Towards Dependable Data Repairing with Fixing Rules

Jiannan Wang*
UC Berkeley
Berkeley, CA, USA
jnwang@eecs.berkeley.edu

Nan Tang
Qatar Computing Research Institute (QCRI)
Doha, Qatar
ntang@qf.org.qa

ABSTRACT

One of the main challenges that data cleaning systems face is to *automatically* identify and repair data errors in a *dependable* manner. Though data dependencies (*a.k.a.* integrity constraints) have been widely studied to capture errors in data, automated and dependable data repairing on these errors has remained a notoriously hard problem. In this work, we introduce an automated approach for dependably repairing data errors, based on a novel class of *fixing rules*. A fixing rule contains an evidence pattern, a set of negative patterns, and a fact value. The heart of fixing rules is *deterministic*: given a tuple, the evidence pattern and the negative patterns of a fixing rule are combined to precisely capture which attribute is wrong, and the fact indicates how to correct this error. We study several fundamental problems associated with fixing rules, and establish their complexity. We develop efficient algorithms to check whether a set of fixing rules is consistent, and discuss approaches to resolve inconsistent fixing rules. We also devise efficient algorithms for repairing data errors using fixing rules. We experimentally demonstrate that our techniques outperform other automated algorithms in terms of the accuracy of repairing data errors, using both real-life and synthetic data.

1. INTRODUCTION

Data quality is essential to all businesses, which demands dependable data cleaning solutions. Traditionally, data dependencies (*a.k.a.* integrity constraints) have been widely studied to capture errors from semantically related values. However, automated and dependable data repairing on these data errors has remained a notoriously hard problem.

A number of recent research [4, 7, 10] have investigated the data cleaning problem introduced in [2]: repairing is to find another database that is consistent and minimally differs from the original database. They compute a consistent database by using different cost functions for value updates and various heuristics to guide repairing. However, it is known that such heuristics might introduce data errors [19]. In order to ensure that a repair is dependable, users are involved in the process of data repairing [19, 26, 28], which is usually time-consuming and cumbersome.

In response to practical need for automated and dependable data repairing, in this work, we propose new data cleaning algorithms, based on a class of *fixing rules*. Given a tuple, fixing rules are designed to precisely capture which attribute is wrong, and specify what value it should take.

	name	country	capital	city	conf
r_1:	George	China	Beijing	Beijing	SIGMOD
r_2:	Ian	China	Shanghai (Beijing)	Hongkong (Shanghai)	ICDE
r_3:	Peter	China (Japan)	Tokyo	Tokyo	ICDE
r_4:	Mike	Canada	Toronto (Ottawa)	Toronto	VLDB

Figure 1: Database D: an instance of schema Travel

Motivating example. We first illustrate by examples how existing solutions work. We then motivate our approach.

Example 1: Consider a database D of travel records for a research institute, specified by the following schema:

Travel (name, country, capital, city, conf),

where a Travel tuple specifies a person, identified by name, who has traveled to conference (conf), held at the city of the country with capital. One Travel instance is shown in Fig. 1. All errors are highlighted and their correct values are given between brackets. For instance, r_2[capital] = Shanghai is wrong, whose correct value is Beijing. □

We next describe how existing methods work.

Data dependencies. A variety of data dependencies have been used to capture errors in data, from traditional constraints (*e.g.,* functional and inclusion dependencies [7, 9]) to their extensions (*e.g.,* conditional functional dependencies [15]). Suppose that a functional dependency (FD) is specified for the Travel table as:

ϕ_1: Travel ([country] → [capital])

which states that country uniquely determines capital. One can verify that in Fig. 1, the two tuples (r_1, r_2) violate ϕ_1, since they have the same country but carry different capital values, so do (r_1, r_3) and (r_2, r_3).

In order to compute a consistent database *w.r.t.* ϕ_1 with the *minimum* cost (*e.g.,* the number of changes),

*Work done while interning at QCRI.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2610494.

	country	capital
s_1:	China	Beijing
s_2:	Canada	Ottawa
s_3:	Japan	Tokyo

Figure 2: Data D_m of schema Cap

	country	{capital$^-$}	capital$^+$
φ_1:	China	Shanghai, Hongkong	Beijing

	country	{capital$^-$}	capital$^+$
φ_2:	Canada	Toronto	Ottawa

Figure 3: Example fixing rules

many algorithms have been presented [2, 5, 7, 10, 11, 19, 20]. For instance, they can change r_2[capital] from `Shanghai` to `Beijing`, and r_3[capital] from `Tokyo` to `Beijing`, which requires two changes. One may verify that this is a repair with the *minimum* cost of two updates. Though these changes correct the error in r_2[capital], they do not rectify r_3[country]. Worse still, they mess up the correct value in r_3[capital].

User guidance. While using data dependencies to detect errors is appropriate, dependencies on their own are not sufficient to guide dependable data repairing. To improve the accuracy of data repairing, users have been involved [19,23,28] and master data (*a.k.a.* reference data) has been used [19].

Consider a recent work [19] that uses editing rules and master data. Figure 2 shows a master data D_m of schema Cap (country, capital), which is considered to be correct. An editing rule eR_1 defined on two relations (Travel, Cap) is:

$\mathsf{eR}_1 : ((\mathsf{country}, \mathsf{country}) \rightarrow (\mathsf{capital}, \mathsf{capital}), t_{p1}[\mathsf{country}] = ())$

Rule eR_1 states that: for any tuple r in a Travel table, if r[country] is correct and it matches s[country] from a Cap table, we can update r[capital] with the value s[capital] from Cap. For instance, to repair r_2 in Fig. 1, the users need to ensure that r_2[country] is correct, and then match r_2[country] and s_1[country] in the master data, so as to update r_2[capital] to s_1[capital]. It proceeds similarly for the other tuples.

Key challenge & observation. The above examples tell us that data dependencies can detect errors but fall short of automatically guiding data repairing, while involving users is generally cost-ineffective. Hence, one of the main challenges in data cleaning is how to automatically detect and repair errors in a dependable manner.

Data cleaning is not magic; it cannot guess something from nothing. What it does is to make decisions from evidence. Certain *data patterns* of semantically related values can provide evidence to precisely capture and rectify data errors. For example, when values (`China, Shanghai`) for attributes (country, capital) appear in a tuple, it suffices to judge that the tuple is about `China`, and `Shanghai` should be `Beijing`, the capital of `China`. In contrast, the values (`China, Tokyo`) are not enough to decide which value is wrong.

Fixing rules. Motivated by the observation above, in this work, we address the problem of automatically finding dependable repairs, by using *fixing rules*. A fixing rule contains an *evidence pattern*, a set of *negative patterns*, and a *fact* value. Given a tuple, the evidence pattern and the negative patterns of a fixing rule are combined to precisely tell which attribute is wrong, and the fact indicates how to correct it.

Example 2: Figure 3 shows two fixing rules. The brackets mean that the corresponding cell is multivalued.

For the first fixing rule φ_1, its evidence pattern, negative patterns and the fact are `China`, {`Shanghai, Hongkong`}, and `Beijing`, respectively. It states that for a tuple t, if its country is `China` and its capital is either `Shanghai` or `Hongkong`, capital should be updated to `Beijing`. For instance, consider the database in Fig. 1. Rule φ_1 detects that r_2[capital] is wrong, since r_2[country] is `China`, but r_2[capital] is `Shanghai`. It will then update r_2[capital] to `Beijing`.

Similarly, the second fixing rule φ_2 states that for a tuple t, if its country is `Canada`, but its capital is `Toronto`, then its capital is wrong and should be `Ottawa`. It detects that r_4[capital] is wrong, and then will correct it to `Ottawa`.

After applying φ_1 and φ_2, two errors, r_2[capital] and r_4[capital], can be repaired. The other two errors, r_2[city] and r_3[country], still remain. We will discuss later how they are repaired, when more fixing rules are available. □

Remark. Fixing rules are designed to *both* capture semantic errors for specific domains (*e.g.,* (`China, Shanghai`) is an error for (country, capital)), *and* specify how to fix it (*e.g.,* change `Shanghai` to `Beijing`), in a deterministic and dependable manner. They are also conservative: they tend to avoid repairing ambiguous errors such as (`China, Tokyo`), which is also difficult for users to repair since it could be either (`China, Beijing`) or (`Japan, Tokyo`).

Contributions. We propose a framework for automatically and dependably repairing data errors.

(1) We formally define fixing rules and their repairing semantics (Section 3). Given a tuple t, fixing rules tell us which attribute is wrong and what value it should take.

(2) We study fundamental problems of fixing rules (Section 4). Specifically, given a set Σ of fixing rules, we determine whether these rules have conflicts. We show that this problem is in PTIME. We also study the problem of whether some other fixing rules are implied by Σ. We show that this problem is coNP-complete, but it is down to PTIME when the relation schema is fixed.

(3) We develop efficient algorithms to check whether a set of fixing rules is consistent *i.e.,* conflict-free (Section 5). We also discuss solutions to resolve inconsistent fixing rules.

(4) We propose two repairing algorithms for a given set Σ of fixing rules (Section 6). The first algorithm is chase-based. It runs in $\mathcal{O}(\mathsf{size}(\Sigma)|R|)$ for one tuple, where $|R|$ is the cardinality of R and $\mathsf{size}(\Sigma)$ is the size of Σ. The second one is a fast linear algorithm that runs in $\mathcal{O}(\mathsf{size}(\Sigma))$ for one tuple, by interweaving inverted lists and hash counters.

(5) We experimentally verify the effectiveness and scalability of the proposed algorithms (Section 7). We find that algorithms with fixing rules can repair data with high precision. In addition, they scale well with the number of fixing rules. One natural concern is how to generate fixing rules. Inspired by the work of [27], we show how a large number of fixing rules can be obtained from examples.

Organization. Section 2 discusses related work. Section 3 introduces fixing rules. Section 4 studies fundamental problems for fixing rules. Section 5 describes algorithms to check consistency of fixing rules and ways to resolve inconsistent rules. Section 6 presents repairing algorithms using fixing rules. Section 7 reports our experimental findings, followed by conclusion in Section 8.

2. RELATED WORK

Despite the need for dependable algorithms to automatically repair data, there has been little discussion about data cleaning solutions that can *both* capture semantic data errors, *and* explicitly specify an action to correct these errors, *without* interacting with users, and *without* any assumption about confidence values placed on the data.

In recent years, there has been an increasing amount of literature on using data dependencies in cleaning data (*e.g.,* [2, 8, 9, 15, 16, 22]; see [14] for a survey). They have been revisited to better capture data errors as violations of these dependencies (*e.g.,* CFDs [15] and CINDs [8]). As remarked earlier, fixing rules differ from those dependencies in that fixing rules can *not only* detect semantic errors, *but also* explicitly specify how to fix these errors.

Editing rules [19] have been introduced for the process of data monitoring to repair data that is guaranteed correct. However, editing rules require users to examine every tuple, which is expensive. Fixing rules differ from them in that they do not depend on users to trigger repairing operations. Instead, fixing rules use both evidence pattern and negative patterns to automatically trigger repairing operations.

Data repairing algorithms have been proposed [6,7,10–13, 18–20, 23, 28]. Heuristic methods are developed in [5, 7, 11, 20], based on FDs [5, 22], FDs and INDs [7], CFDs [15], CFDs and MDs [18], denial constraints [10] and editing rules [20]. Some works employ confidence values placed by users to guide a repairing process [7, 11, 18] or use master data [19]. Statistical inference is studied in [23] to derive missing values, and in [6] to find possible repairs. To ensure the accuracy of generated repairs, [19, 23, 28] require to consult users. In contrast to these prior art, (1) fixing rules are more conservative to repair data, which target at determinism and dependability, instead of computing a consistent database; (2) we neither consult the users, nor assume the confidence values placed by the users. Indeed, our method can be treated as a complementary technique to heuristic methods *i.e.,* one may compute dependable repairs first and then use heuristic solutions to find a consistent database.

There has also been a lot of work on more general data cleaning: data transformation, which brings the data under a single common schema [24]. ETL tools (see [3, 21] for a survey) provide sophisticated data transformation methods, which can be employed to merge data sets and repair data based on reference data. Some recent work has been studied for semantic transformations [27] of strings. However, they are designed for value transformation instead of capturing semantic errors as fixing rules do. Hence, they can be treated as an orthogonal technique which prepares data that is in turn to be repaired by other data cleaning approaches.

3. FIXING RULES

In this section, We first give the formal definition of fixing rules and their semantics (Section 3.1). We then describe the repairing semantics for a set of fixing rules (Section 3.2).

3.1 Definition

Consider a schema R defined over a set of attributes, denoted by $\mathsf{attr}(R)$. We use $A \in R$ to denote that A is an attribute in $\mathsf{attr}(R)$. For each attribute $A \in R$, its domain is specified in R, denoted as $\mathsf{dom}(A)$.

Syntax. A *fixing rule* φ defined on a schema R is formalized as $((X, t_p[X]), (B, T_p^-[B])) \rightarrow t_p^+[B]$ where

1. X is a set of attributes in $\mathsf{attr}(R)$, and B is an attribute in $\mathsf{attr}(R) \setminus X$ (*i.e.,* B is not in X);
2. $t_p[X]$ is a pattern with attributes in X, referred to as the *evidence pattern* on X, and for each $A \in X$, $t_p[A]$ is a constant value in $\mathsf{dom}(A)$;
3. $T_p^-[B]$ is a finite set of constants in $\mathsf{dom}(B)$, referred to as the *negative patterns* of B; and
4. $t_p^+[B]$ is a constant value in $\mathsf{dom}(B) \setminus T_p^-[B]$, referred to as the *fact* of B.

Intuitively, the evidence pattern $t_p[X]$ of X, together with the negative patterns $T_p^-[B]$ impose the condition to determine whether a tuple contains an error on B. The fact $t_p^+[B]$ in turn indicates how to correct this error.

Note that condition (4) enforces that the correct value (*i.e.,* the fact) is different from known wrong values (*i.e.,* negative patterns) relative to a specific evidence pattern.

We say that a tuple t of R *matches* a rule $\varphi : ((X, t_p[X]), (B, T_p^-[B])) \rightarrow t_p^+[B]$, denoted by $t \vdash \varphi$, if (i) $t[X] = t_p[X]$ and (ii) $t[B] \in T_p^-[B]$. In other words, tuple t matches rule φ indicates that φ can identify errors in t.

Example 3: Consider the fixing rules in Fig. 3. They can be formally expressed as follows:

φ_1: ((country, [China]), (capital, {Shanghai, Hongkong})) → Beijing

φ_2: ((country, [Canada]), (capital, {Toronto})) → Ottawa

In both φ_1 and φ_2, X consists of country and B is capital. Here, φ_1 states that, if the country of a tuple is China and its capital value is in {Shanghai, Hongkong}, its capital value is wrong and should be updated to Beijing. Similarly for φ_2.

Consider D in Fig. 1. Tuple r_1 does not match rule φ_1, since r_1[country] = China but r_1[capital] $\notin$ {Shanghai, Hongkong}. As another example, tuple r_2 matches rule φ_1, since r_2[country] = China, and r_2[capital] $\in$ {Shanghai, Hongkong}. Similarly, we have r_4 matches φ_2. □

Semantics. We next give the semantics of fixing rules.

We say that a fixing rule φ is *applied* to a tuple t, denoted by $t \rightarrow_\varphi t'$, if (i) t matches φ (*i.e.,* $t \vdash \varphi$), and (ii) t' is obtained by the update $t[B] := t_p^+[B]$.

That is, if $t[X]$ agrees with $t_p[X]$, and $t[B]$ appears in the set $T_p^-[B]$, then we assign $t_p^+[B]$ to $t[B]$. Intuitively, if $t[X]$ matches $t_p[X]$ and $t[B]$ matches some value in $T_p^-[B]$, it is evident to judge that $t[B]$ is wrong and we can use the fact $t_p^+[B]$ to update $t[B]$. This yields an updated tuple t' with $t'[B] = t_p^+[B]$ and $t'[R \setminus \{B\}] = t[R \setminus \{B\}]$.

Example 4: As shown in Example 2, we can correct r_2 by applying rule φ_1. As a result, r_2[capital] is changed from Shanghai to Beijing, *i.e.,* $r_2 \rightarrow_{\varphi_1} r_2'$ where r_2'[capital] = Beijing and the other attributes of r_2' remain unchanged.

Similarly, we have $r_4 \rightarrow_{\varphi_2} r_4'$ where the only updated attribute value is r_4'[capital] = Ottawa. □

Remark. (1) Fixing rules are different from traditional data dependencies *e.g.,* FDs [1] and CFDs [15]. Data dependencies only detect violations. In contrast, a fixing rule φ specifies an action: applying φ to a tuple t yields an updated t'.

(2) Editing rules [19] also have *dynamic* semantics. However,

they differ in the way of repairing errors. (a) *Editing rules* need users to trigger the action of repairing. That is, when matching some values from dirty data to values in master data, editing rules by themselves cannot tell if the values used for matching are correct, without which the repairing operation cannot be executed. (b) *Fixing rules* encode evidence pattern and negative patterns to decide the correct and erroneous values, which then *automatically* triggers the repair operation. Please see Example 2 for more details.

(3) We have also investigated on how to get fixing rules. Inspired by the work of [27] that learns transformation rules from examples, we discuss in Section 7 how to generate fixing rules from examples.

Notations. For convenience, we introduce some notations. Given fixing rule $\varphi : ((X, t_p[X]), (B, T_p^-[B])) \rightarrow t_p^+[B]$, we denote by X_φ the set X of attributes in φ. Similarly, we write $t_p[X_\varphi]$, B_φ, $T_p^-[B_\varphi]$ and $t_p^+[B_\varphi]$, relative to φ.

3.2 Repairing Semantics with Fixing Rules

We next describe the semantics of applying a set of fixing rules. Note that when applying a fixing rule φ to a tuple t, we update $t[B_\varphi]$ with $t_p^+[B_\varphi]$. To ensure that the change makes sense, the corrected values should remain unchanged in the following process. That is, after applying φ to t, the set $X_\varphi \cup \{B_\varphi\}$ of attributes should be *marked* as correct.

In order to keep track of the set of attributes that has been marked correct, we introduce the notion *assured attributes* to represent them, denoted by $\mathcal{A}_t$ relative to tuple t. We simply write $\mathcal{A}$ when t is clear from the context.

We say that a fixing rule φ is *properly applied* to a tuple t *w.r.t.* the assured attributes $\mathcal{A}$, denoted by $t \rightarrow_{(\mathcal{A},\varphi)} t'$, if (i) t matches φ, and (ii) $B_\varphi \notin \mathcal{A}$.

That is, it is justified that to apply φ to t, for those t match φ, is correct. As $\mathcal{A}$ has been assured, we do not allow it to be changed by enforcing $B_\varphi \notin \mathcal{A}$ (condition (ii)).

Example 5: Consider the fixing rule φ_1 in Example 3 and the tuple r_2 in Fig. 1. Initially, $\mathcal{A}_{r_2} = \emptyset$. The rule φ_1 can be properly applied to r_2 *w.r.t.* $\mathcal{A}_{r_2}$, since r_2[country] = China and r_2[capital] = Shanghai $\in$ {Shanghai, Hongkong} (*i.e.,* r_2 matches φ_1); and moreover, capital $\notin \mathcal{A}_{r_2}$. This yields an updated tuple r_2' where r_2'[capital] = Beijing. □

Observe that if $t \rightarrow_{(\mathcal{A},\varphi)} t'$, then X_φ and B_φ will also be marked correct. Thus, the assured attributes $\mathcal{A}$ should be extended as well, to become $\mathcal{A} \cup X_\varphi \cup \{B_\varphi\}$.

Example 6: Consider Example 5. After φ_1 is applied to r_2, the assured attribute $\mathcal{A}_{r_2}$ will be expanded correspondingly, by including X_{φ_1} (*i.e.,* {country}) and $\{B_{\varphi_1}\}$ (*i.e.,* {capital}), which results in an expanded assured attribute set $\mathcal{A}_{r_2}$ = {country, capital}. □

We write $t \xrightarrow{=}_{(\mathcal{A},\varphi)} t$ if φ cannot be properly applied to t, *i.e.,* t is unchanged by φ relative to $\mathcal{A}$, if either t does not match φ, or $B_\varphi \in \mathcal{A}$.

Consider a set Σ of fixing rules defined on R. Given a tuple t of R, we want a *unique fix* of t by using Σ. That is, no matter in which order the fixing rules of Σ are properly applied, Σ yields a unique t' by updating t.

To formalize the notion of unique fixes, we first recall the repairing semantics of fixing rules. Notably, if φ is *properly applied* to t via $t \rightarrow_{(\mathcal{A},\varphi)} t'$ *w.r.t.* assured attributes $\mathcal{A}$, it yields an updated t' where $t[B_\varphi] \in T_p^-[B_\varphi]$ and $t'[B_\varphi] = t_p^+[B_\varphi]$. More specifically, the fixing rule φ first identifies $t[B_\varphi]$ as incorrect, and as a logical consequence of the application of φ, $t[B_\varphi]$ will be updated to $t_p^+[B_\varphi]$, as a validated correct value in t'. Once an attribute value $t'[B]$ is validated, we do not allow it to be changed, together with the attributes X_φ that are used as the evidence to assert that $t[B_\varphi]$ is incorrect.

Fixes. We say that a tuple t' is a *fix* of t *w.r.t.* a set Σ of fixing rules, if there exists a finite sequence $t = t_0, t_1, \ldots, t_k = t'$ of tuples of R such that for each $i \in [1, k]$, there exists a $\varphi_i \in \Sigma$ such that (1) $t_{i-1} \rightarrow_{(\mathcal{A}_i,\varphi_i)} t_i$, where $\mathcal{A}_1 = \emptyset$, $\mathcal{A}_i = \mathcal{A}_{i-1} \cup X_{\varphi_i} \cup \{B_{\varphi_i}\}$; and (2) for any $\varphi \in \Sigma$, $t' \xrightarrow{=}_{(\mathcal{A}_k,\varphi)} t'$.

Condition (1) ensures that each step of the process is justified, *i.e.,* a fixing rule is *properly applied*. Condition (2) ensures that t' is a fixpoint and cannot be further updated.

We write $t \xrightarrow{*}_{(\mathcal{A},\Sigma)} t'$ to denote that t' is a fix of t.

Unique fixes. We say that an R tuple t has a *unique fix* by a set Σ of fixing rules if there exists a unique t' such that $t \xrightarrow{*}_{(\emptyset,\Sigma)} t'$.

Example 7: Consider Example 5. Indeed, r_2' is a fix of r_2 *w.r.t.* rules φ_1 and φ_2 in Example 3, since no rule can be properly applied to r_2', given the assured attributes to be {country, capital}.

Moreover, r_2' is also a unique fix, since one cannot get a tuple different from r_2' when trying to apply rules φ_1 and φ_2 on tuple r_2 in other orders. □

4. FUNDAMENTAL PROBLEMS

We next identify fundamental problems associated with fixing rules, and establish their complexity.

4.1 Termination

One natural question associated with rule based data repairing processes is the *termination* problem that determines whether a rule-based process will stop. In fact, it is readily to verify that the *fix* process, by applying fixing rules (see Section 3.2), always terminates.

Consider the following. For a sequence of updates $t_0 \rightarrow_{(\mathcal{A}_1,\varphi_1)} t_1 \ldots \rightarrow_{(\mathcal{A}_i,\varphi_i)} t_i \ldots$, each time a fixing rule φ_i $(i \geq 1)$ is applied as $t_{i-1} \rightarrow_{(\mathcal{A}_i,\varphi_i)} t_i$, the number of validated attributes in $\mathcal{A}$ is *strictly increasing* up to $|R|$, the cardinality of schema R.

4.2 Consistency

The problem is to decide whether a set Σ of fixing rules does not have conflicts. We say that Σ is *consistent* if for any input tuple t of R, t has a unique fix by Σ.

Example 8: Consider a fixing rule φ_1' by adding a negative pattern to the φ_1 in Example 3 as the following:

φ_1': (([country], [China]), (capital, {Shanghai, Hongkong, Tokyo})) → Beijing

The revised rule φ_1' states that, for a tuple, if its country is China and its capital value is Shanghai, Hongkong or Tokyo, its capital is wrong and should be updated to Beijing.

Consider another fixing rule φ_3 as: for t in relation Travel, if the conf is ICDE, held at city Tokyo and capital Tokyo, but the country is China, its country should be updated to Japan. This fixing rule can be formally expressed below:

φ_3: (([capital, city, conf], [Tokyo, Tokyo, ICDE]), (country, {China})) → Japan

We show that these two fixing rules, φ_1' and φ_3, are inconsistent. Consider the tuple r_3 in Fig. 1. Both φ_1' and φ_3 can be applied to r_3. It has the following two fixes:

(1) $r_3 \rightarrow_{(\emptyset,\varphi_1')} r_3'$: it will change attribute r_3[capital] from Tokyo to Beijing. This will result in an updated tuple as:

r_3': (Peter, China, Beijing, Tokyo, ICDE).

It also marks attributes {country, capital} as assured, such that φ_3 cannot be properly applied, *i.e.,* r_3' is a fixpoint.

(2) $r_3 \rightarrow_{(\emptyset,\varphi_3)} r_3''$: it will update r_3[country] from China to Japan. This will yield another updated tuple as:

r_3'': (Peter, Japan, Tokyo, Tokyo, ICDE).

The attributes {country, capital, conf} will be marked as assured, such that φ_1' cannot be properly applied, *i.e.,* r_3'' is also a fixpoint.

Observe that the above two fixes (*i.e.,* r_3' and r_3'') will lead to different fixpoints, where the difference is highlighted above. Therefore, φ_1' and φ_3 are inconsistent. Indeed, r_3' contains errors while r_3'' is correct. □

Consistency problem. The *consistency problem* is to determine, given a set Σ of fixing rules defined on R, whether Σ is consistent.

Intuitively, this is to determine whether the rules in Σ are dirty themselves. The practical need for the consistency analysis is evident: we cannot apply these rules to clean data before Σ is ensured consistent itself.

This problem has been studied for CFDs, MDs, and editing rules. It is known that the consistency problem for MDs [17] is trivial: any set of MDs is consistent [18]. They are NP-complete (resp. coNP-complete) for CFDs [15] (resp. editing rules [19]). We shall show that the problem for fixing rules is PTIME, lower than their editing rules counterparts.

Theorem 1: *The consistency problem of fixing rules is* PTIME. □

We prove Theorem 1 by providing a PTIME algorithm for determining if a set of fixing rules is consistent in Section 5.2.

The low complexity from the consistency analysis tells us that it is feasible to efficiently find consistent fixing rules.

4.3 Implication

Given a set Σ of consistent fixing rules, and another fixing rule φ that is not in Σ, we say that φ is *implied* by Σ, denoted by $\Sigma \models \varphi$, if (*i*) $\Sigma \cup \{\varphi\}$ is consistent; and (*ii*) for any input t where $t \xrightarrow{*}_{\Sigma} t'$ and $t \xrightarrow{*}_{\Sigma\cup\{\varphi\}} t''$, t' and t'' are the same.

Condition (*i*) says that Σ and φ must agree on each other. Condition (*ii*) ensures that for *any* tuple t, applying Σ or $\Sigma \cup \{\varphi\}$ will result in the same updated tuple, which indicates that φ is redundant.

Implication problem. The *implication problem* is to decide, given a set Σ of consistent fixing rules, and another fixing rule φ, whether Σ implies φ.

Intuitively, the implication analysis helps us find and remove redundant rules from Σ, *i.e.,* those that are a logical consequence of other rules in Σ, to improve performance.

No matter how desirable it is to remove redundant rules, unfortunately, the implication problem is coNP-complete.

Theorem 2: *The implication problem of fixing rules is coNP-complete. It is down to* PTIME *when the relation schema R is fixed.* □

Proof sketch: **(A) General case**. *Lower bound*. We show the implication problem is coNP-hard by reduction from the 3SAT problem, which is NP-complete [25], to the complement of the implication problem.

Upper bound. To show it is in coNP, we first establish a small model property: a set Σ of fixing rules is consistent if and only if for any tuple t of R consisting of values appeared in Σ, t has a unique fix by Σ. We then give an NP algorithm to its complement problem that first guesses a tuple t with values appear in Σ and then checks whether t has a unique fix by Σ in PTIME.

(B) Special case: when R is fixed. We show that for fixed R, only a polynomially number of tuples need to be guessed and checked with a PTIME algorithm. Thus it is down to PTIME in this special case. □

Details are omitted due to space constraints.

4.4 Determinism

Determinism problem. The *determinism problem* asks whether all terminating cleaning processes end up with the same repair.

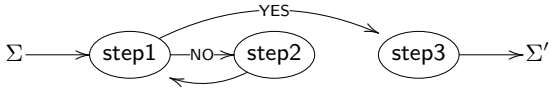
From the definition of consistency of fixing rules, it is trivial to get that, if a set Σ of fixing rules is consistent, for any t of R, applying Σ to t will terminate, and the updated t' is deterministic (*i.e.,* a unique result).

5. ENSURING CONSISTENCY

Our next goal is to study methods for identifying consistent rules. We first describe the workflow for obtaining consistent fixing rules (Section 5.1). We then present algorithms to check whether a given set of rules is consistent (Section 5.2). We also discuss how to resolve inconsistent fixing rules, and ensure the workflow terminates (Section 5.3).

5.1 Overview

Given a set Σ of fixing rules, our workflow contains the following three steps to obtain a set Σ' of fixing rules that is ensured to be consistent.



Step 1: It checks whether the given Σ of fixing rules is consistent. If it is inconsistent, it goes to step (2). Otherwise, it goes to step (3).

Step 2: We allow either an automatic algorithm or experts to examine and resolve inconsistent fixing rules. After some rules are revised, it will go back to step (1).

Step 3: It terminates when the set Σ' of (possibly) modified fixing rules is consistent.

It is desirable that the users are involved in step (2) when resolving inconsistent rules, in order to obtain high quality fixing rules.

5.2 Checking Consistency

We first present a proposition, which is important to design efficient algorithms for checking consistency.

Proposition 3: *For a set Σ of fixing rules, Σ is consistent, iff any two fixing rules φ_i and φ_j in Σ are consistent.* □

Proof sketch: Let n be the number of rules in Σ. When $n = 1$, Σ is trivially consistent. When $n = 2$, Σ is consistent is the same as φ_i and φ_j are consistent ($i \neq j$). When $n \geq 3$, we prove by contradiction.

$\boxed{\Rightarrow}$ Suppose that although the fixing rules are pairwise consistent, when putting together, they are inconsistent. In

other words, they may lead to (at least) two different fixes, *i.e.,* the fixes are not unique. More concretely, there exist (at least) two *non-empty* sequences of fixes as follows:
$S_1 : t = t_0 \rightarrow_{(\emptyset,\varphi_1)} t_1 \cdots \rightarrow_{(\mathcal{A}_{i-1},\varphi_i)} t_i \cdots \rightarrow_{(\mathcal{A}_{m-1},\varphi_m)} t_m = t'$
$S_2 : t = t'_0 \rightarrow_{(\emptyset,\varphi'_1)} t'_1 \cdots \rightarrow_{(\mathcal{A}'_{j-1},\varphi'_j)} t'_j \cdots \rightarrow_{(\mathcal{A}'_{n-1},\varphi'_n)} t'_n = t''$

We consider the following three cases: (*i*) $\mathcal{A}_m \cap \mathcal{A}'_n = \emptyset$; (*ii*) $\mathcal{A}_m \cap \mathcal{A}'_n \neq \emptyset$ and $t'[\mathcal{A}_m \cap \mathcal{A}'_n] = t''[\mathcal{A}_m \cap \mathcal{A}'_n]$; and (*iii*) $\mathcal{A}_m \cap \mathcal{A}'_n \neq \emptyset$ and $t'[\mathcal{A}_m \cap \mathcal{A}'_n] \neq t''[\mathcal{A}_m \cap \mathcal{A}'_n]$, where $\mathcal{A}_m = \mathcal{A}_{m-1} \cup X_{\varphi_m} \cup \{B_{\varphi_m}\}$ and $\mathcal{A}'_n = \mathcal{A}'_{n-1} \cup X_{\varphi'_n} \cup \{B_{\varphi'_n}\}$.

For cases (*i*)(*ii*), we prove that either S_1 or S_2 does not reach a fixpoint, *i.e.,* it is not a *fix*. For case (*iii*), we show that there must exist a φ_i (in sequence S_1) and a φ'_j (in sequence S_2) that are inconsistent.

Putting all contradicting cases (*i,ii,iii*) together, it suffices to see that we were wrong to assume that Σ is inconsistent.

$\boxed{\Leftarrow}$ Assume there exist inconsistent φ_i and φ_j. We show that for any tuple t that leads to different fixes by φ_i and φ_j, we can construct two fixes S'_1 and S'_2 on t by using the rules in Σ. In S'_1, φ_i is applied first; while in S'_2, φ_j is applied first. We prove that these two fixes must yield two different fixpoints. This suffices to show that we were wrong to assume that there exist inconsistent φ_i and φ_j. □

Details are omitted due to space constraints.

Proposition 3 tells us that to determine whether Σ is consistent, it suffices to only check them pairwise. This significantly simplifies the problem and complexity of checking consistency. Next, we describe two algorithms to check the consistency of two fixing rules, by using the result from Proposition 3. One algorithm is based on tuple enumeration, while the other is through rule characterization.

5.2.1 Tuple enumeration

We first consider that whether there exists a finite set of tuples such that it suffices to only inspect these tuples to determine whether rules φ_i and φ_j are consistent or not. That is, for the other tuples, neither φ_i nor φ_j can be applied.

To design an algorithm for tuple enumeration, let's understand what tuples are necessary to be enumerated, and in which cases tuple enumeration can be avoided.

Lemma 4: *Fixing rules φ_i and φ_j are consistent, if there does not exist any tuple t that matches both φ_i and φ_j.* □

Proof. If $\nexists t$ such that $t \vdash \varphi_i$ and $t \vdash \varphi_j$, for any t, there are two cases: *either* no rule can be applied, *or* there exists a unique sequence of applying both rules. Either case will not cause different fixes, *i.e.,* φ_i and φ_j are consistent. □

Note that Lemma 4 is for "if" but not "iff", which tells us that only tuples that draw values from evidence pattern and negative patterns can (possibly) match both rules at the same time. Next we illustrate the tuples that are needed to be generated by an example.

Example 9: Consider rules φ_1 and φ_2 in Example 3. We have two constants in the evidence pattern as {`China, Canada`}, and three constants in the negative patterns as {`Shanghai, Hongkong, Toronto`}. Hence, we only need to enumerate $2 \times 3 = 6$ tuples for relation Travel:

(∘, `China, Shanghai`, ∘, ∘),	(∘, `China, Hongkong`, ∘, ∘)
(∘, `China, Toronto`, ∘, ∘),	(∘, `Canada, Shanghai`, ∘, ∘)
(∘, `Canada, Hongkong`, ∘, ∘),	(∘, `Canada, Toronto`, ∘, ∘)

where '∘' is a special character that is not in any active domain, *i.e.,* it does not match any constant. One can verify that no other tuples can both match φ_1 and φ_2. □

Let $\{A_1, \ldots, A_m\}$ be all attributes appearing in φ_i and φ_j. Let $V_{\varphi_{ij}}(A)$ denote the set of constant values of A that appear either in evidence pattern or negative patterns of φ_i and φ_j. The total number of tuples to be enumerated is $\prod_{l \in [1,m]} (|V_{\varphi_{ij}}(A_l)|)$, where $\prod$ indicates a product and $|V_{\varphi_{ij}}(A_l)|$ denotes the cardinality of $V_{\varphi_{ij}}(A_l)$.

Given a set Σ of fixing rules, we check them pairwise (see Example 8). If any pair of rules is inconsistent, we judge that Σ is inconsistent; otherwise, Σ is consistent. This algorithm is referred to as isConsist^t.

5.2.2 Rule characterization

Now let's shift gears and concentrate on a rather different kind of analysis, by characterizing fixing rules and avoiding enumerating tuples.

Also based on Lemma 4, let us focus on the cases of φ_i and φ_j that there exists some t that can match both fixing rules, *i.e.,* it is possible that applying φ_i and φ_j on t in different orders may result in different fixes. Assume that these rules are represented as follows:

φ_i: $((X_i, t_{p_i}[X_i]), (B_i, T^-_{p_i}[B_i])) \rightarrow t^+_{p_i}[B_i]$
φ_j: $((X_j, t_{p_j}[X_j]), (B_j, T^-_{p_j}[B_j])) \rightarrow t^+_{p_j}[B_j]$

Note that a tuple t matching φ_i and φ_j implies that the following conditions hold: $t[X_i] = t_{p_i}[X_i]$ and $t[X_j] = t_{p_j}[X_j]$. Hence, we have $t_{p_i}[X_i \cap X_j] = t_{p_j}[X_i \cap X_j]$, where a special case is $X_i \cap X_j = \emptyset$. We consider two cases: $B_i = B_j$ and $B_i \neq B_j$.

Case 1: $B_i = B_j$. Let $B = B_i = B_j$. There is a conflict only when (*i*) there exists a tuple t that matches both φ_i and φ_j, and (*ii*) φ_i and φ_j will update t to different values. From (*i*) we have $t[B] \in T^-_{p_i}[B]$ and $t[B] \in T^-_{p_j}[B]$, which gives $T^-_{p_i}[B] \cap T^-_{p_j}[B] \neq \emptyset$, *i.e.,* they can be applied at the same time. From (*ii*) we have $t^+_{p_i}[B] \neq t^+_{p_j}[B]$, *i.e.,* they lead to different fixes. From (*i*) and (*ii*), the extra condition that φ_i and φ_j are inconsistent under such case is $(T^-_{p_i}[B] \cap T^-_{p_j}[B] \neq \emptyset$ and $t^+_{p_i}[B] \neq t^+_{p_j}[B])$.

Case 2: $B_i \neq B_j$. Again, we consider four cases: (a) $B_i \in X_j$ and $B_j \notin X_i$, (b) $B_i \notin X_j$ and $B_j \in X_i$, (c) $B_i \in X_j$ and $B_j \in X_i$, and (d) $B_i \notin X_j$ and $B_j \notin X_i$.

(a) $B_i \in X_j$ and $B_j \notin X_i$. If a tuple t matches φ_i and φ_j, then (*i*) $t[B_i] \in T^-_{p_i}[B_i]$ (to match φ_i), and (*ii*) $t[B_i] = t_{p_j}[B_i]$ (to match φ_j). Observe the following: if φ_j is applied to t first, since $B_i \in X_j$, it will keep $t[B_i]$ unchanged, whereas if φ_i is applied first, it will update $t[B_i]$ to a different value (*i.e.,* $t^+_{p_i}[B_i]$). This will cause different fixes. Hence, φ_i and φ_j are inconsistent only when $t_{p_j}[B_i] \in T^-_{p_i}[B_i]$ (by merging (*i*) and (*ii*)).

(b) $B_i \notin X_j$ and $B_j \in X_i$. This is symmetric to case (a). Therefore, φ_i and φ_j are inconsistent only when $t_{p_i}[B_j] \in T^-_{p_j}[B_j]$.

(c) $B_i \in X_j$ and $B_j \in X_i$. This is the combination of cases (a) and (b). Thus, φ_i and φ_j are inconsistent only when $t_{p_i}[B_j] \in T^-_{p_j}[B_j]$ and $t_{p_j}[B_i] \in T^-_{p_i}[B_i]$.

(d) $B_i \notin X_j$ and $B_j \notin X_i$. For any tuple t that matches both φ_i and φ_j, rule φ_i (resp. φ_j) will deterministically update $t[B_i]$ (resp. $t[B_j]$) to $t^+_{p_i}[B_i]$ (resp. $t^+_{p_j}[B_j]$). That is, φ_i and φ_j are always consistent in this case.

Example 10: Consider φ'_1 and φ_3 in Example 8 and φ_2 in Example 3.

Algorithm isConsist^r
Input: a set Σ of fixing rules.
Output: *true* (consistent) or *false* (inconsistent).

1. **for** any two distinct $\varphi_i, \varphi_j \in \Sigma$ **do**
2. **if** $X_i \cap X_j = \emptyset$ **or** $t_{p_i}[X_i \cap X_j] = t_{p_j}[X_i \cap X_j]$ **do**
3. **if** $B_i = B_j$ **do**
4. **if** $T^-_{p_i}[B_i] \cap T^-_{p_j}[B_i] \neq \emptyset$ **and** $t^+_{p_i}[B_i] \neq t^+_{p_j}[B_i]$ **do**
5. **return** *false*;
6. **elseif** $B_i \in X_j$ **and** $B_j \notin X_i$ **and** $t_{p_j}[B_i] \in T^-_{p_i}[B_i]$
7. **return** *false*;
8. **elseif** $B_j \in X_i$ **and** $B_i \notin X_j$ **and** $t_{p_i}[B_j] \in T^-_{p_j}[B_j]$
9. **return** *false*;
10. **elseif** $B_j \in X_i$ **and** $B_i \in X_j$ **and** $t_{p_i}[B_j] \in T^-_{p_j}[B_j]$ **and** $t_{p_j}[B_i] \in T^-_{p_i}[B_i]$
11. **return** *false*;
12. **return** *true*;

Figure 4: Consistency check via rule characterization

Since φ'_1 (resp. φ_2) is only applied to a tuple whose country is China (resp. Canada), there does not exist any tuple that can match both rules at the same time. Therefore, based on Lemma 4, we have φ'_1 and φ_2 are consistent.

Also, it can be verified that φ'_1 and φ_3 are inconsistent. Consider the following:
(*i*) $B_{\varphi_3} \in X_{\varphi'_1}$ (*i.e.,* country $\in$ {country, capital}),
(*ii*) $t_{p_1}[B_{\varphi_3}] \in T^-_{p_3}[B_{\varphi_3}]$ (*i.e.,* China $\in$ {China}),
(*iii*) $B_{\varphi'_1} \in X_{\varphi_3}$ (*i.e.,* capital $\in$ {capital, city, conf}), and
(*iv*) $t_{p_3}[B_{\varphi'_1}] \in T^-_{p_1}[B_{\varphi'_1}]$ (*i.e.,* Tokyo $\in$ {Shanghai, Hongkong, Tokyo}).

Hence, these two rules will lead to different fixes, which is captured by case 2(c). □

Algorithm. The algorithm to check whether a set of fixing rules is consistent via rule characterization, referred to as isConsist^r, is given in Fig. 4. It takes Σ as input, and returns a boolean value, where *true* indicates that Σ is consistent and *false* otherwise.

It enumerates all pairs of distinct rules (lines 1-11). If any pair is inconsistent, it returns *false* (lines 5,7,9,11); otherwise, it reports that Σ is consistent (line 12). It covers all the cases that two rules can be inconsistent, *i.e.,* case 1 (lines 2-5), case 2(a) (lines 6-7), case 2(b) (lines 8-9) and case 2(c) (lines 10-11). Note that in case 2(d), two rules are trivially consistent. Hence, there is no need to investigate such case.

Correctness & complexity. From the analysis above, isConsist^r covers all the cases that two rules can be inconsistent. Thus, it is proved to be correct based on Proposition 3 and Lemma 4. In terms of complexity, we can use a hash table to check that whether a value matches some negative pattern in constant time. Since isConsist^r enumerates all pairs of rules, its time complexity is $\mathcal{O}(\text{size}(\Sigma)^2)$, where size($\Sigma$) is the size of Σ.

5.3 Resolving Inconsistent Rules

When fixing rules are inconsistent, it may lead to conflicting repairing results. In this section, we discuss how to resolve inconsistent fixing rules.

Consider two inconsistent rules, φ'_1 and φ_3, in Example 10. Fig. 5 highlighted the values that result in a conflict. A conservative algorithm is to remove all the rules that are in conflicts. This process ensures termination since the number of rules is *strictly decreasing*, until the set of rules is consistent or becomes empty. Although the bright side is that the remaining rules are consistent, the problem is that this will also remove some useful rules (*e.g.,* φ_3). It is difficult for automatic algorithms to solve such semantic problem well.

φ'_1:

country	{capital$^-$}	capital$^+$
China	Shanghai	Beijing
	Hongkong	
	Tokyo	

φ_3:

capital	city	conf	{country$^-$}	country$^+$
Tokyo	Tokyo	ICDE	China	Japan

Figure 5: Illustrations in resolving conflicts

Hence, in order to obtain high quality rules, we ask experts to examine rules that are in conflicts. For example, the experts can naturally remove Tokyo from the negative patterns of φ'_1, since one cannot judge, given (China, Tokyo), which attribute is wrong. This will result in a modified rule φ_1 (see Example 3), which is consistent with φ_3. Note that in order to ensure termination, we only allow the experts to remove some negative patterns (*e.g.,* from φ'_1 to φ_1), or remove some fixing rules, without adding values.

6. REPAIRING WITH FIXING RULES

After completing our study of finding a set of consistent fixing rules, the next most important item on nearly everybody's wish list is how to use these rules to repair data.

In the following, we first present a chase-based algorithm to repair one tuple (Section 6.1), with time complexity in $\mathcal{O}(\text{size}(\Sigma)|R|)$. We also present a fast algorithm (Section 6.2) running in $\mathcal{O}(\text{size}(\Sigma))$ time for repairing one tuple.

6.1 Chase-based Algorithm

When a given set Σ of fixing rules is *consistent*, for any t, applying Σ to t will get a unique fix (see Section 3.2), which is also known as the Church-Rosser property [1]. We next present an algorithm to repair a tuple with consistent fixing rules. It iteratively picks a fixing rule that can be properly applied, until a *fix* is reached.

Algorithm. The algorithm, referred to as cRepair, is shown in Fig. 6. It takes as input a tuple t and a set Σ of consistent fixing rules. It returns a repaired tuple t' *w.r.t.* Σ.

The algorithm first initializes a set of assured attributes, a set of fixing rules that can be possibly applied, a tuple to be repaired, and a flag to indicate whether the tuple has been changed (line 1). It then iteratively examines and applies the rules to the tuple (lines 2-7). If there is a rule that can be properly applied (line 5), it updates the tuple (line 6), maintains the assured attributes and rules that can be used correspondingly, and flags this change (line 7). It terminates when no rule can be further properly applied (line 2), and the repaired tuple will be returned (line 8).

Correctness & complexity. The correctness of cRepair is inherently ensured by the Church-Rosser property, since Σ is consistent. For the complexity, observe the following. The outer loop (lines 2-7) iterates at most $|R|$ times. For each loop, it needs to scan each unused rule, and checks whether it can be properly applied to the tuple. From these it follows that Algorithm 6 runs in $\mathcal{O}(\text{size}(\Sigma)|R|)$ time.

6.2 A Fast Repairing Algorithm

Our next goal is to study how to improve the chase-based procedure. One natural way is to consider how to avoid

Algorithm cRepair
Input: a tuple t, a set Σ of consistent fixing rules.
Output: a repaired tuple t'.

1. $\mathcal{A} := \emptyset$; $\Gamma := \Sigma$; $t' := t$; updated := *true*;
2. **while** updated **do**
3. updated := *false*;
4. **for each** $\varphi \in \Gamma$ **do**
5. **if** t' matches φ **and** $B_\varphi \notin \mathcal{A}$ **then**
6. $t'[B_\varphi] := t_p^+[B_\varphi]$ (by applying φ);
7. $\mathcal{A} := \mathcal{A} \cup X_\varphi \cup \{B_\varphi\}$; $\Gamma := \Gamma \setminus \{\varphi\}$; updated := *true*;
8. **return** t';

Figure 6: Chase-based repairing algorithm

repeatedly checking whether a rule is applicable, after each update of the tuple being examined.

Note that a key property of employing fixing rules is that, for each tuple, each rule can be applied only once. After a rule is applied, in consequence, it will mark the attributes associated with this rule as assured, and does not allow these attributes to be changed any more (see Section 3.2).

Hence, two important steps are, after each value update, to (i) efficiently identify the rules that cannot be applied, and (ii) determine unused rules that can be possibly applied.

We employ two types of indices in order to perform the above two targets. Inverted lists are used to achieve (i), and hash counters are employed for (ii).

Before describing how to use these indices to design a fast algorithm, we shall pause and define these indices, which is important to understand the algorithm.

Inverted lists. Each inverted list is a mapping from a *key* to a set Υ of fixing rules. Each key is a pair (A, a) where A is an attribute and a is a constant value. Each fixing rule φ in the set Υ satisfies $A \in X_\varphi$ and $t_p[A] = a$.

For example, an inverted list *w.r.t.* φ_1 in Example 3 is as:

$$\boxed{\texttt{country}, \texttt{China}} \rightarrow \boxed{\varphi_1}$$

Intuitively, when the country of some tuple is China, this inverted list will help to identify that φ_1 might be applicable.

Hash counters. It uses a hash map to maintain a counter for each rule. More concretely, for each rule φ, the counter $\mathsf{c}(\varphi)$ is a nonnegative integer, denoting the number of attributes that a tuple agrees with $t_p[X_\varphi]$.

For example, consider φ_1 in Example 3 and r_2 in Fig. 1. We have $\mathsf{c}(\varphi_1) = 1$ *w.r.t.* tuple r_2, since both r_2[country] and t_{p_1}[country] are China. As another example, consider r_4 in Fig. 1, we have $\mathsf{c}(\varphi_1) = 0$ *w.r.t.* tuple r_4, since r_4[country] = Canada but t_{p_1}[country] = China.

We are now ready to present a fast algorithm by using the two indices introduced above. Note that inverted lists are built *only once* for a given Σ, and keep unchanged for all tuples. The hash counters will be initialized to zero for the process of repairing each new tuple.

Algorithm. The algorithm lRepair is given in Fig. 7. It takes as input a tuple t, a set Σ of consistent fixing rules, and inverted lists $\mathcal{I}$. It returns a repaired tuple t' *w.r.t.* Σ.

It first initializes a set of assured attributes, a set of fixing rules to be used, and a tuple to be repaired (line 1). It also clears the counters for all rules (line 2). It then uses inverted lists to initialize the counters (lines 3-5). After the counters are initialized, it checks and maintains a list of rules that might be used (lines 6-7), and uses a chase process to

Algorithm lRepair
Input: tuple t of R, consistent Σ, inverted lists $\mathcal{I}$.
Output: a repaired tuple t'.

1. $\mathcal{A} := \emptyset$; $\Gamma := \emptyset$; $t' := t$;
2. **for each** $\varphi \in \Sigma$ **do** $\mathsf{c}(\varphi) := 0$;
3. **for each** $A \in R$ **do**
4. **for each** φ in $\mathcal{I}(A, t[A])$ **do**
5. $\mathsf{c}(\varphi) := \mathsf{c}(\varphi) + 1$;
6. **for each** $\varphi \in \Sigma$ **do**
7. **if** $\mathsf{c}(\varphi) = |X_\varphi|$ **then** $\Gamma := \Gamma \cup \{\varphi\}$;
8. **while** $\Gamma \neq \emptyset$ **do**
9. randomly pick φ from Γ;
10. **if** t' matches φ **and** $B_\varphi \notin \mathcal{A}$ **then**
11. update t' by applying φ such that $t'[B_\varphi] = t_p^+[B_\varphi]$;
12. $\mathcal{A} := \mathcal{A} \cup X_\varphi \cup \{B_\varphi\}$;
13. **for each** $\varphi' \in \mathcal{I}(B_\varphi, t'[B_\varphi])$ **do**
14. $\mathsf{c}(\varphi') := \mathsf{c}(\varphi') + 1$;
15. **if** $\mathsf{c}(\varphi') = |X_{\varphi'}|$ **then** $\Gamma := \Gamma \cup \{\varphi'\}$;
16. $\Gamma := \Gamma \setminus \{\varphi\}$;
17. **return** t';

Figure 7: A linear repairing algorithm

repair the tuple (lines 8-16), and returns the repaired tuple (line 17).

During the process (lines 8-16), it first randomly picks a rule that might be used (line 9). The rule will be applied if it is verified to be applicable (lines 10-11). The set of attributes that is assured correct is increased correspondingly (line 12). The counters will be recalculated (lines 13-14). Moreover, if new rules might be used due to this update, it will be identified (line 15). The rule that has been checked will be removed (line 16), no matter it is applicable or not.

Observe the following two cases. (i) If a rule is removed after being applied at line 16 (*i.e.,* line 10 gives a *true*), it cannot be used again and will not be checked at lines 13-15. (ii) If a rule φ is removed without being applied at line 16 (*i.e.,* line 10 gives a *false*), it cannot be used either at lines 13-15. The reason is that: for any rule φ, if φ cannot be properly applied to t', any update on attribute B_φ will mark it as assured, such that φ cannot be properly applied afterwards. From the above (i) and (ii), it follows that it is safe to remove a rule from Γ, after it has been checked, once and for all.

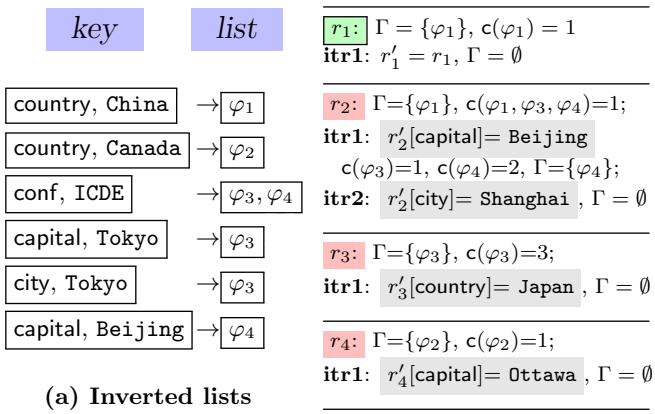
Correctness. Note that Σ is consistent, we only need to prove the repaired tuple t' is a fix of t. This can be proved based on (1) at any point, Γ includes all fixing rules that might match the given tuple; and (2) each fixing rule is added into Γ at most once. Hence, the algorithm terminates until it reaches a fixpoint when Γ is empty.

Complexity. It is clear that the three loops (line 2, lines 3-5 and lines 6-7) all run in time linear to $\mathsf{size}(\Sigma)$. Next let us consider the **while** loop (lines 8-16). Observe that each rule φ will be checked in the inner loop (lines 13-15) up to $|X_\varphi|$ times, by using the inverted lists and hash counters, independent of the number of outer loop iterated. The other lines of this **while** loop can be done in constant time. Putting together, the total time complexity of the algorithm is $\mathcal{O}(\mathsf{size}(\Sigma))$.

We next show by example how Algorithm lRepair works.

Example 11: Consider Travel data D in Fig. 1, rules φ_1, φ_2 in Example 3 and rule φ_3 in Example 8. In order to better understand the chase process, we introduce another rule:

φ_4: (([capital, conf], [Beijing, ICDE]), (city, {Hongkong})



(a) Inverted lists

Figure 8: A running example

→ Shanghai

Rule φ_4 states that: for t in relation Travel, if the conf is ICDE, held at some country whose capital is Beijing, but the city is Hongkong, its city should be Shanghai. This holds since ICDE was held in China only once at 2009, in Shanghai but never in Hongkong.

Given the four fixing rules φ_1–φ_4, the corresponding inverted lists are given in Fig. 8(a). For instance, the third key (conf, ICDE) links to rules φ_3 and φ_4, since conf $\in X_{\varphi_3}$ (*i.e.,* {capital, city, conf}) and t_{p_3}[conf] = ICDE; and moreover, conf $\in X_{\varphi_4}$ (*i.e.,* {capital, conf}) and t_{p_4}[conf] = ICDE. The other inverted lists are built similarly.

Now we show how the algorithm works over tuples r_1 to r_4, which is also depicted in Fig. 8. Here, we highlight these tuples in two colors, where the green color means that the tuple is clean (*i.e.,* r_1), while the red color represents the tuples containing errors (*i.e.,* r_2, r_3 and r_4).

r_1: It initializes (lines 1-7) and finds that φ_1 may be applied, maintained in Γ. In the first iteration (lines 8-16), it finds that φ_1 cannot be applied, since r_1[capital] is Beijing, which is not in the negative patterns {Shanghai, Hongkong} of φ_1. Also, no other rules can be applied. It terminates with tuple r_1 unchanged. Actually, r_1 is a clean tuple.

r_2: It initializes and finds that φ_1 might be applied. In the first iteration (lines 8-16), rule φ_1 is applied to r_2 and updates r_2[capital] to Beijing. Consequently, it uses inverted lists (line 13) to increase the counter of φ_4 (line 14) and finds that φ_4 might be used (line 15). In the second iteration, rule φ_1 is applied and updates r_2[city] to Shanghai. It then terminates since no other rules can be applied.

r_3: It initializes and finds that φ_3 might be applied. In the first iteration, rule φ_3 is applied and updates r_3[coutry] to Japan. It then terminates, since no more applicable rules.

r_4: It initializes and finds that φ_2 might be applied. In the first iteration, rule φ_2 is applied and updates r_4[capital] to Ottawa. It will then terminate.

At this point, we see that all the fours errors shown in Fig. 1 have been corrected, as highlighted in Fig. 8. □

7. EXPERIMENTAL STUDY

We conducted experiments with both real-life and synthetic data to examine our algorithms and help us discover the deficiency of fixing rules and algorithms to be improved. Specifically, we evaluated (1) the efficiency of consistency checking for fixing rules; (2) the accuracy of our data repairing algorithms with fixing rules; and (3) the efficiency of data repairing algorithms using fixing rules.

It is worth noting that the purpose of these experiments is to test, when given high quality fixing rules, how they can be used to *automatically* repair data with high dependability.

7.1 Experimental Setting

Experimental data. We used real-life and synthetic data.

(1) HOSP was taken from US Department of Health & Human Services[1]. It has 115K records with the following attributes: Provider Number (PN), Hospital Name (HN), address1, address2, address3, city, state, zip, county, PhoneNumber (phn), HospitalType (ht), HospitalOwner (ho), EmergencyService (es) Measure Code (MC), Measure Name (MN), condition, and stateAvg.

(2) UIS data was generated by a modified version of the UIS Database generator[2]. It produces a mailing list that has the following schema: RecordID, ssn, FirstName (fname), MiddleInit (minit), LastName (lname), stnum, stadd, apt, city, state, zip. We generated 15K records.

Dirty data generation. We treated clean datasets as the ground truth. Dirty data was generated by adding noise only to the attributes that are related to some integrity constraints, which is controlled by noise rate (10% by default). We introduced two types of noises: typos and errors from the active domain.

Fixing rules generation. We next discuss how fixing rules can be obtained.

Seed fixing rule generation. Since each fixing rule is defined on semantically related attributes, we started with known dependencies (*e.g.,* FDs for our testing). We first detected violations of given FDs, and presented them to experts. The experts produced several fixing rules as seeds (or *samples*), based on their understanding of these violations.

Rule enrichment. Given seed fixing rules, we enriched them by only enlarging their negative patterns, via extracting new negative patterns from related tables in the same domain. For instance, consider Example 2. If users provide a fixing rule that takes China as the evidence pattern, and some Chinese cities (*e.g.,* Shanghai, Hongkong) other than Beijing as negative patterns, one can enlarge its negative patterns by extracting cities from a table about Chinese cities. Note that when an appropriate ontology is available, we can extract the above information as evidence patterns, negative patterns and facts. In such case, the generated fixing rules are usually general. Consequently, they can be applied to multiple databases.

In the experiment, we generated 1000 fixing rules for HOSP data 100 fixing rules for UIS data.

Measuring quality. To assess the accuracy of data cleaning algorithms, we use precision and recall, where precision is the ratio of corrected attribute values to the number of all the attributes that are updated, and recall is the ratio of corrected attribute values to the number of all erroneous attribute values.

Remark. We mainly compare with the *state-of-the-art* automated data cleaning techniques. Note that they are designed for a slightly different target: computing a consistent

[1]http://www.hospitalcompare.hhs.gov/

[2]http://www.cs.utexas.edu/users/ml/riddle/data.html

database. We consider it a relatively fair comparison, since all fixing rules we generated are from FD violations. In other words, the fixing rules and the FDs used are defined on exactly the same set of attributes. We employed the following FDs for HOSP and UIS data, respectively.

FDs for HOSP
PN → HN, address1, address2, address3, city, state, zip, county, phn, ht, ho, es
phn → zip, city, state, address1, address2, address3
MC → MN, condition
PN, MC → stateAvg
state, MC → stateAvg
FDs for UIS
ssn → fname, minit, lname, stnum, stadd, apt, city, state, zip
fname,minit,lname → ssn, stnum, stadd, apt, city, state, zip
zip → state, city

Algorithms. We have implemented the following algorithms in C++: (1) isConsist^t: the algorithm for checking consistency based on tuple enumeration (Section 5.2); (2) isConsist^r: the algorithm for checking consistency based on rule characterization (Fig. 4 in Section 5.2); (3) cRepair: the basic chase-based algorithm for repairing with fixing rules (see Fig. 6); and (4) lRepair: the fast repairing algorithm (see Fig. 7). Moreover, for comparison, we have implemented two algorithms for FD repairing, a cost-based heuristic method [7], referred to as Heu, and a approach for cardinality set minimal [5], referred to as Csm. Both approaches were implemented in Java.

All experiments were conducted on a Windows machine with a 3.0GHz Intel CPU and 4GB of memory.

7.2 Experimental Results

We next report our findings from our experimental study.

Exp-1: Efficiency of checking consistency. We evaluated the efficiency of checking consistency by varying the number of rules. The results for HOSP (resp. UIS) are shown in Fig. 9(a) (resp. Fig. 9(b)). The x-axis is the number of rules multiplied by 100 (resp. 10) for HOSP (resp. UIS), and the y-axis is the running time in millisecond (msec).

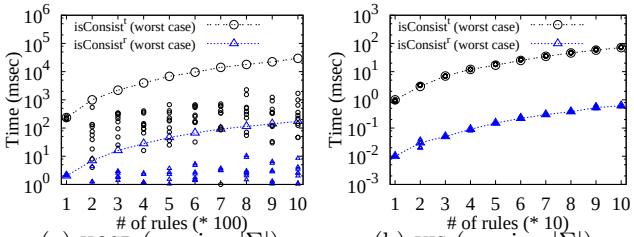
For either isConsist^t or isConsist^r, we plotted its worst case, *i.e.,* checking all pairs of rules, as well as its 10 real cases where it terminated when some pair was detected to be inconsistent. For example, in Fig. 9(a), the big circle for $x = 2$ was for checking 200 rules in the worst case, while the 10 small circles below it were for real cases. In Fig. 9(b), real cases are the same as the worst case, since the 100 rules are consistent and all pairs of distinct rules have to be checked.

These figures show that to check consistency of fixing rules, the algorithm with tuple enumeration (isConsist^t) is slower, as expected. The reason is that enumerating tuples for two rules is more costly than characterizing two rules.

In addition, this set of experiment validated that the consistency of fixing rules can be checked efficiently. For example, it only needs 12 seconds to check the consistency of 1000*1000 pairs of rules, *i.e.,* the top right point in Fig. 9(a).

The result of this study indicates that it is feasible to check consistency for a reasonably large set of fixing rules.

Exp-2: Accuracy. In this set of experiments, we will study the following. (a) The effect of different data errors (*i.e.,* typos or errors from active domain) for repairing algorithms. (b) The influence of fixing rules *w.r.t.* their sizes. (c) Effect of negative patterns. (d) Comparison with editing rules. We use Fix to represent repairing algorithms with fixing rules.



(a) HOSP (varying $|\Sigma|$) (b) UIS (varying $|\Sigma|$)

Figure 9: Efficiency for checking consistency

(a) Noise from active domain. Recall that noise was obtained by either introducing typos to an attribute value or changing an attribute value to another one from the active domain of that specific attribute. For example, an error for Ottawa could be Ottawo (*i.e.,* a typo) or Beijing (*i.e.,* a value from active domain).

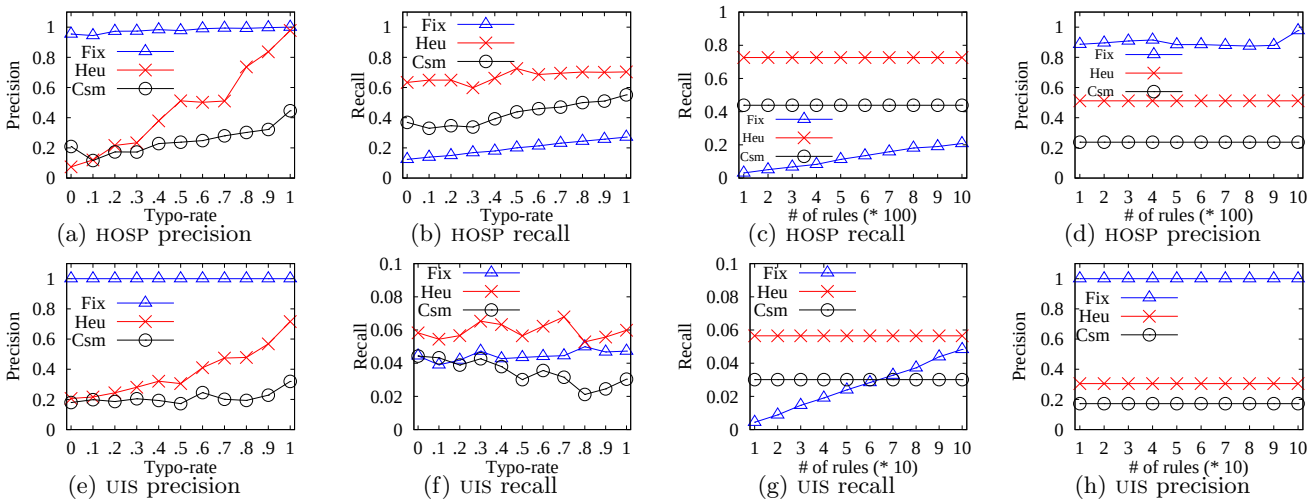
Precision. We fixed the noise rate at 10%, and varied the percentage of typos from 0% to 100% by a step of 10% (x-axis in both charts from Figs. 10(a) and 10(e) for HOSP and UIS, respectively). Both figures showed that our method using fixing rules performed dependable fixes (*i.e.,* high precision), and was not sensitive to types of errors. While for the existing algorithms Heu and Csm, they had lower precision when more errors were from the active domain. The reason is that for such errors, heuristic methods would erroneously connect some tuples as related to violations, which might link previously irrelevant tuples and complicate the process when fixing the data. Indeed, however, both Heu and Csm computed a consistent database, as targeted.

Note that fixing rules also made mistakes, *e.g.,* the precision in Fig. 10(a) is not 100%, which means some changes were not correct. The reason is that, when more errors are from the active domain (*e.g.,* typo rate is 0 in Fig. 10(a)), it will mislead fixing rules to make decisions. For example, consider the two rules in Fig. 3, if the correct (country, capital) values of some tuple are (China, Shanghai) but were changed by using values from the active domain to (Canada, Toronto), using fixing rules will make mistakes. Although this is not very common in practice, it deserves a further study to improve our algorithms in the future.

Recall. In order to better understand the behavior of these algorithms, Figs. 10(b) and 10(f) show the recall corresponding to Figs. 10(a) and 10(e), respectively. Not surprisingly, our algorithm did not outperform existing approaches in terms of recall. This is because heuristic approaches would repair some potentially erroneous values, but at the tradeoff of decreasing precision. Although our method was relatively low in recall, we did our best to ensure the precision, instead of repairing as more errors as possible. Hence, when recall is a major requirement for some system, existing heuristic methods can be used after fixing rules being applied, to compute a consistent database.

Fig. 10(f) shows that the recall is very low (below 8%) for all methods. The reason is that, the UIS dataset generated has few repeated patterns *w.r.t.* each FD. When noise was introduced, many errors cannot be detected, hence no method can repair them. Note, however, that recall can be improved by learning more rules as shown below.

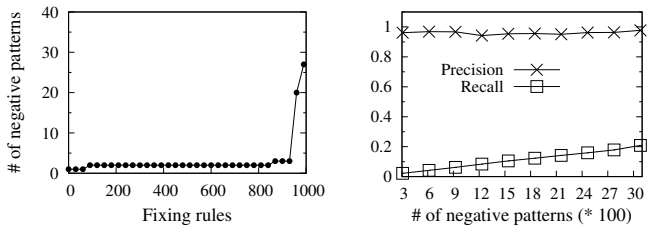
(b) Varying the number of fixing rules. We studied the accuracy of our repairing algorithms by varying the number of fixing rules. We fixed noise rate at 10% and half of



(a) HOSP precision (b) HOSP recall (c) HOSP recall (d) HOSP precision

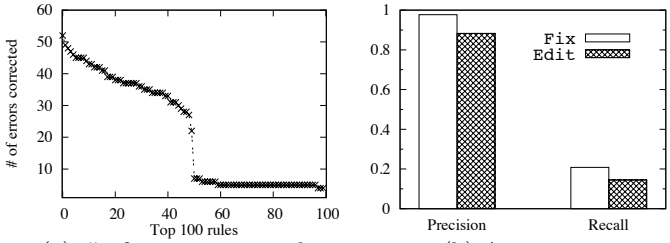
(e) UIS precision (f) UIS recall (g) UIS recall (h) UIS precision

Figure 10: Accuracy of data repairing



(a) Distribution (b) Accuracy

Figure 11: Evaluation for negative patterns (HOSP)



(a) # of errors corrected (b) Accuracy

Figure 12: Comparison with editing rules (HOSP)

them are typos. For HOSP, we varied the number of rules from 100 to 1000, and reported the recall and precision in Fig. 10(c) and Fig. 10(d), respectively. For UIS, we varied the number of rules from 10 to 100, and reported the results in Fig. 10(g) and Fig. 10(h), respectively. For Heu and Csm, as the typo rate was fixed, their precision and recall values were horizontal lines.

The experimental results indicate that when more fixing rules are available, our approach can achieve better recall, while keeping a good precision, as expected.

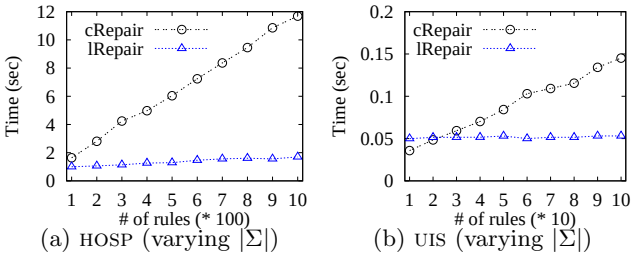
(c) Evaluation for negative patterns. To further investigate fixing rules, we sorted the fixing rules of HOSP based on the number of negative patterns, and plotted every 30 points in Fig. 11(a). We see that most of the fixing rules have a small number of negative patterns. For instance, around 80% of fixing rules contain two negative patterns. We added up all negative patterns, and evaluated the accuracy of our repairing algorithms by varying the number of negative patterns for all rules in total. Figure 11(b) shows the precision and recall of our approach. We can see that adding more negative patterns can lead to a better recall while keeping a high precision. This experimental result further validates the dependable feature of fixing rules.

(d) Comparison with editing rules. We also compared our approach with editing rules [19]. Although editing rules can repair data that is guaranteed to be correct, they are measured by the number of *user interactions* per tuple. That is, for *each tuple* and for *each editing rule* to be applied, the users have to be asked. To this purpose, we evaluated the number of errors that can be corrected by every fixing rule (see Fig. 12(a)) using HOSP data with 100 rules and 10% dirty rate, where the x-axis is for fixing rules and the y-axis is the number of errors they can correct. This experiment shows that a single fixing rule was able to repair errors in more than fifty tuples, but if we employ editing rules to repair these errors, the approach has to interact with users *over fifty times*.

Moreover, we encoded data values from master data into editing rules, to make it an *automated* rule. Note that error information is not in master data, *e.g.,* the negative patterns in fixing rules, which cannot be encoded. Hence, we removed negative patterns in fixing rules, to simulate editing rules. Specifically, each time when seeing an evidence pattern, it simulated users by saying yes, and then updated the right hand side value to the fact. The experimental results are shown in Fig. 12(b), where Fix (resp. Edit) indicates fixing rules (resp. editing rules). The reason that fixing rules have better precision and recall is that, if we have errors in the right hand side of such rules, (automated) editing rules can correct them. However, if there are errors in the left hand side, they will introduce new errors by treating these errors as correct values, resulting in lower precision and in consequence, lower recall.

Exp-3: Efficiency of repairing algorithms. In this last set of experiments, we study the efficiency of our data repairing with fixing rules. As they are linear in data size, we evaluated their efficiency by varying the number of rules.

The results for HOSP and UIS are given in Fig. 13(a) and Fig. 13(b), respectively. In both figures, the x-axis is for the number of rules and the y-axis is for running time. These two figures show that algorithm lRepair is more efficient. For example, it ran in less than 2 seconds to repair $115K$ tuples, using 1000 rules (the bottom right node in Fig. 13(a)). In Fig. 13(b), cRepair was faster only when the number of rules



(a) HOSP (varying $|\Sigma|$) (b) UIS (varying $|\Sigma|$)

Figure 13: Efficiency for data repairing

was very small (*i.e.,* 10), where the reason is the extra overhead of using inverted lists and hash counters. However, in general, lRepair was much faster, since it only examined the rules that can be used instead of checking all rules.

We also compared our fast repairing algorithm lRepair with Heu and Csm. Using both HOSP and UIS data, the results are given in the table below. It shows that lRepair runs much faster than the others. The reasons are twofold: (1) lRepair detects errors on each tuple individually, while the others need to consider a combination of two tuples for violation detection; and (2) lRepair repairs each tuple in linear time, while Heu and Csm repair data by holistically considering all violations, which have much higher time complexity.

	lRepair	Heu	Csm
HOSP	1.7 sec	580 sec	421 sec
UIS	0.05 sec	13 sec	8 sec

Summary. We found the following from the above experiments. (a) It is efficient to detect whether a set of fixing rules is consistent (Exp-1). (b) Data repairing using fixing rules is dependable, *i.e.,* they repair data errors with high precision (Exp-2). (c) The recall of using fixing rules can be improved when more fixing rules are available (Exp-2). (d) It is efficient to repair data via fixing rules, which reveals its potential to be used for large datasets (Exp-3).

8. CONCLUSION AND FUTURE WORK

We have proposed a novel class of data cleaning rules, namely, *fixing rules*, that (1) compared with data dependencies used in data cleaning, are able to find dependable fixes for input tuples, without using heuristic solutions; and (2) differ from editing rules, that are able to repair data automatically without any user involvement. We have identified fundamental problems for deciding whether a set of fixing rules is consistent or redundant, and established their complexity bounds. We have proposed efficient algorithms for checking consistency, and discussed strategies to resolve inconsistent fixing rules. We have also presented data repairing algorithms by capitalizing on fixing rules. Our experimental results with real-life and synthetic data have verified the effectiveness and efficiency of the proposed rules and the presented algorithms. These yield a promising method for automated and dependable data repairing.

The study of automated and dependable data repairing is still in its infancy. This research is just a first attempt to tackle this problem, and it has thrown up many questions in need of further investigation. (1) *Rule discovery.* Our techniques in the paper allow users to define fixing rules manually, or generate rules using examples. We are planning to design algorithm to automatically discover fixing rules. (2) *Interaction with other data quality rules.* A challenging topic is to explore the interaction between fixing rules and other data quality and rules, such as CFDs, MDs, editing rules, and the users.

9. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, 1999.

[3] C. Batini and M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Springer, 2006.

[4] L. E. Bertossi, S. Kolahi, and L. V. S. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. In *ICDT*, 2011.

[5] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 2010.

[6] G. Beskales, M. A. Soliman, I. F. Ilyas, and S. Ben-David. Modeling and querying possible repairs in duplicate detection. In *VLDB*, 2009.

[7] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.

[8] L. Bravo, W. Fan, and S. Ma. Extending dependencies with conditions. In *VLDB*, 2007.

[9] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.*, 2005.

[10] X. Chu, P. Papotti, and I. Ilyas. Holistic data cleaning: Put violations into context. In *ICDE*, 2013.

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

[12] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a commodity data cleaning system. In *SIGMOD*, 2013.

[13] A. Ebaid, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. NADEEF: A generalized data cleaning system. *PVLDB*, 2013.

[14] W. Fan. Dependencies revisited for improving data quality. In *PODS*, 2008.

[15] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 2008.

[16] W. Fan, F. Geerts, N. Tang, and W. Yu. Inferring data currency and consistency for conflict resolution. In *ICDE*, 2013.

[17] W. Fan, X. Jia, J. Li, and S. Ma. Reasoning about record matching rules. *PVLDB*, 2009.

[18] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction between record matching and data repairing. In *SIGMOD*, 2011.

[19] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *VLDB J.*, 2012.

[20] I. Fellegi and D. Holt. A systematic approach to automatic edit and imputation. *J. American Statistical Association*, 1976.

[21] T. N. Herzog, F. J. Scheuren, and W. E. Winkler. *Data Quality and Record Linkage Techniques*. Springer, 2009.

[22] S. Kolahi and L. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, 2009.

[23] C. Mayfield, J. Neville, and S. Prabhakar. ERACER: a database approach for statistical inference and data cleaning. In *SIGMOD*, 2010.

[24] F. Naumann, A. Bilke, J. Bleiholder, and M. Weis. Data fusion in three steps: Resolving schema, tuple, and value inconsistencies. *IEEE Data Eng. Bull.*, 2006.

[25] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[26] V. Raman and J. M. Hellerstein. Potter's Wheel: An interactive data cleaning system. In *VLDB*, 2001.

[27] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 2012.

[28] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 2011.