Operations Research*, 219(1):285–297, 2013.
[60] C.-S. Ma and R. H. Miller. MILP optimal path planning for real-time applications. In *ACC*, pages 6 pp.–, 2006.
[61] Manage the modern workforce with our HCM software. http://go.sap.com/solution/lob/human-resources-hcm.html.
[62] Marilyn mosby's office got two names wrong when filing charges. http://www.mediaite.com/tv/marilyn-mosbys-office-got-two-names-wrong-when-filing-charges-cnn-panel-piles-on/.
[63] B. Meindl and M. Templ. Analysis of commercial and free and open source solvers for linear optimization problems. In *Eurostat*, 2012.
[64] A. Meliou, W. Gatterbauer, S. Nath, and D. Suciu. Tracing data errors with view-conditioned causality. In *SIGMOD*, pages 505–516, 2011.
[65] A. Meliou and D. Suciu. Tiresias: The Database Oracle for How-to Queries. In *SIGMOD*, pages 337–348, 2012.
[66] New systems for government revenue management. https://tax.thomsonreuters.com/new-systems-for-government-revenue-management.
[67] Oakland unified makes $7.6M accounting error in budget; asking schools not to count on it. http://oaklandlocal.com/article/oakland-unified-makes-76-million-accounting-error-budget-asking-schools-not-count-it.
[68] Open data reveals $791 million error in newly adopted NYC budget. http://iquantny.tumblr.com/post/147446103684/open-data-reveals-791-million-error-in-newly, 2016.
[69] I. Quesada and I. E. Grossmann. An LP/NLP based branch and bound algorithm for convex MINLP optimization problems. *Computers & Chemical Engineering*, 16(10-11):937–947, 1992.
[70] J. R. Quinlan. Simplifying decision trees. *International journal of man-machine studies*, 27(3):221–234, 1987.
[71] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.
[72] V. Raman and J. M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *VLDB*, pages 381–390, 2001.
[73] A. Richards, T. Schouwenaars, J. P. How, and E. Feron. Spacecraft trajectory planning with avoidance constraints using mixed-integer linear programming. *Journal of Guidance, Control, and Dynamics*, 25(4):755–764, 2002.
[74] E. Rothberg. An evolutionary algorithm for polishing mixed integer programming solutions. *INFORMS J. on Computing*, 19(4):534–541, 2007.
[75] S. Roy and D. Suciu. A Formal Approach to Finding Explanations for Database Queries. In *SIGMOD*, pages 1579–1590, 2014.
[76] M. Sakal and L. Raković. Errors in Building and Using Electronic Tables: Financial Consequences and Minimisation Techniques. *Strategic Management*, 17(3):29–35, 2012.
[77] Salesforce. https://www.salesforce.com/.
[78] M. W. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6(4):445–454, 1994.
[79] T. Sawik. Mixed integer programming for scheduling flexible flow lines with limited intermediate buffers. *Mathematical and Computer Modelling*, 31(13):39–52, 2000.
[80] T. Sawik. *Scheduling in Supply Chains Using Mixed Integer Programming*. John Wiley & Sons, 2011.
[81] Stock selloff may have been triggered by a trader error. http://www.cnbc.com/id/36999483.
[82] R. A. Stubbs and S. Mehrotra. A branch-and-cut method for 0-1 mixed convex programming. *Mathematical programming*, 86(3):515–532, 1999.
[83] B. ten Cate, C. Civili, E. Sherkhonov, and W. C. Tan. High-Level Why-Not Explanations using Ontologies. In *PODS*, pages 31–43, 2015.
[84] S. Thirumuruganathan, M. Das, S. Desai, S. Amer-Yahia, G. Das, and C. Yu. MapRat: Meaningful Explanation, Interactive Exploration and Geo-Visualization of Collaborative Ratings. *PVLDB*, 5(12):1986–1989, 2012.
[85] Q. T. Tran and C.-Y. Chan. How to ConQueR Why-not Questions. In *SIGMOD*, pages 15–26, 2010.
[86] K. Tzompanaki, N. Bidoit, and M. Herschel. Semi-automatic SQL debugging and fixing to solve the missing-answers problem. In *VLDB PhD Workshop*, 2014.
[87] X. Wang, X. L. Dong, and A. Meliou. Data X-Ray: A diagnostic tool for data errors. In *SIGMOD*, pages 1231–1245, 2015.
[88] A. Wolski. Tatp benchmark description (version 1.0), 2009.
[89] E. Wu and S. Madden. Scorpion: Explaining Away Outliers in Aggregate Queries. *PVLDB*, 6(8):553–564, 2013.
[90] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided Data Repair. *PVLDB*, 4(5):279–289, 2011.
[91] J. Yates. Data entry error wipes out life insurance coverage. In *Chicago Tribune*, 2005.
[92] X. Yin, J. Han, and P. S. Yu. Truth Discovery with Multiple Conflicting Information Providers on the Web. *TKDE*, 20(6):796–808, 2008.
[93] B. Zhao, B. I. P. Rubinstein, J. Gemmell, and J. Han. A Bayesian Approach to Discovering Truth from Conflicting Sources for Data Integration. *PVLDB*, 5(6):550–561, 2012.
[94] M. M. Zloof. Query-by-example: A data base language. *IBM systems Journal*, 16(4):324–343, 1977.

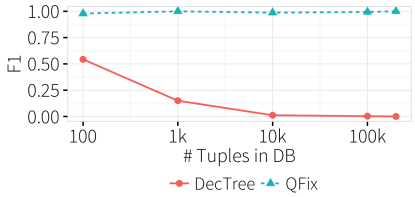
APPENDIX

A. A LEARNING-BASED APPROACH

A drawback of the MILP approach is that the generated models grow with the size of the database and query log. However, we argue that the encoded information is necessary in order to generate a sufficient set of constraints that result in a good repair. In this section, we examine an alternative, simpler, decision tree-based approach called DecTree. We show that even in a simple case of a single query log and a complete complaint set, it is expected to perform poorly. We will first describe how to model the repair process us-



(a) Comparison on Performance.



(b) Comparison on Accuracy.

Figure 10: DecTree compared with QFix

ing a decision tree, and then we will present and discuss experimental results that illustrate its limitations.

A.1 Modeling Repairs with Decision Trees

Rule-based learners are used in classification tasks to generate a set of rules, or conjunctive predicates that best classify a group of labeled tuples. The rules are non-overlapping, and each is associated with a label—a tuple that matches a given rule is assigned the corresponding label. These rules exhibit a natural parallel with SQL `WHERE` clauses, which can be viewed as labeling selected tuples with a positive label and rejected tuples with a negative label. Similarly, the structure of the rules is identical to those that QFix is designed to repair. Thus, given the database tuples labeled to describe the errors, we may use a rule-based learner to generate the most appropriate `WHERE` clause. We focus our attention on rule-based learners; specifically, we experiment with the C4.5 [70] decision tree learner, which is an exemplar of rule-based learners.

A core limitation of this classification-based approach is that there is no means to repair `SET` clauses, which modify data values rather than simply label them. We resolve this with a two step approach. We first use the decision tree to generate a repair for the `WHERE` clause, and then use the modified query to identify repairs for the `SET` clause. The need for this two step procedure limits this approach to encoding and repairing at most one query at a time.

Repairing the WHERE Clause: The `WHERE` clause of an update query is equivalent to a rule-based binary classifier that splits tuples into two groups: (1) tuples that satisfy the conditions in the `WHERE` clause and (2) tuples that do not. A mistake in a query predicate can cause a subset of the tuples to be misclassified, and in turn, translate into data errors. Therefore, repairing the complaints corresponds to repairing the imprecise classification.

The repair works as follows: For an incorrect query q, let D_0 be the database state before q, and D_1^* the *correct* database state that should have been the result after q, if q were correct. We use each tuple $t \in D_0$ as an element in the input training data for the classifier where the values (of each attribute) of t define the feature vector and the label for t:

$$label(t) = \begin{cases} true & \text{if } D_0.t \neq D_1^*.t \\ false & \text{otherwise} \end{cases}$$

The `true` rules generated by the decision tree trained on this labeled dataset forms a disjunction of rules that constitute the repaired `WHERE` clause.

Repairing the SET Clause: The `WHERE` clause repair proposed by the classifier may not completely repair the complaints if there was also an error in the `SET` clause. In this case, we execute a second repair step.

We model the errors as a simple linear system of equations: each expression in the `SET` clause is translated into a linear equation in the same fashion as described in Section 3. Directly solving the system of equations for the undetermined variables will generate the desired repair for the `SET` expression.

A.2 Experimental Results

To illustrate these shortcomings, we compare DecTree with QFix using a simplified version of the setup from Section 6 that favors DecTree. We restrict the query log to contain a single query that is corrupted, use a complete complaint set and vary the database size. We use the following query template, where all `SET` clauses assign the attributes to constants, and the `WHERE` clauses consist of range predicates:

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

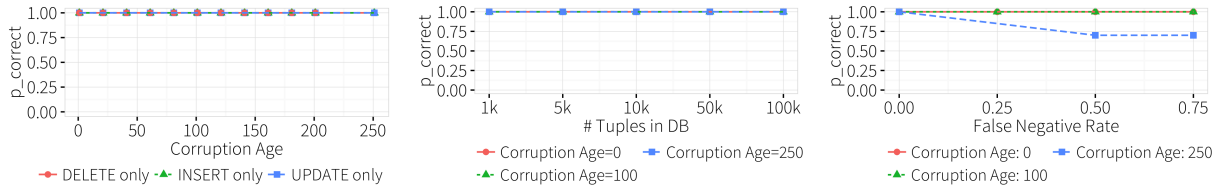
Figure 10a shows that although the runtime performance of DecTree is better than QFix by small a constant factor ($\sim 2.5\times$), both runtimes degrade exponentially. In addition, the DecTree repairs are effectively unusable as their accuracy is low: the F1-score starts at 0.5 and rapidly degrades towards 0. From these empirical results, we find that DecTree generates low-quality repairs even under the simplest conditions—an approach that applies DecTree over more queries is expected to have little hope of succeeding.

There are three important reasons why DecTree, and any approach that focuses on a single query at a time[3], will not perform well.

Single Query Limitation. In principle, one could attempt to apply this technique to the entire log one query at a time, starting from the most recent query. Even ignoring the low repair accuracy shown in Figure 10b, this approach is infeasible. Consider that we generate a labeled training dataset to repair q_i using the query's input and output database states D_{i-1} and D_i^*. Note that D_i^* is the theoretically *correct* database state assuming no errors in the query log. We would need to derive D_i^* by applying the complaint set to D_n to create D_n^*, and roll back the database state. Unfortunately, `UPDATE` queries are commonly surjective such that their inverses are ambiguous, which means that it is often impossible to derive D_i^*. In contrast, the incremental version of QFix can bypass this problem by encoding subsequent queries in the log in a MILP representation.

Structurally Different `WHERE` Clause Results. The basic classifier approach simply learns a set of rules to minimize classification error, and can derive a clause whose struture is arbitrarily different from the original query's `WHERE` clause. Although it may be possible to incorporate a distance measure as part of the decision tree splitting criteria, it is likely to be a heuristic with no guarantees.

[3]Although our incremental approach tries to generate a repair for a single query at a time, it encodes all subsequent queries in the log.



(a) Correct repair ratio vs. query types (b) Correct repair ratio vs. database sizes (c) Top-k vs. query types.

Figure 11: QFix maintains 1.0 correct repair ratio with complete complaint set.

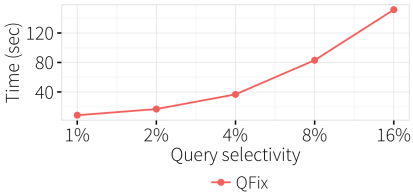


Figure 12: Increasing query selectivity leads to longer QFix execution time.

High Selectivity, Low Precision. Classifiers try to avoid overfitting by balancing the complexity of the rules with classification accuracy. This is problematic for highly selective queries (e.g., primary key updates), because the classifier may simply ignore the single incorrect record and generate a rule such as `FALSE`. In fact, this form of severely imbalanced data continues to be a challenge for most major classification algorithms [37, 48]. Thus, we believe that alternative classification algorithms would not improve on these results. Compound with the fact that many workloads are primarily composed of key update queries [32] this issue severely limits the applicability of learning-based approaches.

B. REPAIRING THE CORRECT QUERY

Our experiments in Section 6 measure the accuracy of QFix based on the effect of the repairs to the dataset. However, it is possible to correctly resolve the reported complaints by modifying a query that was not corrupted. Here we augment our evaluation to study how often QFix chooses the right query to repair. For each setting, we compute the *correct repair ratio* $p_{correct}$ over 20 runs: $p_{correct}$ is the portion of runs where QFix chose the correct query to repair.

In Figure 11a, we evaluate SingleQueryFix with three types of workloads (`UPDATE`, `INSERT`, and `DELETE`) over increasing corruption age: corruption age 1 means that the most recent query was corrupted, while corruption age 250 means that the corruption occurred 250 queries in the past. In this experiment, we assume complete knowledge of the complaint set. QFix selects the correct query to repair in all cases. We next focus on the `UPDATE` workload and three ages of corruption, while increasing the database size. Again, QFix is accurate in every case (Figure 11b).

Finally, we study the effect of incomplete complaints in Figure 11c. Increasing the false negative rate means that more complaints are not reported. Similar to the precision and recall performance in Figure 8b, $p_{correct}$ also drops for problems with old corruptions and high false negative rates. This is expected since with insufficient information, there is a larger number of queries that offer valid fixes and QFix simply chooses the one with the lowest objective.

C. QUERY INTERACTIONS

One of the challenges that motivates our work on QFix is that the progression of queries in the log obscures and propagates errors, as subsequent queries interact with tuples affected by a prior erroneous query. Our synthetic data generator (Section 6.1) does not directly control the degree of interaction among queries in the log, but parameters such as the number of attributes and the dataset size impact this directly. Here we compute the probability that any two tuples in the log interact, through the random generation process. We assume two `UPDATE` queries, q_i and q_j, with range `WHERE` predicates. The probability that q_i and q_j interact (i.e., the intersection of the tuples they update is non-empty) is:

$$Pr(q_i \cap q_j \neq \emptyset) = \\ Pr(\sigma_{q_i}.A = \sigma_{q_j}.A)Pr(q_i \cap q_j \neq \emptyset | \sigma_{q_i}.A = \sigma_{q_j}.A) \\ + Pr(\sigma_{q_i}.A \neq \sigma_{q_j}.A)Pr(q_i \cap q_j \neq \emptyset | \sigma_{q_i}.A \neq \sigma_{q_j}.A),$$

Here, $Pr(\sigma_{q_i}.A = \sigma_{q_j}.A)$ is the probability that the `WHERE` clauses of the two queries have predicates on the same attribute; $Pr(\sigma_{q_i}.A = \sigma_{q_j}.A) = \frac{1}{N_a}$, where N_a is the number of attributes in the database. The probability that the `WHERE` clause ranges of the two queries intersect is $Pr(q_i \cap q_j \neq \emptyset | \sigma_{q_i}.A = \sigma_{q_j}.A) = 2 * s$, where s is the predicate selectivity. Similarly for the second term, $Pr(\sigma_{q_i}.A \neq \sigma_{q_j}.A) = \frac{N_a - 1}{N_a}$ is the probability that the `WHERE` clauses of q_i and q_j use different attributes. Assuming tuple values are evenly distributed, the number of tuples selected by each query is roughly $n = N_d \cdot s$, where N_d is the number of tuples in the database. Thus, the probability that these two queries update at least one common tuple is $Pr(q_i \cap q_j \neq \emptyset | \sigma_{q_i}.A \neq \sigma_{q_j}.A) = 1 - \frac{C(N_d - n, n)}{C(N_d, n)}$, where C computes the combinations $C(n, r) = \frac{n!}{r!(n-r)!}$. Therefore, the probability that two queries in our dataset interact is:

$$Pr(q_i \cap q_j \neq \emptyset) = \frac{2s}{N_a} + \frac{N_a - 1}{N_a} \cdot (1 - \frac{C(N_d - n, n)}{C(N_d, n)}), \quad (8)$$

Where $n = N_d \cdot s$. Based on this result, the probability that any two queries interact with each other with the default parameter settings of our generator is about 0.31.

In Section 6.3.2, we study the influence of the number of attributes N_a on QFix performance (Figure 7a) and observe that more attributes result in faster execution time with all optimizations. This is expected as $Pr(q_i \cap q_j \neq \emptyset)$ is inversely proportional to the number of attributes, which means that there are fewer interactions. Therefore, fewer interactions lead to faster runtimes.

We also evaluate QFix over different query selectivities. In Figure 12, we observe that the execution time of QFix increases with higher query selectivity, which results in higher query interaction.