Proof Positive and Negative in Data Cleaning

Matteo Interlandi
Qatar Computing Research Institute
Doha, Qatar
minterlandi@qf.org.qa

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

***Abstract*—One notoriously hard data cleaning problem is, given a database, how to precisely capture which value is correct (*i.e., proof positive*) or wrong (*i.e., proof negative*). Although integrity constraints have been widely studied to capture data errors as violations, the accuracy of data cleaning using integrity constraints has long been controversial. Overall they deem one fundamental problem: Given a set of data values that together forms a violation, there is no evidence of which value is proof positive or negative. Hence, it is known that integrity constraints themselves cannot guide dependable data cleaning. In this work, we introduce an automated method for proof positive and negative in data cleaning, based on *Sherlock rules* and *reference tables*. Given a tuple and reference tables, Sherlock rules tell us what attributes are proof positive, what attributes are proof negative and (possibly) how to update them. We study several fundamental problems associated with Sherlock rules. We also present efficient algorithms for cleaning data using Sherlock rules. We experimentally demonstrate that our techniques can not only annotate data with proof positive and negative, but also repair data when enough information is available.**

I. Introduction

Real-life data is often dirty: Up to 30% of an organization's data could be dirty [2]. Dirty data is costly: It costs the U.S. economy $3 trillion+ per year [1]. These highlight the importance of cleaning data in all businesses.

One notoriously hard problem in data cleaning is that, given a database, how to precisely capture which value is correct (*i.e., proof positive*), which value is wrong (*i.e., proof negative*) and (possibly) how to repair it. There has been a remarkable series of work for capturing data errors as violations via integrity constraints (ICs) [4], [8], [11], [17], [18], [21], [27]. A violation is a set of data values that, when put together, violates some IC, thus considered to be erroneous. We illustrate this line of work by an example.

Example 1: Consider the database D_{EMP} of Fig. 1 containing employee records, specified by the following schema:

$$\text{EMP}\ (\mathsf{name}, \mathsf{dept}, \mathsf{nation}, \mathsf{capital}, \mathsf{bornat}, \mathsf{officePhn}),$$

where each EMP tuple specifies an employee, identified by his/her name, department (dept), nationality (nation) with its capital, the city he/she was born at (bornat), and office phone number (officePhn). All errors are marked, *e.g.,* $t_2[\mathsf{capital}] = \mathsf{Shanghai}$ is wrong, whose correct value is Beijing.

Assume that a *functional dependency* (FD) is defined as ϕ : EMP (nation $\rightarrow$ capital), which states that nation uniquely determines capital in relation EMP. One can verify that (t_1, t_2) violate ϕ since they have the same nation but different capital values (Beijing for t_1 and Shanghai for t_2). □

	name	dept	nation	capital	bornat	officePhn
t_1	Si	DA	China	Beijing	ChenYang	28098001
t_2	Yan	DA	China	Shanghai	Chengdu	24038698
t_3	Ian	ALT	Chine	Beijing	Hangzhou	33668323

Figure 1. D_{EMP}: An instance of the schema EMP

	country	capital
s_1	China	Beijing
s_2	Japan	Tokyo
s_3	Chile	Santiago

Figure 2. M_{CAP} of CAP

	name	officePhn	mobile
r_1	Si	28098001	66700541
r_2	Yan	24038698	66706563
r_3	Ian	27364928	33668323

Figure 3. M_{PHN} of PHN

Example 1 shows that ICs can detect errors, *e.g.,* there must exist errors in the four values $(t_1[\mathsf{nation}, \mathsf{capital}], t_2[\mathsf{nation}, \mathsf{capital}])$. However, it reveals two shortcomings of IC-based error detection: (*i*) it cannot do *proof positive* (*e.g.,* $t_1[\mathsf{nation}]$ is correct) or *proof negative* (*e.g.,* $t_2[\mathsf{capital}]$ is wrong) in detected violations; and (*ii*) it cannot ensure that data *consistent w.r.t.* a set of ICs is correct. For instance, in Fig. 1, t_3 is consistent with any tuple *w.r.t.* ϕ, but t_3 cannot be marked as correct. Therefore, ICs themselves cannot guide dependable data cleaning. Consequently, automated constraint-based repairing algorithms [4], [8], [11], [18], [21], [27] try to resolve detected violations in a heuristic fashion.

In order to achieve dependable data repairing, users have been involved and reference tables that contain well curated data have been employed [23]. We illustrate this line of work with another example.

Example 2: Consider a reference table M_{CAP} shown in Fig. 2, defined over the schema CAP (country, capital). *Editing rules* [23] work as follows. Let ψ : $((\mathsf{nation}, \mathsf{country}) \rightarrow (\mathsf{capital}, \mathsf{capital}), t_p = ())$ be an editing rule over the two relations (EMP, CAP). Rule ψ states that for any tuple t in D_{EMP}, if $t[\mathsf{nation}]$ is correct and matches $s[\mathsf{country}]$ of a tuple s in M_{CAP}, we can update $t[\mathsf{capital}]$ to $s[\mathsf{capital}]$. For instance, to repair t_2 in Fig. 1, the users need to ensure that $t_2[\mathsf{nation}]$ is correct. Afterwards, $t_2[\mathsf{country}]$ can be safely matched with $s_1[\mathsf{country}]$ in the reference table, so as to update $t_2[\mathsf{capital}]$ to $s_1[\mathsf{capital}]$. For the other tuples similar processes apply. □

Editing rules can ensure dependable data repairing. However, users have to be involved in repairing each tuple, which is not cheap and error-prone.

Examples 1 and 2 highlight one important problem: *How to automatically do proof positive and negative in data cleaning?* This is a hard but important problem to stimulate our creative

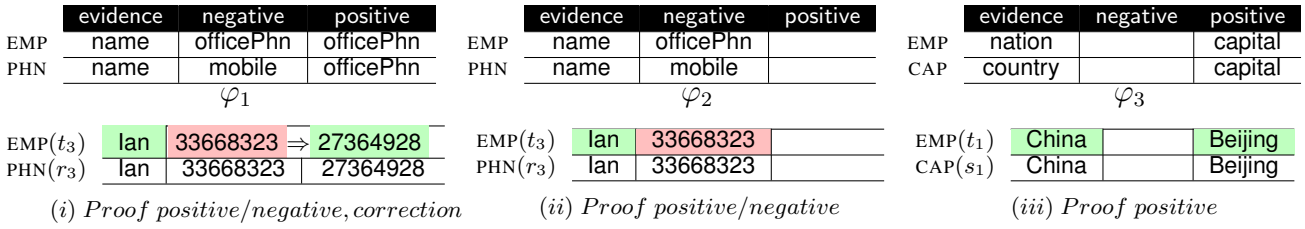


Figure 4. Example Sherlock rules

juices, since it not only lights up the way of dependable data repairing, but also sheds considerable light on other data cleaning approaches that require well annotated data *e.g.,* IC discovery [10], [13], [19] and machine learning based approaches [34]. Sherlock rules proposed by this work are ideal for this purpose.

Sherlock is not magic; it cannot guess something from nothing. What it does is to collect evidences from external sources (*i.e.,* reference tables) so as to make judgement, *e.g.,* people often confuse capitals with big/famous cities, office phone numbers with cell numbers, or zip code with area code. We illustrate by an example how Sherlock rules work, based on the availability of related reference tables.

Example 3: Consider the employee table in Fig. 1, the capital table in Fig. 2, and the phone table in Fig. 3. We next discuss the three cases depicted in Fig. 4 which make use of three Sherlock rules φ_1–φ_3.

Case (i) Proof positive, proof negative and correction. Rule φ_1 states that for a tuple t in D_{EMP}, if its name matches the name of an r tuple in M_{PHN}, and t[officePhn] matches r[mobile], then φ_1 validates that t[name] is correct (proof positive), and t[officePhn] is wrong (proof negative). Moreover, it will rectify t[officePhn] to r[officePhn].

Consider t_3 in D_{EMP} and r_3 in M_{PHN}, φ_1 works as follows (see Fig. 4 case (i)). Firstly, t_3[name] is matched with r_3[name], and t_3[officePhn] with r_3[mobile]. It then detects that t_3 is about Ian, but someone messed up his office number with his mobile number. Consequently, t_3[name] is marked as correct and t_3[officePhn] as wrong. Since the office number of Ian is available in r_3[officePhn], φ_1 will update t_3[officePhn] to r_3[officePhn], which is 27364928.

Case (ii) Proof positive and proof negative. Often times, not all evidences are available. Assume that the column officePhn is missing in PHN, *i.e.,* we consider a revised schema PHN′ (name, mobile). Rule φ_2 states that given a tuple t in D_{EMP}, if its name matches the name of a tuple r in $M_{\text{PHN}'}$, and t[officePhn] matches r[mobile], then φ_2 validates that t[name] is correct and t[officePhn] is wrong.

Again, consider t_3 in D_{EMP} and r_3 in $M_{\text{PHN}'}$, φ_2 works similar to φ_1 (see Fig. 4 case (ii)), which validates that t_3[name] is correct and t_3[officePhn] is wrong. However, due to the missing column officePhn in PHN′, φ_2 cannot update t_3[officePhn].

Case (iii) Proof positive. Rule φ_3 states that for a tuple t in D_{EMP}, if t[country, capital] matches s[country, capital] of an s tuple in M_{CAP}, it will mark t[country, capital] as correct.

Consider t_1 in D_{EMP} and s_1 in M_{CAP}. Since both country (*i.e.,* China) and capital (*i.e.,* Beijing) match, φ_3 will mark t_1[country, capital] as correct. □

Remark: (1) Sherlock rules differ from ICs in two aspects. (a) Sherlock rules can precisely capture which value is correct or wrong, but ICs cannot (see Example 1). (b) Sherlock rules can guide fine grained dependable data repairing by precisely changing data values (see Example 3 case (i)). (2) Sherlock rules differ from other data cleaning rules *e.g.,* editing rules [23] and fixing rules [33] in that instead of only fixing data errors, they are also able to annotate data, even if some information is missing (see Example 3 cases (ii, iii)). This feature is appealing, since Sherlock rules can be used as a pre-processing step for other applications [10], [13], [19], [22], [34] that require well-annotated data as input.

Contributions. We propose Sherlock rules, a novel class of rules for proof positive and negative in data cleaning, with the following notable contributions.

(1) We formally define Sherlock rules and their semantics (Section III). Given a tuple t and a reference table, Sherlock rules tell us that which values are correct, which values are wrong and (possibly) how to repair them.

(2) We study fundamental problems of Sherlock rules (Section IV). Specifically, given a set Σ of Sherlock rules, we determine whether these rules have conflicts. We show that this problem is coNP-complete. We also study the problem of whether some other Sherlock rules are implied by Σ, which is also proved to be a coNP-complete problem.

(3) We propose two repairing algorithms for a given set Σ of Sherlock rules (Section V). The first algorithm is chase-based. The second one is an optimized version of the chased-based algorithm that exploits similarity indices and other auxiliary data structures to improve the efficiency.

(4) We experimentally verify the effectiveness and scalability of the proposed algorithms (Section VI). We find that Sherlock rules can annotate and repair data with high accuracy and good scalability.

Organization: Section II presents related work. Section III introduces Sherlock rules. Section IV studies fundamental problems associated with Sherlock rules. Section V describes repairing algorithms that employ Sherlock rules. Section VI reports experimental findings, followed by concluding remarks in Section VII.

II. Related Work

In recent years, there has been an increasing amount of literature on using ICs in data cleaning (*e.g.,* [4], [8], [11], [18], [21], [27]; see [20], [31] for surveys). They have been revised to better capture data errors as violations of these ICs (*e.g.,* by adding conditions CFDs [18] and CINDs [8]). As remarked earlier, ICs cannot tell which values in a violation are correct or wrong, thus fall short of guiding dependable data repairing.

Closer to this work are editing rules [23] and fixing rules [33]. Editing rules [23] are introduced to repair data that is guaranteed correct. However, editing rules require users to examine every tuple, which is expensive. Sherlock rules differ from editing rules in that they *automatically* annotate data as proof positive or negative, and rectify errors when enough evidences are present. Fixing rules [33] have been recently proposed for automatic and dependable data repairing. Sherlock rules are more general than fixing rules in that (1) Sherlock rules use schema level matching between dirty data and reference tables instead of encoding instances in rules as fixing rules (and constant CFDs) do; (2) Sherlock rules use domain-specific similarity matching instead of exact string matching employed in fixing rules; and (3) Sherlock rules are able to annotate data when insufficient evidences are given.

Many data repairing algorithms have been proposed [6], [7], [12], [14], [15], [22]–[25], [28], [35]. Heuristic methods are developed in [5], [7], [14], [15], [24], based on FDs [5], [27], [32], FDs and INDs [7], CFDs [18], CFDs and MDs [22] and denial constraints [12]. Some works employ confidence values placed by users to guide a repairing process [7], [14], [22] or use master data [23]. Statistical inference is studied in [28] to derive missing values, and in [6] to find possible repairs. To ensure the accuracy of generated repairs, [23], [28], [35] require to consult users. In contrast to these prior arts: (1) Sherlock rules are more conservative in repairing data, which target determinism and dependability, instead of computing a consistent database; (2) Sherlock rules neither consult the users, nor assume the confidence values placed by the users. Indeed, they can be treated as a complementary technique to heuristic methods *i.e.,* one may compute dependable repairs or annotate data first and then use heuristic solutions to find a consistent database.

There has also been work on data transformation [29]. ETL tools (see [26] for a survey) provide sophisticated data transformation methods, which can be employed to merge and repair data. Some recent work has been studied for syntactic transformations of strings [3]. We shall discuss later that they can be expressed as special cases of Sherlock rules.

III. Sherlock Rules

In this section, we first give the formal definition of Sherlock rules (Section III-A). We then describe the repairing semantics (Section III-B) for applying a set of Sherlock rules.

A. Definition

Let D be a table over schema R, and M a reference table with schema R_m. We use $A \in R$ to denote that A is an attribute of R. Note that the relation schema R is often distinct from R_m. Moreover, we assume that the reference table R_m is *correct* but possibly *incomplete*.

Syntax. A *Sherlock rule* (sR) φ defined on schemas (R, R_m) is formalized as $\varphi : ((X, X_m), (B, B_m^-, B_m^+), \vec{\approx})$ where:

- X and X_m are lists of distinct attributes in schemas R and R_m respectively, where $|X| = |X_m|$;
- B is an attribute such that $B \in R \setminus X$, and B_m^-, B_m^+ are two distinct attributes in $R_m \setminus X_m$; and
- $\vec{\approx}$ is a vector of similarity operators over comparable attributes, (A, A_m), (B, B_m^-) and (B, B_m^+), where $A \in X$, and A_m is the corresponding attribute in X_m.

Intuitively, rule φ says that for a pair of tuples t in D and t_m in M, if both $(t[X], t_m[X_m])$ and $(t[B], t_m[B_m^-])$ are similar *w.r.t.* some similarity metrics, φ validates that $t[X]$ is correct, $t[B]$ is wrong, and moreover, the correct value of $t[B]$ is $t_m[B_m^+]$.

Example 4: Consider again the employee table in Fig. 1, the capital table in Fig. 2 and the phone table in Fig. 3. The three sRs in Example 3 can be formally expressed as follows – where φ_1 and φ_2 are defined on (EMP, PHN), and φ_3 is defined on (EMP, CAP):

φ_1: ((name, name), (officePhn, mobile, officePhn), $(=, =, =)$)
φ_2: ((name, name), (officePhn, mobile, $\perp$), $(=, =, \not\approx)$)
φ_3: ((nation, country), (capital, $\perp$, capital), $(=, \not\approx, =)$)

where "$\perp$" indicates that a field is missing, and "$\not\approx$" that the two corresponding attributes are not comparable, *e.g.,* when some attribute is missing from reference tables. □

Match: Given an sR φ defined on two relations (R, R_m), a tuple t from D and tuple t_m from M, we say that t and t_m *match w.r.t.* φ, denoted by $t \sim_\varphi t_m$, if for each attribute $A \in X$, let A_m be its related attribute in X_m, then $t[A] \approx_i t_m[A_m]$, and moreover, (i) if B_m^- is present, $t[B] \approx_j t_m[B_m^-]$ holds, or (ii) if B_m^- is absent but B_m^+ is present, $t[B] \approx_k t_m[B_m^+]$ holds. Here, i, j, k are indices for the corresponding similarity operators in $\vec{\approx}$. Intuitively, t and t_m match *w.r.t.* φ when φ can do proof positive/negative or correct values in t based on the evidences collected from t_m.

Example 5: Consider Example 4 and Fig. 4. We have that t_3 in D_{EMP} and r_3 in M_{PHN} match under φ_1 (*i.e.,* $t_3 \sim_{\varphi_1} r_3$), since $t_3[\mathsf{name}] = r_3[\mathsf{name}]$ and $t_3[\mathsf{officePhn}] = r_3[\mathsf{mobile}]$. Similarly, one can verify that $t_3 \sim_{\varphi_2} r_3$, and $t_1 \sim_{\varphi_3} s_1$, where s_i is a tuple in Fig. 2 and r_j a tuple in Fig. 3. □

Semantics. We say that a Sherlock rule φ *applies* to a tuple t, denoted by $t \overset{\varphi, t_m}{\rightarrow} t'$, if a reference tuple t_m exists such that (1) t and t_m match *w.r.t.* φ (*i.e.,* $t \sim_\varphi t_m$), and (2) t' is the updated version of t. That is, when t agrees with a reference tuple t_m on some set of attributes, we are able to derive an updated and annotated version t' of t.

We use the symbol "$+$" (resp. "$-$") to annotate a value as positive (resp. negative). Otherwise, when a value is not annotated, we call it a *free* value. To ensure that changes and annotations make sense, the values that have been annotated as positive are considered as *bounded*, *i.e.,* they should remain unchanged in the following processes.

Example 6: Consider again case (i) in Fig. 4. Under φ_1 we have that $t_3[\mathsf{name}]$ agrees with $r_3[\mathsf{name}]$, and $t_3[\mathsf{officePhn}]$ is the same as $r_3[\mathsf{mobile}]$. Hence, we know that $t_3[\mathsf{name}]$

$$\frac{(X_m \neq \bot) \wedge (B_{\bar{m}} \neq \bot) \wedge (B_{\dot{m}}^{+} \neq \bot) \wedge (B \notin \text{POS}(t)) \wedge (X \cap \text{NEG}(t) = \bot) \wedge (t[X] \approx t_m[X_m]) \wedge (t[B] \approx t_m[B_{\bar{m}}])}{(t[X, B] := t_m[X_m, B_{\dot{m}}^{+}]) \wedge (\text{POS}(t) := \text{POS}(t) \cup X \cup \{B\}) \wedge (\text{NEG}(t) := \text{NEG}(t) \setminus \{B\})} (1)$$

$$\frac{(X_m \neq \bot) \wedge (B_{\bar{m}} \neq \bot) \wedge (B_{\dot{m}}^{+} = \bot) \wedge (B \notin \text{POS}(t)) \wedge (X \cap \text{NEG}(t) = \bot) \wedge (t[X] \approx t_m[X_m]) \wedge (t[B] \approx t_m[B_{\bar{m}}])}{(t[X] := t_m[X_m]) \wedge (\text{POS}(t) := \text{POS}(t) \cup X) \wedge (\text{NEG}(t) := \text{NEG}(t) \cup \{B\})} (2)$$

$$\frac{(X_m \neq \bot) \wedge (B_{\dot{m}}^{+} \neq \bot) \wedge (B_{\bar{m}} = \bot) \wedge (B \notin \text{POS}(t)) \wedge (B \notin \text{NEG}(t)) \wedge (X \cap \text{NEG}(t) = \bot) \wedge (t[X] \approx t_m[X_m]) \wedge (t[B] \approx t_m[B_{\dot{m}}^{+}])}{(t[X, B] := t_m[X_m, B_{\dot{m}}^{+}]) \wedge (\text{POS}(t) := \text{POS}(t) \cup X \cup \{B\})} (3)$$

$$\frac{(X_m \neq \bot) \wedge (B_{\dot{m}}^{+} \neq \bot) \wedge (B_{\bar{m}} = \bot) \wedge (B \notin \text{POS}(t)) \wedge (X \subseteq \text{POS}(t)) \wedge (t[X] \approx t_m[X_m]) \wedge (t[B] \not\approx t_m[B_{\dot{m}}^{+}])}{(t[B] := t_m[B_{\dot{m}}^{+}]) \wedge (\text{POS}(t) := \text{POS}(t) \cup \{B\}) \wedge (\text{NEG}(t) := \text{NEG}(t) \setminus \{B\})} (4)$$

$$\frac{(X_m = \bot) \wedge (B_{\dot{m}}^{+} \neq \bot) \wedge (B_{\bar{m}} \neq \bot) \wedge (B \notin \text{POS}(t)) \wedge (t[B] \approx t_m[B_{\bar{m}}])}{(t[B] := t_m[B_{\dot{m}}^{+}]) \wedge (\text{POS}(t) := \text{POS}(t) \cup \{B\})} (5)$$

Figure 5. Transformation rules

(*i.e.,* lan) is correct, but the mobile phone has been inserted instead of the office phone, hence t_3[officePhn] is wrong. By applying φ_1 to t_3, we obtain an updated tuple t_3' with officePhn being corrected from 33668323 to 27364928. As a result, t_3[name, officePhn] is corrected and positively annotated as $t_3(\mathsf{lan}^+, 27364928^+)$.

For case (ii) in Fig. 4, rule φ_2 cannot update t_3. However, it will annotate t_3[name, officePhn] as $t_3(\mathsf{lan}^+, 33668323^-)$. Moreover, for case (iii) in Fig. 4, φ_3 will annotate values of t_1[nation, capital] as $t_1(\mathsf{China}^+, \mathsf{Beijing}^+)$. □

Given a rule $\varphi : ((X, X_m), (B, B_{\bar{m}}, B_{\dot{m}}^{+}), \approx)$, for convenience, we denote by A_φ an attribute in φ, where A is in X, X_m or one of $B, B_{\bar{m}}$ or $B_{\dot{m}}^{+}$.

B. Applying Multiple Sherlock Rules

When applying a Sherlock rule to a tuple, the tuple might have been repaired and annotated by other sRs. Hence, we shall discuss the semantics of applying multiple Sherlock rules.

Annotations. Given a tuple t over relation R, we define three sets of annotated attributes.

- POS(t): attributes identified to be correct;
- NEG(t): attributes identified to be wrong; and
- FREE(t): attributes that are free, *i.e.,* not marked.

Intuitively, a value has to be annotated as either positive (*i.e.,* in POS(t)), or negative (*i.e.,* in NEG(t)), or free (*i.e.,* in FREE(t)). At any time, POS$(t) \cap$ NEG$(t) = \emptyset$, *i.e.,* a value cannot be both positive and negative.

Initially, before applying Sherlock rules to a tuple t, all attributes of t are free. After applying a Sherlock rule to t resulting in an annotated and (possibly) updated tuple t', a free value can remain free, or be annotated either as positive, or negative; a negative value can remain negative or be repaired and thus annotated as positive; a positive value can only remain positive. Hence, the application of Sherlock rules gradually identifies positive/negative values from free values, and update negative values when their correct values are known. Along with applying Sherlock rules, the size of unannotated attributes (*i.e.,* FREE(t)) decreases monotonically, while the size of POS(t) increases monotonically, *i.e.,* $|\text{FREE}(t')| \leq |\text{FREE}(t)|$ and $|\text{POS}(t')| \geq |\text{POS}(t)|$.

Transformation rules. Sherlock rules are quite flexible. Based on whether X_m, $\text{B}_{\bar{m}}$ or $\text{B}_{\dot{m}}^{+}$ are present or not, and on the types of annotations that have been marked on a tuple t, we have five *transformation rules* that devise a compositional scheme for Sherlock rules. This defines the operational semantics of rule applications. The five transformation rules are depicted in Fig. 5. They are explained below.

Cases	Proof positive	Proof negative	Repair
(1) $X_m, B_{\bar{m}}, B_{\dot{m}}^{+}$	√	√	√
(2) $X_m, B_{\bar{m}}$	√	√	
(3) $X_m, B_{\dot{m}}^{+}$	√		
(4) $X_m^{+}, B_{\dot{m}}^{+}$	√		√
(5) $B_{\bar{m}}, B_{\dot{m}}^{+}$	√		√

Figure 6. Sherlock rules in different cases

(1) X_m, $\text{B}_{\bar{m}}$ and $\text{B}_{\dot{m}}^{+}$ are given. If B is not yet proof positive, no attribute in X is proof negative, and, in addition, $t[X] \approx t_m[X_m]$ and $t[B] \approx t_m[B_{\bar{m}}]$, then, as a consequence, $t[X \cup \{B\}]$ will be updated and X, B be annotated as positive.

(2) This case is similar to (1), except that $\text{B}_{\dot{m}}^{+}$ is absent. As a consequence, $t[X]$ (resp. $t[B]$) will be annotated as positive (resp. negative).

(3) Similar to (1), but this time $\text{B}_{\bar{m}}$ is absent and $t[B] \approx t_m[B_{\dot{m}}^{+}]$. Both X and B are annotated as positive.

(4) As (3) but $t[B]$ is not similar to $t_m[B_{\dot{m}}^{+}]$ and X has already been annotated as positive. Because of the later, we can then safely update $t[B]$ to $t_m[B_{\dot{m}}^{+}]$ and annotate B as positive.

(5) Only $\text{B}_{\bar{m}}$ and $\text{B}_{\dot{m}}^{+}$ are present. This rule is similar to an ETL dictionary look up, so that $t[B]$ can be updated to $t_m[B_{\dot{m}}^{+}]$ and annotated as positive if $t[B]$ matches $t_m[B_{\bar{m}}]$.

Example 7: Consider rule φ_1 of Example 4. The first transformation rule can be applied in order to obtain the behaviour explained in Example 3 case (i). If, instead, we apply the second transformation rule, *e.g.,* $B_{\dot{m}}^{+}$ is missing in the reference table, we get case (ii) of Example 3. Similarly, transformation rules 3 – 5 can be used over φ_1 to implement different rule semantics. □

Discussion. Figure 6 summarizes the five transformation rules by showing their features. Transformation rule (1) has all the information available and therefore is able to do both proof positive/negative and rectify the wrong values, which can be treated as a generalization of fixing rules [33]. Case (2) is used to identify errors, and case (3) is able to mark data as correct. Interestingly, from this perspective rules (4) and (5)

	name	bornCity	officePhn	mobile
r_1	Si	Shenyang	28098001	66700541
r_2	Yan	Chengdu	24038698	66706563
r_3	Ian	Hangzhou	27364928	33668323

Figure 7. M_{REG} of REG

are identical. Semantically, however, they are quite different: rule (4) is an automatic and similarity-based implementation of editing rules [23], while (5) is an ETL dictionary look up, for example, to normalize strings.

Fixing Transition. Given a tuple t_m in reference table M, a transformation rule φ, and a (possibly annotated) tuple t, a *fixing transition* is the process of selecting and executing one transformation rule, resulting in a modified t'. For ease of understanding, we also use the notation $t \overset{\varphi,t_m}{\rightarrow} t'$ for fixing transition, where both t and t' are annotated tuples.

Example 8: Consider t_1 in Fig. 1 and the reference table M_{REG} in Fig. 7, generated from M_{PHN} of Fig. 3 by adding a column containing the city in which each person was born. We have the following two rules defined over (EMP, REG):

φ_4: ((name, name), (officePhn, $\perp$, officePhn), $(=, \not\approx, =)$)
φ_5: ((name, name), (bornat, $\perp$, borncity), $(=, \not\approx, =)$)

At first, only φ_4 can be applied to t_1 using r_1 in Fig. 7, under the semantics of transformation rule (3). Note that, at this point, φ_5 cannot be applied using r_1 since $t_1[\text{bornat}] = \text{Chenyang}$, which does not equal to $r_1[\text{bornCity}] = \text{Shenyang}$. By applying φ_4 and r_1, the first fixing transition will annotate $t_1[\text{name}, \text{officePhn}]$ as positive, as depicted below.

t_1: (Si, DA, China, Beijing, ChenYang, 28098001)
↓ *(1st fixing transition with φ_4 and r_1)*
t'_1: (Si$^+$, DA, China, Beijing, ChenYang, 28098001$^+$)
↓ *(2nd fixing transition with φ_5 and r_1)*
t''_1: (Si$^+$, DA, China, Beijing, ShenYang$^+$, 28098001$^+$)

Now, since $t_1[\text{name}]$ has been annotated as positive, rule φ_5 can be applied under the semantics of transformation rule (4). This indeed simulates editing rules [23], and $t_1[\text{bornat}]$ can be safely updated from Chenyang to Shenyang and annotated as positive, as depicted for the second fixing transition above. □

Fixing Run. Let t be a tuple and Σ a set of Sherlock rules. A *fixing run* is a sequence of fixing transitions on t, until a fixpoint is reached, *i.e.,* no more rules can be applied.

In other words, given tuple t and rules Σ, a fixing run is a set of fixing transitions as $t^0 \overset{(\varphi,t_m)^0}{\longrightarrow} t^1, \cdots, t^{n-1} \overset{(\varphi,t_m)^{n-1}}{\longrightarrow} t^n$, where t^i denotes the tuple after the i-th fixing transition, t^i and t^{i+1} $(i \in [0, n-1])$ are different *i.e.,* they have different values or different annotations, and t^n is a fixpoint, which for simplicity we denote as t^*.

Example 9: Consider $\Sigma = \{\varphi_4, \varphi_5\}$ of Example 8. The running example shown in Example 8 is a fixing run for tuple t_1 *w.r.t.* Σ, since no more rule can be further applied. □

notation	description
$((X, X_m), (B, B_m^-, B_m^+), \approx)$	a Sherlock rule
A_φ	attribute A in rule φ
$t \sim_\varphi t_m$	t and t_m match *w.r.t.* φ
$t \overset{\varphi,t_m}{\rightarrow} t'$ (fixing transition)	applying φ and t_m to t
$t^0 \overset{(\varphi,t_m)^0}{\longrightarrow} t^1, \cdots, \overset{(\varphi,t_m)^{n-1}}{\longrightarrow} t^n$	a sequence of fixing transitions
t^*	the fixpoint of a fixing transition
$\mathcal{R}_t^{M,\Sigma}$	fixing repairs of t *w.r.t.* M and Σ

Figure 8. Summary of notations

Fixing Repair. Given a possibly dirty tuple t, a reference table M, and a set of Sherlock rules Σ, many different fixing runs can be used to model the repairing of t. Intuitively, this is due to the fact that different updates can occur based on which rule is non-deterministically selected in each transition. To express this, we write $\mathcal{R}_t^{M,\Sigma}$ to denote the *fixing repairs* for t: the set of non-empty runs that models all the possible evolutions of t driven by M and Σ. A fixing repair exactly models the *repairing semantics* of M and Σ on t. In the following we will in general write $\mathcal{R}$ to denote a fixing repair, when t, M and Σ are easily understandable from the context. We will show later that every run in $\mathcal{R}$ is *terminating* (Section IV-A), and that they all *converge* to a *unique* final repaired instance (Section IV-C) – *i.e.,* for every pair of distinct runs $\rho, \rho' \in \mathcal{R}$, $t^* = t'^*$ – if the set of rules Σ is *consistent* (Section IV-B).

We summarize the notations used in this paper in Fig. 8.

IV. Fundamental Problems

In the previous section we have briefly observed that some specific fundamental problems associated with Sherlock rules must be addressed. In this section we shall formally define such problems and study their complexity. Detailed proofs are omitted due to space constraints.

A. Termination

As already discussed in Section III-B, the *termination problem* asks if the repairing process leads to a fixpoint. Note that in every fixing transition of a tuple t, we have that (1) the set of annotated positive attributes increases monotonically, (2) the set of free attributes decreases monotonically, and (3) a variable can only be changed from free to negative or positive, or from negative to positive, but not vice-versa. Hence, termination is assured after a number of fixing transitions polynomial in the size of the relation schema.

B. Consistency

The consistency of Sherlock rules mainly deals with ensuring that the application of a set of rules in different orders will always have the same result.

Consistency problem. Let Σ be a set of Sherlock rules and M a reference table. Σ is said to be *consistent w.r.t.* M, if given any tuple t, all the fixing runs via Σ and M terminate in the same fixpoint t^*, *i.e.,* the repair is unique.

For analyzing the consistency of Sherlock rules, we introduce conflicting rules below.

Conflicting rules. Let φ and φ' be two Sherlock rules in Σ such that (i) $B^\varphi \cap (X^{\varphi'} \cup B^{\varphi'}) \neq \emptyset$, and (ii) two distinct

fixing runs ρ, ρ' in the repair $\mathcal{R}$ exist such that ρ and ρ' agree up to the i^{th} transition $t^i \xrightarrow{(\varphi,t_m)^i} t^{i+1}$, and in the successive transition φ and φ' can be applied concurrently. We say that φ and φ' are *conflicting* if in the $i+1^{th}$ transition, φ is applied in ρ, φ' is applied in ρ', and then $t^* \neq t'^*$.

Intuitively, two rules are conflicting if they try to repair or annotate the same record differently. In this way divergent final results will exist based on the order in which rules are applied first.

Proposition 1: *Let Σ be a set of Sherlock rules. Σ is consistent iff no pair of distinct rules φ, $\varphi' \in \Sigma$ exists such that φ and φ' are conflicting.* □

Proof sketch: For the *only-if* direction, assume that no pair of conflicting rules $\varphi, \varphi' \in \Sigma$ exists. In addition, by contradiction, assume that Σ is not consistent, *i.e.,* two distinct runs ρ, ρ' exist in $\mathcal{R}$ such that the respective final tuples t^*, t'^* are different. One can show that such two assumptions are in contradiction since necessarily a unique final instance occurs when no pair of conflicting rules exists in Σ.

Let us consider now the *if* direction. Let Σ be a consistent set of rules. Assume by contradiction that a pair of conflicting rules φ, φ' exists in Σ. One can show that two distinct final tuples are the result of the repair $\mathcal{R}$. However this is a contradiction, since we have initially assumed that every run in $\mathcal{R}$ bring to a unique final outcome. □

Consistency has been studied over different types of rules. For instance, it is known that any set of MDs is consistent [22], while checking consistency for a set of CFDs and editing rules is respectively NP- and coNP-complete [18], [23] in the general case. The following result shows that with Sherlock rules we are on the same line with editing rules.

Theorem 2: *The consistency problem for Sherlock rules is coNP-complete, even when the reference table is given.* □

Proof sketch: A PTIME algorithm exists to check if each pair of rules $\varphi, \varphi' \in \Sigma$ are not conflicting over the instance M. A procedure then can be built showing that deciding whether φ and φ' are conflicting is NP – and therefore detecting consistency is coNP. For the hardness part, the complement problem can be reduced to the 3SAT problem which is known to be NP-complete. □

If instead we consider the special case in which D is available, the consistency problem becomes PTIME.

Corollary 3: *Given a possibly dirty instance D and reference table M, the consistency problem for a set Σ of Sherlock rules* w.r.t. *D is* PTIME. □

Proof sketch: It is enough to check, for each transition i, that the set of applicable rules does not contain application of two conflicting rules. Finally, if a pair of conflicting applications exists, by Theorem 2 we know that a procedure exists by which we can check in PTIME if they are actually inconsistent against the reference instance M. □

Remark. Given that M is correct, checking whether a set Σ of rules is consistent is infeasible in practice based on Theorem 2. In fact, however, it is rational to check whether putting M and Σ together makes sense relative to a possible dirty data D, which is PTIME from Corollary 3. In our experimental study in Section VI, we use at run-time the PTIME algorithm to check whether M and Σ are consistent *w.r.t.* a given D. If not, we will manually check and revise Σ to ensure their consistency.

C. Determinism

The determinism problem is whether a unique final tuple t^* is obtainable for a given fixing repair $\mathcal{R}$. A unique final result is obviously always obtainable given a consistent Σ. In addition, under this assumption, in every transition all the applications can be concurrently applied since we are assured that no inconsistent result will be obtained.

D. Implication

Given a set Σ of consistent Sherlock rules, and another Sherlock rule ψ that is not in Σ, we say that ψ is implied by Σ, denoted by $\Sigma \models \psi$, if (i) $\Sigma \cup \{\psi\}$ is consistent; and (ii) whichever dirty tuple $t \in D$ and reference table M, every runs in $\mathcal{R}_t^{M,\Sigma}$ and $\mathcal{R}_t^{M,\Sigma\cup\{\psi\}}$ agree on the final tuple t^*. Condition (i) says that Σ and ψ comply with each other. Condition (ii) ensures that the outcomes of applying Σ or $\Sigma \cup \{\psi\}$ are the same, which indicates that ψ is redundant relative to Σ.

Implication problem. The implication problem is: given a set Σ of consistent Sherlock rules, and another rule ψ, determine whether Σ implies ψ for any dirty tuple and reference table M. We first analyze the general case.

Theorem 4: *The implication problem for a set of consistent Sherlock rules $\Sigma \cup \{\psi\}$ is coNP-complete, even if the reference table M is given.* □

Proof sketch: The implication problem can be computed by first checking the consistency of $\Sigma \cup \{\psi\}$ – which from Section IV-B we know is coNP-complete – and then by checking whether the two repairs $\mathcal{R}_t^{M,\Sigma}$ and $\mathcal{R}_t^{M,\Sigma\cup\{\psi\}}$ bring to the same final tuple t^*, whichever input record $t \in D$ is given. This latter is coNP, and can be proved by showing that its complement problem is in NP. For the demonstration that the problem is actually coNP-complete, one can use with few changes the technique employed in Theorem 2, *i.e.,* reduction to the 3SAT problem. □

Although in its general case checking implication is coNP, in the special case in which both D and M are available, we are able to compute implication in PTIME.

Corollary 5: *The implication problem for a set of generalized fixing rules $\Sigma \cup \{\psi\}$ is* PTIME *when D and M are fixed.* □

Proof sketch: From Corollary 3 we already know that checking consistency of $\Sigma \cup \{\psi\}$ is PTIME when both D and M are given. From Sections IV-A and IV-C we already know that every repair over annotated consistent rules is terminating

Algorithm 1 Optimized Repairing Algorithm

Input: A set of consistent rules Σ; A reference table M; A dirty tuple t.
Output: A cleaned and annotated version of t.

```
1: initialize(S, M);
2: I := Σ
3: α :=getCandidateApplications(I, S, t)
4: while α ≠ ∅ do
5:    for each t^φ_{t_m} ∈ α do
6:       for each ψ ∈ I do
7:          if B^φ ∈ ψ.getMatchingAttributes() then
8:             I := I \ ψ
9:    repair(t, α);
10:   α :=getCandidateApplications(I, S, t);
```

and deterministic. It is enough then to evaluate if two arbitrary runs, respectively belonging to $\mathcal{R}_t^{M,\Sigma}$ and $\mathcal{R}_t^{M,\Sigma\cup\{\psi\}}$, both terminate with the same final state t^*. From Section IV-C we know that this computation is PTIME. □

V. Repairing Algorithms with Sherlock Rules

After studying the fundamental problems of Sherlock rules, especially the remark in Section IV-B about how we get consistent Sherlock rules in practice, we are now ready to discuss data repairing using a set of consistent Sherlock rules for a tuple t. We first present a simple algorithm (Section V-A) *w.r.t.* the repairing semantics (Section III-B). We then describe an optimized algorithm by using various indices to speed-up the repairing process (Section V-B).

A. Naive Repairing

The naive algorithm for repairing one tuple works as follows: Firstly, a set of candidate applications is computed. Afterwards, each application is applied over the dirty tuple, following the semantics of Fig. 5. To get the set of candidate applications α, we used a slightly revisited form of the well-known chase-based algorithm: for each input dirty tuple t, we select the rules and the reference tuples matching with it. Before actually adding each application to the candidate set, we run a consolidation routine checking if in the candidate set α, another application already exists, and, in case, just the single application with the highest similarity is added to α. This is necessary since similarity operators are used for the matching, and, as a consequence, multiple reference tuples can exist for the same pair t, φ, and the best one (*w.r.t.* the similarity distance between the applications and the dirty tuple) must be researched.

Correctness and complexity: The naive algorithm just explained is a tuple-by-tuple straightforward implementation of the repairing semantics $\mathcal{R}$ described in Section III-B.

For the complexity analysis, the tuple-based cost of finding the candidate applications is $\mathcal{O}(|\Sigma|\times|M|)$. Since the complexity of the outer loop is $\mathcal{O}(|R|)$ (the number of transitions is bounded by the size of the schema, cf Section IV-A), the total repairing cost for a tuple using the naive implementation is then $\mathcal{O}(|R| \times |\Sigma| \times |M|)$, where $|R|$ and $|\Sigma|$ are typically small in practice.

Algorithm 2 getCandidateApplications (Optimized)

Input: A set of similarity indices $\mathcal{S}$ over M; An inverted index $\mathcal{I}$; A dirty tuple t.
Output: A set α of rules applications

```
1: α := ∅
2: for each φ ∈ I do
3:    A ← φ.getMatchingAttributes()
4:    Γ := ∅
5:    for each A ∈ A do
6:       T_m ∈ S(A, t[A], ≃_A)
7:       if Γ = ∅ then
8:          Γ := T_m
9:       else
10:         Γ := Γ ∩ T_m
11:   for each t_m ∈ Γ do
12:      t^φ_{t_m} = α.consolidate(t, t_m, φ);
13:      α ← t^φ_{t_m};
```

B. Fast Repairing

In order to get a faster repairing process, in this section we will show how we are able to reduce the $(|\Sigma| \times |M|)$ quantity by using both a proper data structure to keep track of relevant rules, and similarity indices on reference records.

Similarity indices: To reduce $|M|$, we use a set of similarity indices as preferred access methods to M. We currently support three different similarity indices and different similarity measures: *BK-tree* [9] which can be parametrized by every string similarity distance defining a *metric space* (*e.g.,* edit distance); *FastSS* [30] that employs an algorithm based on deletions to model the edit distance; and an *n-gram*-based index which can use edit distance as well as token-based similarity measures such as cosine similarity (see Section VI-B3 for a comparison over the performances of the indices). For each attribute in M a proper index is built, based on the similarity operators encoded into the rules by the user. The final result returned by an access query to the reference table M is the set intersection of the records returned by every single attribute index access.

Inverted index: In order to lower the impact of $|\Sigma|$, we take track of all the applicable rules for each dirty tuple, *i.e.,* the rules for which B has not yet been positively bounded. In order to accomplish this, a hash map is maintained. For each dirty tuple, we store all the applicable rules. Initially, for each tuple all the rules can be applied. However, as a tuple is progressively updated/annotated, all the rules resulting with the B attribute being positively bounded are removed from the applicable set. In this way, intuitively, in the best case in which a tuple can be completely covered, the set of applicable rules will eventually be empty.

Algorithm: The new optimized repairing procedure is depicted in Algorithm 1. The algorithm works as in the naive implementation, except for the set $\mathcal{I}$, and the initialization of the similarity indices $\mathcal{S}$ (line 1). $\mathcal{I}$ maintains the set of rules applicable to it (*i.e.,* rules for which the B attribute is not already bounded as positive), and is passed to the function *getCandidateApplications*. $\mathcal{I}$ is initially filled with all the rules in Σ (line 3), and is updated every time a new set of candidate repairs is computed (lines 6-9). To achieve this, for every candidate application – denoted by $t^{\varphi}_{t_m}$ in the algorithm –

we remove from $\mathcal{I}$ all the rules whose B attribute will be positively bounded by the application once implemented.

Algorithm 2 describes the *getCandidateApplications* method. Here the reference table is accessed through a set of similarity indices $\mathcal{S}$, one for each attribute in R_m. For each dirty tuple, we have that $\mathcal{I}$ is accessed, and all the applicable rules retrieved (line 2). For each of such rules, first the relevant attributes used for the matching (as described in Section III) are fetched (line 3). For every relevant attribute, the index is queried using the dirty tuple value $t[A]$, and all the similar reference records T_m are retrieved (lines 5-6). In each iteration, the intersection among all the similar records T_m for different attributes is computed (lines 7-10). At the end, only the reference tuples similar to $t[\mathcal{A}]$ exist in Γ. These tuples are then used to return proper (consolidated) applications (lines 11-13).

Correctness and complexity: The correctness of the optimized algorithm follows from the correctness of the naive implementation. The former, in fact, is just a revisited version of the latter where proper data structures are employed to speed-up the run-time performance. The complexity of the optimized version of the single tuple repair is now $\mathcal{O}(|\Sigma| \times com(\mathcal{S}))$, where with $com(\mathcal{S})$ we denote the average complexity of accessing the similarity indices (*e.g.,* the complexity of the BK-tree index is $\mathcal{O}(log(m))$). Note that in this case, the quantity $|\Sigma|$ is in average lower than in the naive implementation because of the $\mathcal{I}$ data structure.

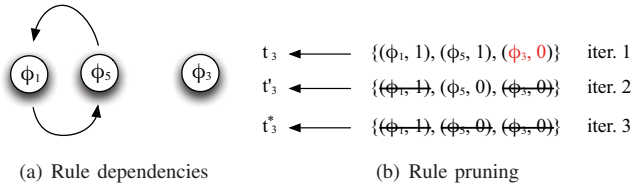
Further optimizations: In order to obtain some increases in the run-time performance, we add to our optimized algorithm a couple of supplementary extensions.

Caching similarity index accesses. Since accessing similarity indices can be expensive (Section VI-B3), we have conceived a set of hash maps acting as caches, so that index accesses can be shared among different rules. For instance, given a dirty tuple t, two applicable rules may share part of the X_m attributes. This means that some unnecessary computation is performed in line 6 of Algorithm 2 when the same index is accessed multiple times for the same tuple and the same set of attributes.

A similar behaviour exists for the intersection part of lines 7-10 of Algorithm 2. If over the same dirty tuple two rules share two or more attributes, we can not only avoid the index accesses for the second rule, but also skip the intersection part. This brings performance improvement especially in the cases in which many similar values exist in the reference table for certain pairs of values.

Rule pruning using dependency counting. The inverted index $\mathcal{I}$ is updated every time a rule becomes not applicable to a dirty tuple because its B attribute has been positively bounded. In certain specific cases, however, we can do some aggressive pruning of a rule, even if its B attribute has not been bounded. The following example illustrates the intuition behind this optimization.

Example 10: Consider rules $\varphi_1, \varphi_3, \varphi_5$ of Examples 4-8, dirty tuple t_3 of Fig. 1, and reference tables M_{CAP}, M_{REG}. Assume that first we want to apply φ_3. Since in M_{CAP} we do not have any entry for the value $t_3[\texttt{Chine}]$, φ_3 cannot be applied. Similarly for φ_5. However, φ_1 can be applied. At the second transition then, as we have seen in Example 8, φ_5 can be applied, while φ_3 still cannot. In the third transition, again, φ_3 cannot be applied. Note that in every transition the system tries to apply φ_3, although no change is manifested in the fields covered by φ_3 by rules φ_1, φ_5. Intuitively, starting from the second transition, φ_3 can be removed from the set of applicable rules of t_3 since: (i) it is not applicable in the first place, (ii) neither φ_1 nor φ_5 will change any value in t_3 which can make φ_3 applicable. □



(a) Rule dependencies (b) Rule pruning

Figure 9. Attribute-rate precision and recall of the annotations

We make concrete the intuition behind the above example by adding, for each rule φ belonging to the inverted index of a tuple, a counter maintaining the number of rules φ depends on, *i.e.,* the number of rules which, if applied, potentially can make φ applicable. Every time a rule is deleted from the inverted index, the counter of the remaining rules is decreased if they depend on the removed rule. Once the counter reaches zero, the rule becomes a candidate that is to be removed. In the successive transition, if the rule φ is still not applicable, it will then be removed.

Example 11: Figure 9 depicts the situation described in the previous example. The dependency graph is represented in Fig. 9(a). Here φ_3 is not connected with other rules since it does not depend on φ_1, φ_5. Figure 9(b) represents how the rules are pruned during the iterative repairing process. Initially all the rules can be used, although φ_3 is marked as red since its dependency count is zero, and hence, if not applicable, it can be pruned starting from the successive iteration. Rule φ_1 is then applied to t_3 giving t'_3 as the result. φ_1 and φ_3 are then dropped from the set of applicable rules. Finally we can apply φ_5, and at the second iteration we reach the final version t^*_3 since no more rule can be applied. □

VI. Experimental Study

In this section we describe an evaluation over how Sherlock rules can be used to deterministically and accurately clean a dirty database. The aim of our experiments is to provide insights on (i) the accuracy of the repairing algorithm; (ii) the accuracy of the annotations over the instance output of the repairing process; and (iii) the efficiency of the implementation, considering that different similarity indices, with different performances, can be adopted.

A. Experiment Setting

Datasets: We used both real-life and synthetic data.

(1) SYN simulates a staff directory of a company. The schema for this dataset is simple and is composed by the following attributes: Office Location, Employee First Name, Employee

Last Name, Office Phone Number, Mobile Phone Number and Email. We generated 100k records for measuring the accuracy of our approach. This dataset has been also used for assessing the scalability of the implementation. For this later study we have generated datasets of different numbers of tuples (100, 1000, 10000, and 100000).

(2) The TAX dataset was taken from the city of Trenton Certified Tax List[1]. Such dataset contains 27k tuples over the following list of attributes: Block, Lot, Qual, Class, Location, Owner Name, Owner Street, Owner City State, Owner Zip, Zoning, Year, Land Dimensions, Description, Additional Lots, Land Value, Improved Value and Total Value.

Data corruption. For generating dirty datasets, we treated the original datasets as the ground truth. Dirty data was generated by adding noise only to the attributes related with some functional dependencies, possibly spanning the full schema of the datasets. We controlled the injection of noise by selecting a noise rate, from 10% to 40%, with steps of 10%. For example, a noise rate of 30% means that at most one third of the attributes touched by some FD is corrupted. We limited the noise rate at 40% since, above this threshold, the original semantics of the record could be lost. Introduced noise has three types: *typos*, *errors from the active domain* (a value in a tuple is substituted with a different one belonging to the active domain of the attribute), and *semantic errors* (a value in a tuple is substituted with one belonging to a semantically related attribute). We treat semantic errors independently from the former two, *i.e.,* we use one parameter for defining the ratio of semantic errors, and one different parameter to specify the ratio of typos and domain errors. In the same tuple different types of errors can exist (*e.g.,* one value contain a typo, while another value has a semantic error), although a value can be corrupted just by one single type of error (*e.g.,* if a field already contains a semantic error, we cannot also add a typo).

Rule generation. Since Sherlock rules are at the schema level, just a small number of rules is required. To give an idea of the order of magnitude, for the experiments over our bigger schema TAX (17 attributes), just 67 rules have been used. Such 67 rules are generated starting from 29 seed rules provided by an expert. The rules generation procedure takes a rule as input and unfolds it based on transformation rules of Section III-B.

Note that, in respect to fixing rules, which are instance based and then even thousands of rules may be necessary to achieve a proper recall, in our case just few tens of rules are enough. This indeed also simplifies the process of consistency checking and implication.

Setting. The experiments were carried out on a i7-3770 machine with 8 3.40GHz CPUs. The operating system was a 64bit Ubuntu 14.04. The test scripts were written in Python, while the algorithms as well as the indices were implemented in Java.

We performed experiments to measure the accuracy of the repair, the accuracy of the annotations, and the efficiency of our approach.

[1]http://opendata.socrata.com/dataset/City-Of-Trenton-2012-Certified-Tax-List/mxtj-vhhx

Repairing and annotation accuracy: For what concern the repairing and annotations accuracy, for each of the two datasets we measured how the precision and recall evolve with the injection of different types of errors and at different rates. In practice, starting from a noise rate of 10%, we increased the ratio by a step of 10% until 40%. For each noise rate, we then selected different typos-active domain rate from 0% (100% of domain errors and 0% of typos errors) to 100% (0% of domain errors and 100% of typos) again by intervals of 10%. For each noise and typos-active domain rate, we then injected semantics errors from 0% to 100% rates, with steps of 10%. We assume that each tuple will be corrupted. Each experiment is computed over 5 different folds, and the reported results were computed as the average. For the accuracy of the repairing, Sherlock rules were compared with fixing and editing rules. That is, we provide a comparison with an automatic and a user-driven approach both able to obtain data repairs with high precision. High precision is indeed the same aim of Sherlock rules. For the specific case of editing rules, in order to automate the repair process, we have used the reference data to emulate the interaction with a human. A similar approach has been also used in [33].

Remark: (i) Sherlock rules subsume both fixing and editing rules when string equality is used as a similarity measure. (ii) Constant CFDs can be simulated by editing rules. We can therefore conclude that our comparison also include CFDs.

Fixing and editing rules do not provide annotations as output of the repairing process. Hence we avoid comparing sRs with editing and fixing rules in the second set of experiments since the comparison would trivially favor the former.

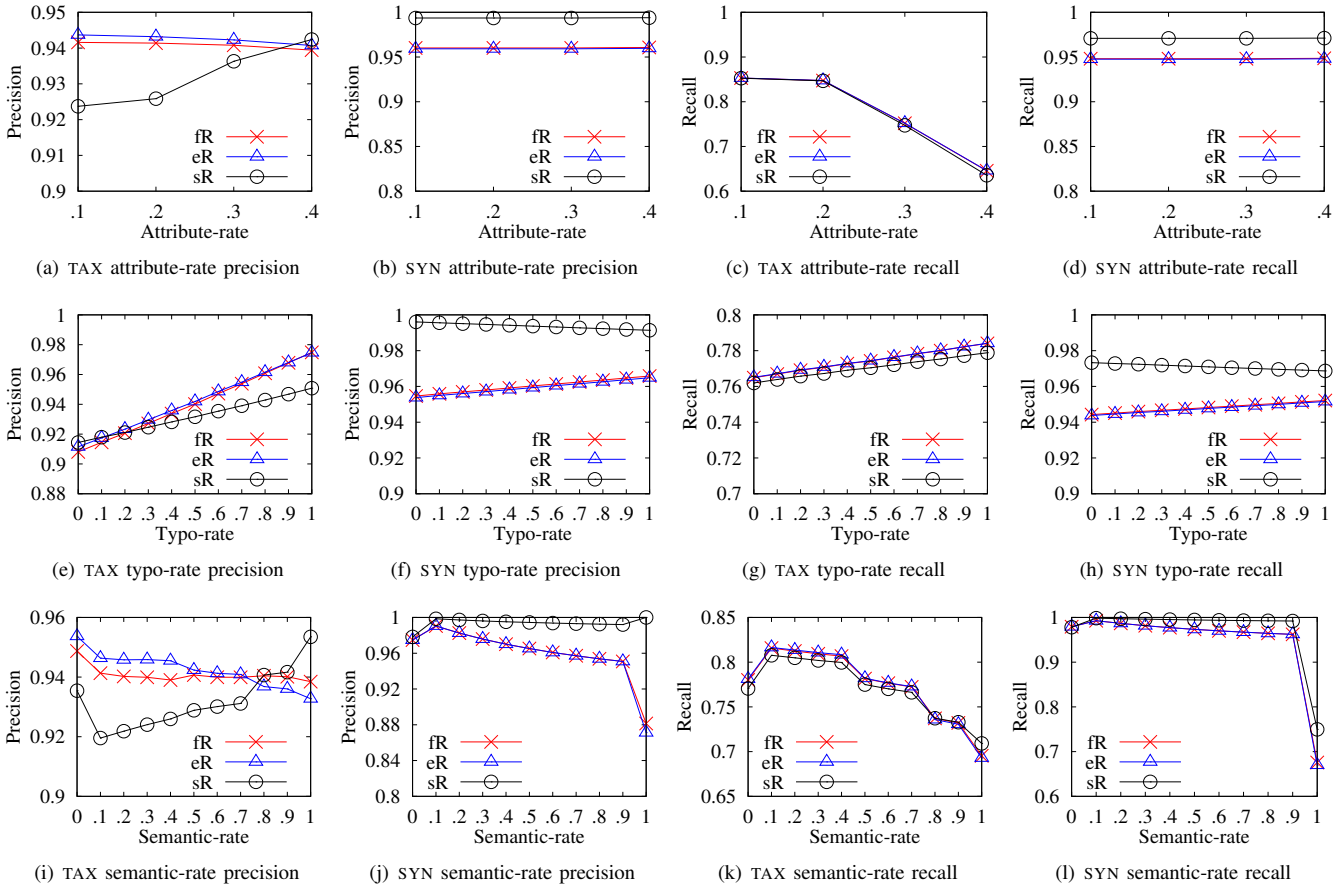
Efficiency: For what concern the efficiency assessment, we have first measured the performances of the naive and the optimized algorithm over the SYN dataset using string equality as similarity measure. For this set of experiments we have generated different dataset sizes and evaluated the scalability of the algorithms as the average over three runs. We have then carried out a set of experiments for measuring the impact of the k parameter in the index access cost. For this set of experiments we have used the SYN dataset with 10k tuples. The access cost was computed as the average cost of all the index access during a repairing run.

B. Experiment Results

In the following we discuss the findings of our experiments.

1) Repairing Accuracy: In order to get better insights over the accuracy of our repairing process, we have aggregated the results of our experiments over three different perspectives: (i) attribute-rate, (ii) typo-rate, and (iii) semantic-rate. In this way we are able to understand which are the strong and the week points of Sherlock rules. As a remark, recall that with Sherlock rules we are aiming high-precision repairing, therefore we are expecting to obtain good precision comparable with other approaches.

Varying the attribute-rate: The precision for datasets TAX and SYN is depicted in Figs. 10(a) and 10(b) respectively. As can be noticed, for both the datasets, with all the three (sR for Sherlock rules, fR for fixing rules, and eR for editing rules)



(a) TAX attribute-rate precision (b) SYN attribute-rate precision (c) TAX attribute-rate recall (d) SYN attribute-rate recall

(e) TAX typo-rate precision (f) SYN typo-rate precision (g) TAX typo-rate recall (h) SYN typo-rate recall

(i) TAX semantic-rate precision (j) SYN semantic-rate precision (k) TAX semantic-rate recall (l) SYN semantic-rate recall

Figure 10. Repairing accuracy for TAX and SYN datasets from different aggregation perspective

dependable repairs (precision is always greater that 0.92) are obtainable also when 40% of the attributes can be corrupted. In addition, in the SYN dataset all the approaches are quite independent from the number of noisy attributes. With the TAX dataset instead, fR and eR slightly decrease in precision with the increase of the noise rate, while sR precision increases. This suggests us that sRs are robust even under high error-rates. Note that in general, except for low noise rate in the TAX dataset, sR is comparable with or even better than the other two approaches.

The recall of the previous experiments is represented in Figs. 10(c) and 10(d). Notice that for the TAX datasets we have a decrease in the recall with the increase of the noise in the attributes, while the recall for the synthetic dataset is independent to the number of erroneous attributes, similarly for the precision.

Varying the typo-rate: If we aggregate by typo-rate instead of attribute-rate, we see from Figs. 10(e) and 10(f) that all the approaches have an increase in precision with the increase of the number of typos *w.r.t.* active domain errors. This shows that all such rules behave better with typos than with active domain errors. Interestingly, precision instead slightly decreases in the SYN dataset when we used Sherlock rules, although it always remains greater than editing and fixing rules.

A similar trend can be seen in the recall charts of Figs. 10(g) and 10(h). Even if Sherlock rules exploit similarity functions, the recall of Sherlock rules for the TAX dataset is lower than editing and fixing rules which instead use exact matching.

Varying the semantic rate: When the aggregation perspective is moved over the semantic rate, we can see from Figs. 10(i) and 10(j) that the two datasets behave similarly: with the increase of the number of errors, the precision of sRs increases, while the precision of editing and fixing rules decreases. This is true especially for the SYN dataset where we have a drop in precision for editing and fixing rules when the maximum semantic-rate is reached. This is due to the fact that Sherlock rules become more effective when semantic errors exist in the records, since sRs are able to exploit the semantic fix (transformation rule 1) to drive the repair over the entire tuple with high accuracy. Although fixing rules are also able to detect and repair such type of errors, they lose precision when many of them are injected into records.

Figures 10(k) and 10(l) depict how the recall varies with the semantic rate. The two datasets still behave similarly: both of them have an initial increase in recall, and then a constant decrease, with an important drop for the SYN dataset at high semantic error rates. Different from the precision case, here we have that all the approaches behave almost equivalently.

From this initial set of experiments we can draw the following conclusions: (i) as also shown in [33] fixing and editing rules behave in a similar way; (ii) Sherlock rules

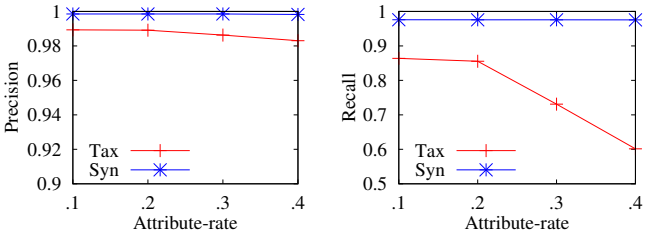


Figure 11. Attribute-rate precision and recall of the annotations

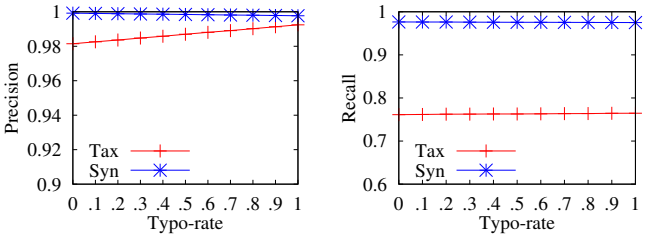


Figure 12. Typo-rate precision and recall of the annotations

are comparable with the aforementioned approaches both in precision and recall. sRs behave better in general for the SYN dataset, while eRs and fRs are slightly better in TAX.

Discussion: In fact, one can also consider editing rules used as a simulation of constant CFDs: If the values of a tuple matches the left hand side of a constant CFD φ, the values of the tuple correspond to the right hand side of φ will be changed correspondingly. Hence, the results in Fig. 10 also demonstrates that sRs outperform constant CFDs in accuracy. Moreover, we defined 14 variable CFDs on the same attributes applied by our sRs over the TAX data. To favour the repairing algorithms for CFDs, we injected 10% errors that are only typos. We ran NADEEF [16] to repair the data. The result shows that the precision of sRs is 97%, which is much better than 83.5% of CFD based repairs. A full comparison with constraint based repairing is omitted due to space constraints.

2) Annotation Accuracy: Now that we have assessed that with Sherlock rules we are able to provide accurate repairs, we are going to describe the accuracy of the annotations of the repaired instance, *i.e.,* with which accuracy further approaches can rely on our annotations. Also in this case, we provide different levels of aggregation over our experiments.

Varying noise rate: The precision and the recall of the annotations for datasets TAX and SYN are represented in Fig. 11. In respect to the precision of the repair, for the annotations we have that in both dataset the precision is higher, *i.e.,* more than 98%. This is because sRs are able to identify and annotate correct values (*i.e.,* proof positive values) in the dirty instance with high precision.

For what concern the recall, annotations maintain the same attitude of the repairing case, both in absolute values and in curve behaviour.

Varying the typo and the semantic rate: For completeness we have added the charts depicting how annotations behave varying the type and the semantic rate. We can see from Figs. 12 and 13 that both precision and recall behave as in the repair case, although we obtain higher precision without losing recall.

Varying the attributes in the reference table: During the paper

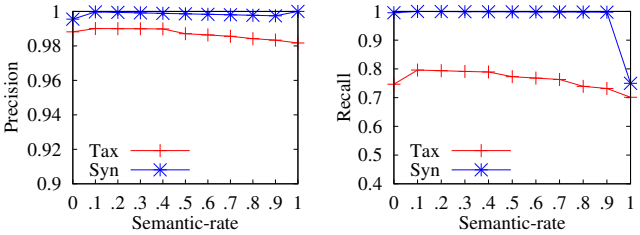


Figure 13. Semantic-rate precision and recall of the annotations

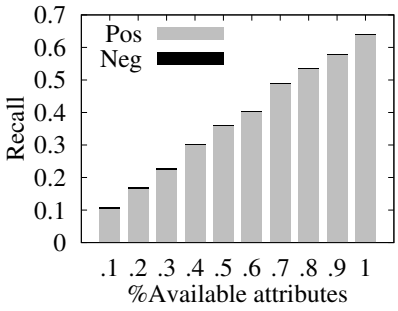


Figure 14. Recall of the coverage varying the reference table attributes

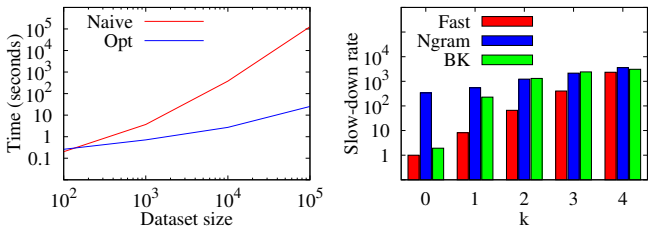
we have illustrated that sRs are quite flexible, even in the case in which the reference table partially covers the information stored in the input dirty instance. To empirically prove this, we measured the recall of the annotations of Sherlock rules while varying the number of attributes in the reference table. In this experiment we have used the TAX dataset with an attribute error rate of 30% and a typo / semantic rate of 50%. The results can be seen in Fig. 14. As expected, we have that the recall constantly increases with the increase of the number of attributes in the reference table. In addition, the number of proof negative values decreases up to almost disappear when 50% of the attributes are available. Intuitively, this is due to the fact that more information is accessible, and hence the wrong values, which before could just be annotated, can now be repaired.

3) Efficiency: In this last set of experiment, we describe the performance of our repairing process.

Figure 15(a) depicts a comparison in terms of scalability of the naive and the optimized algorithm. Both axes are in logarithm scale and they represent how the performance varies in time over different instances size of the SYN dataset. For both algorithms we have used edit distance with $k = 0$ for all attributes, *i.e.,* exact string matching. We used the FastSS index in the optimized algorithm. As can be seen, the optimized program scale linearly.

Since we expect different performance for each index and for different settings of k, in Fig. 15(b) we show what is the trade-off of using a certain index or of varying k, in terms of cost of index accesses. More precisely, we set at 1 the cost of using the FastSS index with exact matching (*i.e.,* $k = 0$), and we plotted the slow-down of using different indices with different values of k, compared with the base case. In other worlds, we plotted how slow are the other indices and k configurations compared to the best case.

As expected, the cost of each access increases with the increase of the k parameter up to 3 order of magnitude more *w.r.t.* the case in which $k = 0$. For what concern instead the



(a) Scalability comparison (b) Index access time comparison

Figure 15. Performances of the implementation

index comparison, we have that with low levels of k, FastSS perform better than the other two indices. At high k values, instead, all the indices perform in a similar way.

Summary. We found the following from the above experiments. (1) Sherlock rules can achieve high precision repairing, comparable with the state-of-the-art (Section VI-B1). (2) Sherlock rules can do annotations with high precision and good recall when enough evidences are present (Section VI-B2). (3) Sherlock rules are scalable since they work on each individual tuple (Section VI-B3).

VII. Conclusion and Future Work

In this paper we have proposed Sherlock rules. Sherlock rules are able to repair and annotate dirty instances in a deterministic fashion with high precision. Differently from fixing rules this can be obtained without having to specify hundreds or thousands of rules, but just exploiting external reliable (but not complete) data sources. Differently from editing rules, we are able to fix data in an automatic fashion, without any user involvement. We have provided a thorough analysis of Sherlock rules, both from a theoretical and practical point of view. We have identified four fundamental problems. Namely deciding whether a repair (i) terminates or (ii) it is deterministic; and if a set of rules (iii) is consistent, or (iv) implies another rule. We have proposed an efficient data repairing algorithms including multiple optimization strategies and data structures. Finally, we have experimentally verified Sherlock rules both over synthetic and real-life data, and both from an efficacy and efficiency perspective.

Many interesting features springing from Sherlock rules deserve a further investigation. One problem is *rule discovery*. In addition it would be interesting to research how different reference tables (maybe with information at different level of granularity) interact among themselves through Sherlock rules. This could be useful in order to factor out some principle that can be applied over heterogeneous data sources.

References

[1] Dirty data costs the U.S. economy $3 trillion+ per year. http://www.ringlead.com/dirty-data-costs-economy-3-trillion/.

[2] Firms full of dirty data. http://www.itpro.co.uk/ 609057/firms-full-of-dirty-data.

[3] A. Arasu, S. Chaudhuri, and R. Kaushik. Learning string transformations from examples. *PVLDB*, 2009.

[4] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. *TPLP*, 2003.

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

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

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

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

[9] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, 1973.

[10] F. Chiang and R. J. Miller. Discovering data quality rules. *PVLDB*, 1(1), 2008.

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

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

[13] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13), 2013.

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

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

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

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

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

[19] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5), 2011.

[20] W. Fan, F. Geerts, S. Ma, N. Tang, and W. Yu. Data quality problems beyond consistency and deduplication. In *In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman*, 2013.

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

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

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

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

[25] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC data-cleaning framework. *PVLDB*, 6(9), 2013.

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

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

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

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

[30] B. S. T. Bocek, E. Hunt. Fast Similarity Search in Large Dictionaries. Technical Report ifi-2007.02, Department of Informatics, University of Zurich, April 2007. http://fastss.csg.uzh.ch/.

[31] N. Tang. Big data cleaning. In *APWeb*, 2014.

[32] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous data cleaning. In *ICDE*, 2014.

[33] J. Wang and N. Tang. Towards dependable data with fixing rules. In *SIGMOD*, 2014.

[34] M. Yakout, L. Berti-Equille, and A. K. Elmagarmid. Don't be SCAREd: use SCalable Automatic REpairing with maximal likelihood and bounded changes. In *SIGMOD*, 2013.

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