Dependable Data Repairing with Fixing Rules

JIANNAN WANG, School of Computing Science, Simon Fraser University, Canada
NAN TANG, Qatar Computing Research Institute, HBKU, Qatar

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 (also known as integrity constraints) have been widely studied to capture errors in data, automated and dependable data repairing on these errors has remained a notoriously difficult 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 are consistent and discuss approaches to resolve inconsistent fixing rules. We also devise efficient algorithms for repairing data errors using fixing rules. Moreover, we discuss approaches on how to generate a large number of fixing rules from examples or available knowledge bases. 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.

Categories and Subject Descriptors: H.2.m [**Database Management**]: Miscellaneous—*Data cleaning*

General Terms: Design, Algorithm, Performance

Additional Key Words and Phrases: Data repairing, fixing rules, dependable

ACM Reference Format:
Jiannan Wang and Nan Tang. 2017. Dependable data repairing with fixing rules. J. Data and Information Quality 8, 3–4, Article 16 (June 2017), 34 pages.
DOI: http://dx.doi.org/10.1145/3041761

1. INTRODUCTION

Data quality is essential to all businesses, which demand 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 (a.k.a. reliable or trusted) data repairing on these data errors has remained a notoriously hard problem.

A number of recent articles [Bohannon et al. 2005; Bertossi et al. 2011; Chu et al. 2013b; Geerts et al. 2013] have investigated the following data cleaning problem [Arenas et al. 1999]: data 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

Authors' addresses: J. Wang, School of Computing Science, Simon Fraser University, Burnaby, Canada; email: jnwang@sfu.ca; N. Tang, Qatar Computing Research Institute, HBKU, Qatar; email: ntang@qf.org.qa.

DOI: http://dx.doi.org/10.1145/3041761

repairing. However, it is known that such heuristics may introduce data errors [Fan et al. 2012]. In order to ensure that a repair is dependable, users have been involved as first-class citizens in the process of data repairing [Fan et al. 2012; Raman and Hellerstein 2001; Yakout et al. 2011], 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.

Motivating example. We first illustrate by examples how existing solutions work.

Example 1.1. Consider a (denormalized) database D of travel records for a research institute, specified by the following schema:

$$\mathsf{Travel}(\mathsf{name}, \mathsf{country}, \mathsf{capital}, \mathsf{city}, \mathsf{conf}),$$

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

We next illustrate two classes of the *state-of-the-art* data repairing algorithms. The first class is about automated dependency-based data-repairing algorithms. The second class is about user guided data-repairing approaches.

Data dependencies. A variety of data dependencies have been used to capture errors in data, from traditional constraints (e.g., functional and inclusion dependencies [Bohannon et al. 2005; Chomicki and Marcinkowski 2005; Wijsen 2005]) to their extensions (e.g., conditional functional dependencies [Fan et al. 2008]). Suppose that a functional dependency (FD) is specified for the Travel table as

$$\phi_1 : \mathsf{Travel}([\mathsf{country}] \rightarrow [\mathsf{capital}]),$$

which states that country uniquely determines capital. One can verify that in Figure 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), many algorithms have been presented [Arenas et al. 1999; Bohannon et al. 2005; Cong et al. 2007; Fellegi and Holt 1976; Chu et al. 2013b; Beskales et al. 2010; Fan et al. 2012]. For instance, they can change $r_2[\mathsf{capital}]$ from Shanghai to Beijing, and $r_3[\mathsf{capital}]$ from Tokyo to Beijing, which requires the changes of two data values. One may verify that this is a repair with the *minimum* cost of two changes. Though these changes correct the error in $r_2[\mathsf{capital}]$, they do not rectify $r_3[\mathsf{country}]$. Worse still, they mess up the correct value in $r_3[\mathsf{capital}]$, by changing it from the correct value Tokyo to a wrong value Beijing.

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 [Mayfield et al. 2010; Yakout et al. 2011; Fan et al. 2012] and master data (a.k.a. reference data) has been used [Fan et al. 2012].

Consider a recent work [Fan et al. 2012] that uses editing rules and master data. Figure 2 shows 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

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

	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	Toronto	VLDB

Fig. 1. D of schema Travel.

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

Fig. 2. D_m of schema Cap.

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

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

Fig. 3. Example fixing rules.

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, then we can update r[capital] with the value s[capital] from Cap. Note that its pattern tuple t_{p1} poses no constraint about the values on country attributes. For instance, to repair r_2 in Figure 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 and observation. The above examples tell us that data dependencies can detect errors but fall short of automatically guiding dependable data repairing. On the other hand, 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 the combination of values (China, Shanghai) for attributes (country, capital) appears 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. A salient feature of fixing rules is the introduction of negative patterns, which is key to enabling dependable data repairs. With their help, we can avoid mistakenly repairing the values such as (China, Tokyo).

Example 1.2. Figure 3 shows two fixing rules. The braces mean that the corresponding cell is multivalued, e.g., $\{\text{capital}^-\}$ in φ_1.

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, then capital should be updated to Beijing. For instance, consider the database in Figure 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), (Japan, Tokyo), or other value combinations.

Contributions. We propose fixing rules and an associated framework for automatically and dependably repairing data errors with the following notable contributions.

(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 devise 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 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 relation 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. The difference between size() and cardinality will be explained at the end of Section 5.2.
(5) We discuss the problem of generating fixing rules (Section 7). We first present how a large number of fixing rules can be obtained from examples, inspired by the work of Singh and Gulwani [2012]. We also describe how to generate fixing rules from available knowledge bases in a batch fashion.
(6) We experimentally verify the effectiveness and scalability of the proposed algorithms (Section 8). We find that algorithms with fixing rules can repair data with high precision. In addition, they scale well with the number of fixing rules.

Organization. Section 2 discusses related work. Section 3 introduces fixing rules. Section 4 studies fundamental problems associated with 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 discusses how to generate fixing rules. Section 8 reports our experimental findings, followed by concluding remarks in Section 9.

2. RELATED WORK

The current submission is an extended version of our conference article [Wang and Tang 2014]. This submission includes the following new materials: (1) proofs of fundamental problems in connection with fixing rules (Section 4); (2) techniques on how to generate fixing rules from external resources such as large-scale knowledge bases (Section 7); (3) an empirical evaluation of using the fixing rules generated from the above techniques (Section 8); and (4) a more detailed explanation of algorithms and examples and a

summary of notations for the ease of reference (Section 3.2). None of the detailed proofs of (1) was presented in the conference article. Some of the proofs are nontrivial and are interesting in their own right. The discussion of (2) on generating fixing rules from available knowledge bases was not addressed in the conference article, which is important for any application to employ fixing rules. The experimental study of (3) was not given before.

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., Arenas et al. [1999], Chomicki and Marcinkowski [2005], Bravo et al. [2007], Fan et al. [2008], Kolahi and Lakshmanan [2009], and Wijsen [2005]; see Fan [2008] for a survey). They have been revisited to better capture data errors as violations of these dependencies (e.g., conditional functional dependencies (CFDs) [Fan et al. 2008] and conditional inclusion dependencies (CINDs) [Bravo et al. 2007], and metric functional dependencies [Koudas et al. 2009]). 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 [Fan et al. 2012] 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.

Closer to this work is Chu et al. [2015] and Interlandi and Tang [2015]. KATARA [Chu et al. 2015] is powered by knowledge bases and crowdsourcing, which cleans a table T based on a knowledge base K. KATARA uses a *table pattern* to explain the semantic matching between the table and the given knowledge base. In fact, table patterns are used analogously to *matching dependencies* for tables. That is, when a full match is found between a tuple t in T and a subgraph g in K for the given table pattern, KATARA can say that t is correct for the attributes that appear in the table pattern. However, if there is no full match, there are two cases: (1) t is wrong; or (2) the knowledge base is incomplete. For both cases, KATARA needs users, i.e., crowdsourcing, to resolve the ambiguity. The essential difference between KATARA and fixing rules is that, given a tuple t, each error is annotated by crowd users, but fixing rules identify errors automatically. Moreover, KATARA does not repair the error. Instead, it finds some possible values from the knowledge bases, which are called *possible repairs*, and it stops there. In contrast, fixing rules will automatically find the unique repair for an error. Sherlock rules [Interlandi and Tang 2015] focus on the following two issues: (1) annotate data as correct or wrong, which is called *proof positive* and *negative*, and repair is another functionality; and (2) they rely on reference tables. Indeed, when reference tables are available for Sherlock rules, we can generate fixing rules but not the other way around.

Data repairing algorithms have been proposed [Bohannon et al. 2005; Cong et al. 2007; Fan et al. 2012; Fellegi and Holt 1976; Mayfield et al. 2010; Yakout et al. 2011; Beskales et al. 2009; Fan et al. 2011; Chu et al. 2013b]. Heuristic methods are developed in Beskales et al. [2010], Bohannon et al. [2005], Cong et al. [2007], and Fellegi and Holt [1976], based on FDs [Beskales et al. 2010; Kolahi and Lakshmanan 2009], FDs and INDs [Bohannon et al. 2005], CFDs [Fan et al. 2008], CFDs and MDs [Fan et al. 2011], denial constraints [Chu et al. 2013b], edit rules [Fellegi and Holt 1976], and neighborhood constraints [Song et al. 2014]. There also exists a generalized data-cleaning system that treats data quality rules as black boxes and repairs detected violations holistically using SAT-solver-based approaches [Dallachiesa et al. 2013].

Another line of work in data repairing studies the problem of repairing both data and dependencies in a unified framework [Volkovs et al. 2014; Chiang and Miller 2011; Beskales et al. 2013]. Statistical inference is studied in Mayfield et al. [2010], to derive missing values, and in Beskales et al. [2009], to find possible repairs. Lian et al. [2010] studies the problem of finding consistent query answers from inconsistent probabilistic databases. In contrast to these prior articles, fixing rules are more conservative to repair data, which target at determinism and dependability, instead of computing a consistent database. 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.

In order to improve the accuracy of data repairing, a number of studies employ confidence placed by users to guide a repairing process [Bohannon et al. 2005; Cong et al. 2007; Fan et al. 2011] or use master data [Fan et al. 2012]. Some other works involve users as first-class citizens in data repairing [Mayfield et al. 2010; Yakout et al. 2011; Fan et al. 2012]. Different from them, fixing rules neither consult the users nor assume the confidence values placed by the users. That is, the repairing process of employing fixing rules is fully automated. Note that machine-learning-based approaches, which take predefined confidence values or user feedback as input, have been studied [Volkovs et al. 2014; Yakout et al. 2011, 2013]. In fact, machine-learning tools are complementary to data repairing and this work, e.g., the evidence, negative, and fact information of fixing rules can be directly used as training data to train machine-learning tools.

There has also been a lot of work on more general data cleaning: data transformation, which brings the data under a single common schema [Naumann et al. 2006]. Extract, translate, load (ETL) tools (see Batini and Scannapieco [2006] and Herzog et al. [2009] 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 both syntactic string transformations [Arasu et al. 2009] and semantic string transformations [Singh and Gulwani 2012]. However, they are designed for value transformation instead of capturing semantic errors from multiple attributes as fixing rules do. Hence, they can be treated as orthogonal techniques, which prepare data first 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 applying 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, which is 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). Here, "\" means set minus;
(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 $t_p^+[B]$ 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 means that φ can identify errors in t.

Example 3.1. Consider the fixing rules in Figure 3. They are expressed as follows:

$$\varphi_1 : (([\text{country}], [\text{China}]), (\text{capital}, \{\text{Shanghai}, \text{Hongkong}\})) \rightarrow \text{Beijing},$$
$$\varphi_2 : (([\text{country}], [\text{Canada}]), (\text{capital}, \{\text{Toronto}\})) \rightarrow \text{Ottawa}.$$

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 Figure 1. Tuple r_1 does not match rule φ_1, since $r_1[\text{country}] = \text{China}$ but $r_1[\text{capital}] \notin \{\text{Shanghai}, \text{Hongkong}\}$. As another example, r_2 matches rule φ_1, since $r_2[\text{country}] = \text{China}$, and $r_2[\text{capital}] \in \{\text{Shanghai}, \text{Hongkong}\}$. Similarly, we have $r_4 \vdash \varphi_2$.

Semantics. We next give the semantics of one fixing rule.

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 $T_p^-[B]$, then 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 3.2. As shown in Example 1.2, we can correct r_2 by applying the fixing rule φ_1. As a result, $r_2[\text{capital}]$ is changed from Shanghai to Beijing, i.e., $r_2 \rightarrow_{\varphi_1} r_2'$, where $r_2'[\text{capital}] = \text{Beijing}$ and the other attributes of r_2' remain unchanged.

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

Remark. (1) fixing rules are different from traditional data dependencies e.g., FDs [Abiteboul et al. 1995] and CFDs [Fan et al. 2008]. 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 [Fan et al. 2012] 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 refer to Example 1.2 for more details.

(3) We have also investigated how to generate fixing rules. Inspired by the work of Singh and Gulwani [2012], which learns transformation rules from examples, we discuss in Section 7 how to generate fixing rules from examples. Moreover, we describe how to generate fixing rules by leveraging available knowledge bases, also in Section 7.

Notations. For convenience, we introduce some notations. Given a 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 Multiple Fixing Rules

We next describe the semantics of applying a set of fixing rules. Note that when applying a 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. As will be seen later, the order becomes irrelevant, if these rules have certain properties (see Section 4 for more details).

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.

Consider a fixing rule φ, 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) $B_\varphi \notin \mathcal{A}$.

This shows when it is correct to apply a fixing rule φ to a tuple. As $\mathcal{A}$ has been assured, we do not allow it to be changed by enforcing $B_\varphi \notin \mathcal{A}$ (condition (ii)).

Example 3.3. Consider rule φ_1 in Example 3.1 and the tuple r_2 in Figure 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 $r_2[\mathsf{country}] = \mathsf{China}$ and $r_2[\mathsf{capital}] = \mathsf{Shanghai} \in \{\mathsf{Shanghai}, \mathsf{Hongkong}\}$ (i.e., r_2 matches φ_1); and moreover, $\mathsf{capital} \notin \mathcal{A}_{r_2}$. This yields an updated tuple r_2' where $r_2'[\mathsf{capital}] = \mathsf{Beijing}$.

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

Example 3.4. Consider Example 3.3. 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., $\{\mathsf{country}\}$) and $\{B_{\varphi_1}\}$ (i.e., $\{\mathsf{capital}\}$), which results in an expanded assured attribute set $\mathcal{A}_{r_2} = \{\mathsf{country}, \mathsf{capital}\}$.

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 rule φ_i in Σ, 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 $\varphi \in \Sigma$, $\nexists t'' : t' \xrightarrow{=}_{(\mathcal{A}_k,\varphi)} t''$ and $t' \neq t''$.

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 $t \xrightarrow{*}_{(\emptyset,\Sigma)} t'$.

Symbols	Semantics
$t \vdash \varphi$	a tuple t *matches* a fixing rule φ
$t \rightarrow_{\varphi} t'$	φ is *applied* to a tuple t, and t' is the updated tuple
$t \rightarrow_{(\mathcal{A},\varphi)} t'$	φ is *properly applied* to a tuple t *w.r.t.* the assured attributes $\mathcal{A}$
$t \xrightarrow{=}_{(\mathcal{A},\varphi)} t$	φ cannot be properly applied to t
$t \xrightarrow{*}_{(\mathcal{A},\Sigma)} t'$	t' is a fix of t

Fig. 4. Summary of notations.

Example 3.5. Consider Example 3.3. Indeed, r_2' is a fix of r_2 w.r.t. rules φ_1 and φ_2 in Example 3.1, 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.

In Figure 4 we summarize the notations to be used in this article.

4. FUNDAMENTAL PROBLEMS

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

4.1. Termination

One natural question for rule-based data-repairing processes is the *termination* problem that determines whether a rule-based process will stop. In fact, it is easy 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* bounded by $|R|$, the cardinality of schema R. Hence, a fix process will always terminate.

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, then t has a unique fix by Σ.

Example 4.1. Consider a fixing rule φ_1' by adding a negative pattern to the φ_1 in Example 3.1 as the following:

$$\varphi_1' : (([\text{country}], [\text{China}]), (\text{capital}, \{\text{Shanghai}, \text{Hongkong}, \text{Tokyo}\})) \rightarrow \text{Beijing}.$$

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 as follows:

$$\varphi_3 : (([\text{capital}, \text{city}, \text{conf}], [\text{Tokyo}, \text{Tokyo}, \text{ICDE}]), (\text{country}, \{\text{China}\})) \rightarrow \text{Japan}.$$

We show that these two fixing rules, φ_1' and φ_3, are inconsistent. Consider the tuple r_3 in Figure 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

$$r_3' : (\text{Peter}, \text{China}, \text{Beijing}, \text{Tokyo}, \text{ICDE}).$$

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

$$r_3'' : (\text{Peter}, \boxed{\text{Japan, Tokyo}}, \text{Tokyo}, \text{ICDE}).$$

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'') 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 Σ to repair 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 [Fan et al. 2009] is trivial: any set of MDs is consistent [Fan et al. 2011]. They are NP-complete (resp. coNP-complete) for CFDs [Fan et al. 2008] (respectively, editing rules [Fan et al. 2012]). We shall show that the problem for fixing rules is PTIME, lower than their editing rules counterparts.

THEOREM 4.2. *The consistency problem of fixing rules is* PTIME.

We prove Theorem 4.2 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.

A dual problem of consistency is the determinism problem.

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).

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

(1) $\Sigma \cup \{\varphi\}$ is consistent; and

(2) 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 with 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 4.3. *The implication problem of fixing rules is coNP-complete. It is down to* PTIME *when the relation schema R is fixed.*

PROOF. We first show that the implication problem for fixing rules is coNP-complete for general case, and then show it is PTIME when the relational schema R if fixed.

(A) General case. We first show it is in coNP and then show it is coNP-hard.

Upper bound. The coNP upper bound is verified by providing an NP algorithm for its complement problem, based on the following *small model property*:

Given a consistent set Σ of fixing rules and another rule φ defined on the same relation schema R, Σ implies φ (i.e., $\Sigma \models \varphi$) iff (a) $\Sigma \cup \{\varphi\}$ is consistent; and (b) for any tuple t of R that draws values from active domain adom*, t has the same unique fix by both Σ and $\Sigma \cup \{\varphi\}$, where* adom *is the set of all constants appearing in Σ and φ.*

The small model property states that, while there may exist infinitely many t, it suffices to inspect those t constructed with those values in adom only.

We next present an NP algorithm for the complement of the implication problem, i.e., the algorithm returns "Yes" iff $\Sigma \not\models \varphi$.

In a nutshell, the NP algorithm works as follows:

(1) Check whether $\Sigma \cup \{\varphi\}$ is consistent; if no, return "Yes" (i.e., $\Sigma \not\models \varphi$); otherwise, continue;
(2) Guess a tuple t that draws values from adom;
(3) Check whether t has the same unique fix by $\Sigma \cup \{\varphi\}$ and Σ; if yes, go to step 2; otherwise, return "Yes" (i.e., $\Sigma \not\models \varphi$); and
(4) return "No" if all tuples constructed with values from adom pass the check in step 3 (i.e., $\Sigma \models \varphi$).

Step 1 is in PTIME. In step 3, the check is in PTIME by the semantics of fixing rules. The algorithm returns "Yes" iff there exists a tuple t that has distinct fixes by Σ and $\Sigma \cup \{\varphi\}$. Thus, it is an NP algorithm for the complement of the implication problem. That is, the implication problem is in coNP.

Lower bound. We next show the implication problem is coNP-hard by reduction from the 3SAT problem, which is NP-complete (cf. [Papadimitriou 1994]), to the complement of the implication problem.

An instance ϕ of the 3SAT problem is a well-formed Boolean formula $\phi = \mathcal{C}_1 \wedge \cdots \wedge \mathcal{C}_r$, where all variables in ϕ are $x_1, \ldots, x_m$ and C_j is of the form $\ell_1^j \vee \ell_2^j \vee \ell_3^j$, and ℓ_i^j is either x_k or $\overline{x}_k$, for $k \in [1, m]$. The problem is to determine whether there exists a truth assignment such that ϕ is *true*; i.e., ϕ is satisfiable.

Given an instance ϕ of the 3SAT problem, we define an instance of the implication problem for fixing rules, namely, a relation schema R, a set Σ of consistent fixing rules on R and another fixing rule φ on R, such that $\Sigma \not\models \varphi$ iff ϕ is satisfiable.

(1) We define the relation schema R to be $(X_1, \ldots, X_m, C, S)$, where the data type of each attribute is *Boolean*. In other words, the possible values of each attribute are true and false. Intuitively, for each tuple t in an instance I of R, $t[X_1, \ldots, X_m]$ encodes a truth assignment of the variables $x_1, \ldots, x_m$. We next define the consistent set Σ of fixing rules where $|\Sigma| = r$ and an extra fixing rule φ, and we illustrate attributes C and S below.

(2) We define Σ to be a set of fixing rules, where $|\Sigma| = r$, is of the following form:

$$((\mathcal{X}_j, t_p[\mathcal{X}_j]), (C, \{\mathsf{true}\})) \rightarrow \mathsf{false},$$

where for each clause $\mathcal{C}_j = \ell_1^j \vee \ell_2^j \vee \ell_3^j$ ($j \in [1, r]$), $\mathcal{X}_j = [\mathsf{ind}(\ell_1^j), \mathsf{ind}(\ell_2^j), \mathsf{ind}(\ell_3^j)]$, where $\mathsf{ind}(\ell_i^j) = X_k$ if $\ell_i^j = \overline{x}_k$ or x_k. For each attribute X_k in $\mathcal{X}_j$, $t_p[X_k] = \mathsf{false}$

if x_k appears in $\mathcal{C}_j$, and $t_p[X_k] = \mathsf{true}$ if $\overline{x}_k$ appears in $\mathcal{C}_j$. For example, consider $\mathcal{C}_1 = x_1 \vee \overline{x}_2 \vee x_3$. Based on the mapping, we will construct a fixing rule of the form $(([X_1, X_2, X_3], [\mathsf{false}, \mathsf{true}, \mathsf{false}]), (C_1, \{\mathsf{true}\})) \rightarrow \mathsf{false}$.

(3) We define φ as

$$(([S], [\mathsf{false}]), (C, \{\mathsf{true}\})) \rightarrow \mathsf{false}.$$

Intuitively, for any tuple t defined on R with $t[S] = \mathsf{false}$ and $t[C] = \mathsf{true}$, applying φ will change $t[C]$ to false.

We now show the correctness of this reduction, i.e., ϕ is satisfiable if and only if $\Sigma \not\models \varphi$. It is easy to see that Σ is consistent, because each rule in Σ only modifies the attributes of C_j ($j \in [1, r]$), which has no impact on the other rules. We next show that there exists a tuple t such that t has two distinct fixes by Σ and $\Sigma \cup \{\varphi\}$ iff there exists a truth assignment μ of variables $x_1, \ldots, x_m$ that satisfies ϕ.

$\boxed{\Rightarrow}$ Assume ϕ is satisfiable. That is, there exists a truth assignment μ_0 for variables $x_1, \ldots, x_m$ such that $\mu_0(\phi) = \mathit{true}$. Consider the tuple $t = (\mu_0(x_1), \ldots, \mu_0(x_m), \mathsf{true}, \mathsf{false})$. We can observe that there is no fixing rule in Σ that can be properly applied to t. Thus, t has a unique fix $t_1 = (\mu_0(x_1), \ldots, \mu_0(x_m), \mathsf{true}, \mathsf{false})$ by Σ. This observation can be proved by contradiction. Assuming there exists a fixing rule that can be properly applied to the tuple. Let the corresponding clause be $\mathcal{C}_j = \ell_1^j \vee \ell_2^j \vee \ell_3^j$. We next prove that $\ell_1^j \vee \ell_2^j \vee \ell_3^j$ must be false, which contradicts the initial assumption (i.e., ϕ is satisfiable). Consider ℓ_i^j ($i \in [1, 3]$). It can be either x_k or $\overline{x}_k$.

- (i) If $\ell_i^j = x_k$, then in the fixing rule constructed for $\mathcal{C}_j$, we have $t_p[X_k] = \mathsf{false}$. Since the rule can be properly applied to the tuple t, we have $t[X_k] = \mathsf{false}$. Based on the way to construct the tuple t, we have $x_k = \mathsf{false}$. Thus, $\ell_i^j = x_k = \mathsf{false}$.
- (ii) If $\ell_i^j = \overline{x}_k$, then we have $t_p[X_k] = \mathsf{true}$ in the corresponding fixing rule. Since the rule can be properly applied to the tuple t, we have $t[X_k] = \mathsf{true}$. Based on the way to construct the tuple t, we have $x_k = \mathsf{true}$. Thus, $\ell_i^j = \overline{x}_k = \mathsf{false}$.

In any case, $\ell_i^j = \mathsf{false}$ ($i \in [1, 3]$). Thus, the observation is proved, and t has a unique fix $t_1 = (\mu_0(x_1), \ldots, \mu_0(x_m), \mathsf{true}, \mathsf{false})$ by Σ. However, if we consider $\Sigma \cup \{\varphi\}$, then t will have a distinct unique fix $t_2 = (\mu_0(x_1), \ldots, \mu_0(x_m), \mathsf{false}, \mathsf{false})$, since the extra rule φ enforces $t[C]$ from true to false. Therefore, $\Sigma \not\models \varphi$.

Here is an example to show how we construct the tuple t. Suppose $\mathcal{C}_1 = x_1 \vee \overline{x}_2 \vee x_3$ and $\mathcal{C}_2 = \overline{x}_1 \vee \overline{x}_2 \vee x_3$. Consider a truth assignment is $\mu_0(x_1) = \mathsf{true}$, $\mu_0(x_2) = \mathsf{false}$, and $\mu_0(x_3) = \mathsf{true}$. Then the tuple t is (true, false, true, true, false)

$\boxed{\Leftarrow}$ Suppose ϕ is not satisfiable. We next prove $\Sigma \models \varphi$. That is, for any tuple $t = (t[X_1], \ldots, t[X_m], t[C], t[S])$, t has the same unique fix by Σ and $\Sigma \cup \{\varphi\}$. We consider t in four cases:

- $t = (t[X_1], \ldots, t[X_m], \mathsf{false}, \mathsf{false})$. In this case, $t[C] = \mathsf{false}$. For the fixing rules in $\Sigma \cup \{\varphi\}$, to properly apply them to t, it is required that $t[C] = \mathsf{true}$. Thus, none of the rules can be properly applied to the tuple. t has the same unique fix by Σ and $\Sigma \cup \{\varphi\}$.
- $t = (t[X_1], \ldots, t[X_m], \mathsf{false}, \mathsf{true})$. Since $t[C] = \mathsf{false}$, similarly, we can prove t has the same unique fix by Σ and $\Sigma \cup \{\varphi\}$ in this case.
- $t = (t[X_1], \ldots, t[X_m], \mathsf{true}, \mathsf{false})$. Consider the assignment μ', where $\mu'(x_k) = t[X_k]$ for $k \in [1, m]$. Since ϕ is not satisfiable, there exists at least one clause such that $\mathcal{C}_j = \ell_1^j \vee \ell_2^j \vee \ell_3^j$ is false. That is, $\ell_i^j = \mathsf{false}$ ($i \in [1, 3]$). We next prove the fixing rule constructed for $\mathcal{C}_j$ can be properly applied to the tuple t.

  Consider ℓ_i^j ($i \in [1, 3]$). It can be either x_k or $\overline{x}_k$.

(1) In the case of $t[X_k] = \mathsf{true}$, to ensure $\ell_i^j = \mathsf{false}$, we can obtain $\ell_i^j = \overline{x}_k$. Since $\ell_i^j = \overline{x}_k$, based on the way to construct the fixing rule w.r.t. $\mathcal{C}_j$, we have $t_p[X_k] = \mathsf{true}$. Thus, $t_p[X_k] = t[X_k] = \mathsf{true}$.

(2) In the case of $t[X_k] = \mathsf{false}$, to ensure $\ell_i^j = \mathsf{false}$, we can obtain $\ell_i^j = x_k$. Since $\ell_i^j = x_k$, based on the way to construct the fixing rule w.r.t. $\mathcal{C}_j$, we have $t_p[X_k] = \mathsf{false}$. Thus, $t_p[X_k] = t[X_k] = \mathsf{false}$.

We can see in either case, the evidence pattern of the fixing rule matches the tuple, thus, the rule can be properly applied to the tuple, enforcing $t[C]$ to false. Therefore, the unique fix of t corresponding to Σ is $(t[X_1], \ldots, t[X_m], \mathsf{false}, \mathsf{false})$. Note that the extra fixing rule φ enforces $t[C]$ to false, which coincides with the unique fix. Hence, t has the same unique fix by Σ and $\Sigma \cup \{\varphi\}$.

- $t = (t[X_1], \ldots, t[X_m], \mathsf{true}, \mathsf{true})$. Similar to the above proof, we can obtain that the unique fix of t corresponding to Σ is $(t[X_1], \ldots, t[X_m], \mathsf{false}, \mathsf{true})$. Note that the extra fixing rule φ does not have any impact on the tuple, because its evidence pattern does not match the tuple. Hence, t has the same unique fix by Σ and $\Sigma \cup \{\varphi\}$.

Thus, the implication problem is coNP-complete.

(B) Special case: R is fixed. When R is fixed, with the small model property, only polynomially many tuples need to be guessed and checked in the algorithm presented in the upper bound proof. Thus, it is down to PTIME in this special case. □

5. ENSURING CONSISTENCY

Our next goal, after studying the consistency problem in Section 4.2, 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, then 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, to obtain high-quality fixing rules.

5.2. Checking Consistency

We first present a proposition, which is important to design efficient algorithms for checking the consistency of a set of fixing rules.

PROPOSITION 5.1. *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. Let n be the number of rules in Σ. When $n = 1$, Σ is trivially consistent. When $n = 2$, Σ is consistent, the same as φ_i and φ_j are consistent ($i \neq j$). When $n \geq 3$, we prove by contradiction.

$\boxed{\Rightarrow}$ Suppose this proposition is *false*.

This conditional statement being *false* means that although the fixing rules are pairwise consistent, when putting together, 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:

$$S_1 : t = t_0 \rightarrow_{(\emptyset,\varphi_1)} t_1 \ldots \rightarrow_{(\mathcal{A}_{i-1},\varphi_i)} t_i \ldots \rightarrow_{(\mathcal{A}_{m-1},\varphi_m)} t_m = t',$$
$$S_2 : t = t'_0 \rightarrow_{(\emptyset,\varphi'_1)} t'_1 \ldots \rightarrow_{(\mathcal{A}'_{j-1},\varphi'_j)} t'_j \ldots \rightarrow_{(\mathcal{A}'_{n-1},\varphi'_n)} t'_n = t''.$$

We consider the following three cases: (1) $\mathcal{A}_m \cap \mathcal{A}'_n = \emptyset$; (2) $\mathcal{A}_m \cap \mathcal{A}'_n \neq \emptyset$ and $t'[\mathcal{A}_m \cap \mathcal{A}'_n] = t''[\mathcal{A}_m \cap \mathcal{A}'_n]$; and (3) $\mathcal{A}_m \cap \mathcal{A}'_n \neq \emptyset$ 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 $\mathcal{A}'_n = \mathcal{A}'_{n-1} \cup X_{\varphi'_n} \cup \{B_{\varphi'_n}\}$. We shall prove in the following that each case will lead to a contradiction to the assumption.

Case 1 [$\mathcal{A}_m \cap \mathcal{A}'_n = \emptyset$]. We prove that t' is not a fixpoint. Observe that (1) each rule in S_1 will only change attributes in $\mathcal{A}_m$ and (2) $\mathcal{A}_m \cap \mathcal{A}'_n = \emptyset$, so it is clear that $t'[\mathcal{A}'_n] = t[\mathcal{A}'_n]$. That is, the first sequence will not change any attribute in $\mathcal{A}'_n$. As a consequence, the rule φ'_1 in the second sequence can be properly applied to t', which proves that t' is not a fixpoint.

The above analysis proved that if the two non-empty sequences of fixes lead to two distinct fixpoints, the intersection of the two attribute sets, $\mathcal{A}_m$ and $\mathcal{A}'_n$, cannot be empty. In other words, we were wrong to assume that their intersection is empty.

Case 2 [$\mathcal{A}_m \cap \mathcal{A}'_n \neq \emptyset$ and $t'[\mathcal{A}_m \cap \mathcal{A}'_n] = t''[\mathcal{A}_m \cap \mathcal{A}'_n]$]. Let $X_{mn} = \mathcal{A}_m \cap \mathcal{A}'_n$. Since $t'[\mathcal{A}_m \cap \mathcal{A}'_n] = t''[\mathcal{A}_m \cap \mathcal{A}'_n]$, we have either (a) $t'[\mathcal{A}_m \backslash X_{mn}] \neq t''[\mathcal{A}_m \backslash X_{mn}]$ or (b) $t'[\mathcal{A}'_n \backslash X_{mn}] \neq t''[\mathcal{A}'_n \setminus X_{mn}]$, which are symmetric cases.

In the following, we focus on case 2(a). Assume that B is the first attribute appearing in S_1, such that $B \in \mathcal{A}_m \setminus X_{mn}$ and $t_i[B] \neq t''[B]$. Since B is the first such attribute, it is natural to follow that $t_i[X_{\varphi_i}] = t''[X_{\varphi_i}]$. Recall that $B \in \mathcal{A}_m \setminus X_{mn}$, which means that the rule φ_i can be properly applied to t'', which contradicts to the assumption that S_2 reaches a fixpoint. Similarly, we can prove case 2(b) is a contradiction.

By putting case 2(a) and case 2(b) together, it suffices to prove that the whole case 2 contradicts to the assumption that they lead to two distinct fixes.

Case 3 [$\mathcal{A}_m \cap \mathcal{A}'_n \neq \emptyset$ and $t'[\mathcal{A}_m \cap \mathcal{A}'_n] \neq t''[\mathcal{A}_m \cap \mathcal{A}'_n]$]. We shall prove that in this case, there must exist two fixing rules that are inconsistent.

Let, without loss of generality, φ_i at position i of sequence S_1 be the first rule such that $t_i[\mathcal{A}_m \cap \mathcal{A}'_n] \neq t''[\mathcal{A}_m \cap \mathcal{A}'_n]$. In other words, for all $k < i$, $t_k[\mathcal{A}_m \cap \mathcal{A}'_n] = t''[\mathcal{A}_m \cap \mathcal{A}'_n]$.

Here, $t_{i-1}[\mathcal{A}_m \cap \mathcal{A}'_n] = t''[\mathcal{A}_m \cap \mathcal{A}'_n]$ and $t_i[\mathcal{A}_m \cap \mathcal{A}'_n] \neq t''[\mathcal{A}_m \cap \mathcal{A}'_n]$ imply that $t_i[B_{\varphi_i}] \neq t''[B_{\varphi_i}]$. Moreover, since $B_{\varphi_i} \in \mathcal{A}_m \cap \mathcal{A}'_n$, we have $B_{\varphi_i} \in \mathcal{A}'_n$. This further shows that in sequence S_2, there exists a fixing rule φ'_j where $B_{\varphi_i} \in X_{\varphi'_j} \cup \{B_{\varphi'_j}\}$. In the following, we will show that φ_i and φ'_j are indeed inconsistent, by constructing a tuple r that leads to different fixes.

Since $B_{\varphi_i} \in X_{\varphi'_j} \cup \{B_{\varphi'_j}\}$, we consider two cases: (a) $B_{\varphi_i} = B_{\varphi'_j}$ and (b) $B_{\varphi_i} \in X_{\varphi'_j}$.

For case (a), we have $t_i[X_{\varphi_i}] = t''[X_{\varphi_i}]$ and $t'_j[X_{\varphi'_j}] = t''[X_{\varphi'_j}]$. Let the tuple r be $r[X_{\varphi_i}] = t''[X_{\varphi_i}]$ and $r[X_{\varphi'_j}] = t''[X_{\varphi'_j}]$, it is clear that both φ_i and φ'_j can be applied to r, while leading to two different fixpoints. In other words, φ_i and φ'_j are inconsistent.

For case (b), we construct the tuple r as the above. (i) By using φ'_j first, the attribute $r[B_{\varphi_i}]$ is *assured* and cannot be changed. (ii) By using φ_i first, the attribute $r[B_{\varphi_i}]$ will be changed. From the above two cases, it shows that φ_i and φ'_j are inconsistent.

Putting contradiction cases 1–3 together, it suffices to prove that we were wrong to assume that the proposition is false.

$\boxed{\Leftarrow}$ 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 sequences of fixes S_1 and S_2 on t by using the rules in Σ. In S_1, we apply φ_i first; while in S_2, we apply φ_j first. We prove that these two sequences must yield two different fixes. This suffices to show that we were wrong to assume that there exist inconsistent φ_i and φ_j. □

Proposition 5.1 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 the consistency of fixing rules. Next, we describe two algorithms to check the consistency of two fixing rules, by using the result from Proposition 5.1. One algorithm is based on tuple enumeration, while the other is through rule characterization.

5.2.1. Tuple Enumeration. We first consider 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 5.2. *Two 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 both $t \vdash \varphi_i$ and $t \vdash \varphi_j$ hold, for any such t, then 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 5.2 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 5.3. Consider rules φ_1 and φ_2 in Example 3.1. 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 as follows:

(○, 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 tuple can both match φ_1 and φ_2.

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 $V_{\varphi_{ij}}(A_l)$.

Algorithm isConsist^t**.** Figure 5 shows the pseudo-code of the algorithm. Given a set Σ of fixing rules, we check them pairwise. For each pair, φ_i and φ_j, we enumerate $\prod_{l\in[1,m]}(|V_{\varphi_{ij}}(A_l)|)$ possible tuples. If there exists a tuple that has two unique fixes w.r.t. $\{\varphi_i, \varphi_j\}$, then we judge that Σ is inconsistent. If such a case does not happen for any pair of rules, then Σ is consistent.

Algorithm isConsist^t
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. Let $\{A_1, \ldots, A_m\}$ be all attributes appearing in φ_i and φ_j;
3. 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;
4. **for** each tuple $t \in V_{\varphi_{ij}}(A_1) \times \ldots \times V_{\varphi_{ij}}(A_m)$ **do**
5. **if** t does not match φ_i or φ_j **then**
6. **continue**;
7. Let t_1 be the updated tuple after first applying φ_i and then φ_j to t;
8. Let t_2 be the updated tuple after first applying φ_j and then φ_i to t;
9. **if** $t_1 \neq t_2$ **then**
10. **return** *false*;
11. **return** *true*;

Fig. 5. Consistency check via tuple enumeration.

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

Based on Lemma 5.2, 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 they are represented as follows:

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

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 a case is ($T^-_{p_i}[B] \cap T^-_{p_j}[B] \neq \emptyset$ and $t^+_{p_i}[B] \neq t^+_{p_j}[B]$). The following example shows two inconsistent fixing rules for this case:

$$\varphi_i : (([\text{country}], [\text{China}]), (\text{capital}, \{\text{Shanghai}, \text{Hongkong}\})) \rightarrow \text{Beijing},$$
$$\varphi_j : (([\text{country}], [\text{China}]), (\text{capital}, \{\text{Shanghai}\})) \rightarrow \text{Xian}.$$

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 $B_i \in X_j$, then it will keep $t[B_i]$ unchanged, whereas if φ_i is applied first, then 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)). The following example provides

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*;

Fig. 6. Consistency check via rule characterization.

two inconsistent fixing rules under this case:

φ_i : (([country], [China]), (capital, {Shanghai, Tokyo})) → Beijing,
φ_j : (([capital], [Tokyo]), (conf, {SIGMOD})) → ICDE.

(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 $t_{p_j}[B_i] \in T^-_{p_i}[B_i]$. The following example provides two inconsistent fixing rules under this case:

φ_i : (([country], [China]), (capital, {Tokyo, Shanghai})) → Beijing,
φ_j : (([capital], [Tokyo]), (country, {China})) → Japan.

(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 (respectively, φ_j) will deterministically update $t[B_i]$ (respectively, $t[B_j]$) to $t^+_{p_i}[B_i]$ (respectively, $t^+_{p_j}[B_j]$). That is, φ_i and φ_j are trivially consistent in this case.

Example 5.4. Consider φ'_1 and φ_3 in Example 4.1 and φ_2 in Example 3.1. Since φ'_1 (respectively, φ_2) is only applied to a tuple whose country is China (respectively, Canada), there does not exist any tuple that can match both rules at the same time. Therefore, based on Lemma 5.2, 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 ∈ {country}),
(ii) $t_{p_1}[B_{\varphi_3}] \in T^-_{p_3}[B_{\varphi_3}]$ (i.e., China ∈ {China}),
(iii) $B_{\varphi'_1} \in X_{\varphi_3}$ (i.e., capital ∈ {capital, city, conf}), and
(iv) $t_{p_3}[B_{\varphi'_1}] \in T^-_{p_1}[B_{\varphi'_1}]$ (i.e., Tokyo ∈ {Shanghai, Hongkong, Tokyo}).

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

Algorithm isConsist^r**.** The algorithm to check whether a set of fixing rules is consistent via rule characterization, referred to as isConsist^r, is given in Figure 6. It takes Σ as input and returns a Boolean value, where *true* indicates that Σ is consistent and *false* otherwise.

Fig. 7. Illustrations in resolving conflicts.

It enumerates all pairs of distinct rules (lines 1–11). If any pair is inconsistent, then 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 and 7), case 2(b) (lines 8 and 9) and case 2(c) (lines 10 and 11). Note that in case 2(d), two rules are trivially consistent; there is no need to investigate such case.

Correctness and 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 5.1 and Lemma 5.2. 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($\Sigma$) represents the physical size of Σ, which is different from its cardinality.

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 5.4. Figure 7 highlights the values that result in the conflict. A conservative solution is to remove all the rules that are in conflict. 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.

Hence, to obtain high-quality rules, we ask experts to examine rules that are in conflict. For example, the experts can naturally remove Tokyo from the negative patterns of φ_1', which will result in a modified rule φ_1 (see Example 3.1) and φ_1 is consistent with φ_3. Note that 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 $\mathcal{O}(\text{size}(\Sigma)|R|)$, where R is the relation. To make the solution efficient enough in practice, 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 [Abiteboul et al. 1995]. Note that each tuple will be repaired independently. In the following, our discussion is about how to repair one tuple, and all tuples will be repaired one by one. We next present an algorithm to repair a tuple with consistent fixing rules. It iteratively picks a fixing rule that can be applied, until a *fix* is reached.

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. $\mathcal{A} := \mathcal{A} \cup X_\varphi \cup \{B_\varphi\}$;
8. $\Gamma := \Gamma \setminus \{\varphi\}$;
9. updated := *true*;
10. **return** t';

Fig. 8. Chase-based repairing algorithm.

Algorithm. The algorithm, referred to as cRepair, is shown in Figure 8. It takes as input a tuple t and a set Σ of consistent fixing rules. It returns a repaired tuple t' w.r.t. the given set of rules 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 2–9). If there is a rule that can be properly applied (line 5), then it updates the tuple (line 6), maintains the assured attributes and rules that can be used correspondingly, and flags this change (lines 7–9). It terminates when no rule can be further properly applied (line 2), and the repaired tuple will be returned (line 10).

Correctness and complexity. The correctness of cRepair is naturally ensured by the Church-Rosser property, since Σ is consistent. For the complexity, observe the following. The outer loop (lines 2–9) 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 cRepair runs in $\mathcal{O}(\mathsf{size}(\Sigma)|R|)$ 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 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 to achieve 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, let's 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 $t_p[A] = a$.

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';

Fig. 9. A linear repairing algorithm.

For example, an inverted list w.r.t. φ_1 in Example 3.1 is as

$$\boxed{\mathsf{country}, \mathsf{China}} \rightarrow \boxed{\varphi_1}.$$

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 3.1 and r_2 in Figure 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 Figure 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 Figure 9. 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—a rule will be added to the list if it is applicable to t (lines 6 and 7), and uses a chase process to 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 and 11).

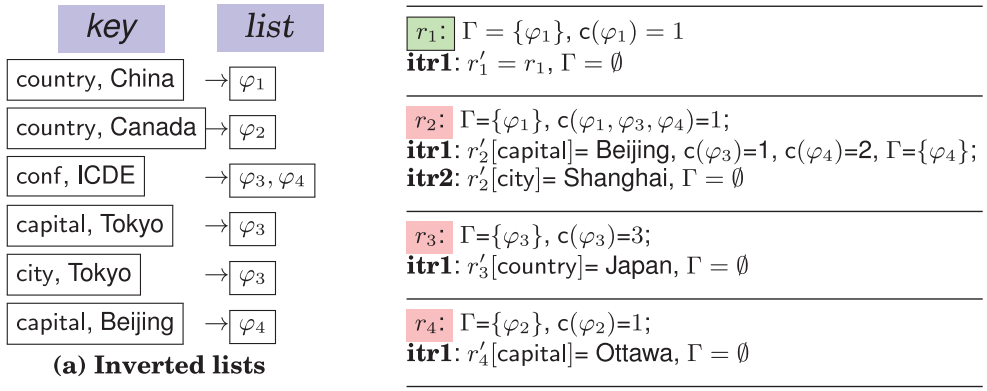


Fig. 10. A running example.

The set of attributes that is assured correct is increased correspondingly (line 12). The counters will be recalculated (lines 13 and 14). Moreover, if new rules might be used due to this update, then they will be identified (line 15). The rule that has been checked will be removed (line 16), no matter whether 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*), then 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*), then 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, lines 6 and 7) all run in time linear to size(Σ). 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. Added together, the total time complexity of the algorithm is $\mathcal{O}$(size(Σ)).

We next show by example how algorithm lRepair works.

Example 6.1. Consider Travel data D in Figure 1, rules φ_1, φ_2 in Example 3.1, and rule φ_3 in Example 4.1. In order to better understand the chase process, we introduce another rule:

$$\varphi_4 : (([\text{capital}, \text{conf}], [\text{Beijing}, \text{ICDE}]), (\text{city}, \{\text{Hongkong}\}) \rightarrow \text{Shanghai}.$$

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 Figure 10(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 Figure 10. 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 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, 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 four errors shown in Figure 1 have been corrected.

7. FIXING RULE GENERATION

In this section, we first describe how to generate fixing rules from examples (Section 7.1). We then discuss how to generate fixing rules from available knowledge bases (Section 7.2).

Note that the purpose of fixing rule generation is not to use them for some specific dataset. It is, by collecting expert knowledge for specific errors, to produce high-quality domain-related rules that can be used to automatically detect and repair data for other datasets in the same domain.

7.1. Generating fixing Rules from Examples

Indeed, human efforts are necessary to design high-quality fixing rules, which is also the case for other rule-based data-cleaning systems, for obtaining, e.g., CFDs [Fan et al. 2008], editing rules [Fan et al. 2012], and ETL rules [Batini and Scannapieco 2006; Herzog et al. 2009]. In this section, we will discuss how to leverage human efforts to generate fixing rules from examples. In order to generate fixing rules more efficiently, we divide the rule generation process into the following two phases:

Seed fixing rule generation. Since each fixing rule is defined on semantically related attributes, we start with known data dependencies (e.g., FDs) and detect violations of given FDs. Intuitively, a violation represents inconsistent answers to the same question. Based on the understanding of these violations, the experts can produce seed fixing rules by correcting inconsistent answers.

Rule enrichment. Given seed fixing rules, we enrich them by only enlarging their negative patterns (i.e., possible false answers), via extracting new negative patterns from other tables in the same domain. For instance, consider Example 1.2. 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, then one can enlarge its negative patterns by extracting large cities from a table about Chinese cities.

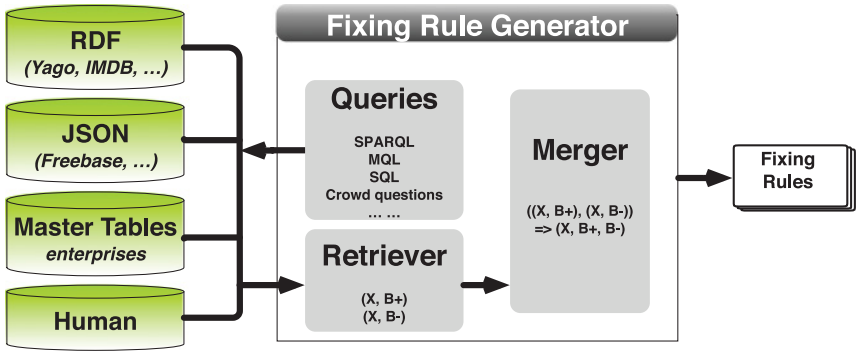


Fig. 11. Fixing Rule Generator.

Compared with editing rules, fixing rules might need more human efforts to get. However, as will be shown in the experiment (Exp-2(d)), when a fixing rule is provided, it can help to automatically repair many tuples.

7.2. Generating fixing Rules from Knowledge Bases

Nowadays, there are many publicly available knowledge bases. They contain a large amount of high-quality information, and provide convenient query interfaces for users to access such information. In this section, we will use two widely used knowledge bases (Freebase[1] and YAGO[2]) as examples to illustrate how to generate fixing rules from knowledge bases. Here, Freebase is in JSON format and YAGO is in RDF format.

7.2.1. Fixing Rule Generator Framework. Recall the definition of fixing rules in Section 3.1, a fixing rule mainly consists of three parts: evidence pattern, fact, and a set of negative patterns. Intuitively, the evidence pattern can be considered as a question, and the fact is the true answer to the question, and the set of negative patterns are the possible false answers to the question. For example, a question can be "what is the capital of China?", and the corresponding true answer is "Beijing," and possible false answers can be other big cities in China (e.g., "Shanghai" and "HongKong"). Therefore, the key point for generating fixing rules becomes how to effectively generate ⟨question, true answer⟩ pairs and ⟨question, false answers⟩ pairs.

Following this point, we propose the fixing rule generator framework in Figure 11, which aims to generate these two types of pairs by leveraging human efforts or querying knowledge bases. Let $(X, B+)$ and $(X, B-)$ denote a pair of ⟨question, true answer⟩ and a pair of ⟨question, false answers⟩, respectively. The framework will first construct specific queries according to available resources (e.g., master tables, human or knowledge bases such as YAGO and Freebase), and then retrieve a collection of $(X, B+)$ and $(X, B-)$ from the results returned by the queries. In the end, the framework will merge $(X, B+)$ and $(X, B-)$ that share the same "question" (i.e., X), and generate fixing rules in the form of $(X, B+, B-)$.

7.2.2. Extracting Fixing Rules From Freebase. Freebase is a community-curated database consisting of tens of millions of topics and billions of facts (as of January 2014). It was initially developed by the Metaweb company and was acquired by Google in 2010. Freebase provides the Metaweb Query Language (MQL) for users to access the database. Note that we use FreeBase as an example to show how we can extract fixing rules from

[1]http://www.freebase.com/.
[2]http://www.mpi-inf.mpg.de/yago-naga/yago/.

knowledge bases that use MQL. Next we will show how to use the MQL to get a set of fixing rules used in the article such as the following rules:

φ_1 : (([country], [China]), (capital, {Shanghai, Hongkong})) → Beijing,
φ_2 : (([country], [Canada]), (capital, {Toronto})) → Ottawa.

The generation process consists of two steps:
Step1: Get a list of ⟨country, capital⟩ pairs. This can be done by issuing an MQL (http://wiki.freebase.com/wiki/MQL) query to Freebase:

```
[{
  ''type'': ''/location/country'',
  ''name'': null,
  ''/location/country/capital'': []
}]
```

The above query will return a list of ⟨country, capital⟩ pairs.
Step2: For each pair obtained in step 1, issue the following query for, e.g., China:

```
[{
  ''/location/country/iso3166_1_shortname'': ''CHINA'',
  ''/location/location/contains'': [{
  ''name'': null,
  ''type'': ''/location/citytown''
  }]
}]
```

The query returns a list of cities in China. We can get negative patterns by removing capitals from the corresponding countries. The Merger will then merge the results from the above two steps to generate a set of fixing rules.

7.2.3. Extracting Fixing Rules From YAGO. YAGO is a popular knowledge base developed at the Max Planck Institute for Computer Science, which is stored in RDF format. Its information was extracted from Wikipedia, WordNet, and GeoNames. We can write SPARQL[3] queries to access the YAGO database. In the following, we will show how to generate the same fixing rules as Section 7.2.2 from YAGO.

```
select ?x, ?y, ?z
where {
      ?x rdf:type country,
      ?y rdf:type capital,
      ?z rdf:type city,
      ?x yago:hasCapital ?y,
      ?z yago:locatedIn ?x
}
```

From the above query, you will get a set of triples for (country, capital, city). Then the Merger will merge them to get fixing rules.

Discussion. There exist many reliable data sources stored in other formats, e.g., relational tables, XML data, or Google knowledge graph. Similarly to the process discussed above, one can readily write SQL queries for relational tables, XQuery (or XPath) queries for XML data, or Google knowledge graph search API for Google knowledge graph to retrieve the information for generating fixing rules, by following the general framework shown in Figure 11.

[3]https://www.w3.org/TR/sparql11-query/.

Moreover, in practice, it is hard to generate the queries automatically, especially for the negative patterns. Suppose an evidence pattern and the true answer of a fixing rule are extracted from a knowledge base. To generate the negative patterns for the rule, intuitively, we want to find other values than the true answer that are *related* to the evidence pattern as the negative patterns. For the country-capital example, those values can be the non-capital cities of a country, because they are related to the country but are not the same as the true answer. For another example, if one wants to generate a rule to correct the year of birth of famous painters, he/she can use some other years related to the painters (e.g., the year of death) as their negative patterns. Nevertheless, the whole semantics of rules is fully in the hands of a user, since it relies on the user to determine which values are related. Typically, practitioners, who know the semantics of the rules, will write the queries in a *trial-and-error* fashion, until they obtain satisfactory results, e.g., negative patterns, from the knowledge bases.

8. 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; (3) the accuracy of our fixing rule generation framework; and (4) the efficiency of data repairing algorithms using fixing rules. Note that the above (3) is new w.r.t. the conference article [Wang and Tang 2014].

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.

8.1. Experimental Setting

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

(1) HOSP was taken from US Department of Health & Human Services.[4] It has 115K records with the following attributes: Provider Number (PN), Hospital Name (HN), address1, address2, address3, city, state, zip, county, Phone Number (phn), Hospital Type (ht), Hospital Owner (ho), Emergency Service (es) Measure Code (MC), Measure Name (MN), condition, and stateAvg.

(2) UIS data was generated by the UIS Database generator[5]. It produces a mailing list that has the following schema: RecordID, ssn, First Name (fname), Middle Init (minit), Last Name (lname), stnum, stadd, apt, city, state, zip. We generated 15K records.

Dirty data generation. We treated original HOSP and UIS datasets as clean data. 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). For example, a noise rate of 10% means that 90% of the tuples are clean and the remaining 10% of the tuples have noises. We introduced two types of noises: typos and errors from the active domain. Specifically, given a tuple, we randomly selected some attributes to add noise. For each selected attribute, the type of the noise added to the attribute is controlled by typo rate (50% by default). Many state-of-the-art systems use similar techniques [Chu et al. 2013b; Dallachiesa et al. 2013].

Fixing rule generation. We generated 1000 fixing rules for HOSP data and 100 fixing rules for UIS data. These fixing rules are generated with the help of domain experts. Section 7.1 describes the details of the generation process. In addition, we also

[4]http://www.hospitalcompare.hhs.gov/.
[5]http://www.cs.utexas.edu/users/ml/riddle/data.html.

Table I. FDs for HOSP and UIS data

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

generated fixing rules from knowledge bases and compared their accuracy with these human-generated fixing rules in Exp-3.

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 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. Table I shows the corresponding FDs to HOSP and UIS data, respectively. The following shows a couple of examples of the fixing rules generated for the datasets:

HOSP : (([MC], [HF-2]), (condition, {Pneumonia, HeartAttack}) → HeartFailure,
HOSP : (([MC], [AMI-4]), (condition, {HeartAttack_typo}) → HeartFailure,
UIS : (([ZIP], [35808]), (STATE, {OH}) → AL,
UIS : (([ZIP], [96944]), (CITY, {kosrae_typo}) → kosrae.

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 (Figure 6 in Section 5.2); (3) cRepair: the basic chase-based algorithm for repairing with fixing rules (see Figure 8); and (4) lRepair: the fast repairing algorithm (see Figure 9). Moreover, for comparison, we have implemented two algorithms for FD repairing, a cost-based heuristic method [Bohannon et al. 2005], referred to as Heu, and an approach for cardinality-set-minimal repairs [Beskales et al. 2010], 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.

8.2. 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 Figure 12(a) (respectively, Figure 12(b)). The x-axis is the number of rules divided by 100 (respectively, 10) for HOSP (respectively, UIS), and the y-axis is the running time in millisecond (msec).

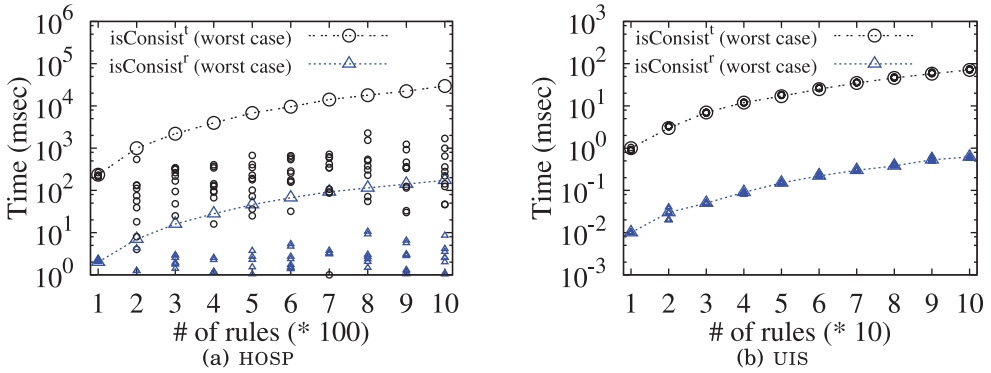


Fig. 12. Efficiency for checking consistency.

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 Figure 12(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 Figure 12(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 12s to check the consistency of 1000*1000 pairs of rules, i.e., the top right point in Figure 12(a).

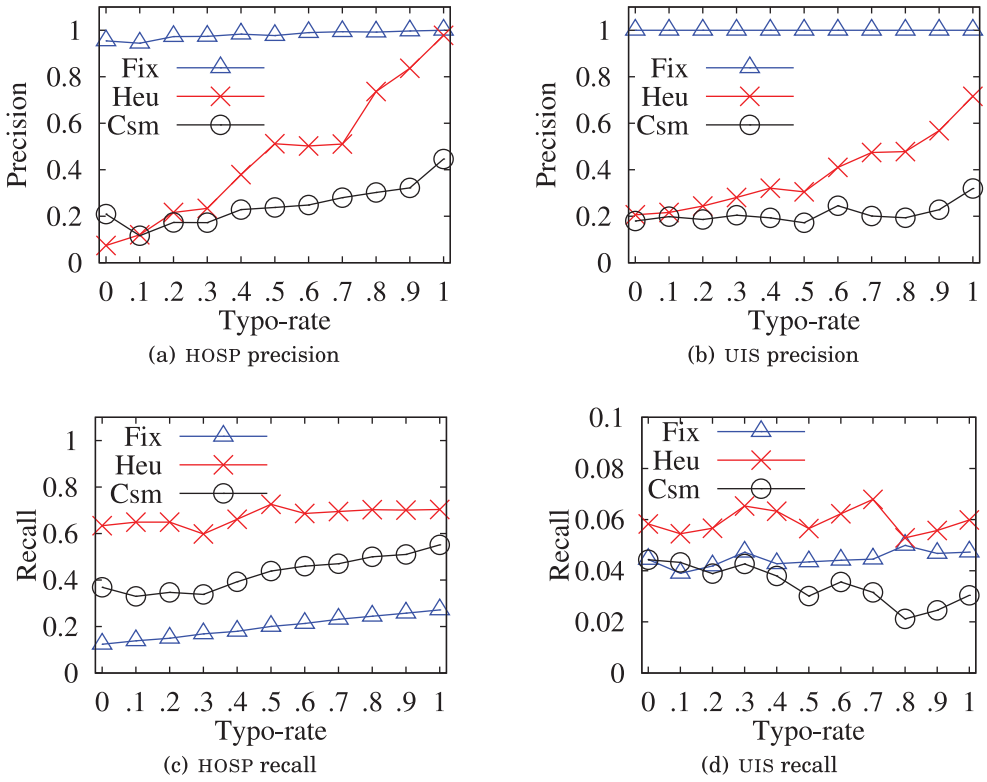
The results of this study indicates that it is feasible to check consistency for a reasonably large set of fixing rules.

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 the domain) for repairing algorithms. (b) The influence of fixing rules w.r.t. their number and negative patterns. (c) Comparison with editing rules. We use Fix to represent repairing algorithms with fixing rules.

(a) Noise from the 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., values from the domain that look like typos).

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 Figures 13(a) and 13(b) 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. 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 that assume that left-hand values of a constraint are correct 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 Figure 13(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 Figure 13(a)), it will mislead



(a) HOSP precision (b) UIS precision

(c) HOSP recall (d) UIS recall

Fig. 13. Accuracy of data repairing on different data errors.

fixing rules to make decisions. For example, consider the two rules in Figure 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, Figures 13(c) and 13(d) show the recall corresponding to Figures 13(a) and 13(b), 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 many 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.

Figure 13(d) 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 is 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.

F-Measure. We compared the f-measure of three data-repairing algorithms on UIS and HOSP datasets, where the f-measure is defined as the harmonic mean of precision and recall. Table II shows the result (typo rate = 50%). We can see that Heu has a higher f-measure than our algorithm, Fix. This is because Fix aims for dependable data repairing. It only fixed the values that were very likely to be wrong. Thus, it got

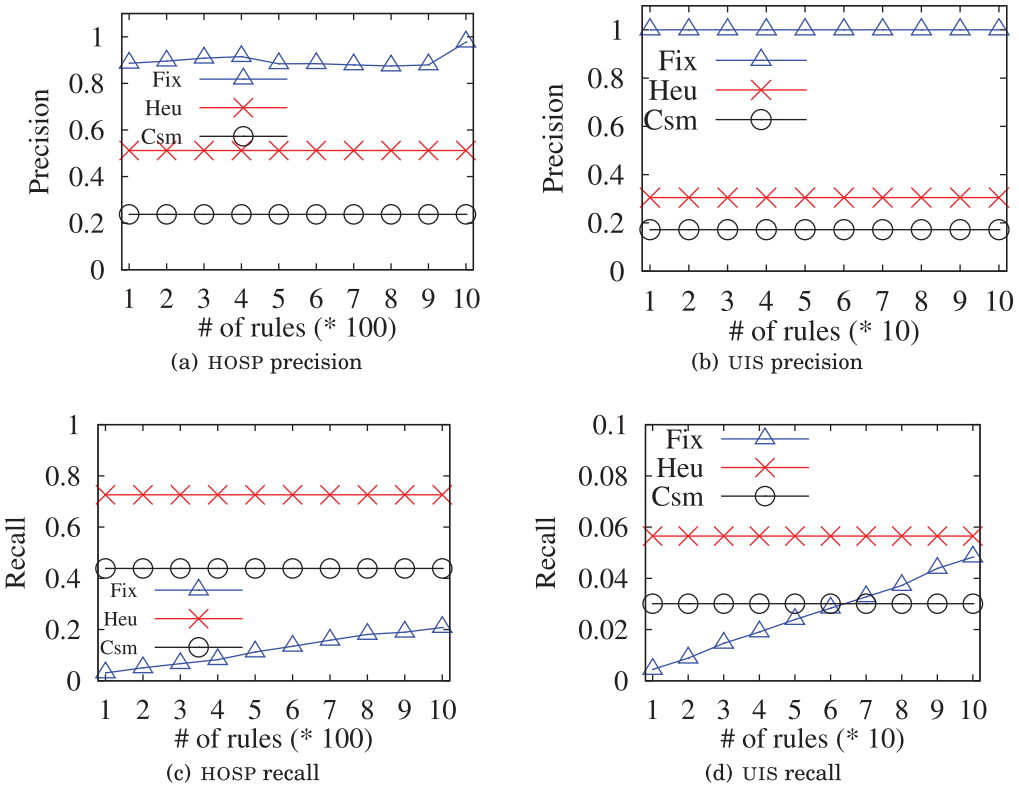


Fig. 14. Accuracy of data repairing by varying the number of fixing rules.

the highest precision but the recall was not as high as other algorithms. In the next experiments (Exp-2(b) and Exp-2(c)), we will investigate how increasing the number of fixing rules or the number of negative patterns can improve the recall.

(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 them are typos. For HOSP, we varied the number of rules from 100 to 1000 and reported the recall and precision in Figures 14(a) and 14(c), respectively. For UIS, we varied the number of rules from 10 to 100, and reported the results in Figures 14(b) and 14(d), respectively. For Heu and Csm, as these two algorithms don't employ fixing rules, 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 Figure 15(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 collected the negative patterns of all fixing rules and sorted them in a random order. We then picked up the first k negative patterns from the collection, denoted by C_k, and modified each fixing rule by removing its negative patterns that do not appear in C_k. Thus, for any given k, we can obtain a set of new fixing rules. We varied the number of negative patterns (i.e., k) and evaluated the accuracy of our repairing algorithms using the set of fixing rules for each k. Figure 15(b) shows the

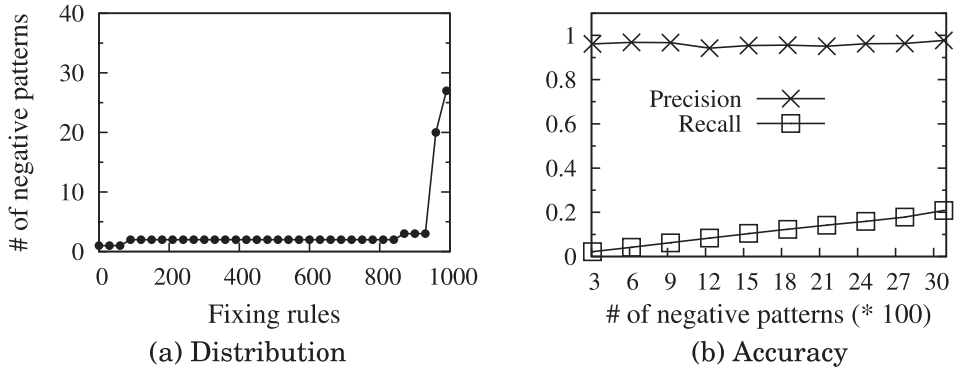


Fig. 15. Evaluation for negative patterns (HOSP).

Table II. Comparing the F-measure of Data-repairing Methods (Typo Rate = 50%)

	Algorithms	Precision	Recall	F-Measure
HOSP	Fix	**0.977**	0.208	0.343
	Heu	0.512	**0.726**	**0.600**
	Csm	0.238	0.439	0.308
UIS	Fix	**1**	0.048	0.092
	Heu	0.716	**0.056**	**0.105**
	Csm	0.319	0.030	0.055

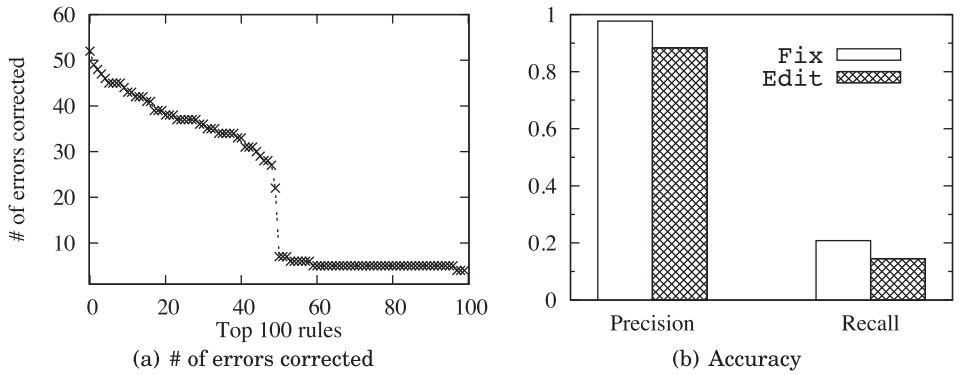


Fig. 16. Comparison with editing rules (HOSP).

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. The experimental result further validates the dependable feature of fixing rules.

(d) Comparison with editing rules. We also compared our approach with editing rules [Fan et al. 2012]. Note that editing rules can repair data and ensure the repairing operation is correct. However, 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 Figure 16(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. The experiment shows that a single fixing rule was able to repair errors in

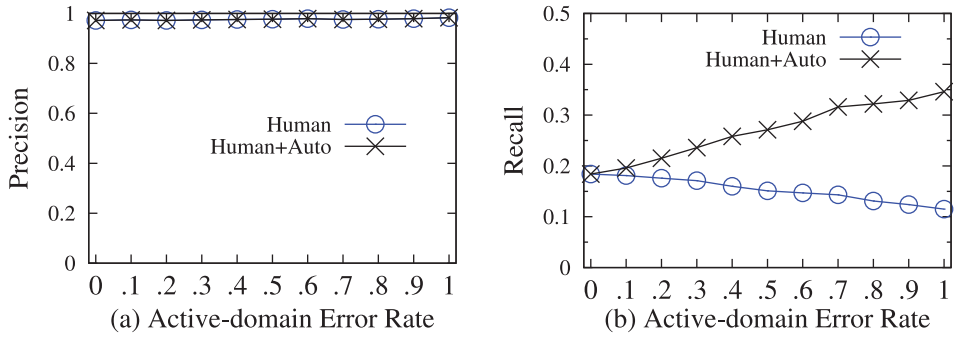


Fig. 17. Effectiveness of fixing rule generation.

more than 50 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 Figure 16(b), where Fix (respectively, Edit) indicates fixing rules (respectively, 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, as a consequence, lower recall. Note that the purpose of designing editing rules is for critical data at entry point by interacting with the users. Hence, we don't compare with them.

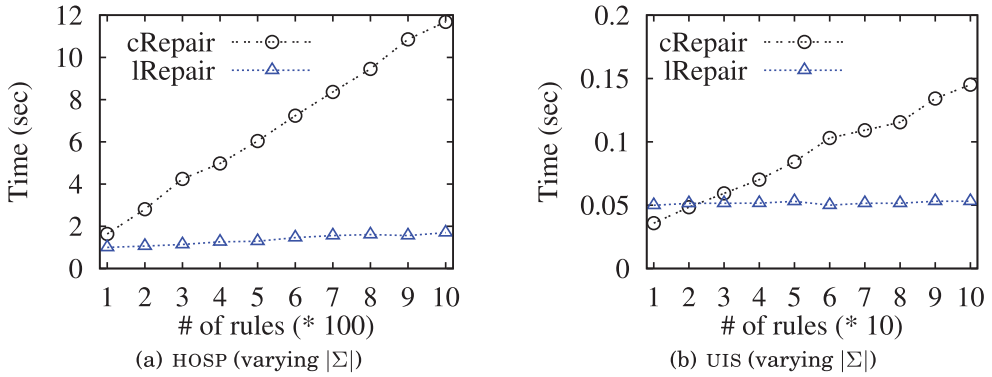
Exp-3: Effectiveness of fixing rule generation. We also evaluated the effectiveness of fixing rule generator framework described in Section 7.2. We constructed a new dataset by enriching the film actors in the IMDB database[6] with four additional fields, i.e., country, capital, currency, and language. The dataset consists of 199,698 records. We used the same method for HOSP and UIS datasets to corrupt the data and to generate 100 fixing rules for the data. In addition, we applied the fixing rule generator framework to automatically generate a size of 87,481 fixing rules by issuing MQL queries to Freebase. The rules covered 25 evidence patterns. Both facts and negative values followed a uniform distribution. We wrote 50 MQL queries in total. Here are two examples of generated rules:

φ_1 : (([country], [AUSTRIA]), (capital, {BadEisenkappel, Lungitz, . . .})) → Vienna,
φ_2 : (([country], [FRANCE]), (capital, {Fontainebleau, Saint − Denis, . . .})) → Paris.

Note that φ_1 and φ_2 contain 1205 and 1606 negative patterns, respectively. Due to space constraints, we only showed two negative patterns for each rule.

Figure 17 shows how these auto-generated rules coupled with human-generated fixing rules can improve the accuracy of data repairing. In the figure, we varied the active-domain error rate and compared the accuracy (precision and recall) of Human and Human+Auto, where Human represents the 100 human-generated fixing rules and Human+Auto represents the union of the 100 human-generated fixing rules

[6]http://www.imdb.com.



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

Fig. 18. Efficiency for data repairing.

Table III. Comparing with Existing Repairing Methods in Efficiency

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

and the 87,481 auto-generated fixing rules. Comparing with Human, we can see that Human+Auto does not lose any precision but increases recall a lot, which validates the effectiveness of our fixing rule generation framework. Another interesting observation is that Human+Auto achieved better recall when the active-domain error rate was increasing. Intuitively, this is because knowledge bases can help us obtain more negative patterns or facts than human-domain knowledge.

Exp-4: 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 Figures 18(a) and 18(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 2s to repair 115K tuples, using 1000 rules (the bottom right node in Figure 18(a)). In Figure 18(b), cRepair was faster only when the number of rules 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 have also compared our fast repairing algorithm lRepair with Heu and Csm. Using both HOSP and UIS data, the results are given in Table III. 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 repairs data by holistically considering all violations, which have much higher time complexity.

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) Auto-generated fixing rules can further improve the recall of human-generated

fixing rules while keeping high precision (Exp-3). (e) It is efficient to repair data via fixing rules, which reveals its potential to be used for large datasets (Exp-4).

9. 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, fixing rules are able to find dependable fixes for input tuples, without using heuristic solutions; and (2) unlike editing rules, fixing rules are able to repair data automatically without any user involvement. We have formalized the problems for deciding whether a set of fixing rules are 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 dependable data repairing algorithms by capitalizing on fixing rules. Moreover, we have presented how to generate fixing rules by soliciting examples from users and by querying and massaging available knowledge bases. 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 brought up many questions in need of further investigation. (1) *Rule discovery*. Our techniques in the article allow users to define fixing rules manually or generate rules using examples. We are planning to discover fixing rules or samples from mining algorithms, along the same line with other data quality rule discovery algorithms [Chiang and Miller 2008; Fan et al. 2011; Chu et al. 2013a; Golab et al. 2014; Song and Chen 2013; Golab et al. 2008]. (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.

REFERENCES

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. 2009. Learning string transformations from examples. *Proc. VLDB* 2, 1 (2009).

Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. 1999. Consistent query answers in inconsistent databases. In *PODS*. 68–79.

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

Leopoldo E. Bertossi, Solmaz Kolahi, and Laks V. S. Lakshmanan. 2011. Data cleaning and query answering with matching dependencies and matching functions. In *ICDT*. 268–279.

George Beskales, Ihab F. Ilyas, and Lukasz Golab. 2010. Sampling the repairs of functional dependency violations under hard constraints. *Proc. VLDB* 3, 1 (2010), 197–207.

George Beskales, Ihab F. Ilyas, Lukasz Golab, and Artur Galiullin. 2013. On the relative trust between inconsistent data and inaccurate constraints. In *ICDE*.

George Beskales, Mohamed A. Soliman, Ihab F. Ilyas, and Shai Ben-David. 2009. Modeling and querying possible repairs in duplicate detection. *Proc. VLDB* 2, 1 (2009), 598–609.

Philip Bohannon, Wenfei Fan, Michael Flaster, and Rajeev Rastogi. 2005. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*.

Loreto Bravo, Wenfei Fan, and Shuai Ma. 2007. Extending dependencies with conditions. In *VLDB*. 243–254.

Fei Chiang and Renée J. Miller. 2008. Discovering data quality rules. *Proc. VLDB* 1, 1 (2008).

Fei Chiang and Renée J. Miller. 2011. A unified model for data and constraint repair. In *ICDE*.

J. Chomicki and J. Marcinkowski. 2005. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.* 197, 1–2 (2005), 90–121.

Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013a. Discovering denial constraints. *Proc. VLDB* 6, 13 (2013).

Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013b. Holistic data cleaning: Putting violations into context. In *ICDE*.

Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. 2015. KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. In *SIGMOD*. 1247–1261.

Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. 2007. Improving data quality: Consistency and accuracy. In *VLDB*. 315–326.

Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. 2013. NADEEF: A commodity data cleaning system. In *SIGMOD*.

Wenfei Fan. 2008. Dependencies revisited for improving data quality. In *PODS*. 159–170.

Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional functional dependencies for capturing data inconsistencies. *TODS* (2008).

Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. 2011. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.* 23, 5 (2011).

Wenfei Fan, Xibei Jia, Jianzhong Li, and Shuai Ma. 2009. Reasoning about record matching rules. *Proc. VLDB* 2, 1 (2009), 407–418.

Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. 2011. Interaction between record matching and data repairing. In *SIGMOD*. 469–480.

Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. 2012. Towards certain fixes with editing rules and master data. *VLDB J.* 21, 2 (2012), 213–238.

I. Fellegi and D. Holt. 1976. A systematic approach to automatic edit and imputation. *J. Am. Stat. Assoc.* 71, 353 (1976), 17–35.

Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2013. The LLUNATIC data-cleaning framework. *Proc. VLDB* 6, 9 (2013), 625–636.

Lukasz Golab, Howard J. Karloff, Flip Korn, Barna Saha, and Divesh Srivastava. 2014. Discovering conservation rules. *IEEE Trans. Knowl. Data Eng.* 26, 6 (2014).

Lukasz Golab, Howard J. Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. 2008. On generating near-optimal tableaux for conditional functional dependencies. *Proc. VLDB* 1, 1 (2008).

Thomas N. Herzog, Fritz J. Scheuren, and William E. Winkler. 2009. *Data Quality and Record Linkage Techniques*. Springer.

Matteo Interlandi and Nan Tang. 2015. Proof positive and negative in data cleaning. In *ICDE*. 18–29.

Solmaz Kolahi and Laks Lakshmanan. 2009. On approximating optimum repairs for functional dependency violations. In *ICDT*. 53–62.

Nick Koudas, Avishek Saha, Divesh Srivastava, and Suresh Venkatasubramanian. 2009. Metric functional dependencies. In *ICDE*.

Xiang Lian, Lei Chen, and Shaoxu Song. 2010. Consistent query answers in inconsistent probabilistic databases. In *SIGMOD*.

Chris Mayfield, Jennifer Neville, and Sunil Prabhakar. 2010. ERACER: A database approach for statistical inference and data cleaning. In *SIGMOD Conference*. 75–86.

Felix Naumann, Alexander Bilke, Jens Bleiholder, and Melanie Weis. 2006. Data fusion in three steps: Resolving schema, tuple, and value inconsistencies. *IEEE Data Eng. Bull.* 29, 2 (2006), 21–31.

Vijayshankar Raman and Joseph M. Hellerstein. 2001. Potter's wheel: An interactive data cleaning system. In *VLDB*.

Rishabh Singh and Sumit Gulwani. 2012. Learning semantic string transformations from examples. *Proc. VLDB* 5, 8 (2012), 740–751.

Shaoxu Song and Lei Chen. 2013. Efficient discovery of similarity constraints for matching dependencies. *Data Knowl. Eng.* 87 (2013).

Shaoxu Song, Hong Cheng, Jeffrey Xu Yu, and Lei Chen. 2014. Repairing vertex labels under neighborhood constraints. *Proc. VLDB* 7, 11 (2014).

Maksims Volkovs, Fei Chiang, Jaroslaw Szlichta, and Renée J. Miller. 2014. Continuous data cleaning. In *ICDE*. 244–255.

Jiannan Wang and Nan Tang. 2014. Towards dependable data repairing with fixing rules. In *SIGMOD*.

Jef Wijsen. 2005. Database repairing using updates. *ACM Trans. Database Syst.* 30, 3 (2005), 722–768.

Mohamed Yakout, Laure Berti-Equille, and Ahmed K. Elmagarmid. 2013. Don't be SCAREd: Use scalable automatic repairing with maximal likelihood and bounded changes. In *SIGMOD*. 553–564.

Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F. Ilyas. 2011. Guided data repair. *Proc. VLDB* 4, 5 (2011), 279–289.

Received November 2015; revised September 2016; accepted January 2017