Towards Certain Fixes with Editing Rules and Master Data

Wenfei Fan[1,2] Jianzhong Li[2] Shuai Ma[1] Nan Tang[1] Wenyuan Yu[1]
[1]University of Edinburgh [2]Harbin Institute of Technology
{wenfei@inf.,shuai.ma@, ntang@inf., wenyuan.yu@}ed.ac.uk lijzh@hit.edu.cn

Abstract

A variety of integrity constraints have been studied for data cleaning. While these constraints can detect the presence of errors, they fall short of guiding us to correct the errors. Indeed, data repairing based on these constraints may not find *certain fixes* that are absolutely correct, and worse, may introduce new errors when repairing the data. We propose a method for finding certain fixes, based on master data, a notion of *certain regions*, and a class of *editing rules*. A certain region is a set of attributes that are assured correct by the users. Given a certain region and master data, editing rules tell us what attributes to fix and how to update them. We show how the method can be used in data monitoring and enrichment. We develop techniques for reasoning about editing rules, to decide whether they lead to a unique fix and whether they are able to fix all the attributes in a tuple, *relative to* master data and a certain region. We also provide an algorithm to identify minimal certain regions, such that a certain fix is warranted by editing rules and master data as long as one of the regions is correct. We experimentally verify the effectiveness and scalability of the algorithm.

1. Introduction

Real-life data is often dirty: 1%–5% of business data contains errors [25]. Dirty data costs US companies alone 600 billion dollars each year [10]. These highlight the need for data cleaning, to catch and fix errors in the data. Indeed, the market for data cleaning tools is growing at 17% annually, way above the 7% average forecast for other IT sectors [17].

An important functionality expected from a data cleaning tool is *data monitoring* [6, 26]: when a tuple t is created (either entered manually or generated by some process), it is to find errors in t and correct the errors. That is, we want to ensure that t is clean before it is used, to prevent errors introduced by adding t. As noted by [26], it is far less costly to correct t at the point of entry than fixing it afterward.

A variety of integrity constraints have been studied for data cleaning, from traditional constraints (*e.g.,* functional and inclusion dependencies [4, 8, 30]) to their extensions (*e.g.,* conditional functional and inclusion dependencies [12, 5, 19]). These constraints help us determine whether data is dirty or not, *i.e.,* whether errors are present in the data. However, they fall short of telling us which attributes of t are erroneous and moreover, how to correct the errors.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '10, September 13-17, 2010, Singapore
Copyright 2010 VLDB Endowment, ACM 000-0-00000-000-0/00/00.

Example 1.1: Consider an input tuple t_1 given in Fig. 1(a). It specifies a supplier in the UK: name (FN, LN), phone number (area code AC and phone phn), address (street str, city, zip code) and items supplied. Here phn is either home phone or mobile phone, indicated by type (1 or 2, respectively).

It is known that if AC is 020, city should be Ldn, and when AC is 131, city must be Edi. These can be expressed as conditional functional dependencies (CFDs [12]). The tuple t_1 is *inconsistent*: $t_1[\mathsf{AC}] = 020$ but $t_1[\mathsf{city}] = $ Edi.

The CFDs detect that either $t_1[\mathsf{AC}]$ or $t_1[\mathsf{city}]$ is incorrect. However, they do not tell us which of the two attributes is wrong and to what value it should be changed. □

Several heuristic methods have been studied for repairing data based on constraints [3, 4, 9, 15, 22, 20]. For the reasons mentioned above, however, these methods do not guarantee to find correct fixes in data monitoring; worse still, they may introduce new errors when trying to repair the data. For instance, tuple s_1 of Fig. 1(b) indicates corrections to t_1. Nevertheless, all of the prior methods may opt to change $t_1[\mathsf{city}]$ to Ldn; this does not fix the erroneous $t_1[\mathsf{AC}]$ and worse, messes up the correct attribute $t_1[\mathsf{city}]$.

This motivates the quest for effective methods to find *certain fixes* that are guaranteed correct [18, 20]. The need for this is especially evident in monitoring *critical* data, in which an error may have disastrous consequences [20]. To this end we need *editing rules* that tell us how to fix errors, *i.e.,* which attributes are wrong and what values they should take. In contrast, constraints only detect the presence of errors.

This is possible given the recent development of master data management (MDM [23]). An enterprise nowadays typically maintains *master data* (*a.k.a. reference data*), a single repository of high-quality data that provides various applications with a synchronized, consistent view of its core business entities. MDM is being developed by IBM, SAP, Microsoft and Oracle. In particular, master data has been explored to provide *a data entry solution* in the SOA (Service Oriented Architecture) at IBM [26], for data monitoring.

Example 1.2: A master relation D_m is shown in Fig. 1(b). Each tuple in D_m specifies a person in the UK in terms of the name, home phone (Hphn), mobile phone (Mphn), address, date of birth (DOB) and gender. An example editing rule is:

- eR_1: for an input tuple t, if there exists a master tuple s in D_m with $s[\mathsf{zip}] = t[\mathsf{zip}]$, then t should be updated by $t[\mathsf{AC}, \mathsf{str}, \mathsf{city}] := s[\mathsf{AC}, \mathsf{str}, \mathsf{city}]$, provided that $t[\mathsf{zip}]$ is *certain*, *i.e.,* it is assured correct by the user.

This rule makes *corrections* to attributes $t[\mathsf{AC}]$ and $t[\mathsf{str}]$, by taking values from master data s_1. Another editing rule is

- eR_2: if $t[\mathsf{type}] = 2$ (indicating mobile phone) and if there is a master tuple s with $s[\mathsf{Mphn}] = t[\mathsf{phn}]$, then

	FN	LN	AC	phn	type	str	city	zip	item
t_1:	Bob	Brady	020	079172485	2	501 Elm St.	Edi	EH7 4AH	CD
t_2:	Robert	Brady	131	6884563	1	null	Ldn	null	CD
t_3:	Robert	Brady	020	6884563	1	null	null	EH7 4AH	DVD
t_4:	Mary	Burn	029	9978543	1	null	Cad	null	BOOK

(a) Example input tuples t_1 and t_2

	FN	LN	AC	Hphn	Mphn	str	city	zip	DOB	gender
s_1:	Robert	Brady	131	6884563	079172485	51 Elm Row	Edi	EH7 4AH	11/11/55	M
s_2:	Mark	Smith	020	6884563	075568485	20 Baker St.	Lnd	NW1 6XE	25/12/67	M

(b) Example master relation D_m

Figure 1: Example input tuples and master relation

$t[\mathsf{FN}, \mathsf{LN}] := s[\mathsf{FN}, \mathsf{LN}]$, as long as $t[\mathsf{phn}, \mathsf{type}]$ is certain. This *standardizes* $t_1[\mathsf{FN}]$ by changing Bob to Robert.

As another example, consider input tuple t_2 given in Fig. 1(a), in which attributes $t_2[\mathsf{str}, \mathsf{zip}]$ are missing, and $t_2[\mathsf{AC}]$ and $t_2[\mathsf{city}]$ are inconsistent. Consider an editing rule

- eR_3: if $t[\mathsf{type}] = 1$ (indicating home phone) and if there exists a master tuple s in D_m such that $s[\mathsf{AC}, \mathsf{phn}] = t[\mathsf{AC}, \mathsf{Hphn}]$, then $t[\mathsf{str}, \mathsf{city}, \mathsf{zip}] := s[\mathsf{str}, \mathsf{city}, \mathsf{zip}]$, provided that $t[\mathsf{type}, \mathsf{AC}, \mathsf{phn}]$ is *certain*.

This helps us fix $t_2[\mathsf{city}]$ and *enrich* $t[\mathsf{str}, \mathsf{zip}]$ by taking the corresponding values from the master tuple s_1. □

Contributions. We propose a method for data monitoring, by capitalizing on editing rules and master data.

(1) We introduce a class of editing rules defined in terms of data patterns and updates (Section 2). Given an input tuple t that matches a pattern, editing rules tell us what attributes of t should be updated and what values from master data should be assigned to them. In contrast to constraints, editing rules have a *dynamic* semantics, and are *relative to* master data. All the rules in Example 1.2 can be written as editing rules, but they are not expressible as constraints.

(2) We identify and study fundamental problems for reasoning about editing rules (Section 3). The analyses are relative to a *region* (Z, T_c), where Z is a set of attributes and T_c is a pattern tableau. One problem is to decide whether a set Σ of editing rules guarantees to find a *unique* (deterministic [18, 20]) fix for input tuples t that match a pattern in T_c. The other problems concern whether Σ is able to fix all the attributes of such tuples. Intuitively, as long as $t[Z]$ is assured correct, we want to ensure that editing rules can find a certain fix for t. We show that these problems are coNP-complete, NP-complete or #P-complete, but we identify special cases that are in polynomial time (PTIME).

(3) We develop an algorithm to derive certain regions from a set Σ of rules and master data D_m (Section 4). A certain region (Z, T_c) is such a region that a certain fix is warranted for an input tuple t as long as $t[Z]$ is assured correct and t matches a pattern in T_c. We naturally want to recommend minimal such Z's to the users. However, we show that the problem for finding minimal certain regions is NP-complete. Nevertheless, we develop an efficient heuristic algorithm to find a set of certain regions, based on a quality model.

(4) We experimentally verify the effectiveness and scalability of the algorithm, using real-life hospital data, DBLP as well as synthetic data TPC-H and RAND (Section 5). We find that the algorithm scales well with the size of master data and the size of editing rules. We also show that certain regions automatically derived by the heuristic algorithm are comparable to certain regions manually designed, when they are used to clean input tuples.

Taken together, these yield a data entry solution. A set of certain regions are first recommended to the users, derived from editing rules and master data available. Then for any input tuple t, if the users ensure that any of those regions in t is correct, the rules guarantee to find a certain fix for t.

Related work. Several classes of constraints have been studied for data cleaning (*e.g.,* [3, 4, 8, 5, 12, 22, 30]; see [11] for a survey). As remarked earlier, editing rules differ from those constraints in the following: (a) they are defined in terms of updates, and (b) their reasoning is relative to master data and is based on its dynamic semantics, a departure from our familiar terrain of dependency analysis. They are also quite different from edits studied for census data repairing [15, 18, 20], which are conditions defined on a single record and are used to detect errors.

Closer to editing rules are matching dependencies (MDs [13]). We shall elaborate their differences in Section 2.

Rules have also been studied for active databases (see [29] for a survey). Those rules are far more general than editing rules, specifying events, conditions and actions. Indeed, even the termination problem for those rules is undecidable, as opposed to the coNP upper bounds for editing rules. Results on those rules do not carry over to editing rules.

Data monitoring is advocated in [6, 14, 26]. A method for matching input tuples with master data was presented in [6], but it did not consider repairing the tuples. There has been a host of work on data repairing [3, 4, 8, 5, 12, 15, 18, 20, 22, 30], aiming to find a consistent database D' that minimally differs from original data D. It is to repair a database rather than cleaning an input tuple at the point of entry. Although the need for finding certain fixes has long been recognized [18, 20], prior methods do not guarantee that all the errors in D are fixed, or that D' does not have new errors. Master data is not considered in those methods.

Editing rules can be extracted from business rules. They can also be discovered from sample data along the same lines as mining constraints for data cleaning (*e.g.,* [7, 19]).

2. Editing Rules

We study editing rules for data monitoring. Given a master relation D_m and an input tuple t, we want to fix errors in t use editing rules and data values in D_m.

We specify input tuples t with a relation schema R. We use $A \in R$ to denote that A is an attribute of R.

The master relation D_m is an instance of a relation schema R_m, which is often distinct from R. As remarked earlier, D_m can be assumed consistent and complete [23].

Editing rules. An *editing rule* (eR) φ defined on (R, R_m) is a pair $((X, X_m) \to (B, B_m), t_p[X_p])$, 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 $B_m \in R_m$,
- t_p is a pattern tuple over a set of distinct attributes X_p in R, where for each $A \in X_p$, $t_p[A]$ is either a or $\bar{a}$ for a constant a drawn from the domain of A.

We say that a tuple t of R *matches* pattern $t_p[X_p]$, denoted by $t[X_p] \approx t_p[X_p]$, if for each attribute $A \in X_p$, (a) $t[A] = a$ if $t_p[A]$ is a, and (b) $t[A] \neq a$ if $t_p[A]$ is $\bar{a}$.

Example 2.1: Consider the supplier schema R and master relation schema R_m shown in Fig. 1. The rules eR_1, eR_2 and eR_3 described in Example 1.2 can be expressed as the following editing rules defined on (R, R_m).

φ_1: $((\mathsf{zip}, \mathsf{zip}) \rightarrow (B_1, B_1),\ t_{p1} = ())$
φ_2: $((\mathsf{phn}, \mathsf{Mphn}) \rightarrow (B_2, B_2),\ t_{p2}[\mathsf{type}] = (2))$
φ_3: $((\mathsf{phn}, \mathsf{Hphn}) \rightarrow (B_3, B_3),\ t_{p3}[\mathsf{type}, \mathsf{AC}] = (1, \overline{0800}))$
φ_4: $((\mathsf{AC}, \mathsf{AC}) \rightarrow (\mathsf{city}, \mathsf{city}),\ t_{p4}[\mathsf{AC}] = (\overline{0800}))$

Here eR_1 is expressed as three editing rules of the form φ_1, for B_1 ranging over AC, str and city. In φ_1, both X and X_m consist of zip, and B and B_m are B_1. Its pattern tuple t_{p1} poses no constraint. Similarly, eR_2 is expressed as two editing rules of the form φ_2, in which B_2 is either FN or LN. The pattern tuple $t_{p2}[\mathsf{type}] = (2)$, requiring that phn is mobile phone. The rule eR_3 is written as φ_3 for B_3 ranging over $\mathsf{str}, \mathsf{city}, \mathsf{zip}$, where $t_{p3}[\mathsf{type}, \mathsf{AC}]$ requires that $\mathsf{type} = 1$ (home phone) yet $\mathsf{AC} \neq 0800$ (toll free, non-geographic).

The eR φ_4 states that for a tuple t, if $t[\mathsf{AC}] \neq 0800$ and $t[\mathsf{AC}]$ is correct, we can update $t[\mathsf{city}]$ using master data. □

We next give the semantics of editing rules.

We say that an eR φ and a master tuple $t_m \in D_m$ *apply to* t, denoted by $t \rightarrow_{(\varphi, t_m)} t'$, if (a) $t[X_p] \approx t_p[X_p]$, (b) $t[X] = t_m[X_m]$, and (c) t' is obtained by the update $t[B] := t_m[B_m]$.

That is, if t matches t_p and if $t[X]$ agrees with $t_m[X_m]$, then we assign $t_m[B_m]$ to $t[B]$. Intuitively, if $t[X, X_p]$ is assured correct, we can safely *enrich* $t[B]$ with master data $t_m[B_m]$ as long as (a) $t[X]$ and $t_m[X_m]$ are identified, and (b) $t[X_p]$ matches the pattern in φ. This yields a new tuple t' with $t'[B] = t_m[B_m]$ and $t'[R \setminus \{B\}] = t[R \setminus \{B\}]$.

We write $t \rightarrow_{(\varphi, t_m)} = t$ if φ and t_m do not apply to t, *i.e., t* is unchanged by φ if either $t[X_p] \not\approx t_p[X_p]$ or $t[X] \neq t_m[X_m]$.

Example 2.2: As shown in Example 1.2, we can correct t_1 by applying the eRs φ_1 and master tuple s_1 to t_1. As a result, $t_1[\mathsf{AC}, \mathsf{str}]$ is changed to (131, 51 Elm Row). Furthermore, we can normalize $t_1[\text{FN}]$ by applying φ_2 and s_1 to t_1, such that $t_1[\text{FN}]$ is changed from Bob to Robert.

The eRs φ_3 and master tuple s_1 can be applied to t_2, such that $t_2[\mathsf{city}]$ is corrected and $t_2[\mathsf{str}, \mathsf{zip}]$ is enriched. □

Remarks. (1) As remarked earlier, editing rules are quite different from CFDs [12]. A CFD $\psi = (X \rightarrow Y, t_p)$ is defined on a single relation R, where $X \rightarrow Y$ is a standard FD and t_p is a pattern tuple on X and Y. It requires that for any tuples t_1, t_2 of R, if t_1 and t_2 match t_p, then $X \rightarrow Y$ is enforced on t_1 and t_2. It has a *static* semantics: t_1 and t_2 either satisfy or violate ψ, but they are not changed. In contrast, an eR φ specifies an action: applying φ and a master tuple t_m to t yields an updated t'. It is defined in terms of master data.

(2) The MDs of [13] also have a dynamic semantics. An MD ϕ is of the form $((X, X'), (Y, Y'), \mathsf{OP})$, where X, Y and X', Y' are lists of attributes in schemas R, R', respectively, and OP is a list of similarity operators. For a tuple t_1 of R_1 and a tuple t_2 of R_2, ϕ assures that if $t_1[X]$ and $t_2[X']$ match *w.r.t.* the operators in OP, then $t_1[Y]$ and $t_2[Y']$ are identified as the same object. In contrast to editing rules, (a) MDs are for record matching, not for data cleaning. They specify what attributes should be identified, but do not tell us how to update them. (b) MDs do not carry data patterns. (c) MDs do not consider master data, and hence, their analysis is far less challenging. Indeed, the static analyses of editing rules are intractable, while the analysis of MDs is in PTIME [13].

CFDs and MDs *cannot* be expressed as eRs, and *vice versa*.

(3) To simplify the discussion we consider a single master relation D_m. Nonetheless the results of this work readily carry over to multiple master relations.

3. Ensuring Unique and Certain Fixes

Consider a master relation D_m of schema R_m, and a set Σ of editing rules defined on (R, R_m). Given a tuple t of R, we want to find a "certain fix" t' of t by using Σ and D_m, *i.e.,* (a) no matter how eRs of Σ and master tuples in D_m are applied, Σ and D_m yield a unique t' by updating t; and (b) all the attributes of t' are ensured correct.

Below we first formalize the notion of certain fixes. We then study several problems for deciding whether Σ and D_m suffice to find a certain fix, *i.e.,* they ensure (a) and (b).

3.1 Certain Fixes and Certain Regions

When applying an eR φ and a master tuple t_m to t, we update t with a value in t_m. To ensure that the change makes sense, some attributes of t have to be assured correct. In addition, we cannot update t if either it does not match the pattern of φ or it cannot find a master tuple t_m in D_m that carries the information needed for correcting t.

Example 3.1: Consider the master relation D_m given in Fig. 1(a) and a set Σ_0 consisting of $\varphi_1, \varphi_2, \varphi_3$ and φ_4 of Example 2.1. Given input tuple t_3 of Fig. 1(a), both (φ_1, s_1) and (φ_3, s_2) apply to t_3. However, they suggest to update $t_3[\mathsf{city}]$ with distinct values Edi and Lnd. The conflict arises because $t_3[\mathsf{AC}]$ and $t_3[\mathsf{zip}]$ are inconsistent. Hence to fix t_3, we need to assure that one of $t_3[\mathsf{AC}]$ and $t_3[\mathsf{zip}]$ is correct.

Now consider tuple t_4 of Fig. 1(a). We find that no eRs in Σ_0 and tuples in D_m can be applied to t_4, and hence, we cannot decide whether t_4 is correct. This is because Σ_0 and D_m do not cover all the cases of input tuples. □

This motivates us to introduce the following notion.

Regions. A *region* is a pair (Z, T_c), where Z is a list of attributes in R, and T_c is a *pattern tableau* consisting of a set of pattern tuples with attributes in Z, such that for each tuple $t_c \in T_c$ and each attribute $A \in Z$, $t_c[A]$ is one of $_$, a or $\bar{a}$. Here a is a constant in the domain of A, and $_$ is an unnamed variable (wildcard).

Intuitively, a region tells us that to correctly fix errors in a tuple t, $t[Z]$ should be assured correct, and $t[Z]$ should "satisfy" a pattern in T_c (defined below). Here T_c specifies what cases of input tuples are covered by eRs and D_m.

A tuple t of R *satisfies* a pattern tuple t_c in T_c, denoted by $t \rightleftharpoons t_c$, if for each $A \in Z$, either $t_c[A] = _$, or $t[A] \approx t_c[A]$. That is, $t \rightleftharpoons t_c$ if either $t_c[A]$ is a wildcard, or $t[A]$ matches $t_c[A]$ when $t_c[A]$ is a or $\bar{a}$. We refer to t as a tuple *covered* by (Z, T_c) if there exists $t_c \in T_c$ such that $t \rightleftharpoons t_c$.

Consider an eR $\varphi = ((X, X_m) \rightarrow (B, B_m), t_p[X_p])$ and a master tuple t_m. We say that φ and t_m *correctly apply* to a tuple t *w.r.t.* (Z, T_c), denoted by $t \rightarrow_{((Z, T_c), \varphi, t_m)} t'$, if (a) $t \rightarrow_{(\varphi, t_m)} t'$, (b) $X \subseteq Z$, $X_p \subseteq Z$, $B \notin Z$, and (c) there exists a pattern tuple $t_c \in T_c$ such that $t \rightleftharpoons t_c$.

That is, it is justified to apply φ and t_m to t for those t covered by (Z, T_c) if $t[X, X_p]$ is correct. As $t[Z]$ is correct, we do not allow it to be changed by enforcing $B \notin Z$.

Example 3.2: Referring to Example 3.1, a region for tuples of R is $(Z_{\mathsf{AH}}, T_{\mathsf{AH}}) = ((\mathsf{AC}, \mathsf{phn}, \mathsf{type}), \{(\overline{0800}, _, 2)\})$. Hence, if $t_3[\mathsf{AC}, \mathsf{phn}, \mathsf{type}]$ is correct, then (φ_3, s_2) can be correctly applied to t_3, yielding $t_3 \rightarrow_{((\mathsf{AC},\mathsf{phn}),T_{\mathsf{AC}},\varphi_3,s_2)} t'_3$, where $t'_3[\mathsf{str}, \mathsf{city}, \mathsf{zip}] = s_2[\mathsf{str}, \mathsf{city}, \mathsf{zip}]$, and t'_3 and t_3 agree on all the other attributes of R. □

Observe that if $t \rightarrow_{((Z,T_c),\varphi,t_m)} t'$, then $t'[B]$ is also assured correct. Hence we can extend (Z, T_c) by including B in Z and by expanding each t_c in T_c such that $t_c[B] = _$. We denote the extended region as $\mathsf{ext}(Z, T_c, \varphi)$.

For instance, $\mathsf{ext}((\mathsf{AC}, \mathsf{phn}, \mathsf{type}), T_{\mathsf{AH}}, \varphi_3)$ is (Z', T'), where Z' consists of AC, phn, type, str, city and zip, and T' has a single tuple $t'_c = (\overline{0800}, _, 2, _, _, _)$.

Certain fix. For a tuple t of R covered by (Z, T_c), we want to make sure that we can get a unique fix t' no matter how eRs in Σ and tuples in D_m are applied to t.

We say that a tuple t' is a *fix* of t by (Σ, D_m), denoted by $t \rightarrow^*_{((Z,T_c),\Sigma,D_m)} t'$, if there exists a finite sequence $t_0 = t$, $t_1, \ldots, t_k = t'$ of tuples of R, and for each $i \in [1, k]$, there exist $\varphi_i \in \Sigma$ and $t_{m_i} \in D_m$ such that

(a) $t_{i-1} \rightarrow_{((Z_{i-1},T_{i-1}),\varphi_i,t_{m_i})} t_i$, where $(Z_0, T_0) = (Z, T_c)$, and $(Z_i, T_i) = \mathsf{ext}(Z_{i-1}, T_{i-1}, \varphi_i)$;

(b) $t_i[Z] = t[Z]$; and

(c) for all $\varphi \in \Sigma$ and $t_m \in D_m$, $t' \rightarrow_{((Z_m,T_m),\varphi,t_m)} t'$.

Intuitively, (a) each step of the correcting process is justified; (b) $t[Z]$ is assumed correct and hence, remains unchanged; and (c) t' is a fixpoint and cannot be further updated.

We say that t has a *unique fix* by (Σ, D_m) *w.r.t.* (Z, T_c) if there exists a unique t' such that $t \rightarrow^*_{((Z,T_c),\Sigma,D_m)} t'$.

When there exists a unique fix t' of t, we refer to Z_m as the set of attributes of t *covered by* (Z, T_c, Σ, D_m).

The fix t' is called the *certain fix* if the set of attributes covered by (Z, T_c, Σ, D_m) includes *all* the attributes in R.

Intuitively, if t has a certain fix t' then (a) it has a unique fix and (b) all the attributes of t' are correct provided that $t[Z]$ is correct. A notion of deterministic fix was addressed in [18, 20]. It refers to unique fixes, *i.e.,* (a) above, without requiring (b). Further, it is not defined relative to (Z, T_c).

Example 3.3: By the set Σ_0 of eRs of Example 3.1 and the master data D_m of Fig. 1(b), tuple t_3 of Fig. 1(a) has a unique fix *w.r.t.* $(Z_{\mathsf{AH}}, T_{\mathsf{AH}})$, namely, t'_3 given in Example 3.2. However, as observed in Example 3.1, if we extend the region by adding zip, denoted by $(Z_{\mathsf{AHZ}}, T_{\mathsf{AH}})$, then t_3 no longer has a unique fix by (Σ_0, D_m) *w.r.t.* $(Z_{\mathsf{AHZ}}, T_{\mathsf{AH}})$.

As another example, consider a region $(Z_{\mathsf{zm}}, T_{\mathsf{zm}})$, where $Z_{\mathsf{zm}} = (\mathsf{zip}, \mathsf{phn}, \mathsf{type})$, and T_{zm} has a single tuple $(_, _, 2)$. As shown in Example 2.2, tuple t_1 of Fig. 1(a) has a unique fix by Σ_0 and D_m *w.r.t.* $(Z_{\mathsf{zm}}, T_{\mathsf{zm}})$, by correctly applying (φ_1, s_1) and (φ_2, s_2). It is *not* a certain fix, since the set of attributes covered by $(Z_{\mathsf{zm}}, T_{\mathsf{zm}}, \Sigma_0, D_m)$ does not include item. Indeed, the master data D_m of Fig. 1(b) has no information about item, and hence, does not help here. To find a certain fix, one has to extend Z_{zm} by adding item. In other words, its correctness has to be assured by the users. □

Certain region. We say that (Z, T_c) is a *certain region* for (Σ, D_m) if for all tuples t of R that are covered by (Z, T_c), t has a certain fix by (Σ, D_m) *w.r.t.* (Z, T_c).

We are naturally interested in certain regions since they warrant absolute corrections, which are assured either by the users (the attributes in Z) or by master data $(R \setminus Z)$.

Example 3.4: As shown in Example 3.3, $(Z_{\mathsf{zm}}, T_{\mathsf{zm}})$ is not a certain region. One can verify that a certain region for (Σ_0, D_m) is $(Z_{\mathsf{zmi}}, T_{\mathsf{zmi}})$, where Z_{zmi} extends Z_{zm} with item, and T_{zmi} consists of patterns of the form $(z, p, 2, _)$ for z, p ranging over $s[\mathsf{zip}, \mathsf{Mphn}]$ for all master tuples s in D_m. For tuples covered by the region, a certain fix is warranted. □

3.2 Reasoning about Editing Rules

Given a set Σ of eRs and a master relation D_m, we want to make sure that they can *correctly* fix *all* errors in those input tuples covered by a region (Z, T_c). This motivates us to study several problems for reasoning about editing rules, and establish their complexity bounds (all the proofs are in the appendix, and some proofs are highly nontrivial).

The consistency problem. One problem is to decide whether (Σ, D_m) and (Z, T_c) do not have conflicts.

We say that (Σ, D_m) is *consistent relative to* (Z, T_c) if for each input tuple t of R that is covered by (Z, T_c), t has a unique fix by (Σ, D_m) *w.r.t.* (Z, T_c).

Example 3.5: There exist (Σ, D_m) and (Z, T_c) that are inconsistent. Indeed, (Σ_0, D_m) is not consistent relative to $(Z_{\mathsf{AHZ}}, T_{\mathsf{AHZ}})$ of Example 3.3, since tuple t_3 does not have a unique fix by (Σ_0, D_m) *w.r.t.* $(Z_{\mathsf{AHZ}}, T_{\mathsf{AHZ}})$. □

The *consistency problem* is to determine, given (Z, T_c) and (Σ, D_m), whether (Σ, D_m) is consistent relative to (Z, T_c).

Theorem 3.1: *The consistency problem is co*NP*-complete, even for relations with infinite-domain attributes only.* □

The consistency analysis of eRs is more intriguing than its CFD counterpart, which is NP-complete but is in PTIME when all attributes involved have an infinite domain [12]. It is also much harder than MDs, which is in quadratic-time [13]. Nevertheless, it is decidable, as opposed to the undecidability for reasoning about rules for active databases [29].

The coverage problem. Another problem is to determine whether (Σ, D_m) is able to fix errors in all attributes of input tuples that are covered by (Z, T_c).

The *coverage problem* is to determine, given any (Z, T_c) and (Σ, D_m), whether (Z, T_c) is a certain region for (Σ, D_m).

No matter how desirable to find certain regions, the coverage problem is intractable, although it is decidable.

Theorem 3.2: *The coverage problem is co*NP*-complete.* □

To derive a certain region (Z, T_c) from (Σ, D_m), one wants to know whether a given list Z of attributes could make a certain region by finding T_c, and if so, how large T_c is.

The Z*-validating* problem is to decide, given (Σ, D_m) and a list Z of attributes, whether there exists a nonempty tableau T_c such that (Z, T_c) is a certain region for (Σ, D_m).

The Z*-counting* problem is to determine, given (Σ, D_m) and Z, how many pattern tuples can be found from (Σ, D_m) to construct T_c such that (Z, T_c) is a certain region.

Both problems are beyond reach in practice. In particular, the Z-counting problem is as hard as finding the number of truth assignments that satisfy a given 3SAT instance [16].

Theorem 3.3: *(1) The Z-validating problem is* NP*-complete. (2) The Z-counting problem is* #P*-complete.* □

One would naturally want a certain region (Z, T_c) with a "small" Z, such that the users only need to assure the correctness of a small number of attributes in input tuples.

The Z-*minimum* problem is to decide, given (Σ, D_m) and a positive integer K, whether there exists a set Z of attributes such that (a) $|Z| \leq K$ and (b) there exists a pattern tableau T_c such that (Z, T_c) is a certain region for (Σ, D_m).

This problem is also intractable. Worse still, there exists no approximate algorithm for it with a reasonable bound.

Theorem 3.4: *The Z-minimum problem is (1) NP-complete, and (2) cannot be approximated within $c \log\ n$ in PTIME for a constant c unless NP $\subseteq$DTIME $(n^{polylog(n)})$.* □

Tractable cases. The intractability results suggest that we consider special cases that allow efficient reasoning.

Fixed Σ. One case is where the set Σ is fixed. Indeed, editing rules are often predefined and fixed in practice.

Concrete T_c. Another case is where no pattern tuples in T_c contain wildcard '_' or $\bar{a}$, *i.e.,* they contain a only.

Direct fix. We also consider a setting in which (a) for all eRs $\varphi = ((X, X_m) \rightarrow (B, B_m), t_p[X_p])$ in Σ, $X_p \subseteq X$, *i.e.,* the pattern attributes X_p are also required to find a match in D_m, and (b) each step of a fixing process employs (Z, T_c), *i.e.,* $t_{i-1} \rightarrow_{((Z,T_c),\varphi_i,t_{m_i})} t_i$, without extending (Z, T_c).

Each of these restrictions makes our lives much easier.

Theorem 3.5: *The consistency problem and the coverage problem are in* PTIME *if we consider (a) a fixed set Σ of eRs, (b) a concrete pattern tableau T_c, or (c) direct fixes.* □

However, it does not simplify the other problems.

Corollary 3.6: *When only direct fixes are considered, the Z-validating, Z-counting and Z-minimum problems remain* NP*-complete,* #P*-complete, both* NP*-complete and approximation-hard, respectively.* □

One might think that fixing master data D_m would also simplify the analysis of eRs. Unfortunately, it does not help.

Corollary 3.7: *Both the consistency problem and the coverage problem remain* coNP*-complete when D_m is fixed.* □

4. Computing Certain Regions

An important issue concerns how to automatically derive a set of certain regions from a set Σ of eRs and a master relation D_m. These regions are recommend to users, such that Σ and D_m warrant to find an input tuple t a certain fix as long as the users assure that t is correct in any of these regions. However, the intractability and approximation-hardness of Theorems 3.2, 3.3 and 3.4 tell us that any efficient algorithms for deriving certain regions are necessarily heuristic.

We develop a heuristic algorithm based on a *characterization* of certain regions as cliques in graphs. Below we first introduce the characterization and then present the algorithm. We focus on direct fixes, a special case identified in Section 3.2 that is relatively easier (Theorem 3.5) but remains intractable for deriving certain regions (Corollary 3.6).

Proofs of all the results of this section are in the appendix.

4.1 Capturing Certain Regions as Cliques

We first introduce a notion of compatible graphs to characterize eRs and master data. We then establish the connection between certain regions and cliques in such a graph.

Compatible graphs. Consider $\Sigma = \{ \varphi_i \mid i \in [1, n]\}$ defined on (R, R_m), where $\varphi_i = ((X_i, X_{m_i}) \rightarrow (B_i, B_{m_i}), t_{p_i}[X_{p_i}])$. We use the following notations.

(1) For a list X'_i of attributes in X_i, we use $\lambda_{\varphi_i}(X'_i)$ to denote the corresponding attributes in X_{m_i}. For instance, when $(X_i, X_{m_i}) = (ABC, A_mB_mC_m)$, $\lambda_{\varphi_i}(AC) = A_mC_m$.

We also use the following: (a) $\text{LHS}(\varphi_i) = X_i$, $\text{RHS}(\varphi_i) = B_i$; (b) $\text{LHS}_m(\varphi_i) = X_{m_i}$, $\text{RHS}_m(\varphi) = B_{m_i}$; and (c) $\text{LHS}_p(\varphi_i) = X_{p_i}$. For a set Σ_c of eRs, we denote $\cup_{\varphi\in\Sigma_c}\text{LHS}(\varphi)$ by $\text{LHS}(\Sigma_c)$; similarly for $\text{RHS}(\Sigma_c)$, $\text{LHS}_m(\Sigma_c)$ and $\text{RHS}_m(\Sigma_c)$.

(2) Consider pairs (φ_i, t_m) and (φ_j, t'_m) of eRs and master tuples such that $t_{p_i}[X_{p_i}] \approx t_m[\lambda_{\varphi_i}[X_{p_i}]]$ and $t_{p_j}[X_{p_j}] \approx t'_m[\lambda_{\varphi_j}[X_{p_j}]]$. We say that t_m and t'_m are *conflict tuples* if (a) $B_i = B_j$ and $t_m[B_{m_i}] \neq t'_m[B_{m_j}]$, and (b) for each attribute $A \in X_i \cap X_j$, $t_m[\lambda_{\varphi_i}(A)] = t'_m[\lambda_{\varphi_j}(A)]$.

That is, (φ_i, t_m) and (φ_j, t'_m) may incur conflicts when they are applied to the same input tuple. To avoid taking conflict tuples in T_c, we remove conflict tuples from D_m, and refer to the result as the *reduced* master data D_s.

(3) We say that eR-tuple pairs (φ_i, t_m) and (φ_j, t'_m) are *compatible* if (a) $B_i \neq B_j$, $B_i \notin X_j$, $B_j \notin X_i$, and (b) for each attribute $A \in X_i \cap X_j$, $t_m[\lambda_{\varphi_i}(A)] = t'_m[\lambda_{\varphi_j}(A)]$.

Intuitively, we can apply (φ_i, t_m) and (φ_j, t'_m) to the same input tuple t if they are compatible.

We are now ready to define compatible graphs.

The *compatible graph* $G(V, E)$ of (Σ, D_m) is an undirected graph, where (1) the set V of nodes consists of eR-tuple pairs (φ_i, t_m) such that $\varphi_i \in \Sigma$, $t_m \in D_s$, and $t_{p_i}[X_{p_i}] \approx t_m[\lambda_{\varphi_i}[X_{p_i}]]$; and (2) the set E of edges consists of (u, v) such that u and v in V are compatible with each other.

The graph $G(V, E)$ depicts what eR-tuple pairs are compatible and can be applied to the same tuple. Note that V (resp. E) is bounded by $O(|\Sigma||D_m|)$ (resp. $O(|\Sigma|^2|D_m|^2)$).

The connection. We now establish the connection between identifying certain regions (Z, T_c) for (Σ, D_m) and finding cliques $\mathcal{C}$ in the compatible graph $G(V, E)$.

Consider a clique $\mathcal{C} = \{v_1, \ldots, v_k\}$ in G, where for each $i \in [1, k]$, $v_i = (\varphi_i, t_{m_{j_i}})$. Let $\Sigma_\mathcal{C}$ be the set of eRs in the clique $\mathcal{C}$. Then it is easy to verify (a) $\text{LHS}(\Sigma_\mathcal{C}) \cap \text{RHS}(\Sigma_\mathcal{C}) = \emptyset$, and (b) $|\text{RHS}(\Sigma_\mathcal{C})| = |\mathcal{C}| = k$, *i.e.,* the number of attributes in $\text{RHS}(\Sigma_\mathcal{C})$ is equal to the number of nodes in $\mathcal{C}$.

Let $Z = R \setminus \text{RHS}(\Sigma_\mathcal{C})$, and t_c be a tuple with attributes in Z such that (a) $t_c[\text{LHS}(\Sigma_\mathcal{C})] = t_{j_1} \bowtie \ldots \bowtie t_{j_k}[\text{LHS}(\Sigma_\mathcal{C})]$, where for each $i \in [1, k]$, $t_{j_i}[X_iB_i] = t_{m_{j_i}}[X_{m_i}B_{m_i}]$, and (b) $t_c[A] =$ '_' for all remaining attributes $A \in Z$. Here $\bowtie$ is the natural join operator. Then it is easy to verify that $(Z, \{t_c\})$ is a certain region for (Σ, D_m). Hence we have:

Proposition 4.1: *Each clique in the compatible graph G of (Σ, D_m) corresponds to a certain region for (Σ, D_m).* □

This allows us to find certain regions by employing algorithms (*e.g.,* [21, 24]) for finding maximal cliques in a graph.

Compressed graphs. However, the algorithms for finding cliques take $O(|V||E|)$ time on each clique. When it comes to compatible graph, it takes $O(|\Sigma|^3|D_m|^3)$ time for each certain region, too costly to be practical on large D_m.

In light of this we compress a compatible graph $G(V, E)$ by removing master tuples from the nodes. More specifically, we consider *compressed compatible graph* $G^c(V^c, E^c)$, where (1) V^c is Σ, *i.e.,* each node is an eR in Σ, and (2) there is an edge (φ_i, φ_j) in E^c iff there exist master tuples t_m, t'_m such that $((\varphi_i, t_m), (\varphi_j, t'_m))$ is an edge in E.

Observe that G^c is much smaller than G and is independent of master data D_m: V^c is bounded by $O(|\Sigma|)$ and E^c is bounded by $O(|\Sigma|^2)$. On the other hand, however, it is no longer easy to determine whether a clique yields a cer-

tain region. More specifically, let $\mathcal{C}$ be a clique in G^c and $Z = R \backslash \text{RHS}(\Sigma_{\mathcal{C}})$. The Z-validating problem *for a clique* is to determine whether there exists a nonempty pattern tableau T_c such that (Z, T_c) is a certain region for (Σ, D_m).

Theorem 4.2: *The Z-validating problem for a clique in a compressed graph G^c is* NP*-complete.* □

A heuristic. To cope with the intractability we develop a heuristic algorithm to validate Z. We partition Z into Z_1 and Z_2 such that only Z_2 is required to match a list Z_m of attributes in R_m, where the correctness of Z_1 is to be assured by the users. Here Z_2 and Z_m are $\text{LHS}(\Sigma_{\mathcal{C}})$ and $\text{LHS}_m(\Sigma_{\mathcal{C}})$, respectively, derived from a clique $\mathcal{C}$ in the compressed graph G^c. We denote this by $W = (Z_1, Z_2 \parallel Z_m)$, where $Z = Z_1 \cup Z_2$, $Z_1 \cap Z_2 = \emptyset$ and $|Z_2| = |Z_m|$.

The W-validating problem asks whether there exists t_m in D_s such that $(Z_1 Z_2, \{t_c\})$ is a certain region for (Σ, D_s), where $t_c[Z_1]$ consists of '_' only and $t_c[Z_2] = t_m[Z_m]$. That is, t_c is extracted from a single master tuple, not a combination from multiple. In contrast to Theorem 4.2, we have:

Proposition 4.3: *There exists an $O(|\Sigma||D_s|\log|D_s|)$-time algorithm for the W-validating problem.* □

Based on this, the algorithm works as follows. Given Z and a clique $\mathcal{C}$, it first partitions Z into $W = (Z_1, Z_2 \parallel Z_m)$. It then finds a tuple t_c using the $O(|\Sigma||D_s|\log|D_s|)$-time algorithm. To ensure the correctness we require that for any φ_i and φ_j in G^c, $\text{LHS}(\varphi_i)$ and $\text{LHS}(\varphi_j)$ are disjoint. In fact, a set of certain regions can be generated when validating W, one for each master tuple in D_s. We employ this idea to generate certain regions from cliques in the compressed compatible graph (in Procedure cvrtClique, see the appendix).

4.2 A Graph-based Heuristic Algorithm

Based on the graph characterization we provide Algorithm compCRegions (see Fig. 6 in the appendix).

The algorithm takes as input a positive integer K, a set Σ of eRs and master data D_m. It returns an array M of certain regions (Z, T_c) such that $M[Z] = (Z, T_c)$.

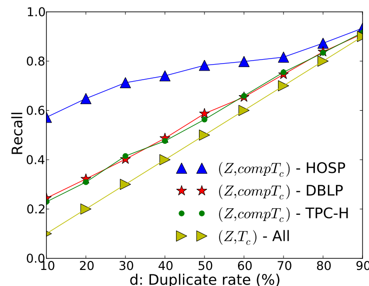
It first computes a *reduced* master relation D_s, and builds the compressed compatible graph G^c of (Σ, D_s). It then finds up to K maximal cliques in G^c, and converts these cliques into certain regions. Finally, it constructs M by merging certain regions having the same Z (see the appendix for the details of the algorithm and a running example).

The algorithm guarantees to return a nonempty set M of certain regions, by Propositions 4.1 and 4.3. It is in $O(|\Sigma|^2|D_m|\log|D_m| + K|\Sigma|^3 + K|\Sigma||D_m|\log|D_m|)$ time. In practice, $|\Sigma|$ and K are often small. We shall verify its effectiveness and efficiency in Section 5.

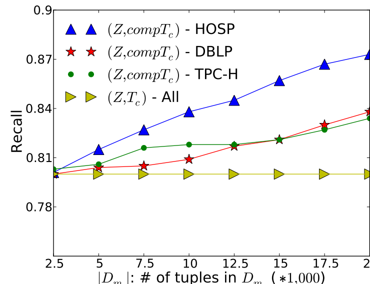
A preference model. When there exist more than K maximum cliques we need to decide which K cliques to pick. To this end, the algorithm adopts a preference model that ranks certain regions (Z, T_c) based on the following factors.

- The *number* $|Z|$ *of attributes* in Z. We naturally want Z to be as small as possible. The larger the size of a clique $\mathcal{C}$ is, the smaller $|Z|$ is for Z derived from $\mathcal{C}$.
- The *accuracy* $\mathsf{ac}(A)$ of $A \in R$, indicating the confidence placed by the user in the accuracy of the attribute. The smaller $\mathsf{ac}(A)$ is, the more reliable A is.

The algorithm uses a total order $\mathcal{O}$ for the eRs in Σ such that $\mathcal{O}(\varphi) < \mathcal{O}(\varphi')$ if $\mathsf{ac}(\text{RHS}(\varphi)) < \mathsf{ac}(\text{RHS}(\varphi'))$. It finds maximum cliques (small regions). Cliques having eRs φ with unreliable $\text{RHS}(\varphi)$ are returned first. Hence, small Z with reliable attributes derived from the cliques are selected.



(a) Varying $d\%$



(b) Varying $|D_m|$

Figure 2: Tuple Level Recall

5. Experimental Study

We next present an experimental study using both real-life data and synthetic data. Two sets of experiments were conducted to verify (1) the effectiveness of the certain regions obtained; and (2) the efficiency and scalability of algorithm compCRegions in deriving certain regions. For the effectiveness study, we used the incremental repairing algorithm developed in [9], IncRep, for comparison.

Experimental setting. Real-life data (HOSP and DBLP) was used to test the efficacy of certain regions derived by our algorithm in real world. Synthetic data (TPC-H and RAND) was employed to control the characteristics of data and editing rules, for an in-depth analysis.

(1) HOSP (Hospital Compare) data is publicly available from U.S. Department of Health & Human Services[1]. There are 37 eRs designed for HOSP.

(2) DBLP data is from the DBLP Bibliography[2]. There are 16 eRs designed for DBLP.

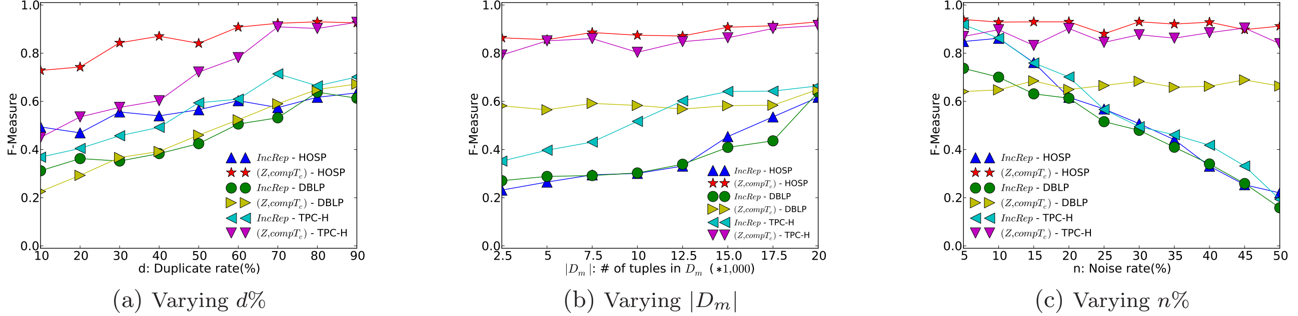
(3) TPC-H data is from the DBGEN generator [3]. There are 55 eRs designed for TPC-H.

(4) RAND data was generated by varying the following parameters: (a) the number of attributes in the master relation R_m; (b) the number of attributes in the relation R; (c) the number of master tuples; (d) the number of editing rules (eRs); and (e) the number of attributes in LHS of eRs.

We refer the reader to the appendix for the details of the datasets, the editing rules designed, and Algorithm IncRep.

Implementation. All algorithms were implemented in Python 2.6, except that the C implementation[4] in [24] was

[1]http://www.hospitalcompare.hhs.gov/
[2]http://www.informatik.uni-trier.de/∼ley/db/
[3]http://www.tpc.org/tpch/
[4]http://research.nii.ac.jp/∼uno/code/mace.htm



(a) Varying $d\%$ (b) Varying $|D_m|$ (c) Varying $n\%$

Figure 3: F-Measure *w.r.t.* $d\%$, $|D_m|$ and $n\%$

used to compute maximal cliques. All experiments were run on a machine with an Intel Core2 Duo P8700 (2.53GHz) CPU and 4GB of memory. Each experiment was repeated over 5 times and the average is reported here.

Experimental results. We next present our findings.

Exp-1: Effectiveness. We used real-life datasets HOSP and DBLP, and synthetic TPC-H data to verify the effectiveness of certain regions found by our heuristic compCRegions.

The tests were conducted upon varying three parameters: $d\%$, $|D_m|$ and $n\%$, where $d\%$ means the probability that an input tuple can match a tuple in D_m; $|D_m|$ is the cardinality of master data; $n\%$ is the noise rate, which represents the percentage of attributes with errors in the input tuples. When one parameter was varied, the others were fixed.

The comparisons were quantified with two measures, in tuple level and in attribute level, respectively.

Tuple level comparison. The tuple level recall is defined as:

$$recall_t = \#\text{ of corrected tuples } / \#\text{ of error tuples}$$

The results for varying $d\%$ and $|D_m|$ are shown in Fig. 2(a) and 2(b), respectively. Notably for the (Z, T_c) derived, its recall is close to the duplicate rate $d\%$, irrelevant to the datasets tested. Hence, in Fig. 2(a) and Fig. 2(b), the curve annotated by (Z, T_c)-ALL is used to represent the curves for all datasets. Moreover, $compT_c$ stands for the complete T_c.

In Fig. 2(a), we fixed $|D_m| = 20000$ while varying $d\%$ from 10% to 90%. When the master data covers more portions of the input tuples (from 10% to 90%), the recall increases (from 0.1 to 0.9). This tells us the following: (1) the efficacy of certain regions is sensitive to duplicate rates. Hence, the master data should be as complete as possible to cover the input tuples; and (2) the effect of certain regions (Z, T_c) derived by compCRegions is worse than complete T_c, as expected, since some valid pattern tuples were not selected by our heuristic method. However, when $d\%$ is increased, the recall via heuristic (Z, T_c) becomes close to that of the complete T_c, validating the effectiveness of compCRegions.

In Fig. 2(b), we fixed $d\% = 80\%$ while varying $|D_m|$ from 2500 to 20000. The recall of $(Z, compT_c)$ increases when increasing $|D_m|$, as expected. Observe that the curve for HOSP grows faster than the ones for TPC-H or DBLP. This is data-dependent, due to the fact that the number of hospitals in US is much smaller than the distinct entities in TPC-H sale records or DBLP publications. By increasing $|D_m|$, HOSP has a higher probability to cover more portions of the input tuples, which is reflected in Fig. 2(b).

This reveals that the completeness of master data is pivot. When D_m is assured consistent and complete [23], our algorithm can find certain regions with good recalls.

Attribute level comparison. For the attribute level quantification, we used a fine-grained measure F-measure [1]:

$$\text{F-measure} = 2(recall_a \cdot precision)/(recall_a + precision)$$
$$recall_a = \#\text{ of corrected attributes } / \#\text{ of error attributes}$$
$$precision = \#\text{ of corrected attributes}/\#\text{ of changed attributes}$$

We compared the F-measure values of adopting $(Z, compT_c)$ with IncRep [9]. We remark that the *precision* of compCRegions is always 100%, if the user assures the correctness of attributes in certain regions, as defined.

Figures 3(a), 3(b) and 3(c) show the results of F-measure comparisons when varying the parameters $d\%$, $|D_m|$ and $n\%$, respectively. Observe the following: (1) in most cases, when varying the three parameters described above, the F-measure of $(Z, compT_c)$ is better than that of IncRep, for all the datasets. This tells us that the certain regions and master data are more effective in guaranteeing the correctness of fixes than up-to-date techniques without leveraging master data, *e.g.,* IncRep. (2) Even when $|D_m|$ is small, $(Z, compT_c)$ can leverage D_m and perform reasonably well, if D_m can match a large part of certain regions of input tuples (*e.g.,* $d\%$ is 80%), as depicted in Fig. 3(b). (3) The certain regions derived by compCRegions is insensitive to the noise rate, whereas IncRep is sensitive, as verified in Fig. 3(c).

This set of experiments verified that the proposed method performed well in fixing errors in data while assuring its correctness. The results also validate that the two most important factors are $d\%$ and D_m. When $|D_m|$ is large and $d\%$ is high, certain fixes could be expected.

Exp-2: Efficiency and scalability. We evaluated the efficiency and scalability of both compCRegions in this set of experiments. Since real-life data is not flexible to vary the three parameters $|D_m|$, $|\Sigma|$ and K (the number of maximal cliques), we used TPC-H and RAND data.

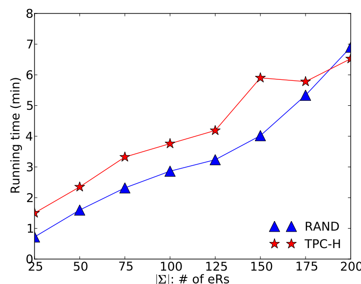
For TPC-H data, we have 55 eRs. When varying $|\Sigma|$, we randomly assigned these rules with pattern tuples so that we could always reach the number of eRs needed. For the RAND data, the default setting of $|R_m|$, $|R|$, $|D_m|$, $|\Sigma|$, $|\text{LHS}|$ and K are 40, 20, 50, 5000, 4 and 100, respectively. When varying one parameter, the others were fixed.

We only report here the impact of the three most important factors: $|\Sigma|$, $|D_m|$ and K. The results for the other parameters are omitted due to space limitations.

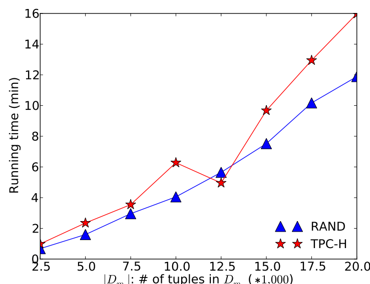
Figures 4(a), 4(b), and 4(c) show the running time of computing compCRegions when varying $|\Sigma|$, $|D_m|$ and K, respectively. From these figures we can see the following.

(1) In all the cases, compCRegions could be computed efficiently. Note that for all datasets, Algorithm compCRegions was executed only once, irrelevant to the size of input data to be fixed. Therefore, it could be considered as a precomputation. The time in minute level is thus acceptable.

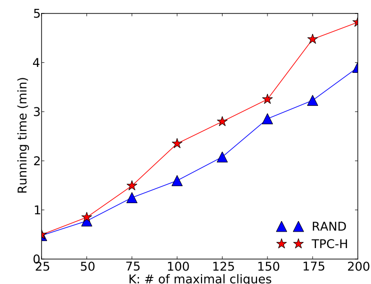
(2) Figures 4(a) and 4(c) show sub-linear scalability, and



(a) Running time *w.r.t.* $|\Sigma|$



(b) Running time *w.r.t.* $|D_m|$



(c) Running time *w.r.t.* K

Figure 4: The Scalability *w.r.t.* $|\Sigma|$, $|D_m|$ and K

better still, Figure 4(b) shows super-linear scalability. The trends in these results match our complexity analysis in Section 4.2, *i.e.,* in $O(|\Sigma|^2|D_m|\log|D_m| + K|\Sigma|^3 + K|\Sigma||D_m|\log|D_m|)$ time. This indicates that Algorithm compCRegions is scalable, and works well in practice.

Summary. From the experimental results we found the following. (1) The certain regions derived by our algorithm are effective and of high quality: at both the tuple level and the attribute level, our experiments have verified that the algorithm works well even with limited master data and high noise rate. (2) The completeness of master data (the amount of master data available) is critical to computing certain fixes. (3) Our algorithm scales well with the sizes of master data, editing rules and the number of certain regions.

6. Conclusion

We have proposed editing rules that, in contrast to constraints used in data cleaning, are able to find certain fixes by updating input tuples with master data. We have identified fundamental problems for deciding certain fixes and certain regions, and established their complexity bounds. We have also developed a graph-based algorithm for deriving certain regions from editing rules and master data. As verified by our experimental study, these yield a promising method for fixing data errors while ensuring its correctness.

We are extending Quaid [9], our working system for data cleaning, to support master data and to experiment with real-life data that Quaid processes. We are also exploring optimization methods to improve our derivation algorithm. Another topic is to develop methods for discovering editing rules from sample inputs and master data, along the same lines as discovering other data quality rules [7, 19].

Acknowledgments. Fan and Ma are supported in part by EPSRC E029213/1.

7. References

[1] F-measure. *http://en.wikipedia.org/wiki/F-measure*.

[2] T. Akutsu and F. Bao. Approximating minimum keys and optimal substructure screens. In *COCOON*, 1996.

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

[4] 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.

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

[6] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, 2003.

[7] F. Chiang and R. Miller. Discovering data quality rules. In *VLDB*, 2008.

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

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

[10] W. W. Eckerson. Data quality and the bottom line: Achieving business success through a commitment to high quality data. The Data Warehousing Institute, 2002.

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

[12] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(2), 2008.

[13] W. Fan, X. Jia, J. Li, and S. Ma. Reasoning about record matching rules. *PVLDB*, 2(1), 2009.

[14] T. Faruquie et al. Data cleansing as a transient service. In *ICDE*, 2010.

[15] I. Fellegi and D. Holt. A systematic approach to automatic edit and imputation. *J. American Statistical Association*, 71(353):17–35, 1976.

[16] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[17] Gartner. Forecast: Data quality tools, worldwide, 2006-2011. Technical report, Gartner, 2007.

[18] P. Giles. A model for generalized edit and imputation of survey data. *The Canadian J. of Statistics*, 16:57–73, 1988.

[19] L. Golab, H. J. Karloff, F. Korn, D. Srivastava, and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. In *VLDB*, 2008.

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

[21] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, 1988.

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

[23] D. Loshin. *Master Data Management*. Knowledge Integrity, Inc., 2009.

[24] K. Makino and T. Uno. New algorithms for enumerating all maximal cliques. In *SWAT*, 2004.

[25] T. Redman. The impact of poor data quality on the typical enterprise. *Commun. ACM*, 41(2):79–82, 1998.

[26] G. Sauter, B. Mathews, and E. Ostic. Information service patterns, part 3: Data cleansing pattern. IBM, 2007.

[27] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

[28] V. V. Vazirani. *Approximation Algorithms*. Springer, 2003.

[29] J. Widom and S. Ceri. *Active database systems: triggers and rules for advanced database processing*. Morgan Kaufmann, 1996.

[30] J. Wijsen. Database repairing using updates. *TODS*, 30(3):722–768, 2005.

APPENDIX: Proofs and Algorithms

Proof of Theorem 3.1

(I) We first show that the problem is in coNP, by providing an NP algorithm for its complement, *i.e.,* the algorithm returns 'yes' iff (Σ, D_m) is *not* consistent relative to (Z, T_c).

We define dom to be the set of all constants appearing in D_m and Σ, and introduce a variable v representing a distinct constant not in dom. It suffices to consider R tuples t such that for each attribute A of R, $t[A]$ is either a constant in dom or the variable v.

The NP algorithm works as follows:
(a) guess a tuple t_c in T_c;
(b) guess an R tuple t such that for each attribute A of R, $t[A]$ is a constant in dom or the variable v; and
(c) If $t \rightleftharpoons t_c$, then check whether (Σ, D_m) is consistent relative to $(Z, \{t[Z]\})$. If not, the algorithm returns 'yes'.

By Theorem 3.5 (see its proof below), checking whether (Σ, D_m) is consistent relative to $(Z, \{t[Z]\})$ is in PTIME since $t[Z]$ consists of only constants. Thus, the algorithm is in NP.

(II) We next show that the problem is coNP-hard, by reduction from the 3SAT problem to its complement.

An instance ϕ of 3SAT is of the form $C_1 \wedge \cdots \wedge C_n$, where C_i is a disjunction of at most three literals. The 3SAT problem is to determine whether ϕ is satisfiable. It is known to be NP-complete (cf. [16]).

Given an instance ϕ of the 3SAT problem, we construct the following: (a) schemas R and R_m, (b) a master relation D_m of schema R_m, (c) a pattern tableau T_c consists of a single tuple t_c for a set Z of attributes of schema R, and (d) a set Σ of eRs, such that (Σ, D_m) is consistent relative to (Z, T_c) if and only if the instance ϕ is *not* satisfiable. □

Proof of Theorem 3.2

(I) We show that the problem is in coNP by giving a coNP algorithm. The algorithm is the same as the one developed in the proof of Theorem 3.1, except that in the last step, it uses a variation of the PTIME algorithm given in Theorem 3.5 such that it only returns 'yes' if both the set S is empty, and if the tuple t is wildcard free.

(II) We show that the problem is coNP-hard by reduction from the 3SAT problem to its complement. Given an instance ϕ of the 3SAT problem, we construct schemas R and R_m, a master relation D_m of R_m, a set $Z \cup \{B\}$ of attributes of R, and a set Σ of eRs such that (Z, T_c) is a certain region for (Σ, D_m) if and only if ϕ is *not* satisfiable. □

Proof of Theorem 3.3

(I) The Z-validating problem is NP-complete.

(1) We show the problem is in NP, by providing an NP algorithm that, given Z, returns 'yes' iff there exists a non-empty pattern tableau T_c such that (Z, T_c) is a certain region for (Σ, D_m). Observe that if so, there must exist a tuple t_c consisting of only constants such that $(Z, \{t_c\})$ is a certain region for (Σ, D_m). Thus it suffices to consider pattern tuples consisting of constants only.

Define active domain dom and variable v as in the proof of Theorem 3.1. The NP algorithm works as follows.
(a) Guess a tuple t_c such that for each attribute $A \in Z$, $t_c[A]$ is either a constant in dom or the variable v.
(b) If $(Z, \{t_c\})$ is a certain region for (Σ, D_m), then the algorithm returns 'yes'; and it returns 'no', otherwise.

Similar to the proof of Theorem 3.1, it is easy to see that the algorithm is in NP and is correct.

(2) We show the problem is NP-hard by reduction from 3SAT. Given an instance ϕ of 3SAT, we construct schemas R and R_m, a master relation D_m of R_m, a set Z of attributes of R, and a set Σ of eRs such that Z is valid iff ϕ is satisfiable.

(II) The Z-counting problem is #P-complete.

The reduction above is *parsimonious*. That is, the number of satisfiable truth assignments for the 3SAT instance is equal to the number of pattern tuples t_c such that $(Z, \{t_c\})$ is a certain region for (Σ, D_m).

The #3SAT problem, which is the counting version of the 3SAT problem, is #P-complete [16, 27]. From this it follows that the Z-counting problem is also #P-complete. □

Proof of Theorem 3.4

(I) The Z-*minimum* problem is NP-complete.

(1) We show the problem is in NP by giving an NP algorithm. Consider a set Σ of eRs over schemas (R, R_m), and a positive integer $K \leq |R|$. The algorithm works as follows.
(a) Guess a set Z of attributes in R such that $|Z| \leq K$.
(b) Guess a pattern tuple t_c, and check whether $(Z, t_c[Z])$ is a certain region for (Σ, D_m).
(c) If so, it returns 'yes'; and it returns 'no' otherwise.

The correctness of the NP algorithm can be verified along the same lines as the proof of Theorem 3.3.

(2) We show that the problem is NP-hard by reduction from the minimum key problem, which is NP-complete [2].

(II) We show that the Z-*minimum* problem cannot be approximated within a factor of $c \log n$ in polynomial time for any constant c unless NP $\subseteq$ DTIME $(n^{polylog(n)})$. This can be verified by an approximation preserving reduction [28] from the minimum set cover problem, along the same lines as the one for the minimum key problem for functional dependencies (FDs) [2]. □

Proof of Theorem 3.5

Consider (Z, T_c) and (Σ, D_m). Assume *w.l.o.g.* that there is a single tuple $t_c \in T_c$. When there are multiple tuples in T_c, we can test them one by one by the same algorithms below.

Statements (a) and (b). We first show that if the consistency problem (resp. the coverage problem) is in PTIME for case (b), then it is in PTIME for case (a). We then show that both problems are in PTIME for case (b).

(I) We first show that it suffices to consider case (b) only.

We define active domain dom and variable v as in the proof of Theorem 3.1. It suffices to consider R tuples t such that for each attribute A of R, $t[A]$ is either a constant in dom or the variable v. Since we only need to consider attributes that appear in Σ, there are at most $O(|\text{dom}|^{|\Sigma|})$ tuples of R to be considered, a polynomial when fixing Σ.

For such an R tuple t, if $t \not\rightleftharpoons t_c \in T_c$, there exists a unique fix, but no certain fix, for t. If $t \rightleftharpoons t_c \in T_c$, it is easy to see that there is a unique fix (resp. certain fix) for t by (Σ, D_m) *w.r.t.* $(Z, \{t_c\})$ if and only if (Σ, D_m) is consistent relative to $(Z, \{t[Z]\})$ (resp. $(Z, \{t[Z]\})$ is a certain region for (Σ, D_m)). From this it follows that we only need to consider case (b).

(II) We show that the consistency problem for case (b) is in PTIME, by giving a PTIME algorithm such that (Σ, D_m) is consistent relative to (Z, T_c) iff the algorithm returns 'yes'.

(III) We next show that the coverage problem for case (b) is in PTIME. Indeed, the PTIME algorithm developed above can be applied here, but it only returns 'yes' at step (c) if when the set S is empty and the tuple t only consists of constants.

This completes the proof for statements (a) and (b).

Statement (c). We show that the consistency and coverage problems are in PTIME for direct fixes, one by one as follows.

(I) We first show how to check the relative consistency via SQL queries, which yields a PTIME algorithm for the problem.

Given a set Z of certain attributes, let Σ_Z be the set of eRs φ in Σ such that $\text{LHS}(R,\varphi) \subseteq Z$, but $\text{RHS}(R,\varphi) \notin Z$.

We first define an SQL query Q_φ for an eR $\varphi = ((X, X_m) \to (B, B_m),\ t_p[X_p])$ in Σ_Z, as follows.

1. **select distinct** (X_m, B_m) **as** (X, B)
2. **from** $\mathsf{R_m}$
3. **where** $\mathsf{R_m.X_p} \approx t_p[X_p]$ **and** $\mathsf{R_m.X_m} \rightleftharpoons t_c[X]$

We use $Q_\varphi(X, B)$ and $Q_\varphi(X)$ to denote the result of the SQL query projected on attributes $X \cup \{B\}$ and X, respectively.

We then define an SQL query $Q_{(\varphi_1,\varphi_2)}$ for two eRs $\varphi_1 = ((X_1X, X_{m_1}X_m) \to (B, B_{m_1}),\ t_{p_1}[X_{p_1}])$ and $\varphi_2 = ((X_2X, X_{m_2}X'_m) \to (B, B_{m_2}),\ t_{p_2}[X_{p_2}])$ such that $X_1 \cap X_2$ is empty and $|X| = |X_m| = |X'_m|$. Here X may be empty.

1. **select** $R_1.X_1$,$R_1.X$,$R_2.X_2$
2. **from** $Q_{\varphi_1}(X_1X, B)$ **as** R_1, $Q_{\varphi_2}(X_2X, B)$ **as** R_2
3. **where** $R_1.X = R_2.X$ **and** $R_1.B \neq R_2.B$

For two eRs $\varphi_1 = ((X_1, X_{m_1}) \to (B_1, B_{m_1}),\ t_{p_1}[X_{p_1}])$ and $\varphi_2 = ((X_2, X_{m_2}) \to (B_2, B_{m_2}),\ t_{p_2}[X_{p_2}])$ such that $B_1 \neq B_2$, we define SQL query $Q_{(\varphi_1,\varphi_2)}$ that always returns $\emptyset$.

Observe that (Σ, D_m) is consistent relative to $(Z, \{t_c\})$ if and only if for all eRs φ_1 and φ_2 in Σ_Z, the queries $Q_{(\varphi_1,\varphi_2)}$ return an empty result. The SQL query $Q_{(\varphi_1,\varphi_2)}$ is obviously in PTIME, and hence, the consistency problem is in PTIME for direct fixes.

(II) For the coverage problem, observe that (Z, T_c) is a certain region for (Σ, D_m) if and only if:

1. (Σ, D_m) is consistent relative to (Z, T_c), and
2. for each $B \in R \setminus Z$, there exists an eR $\varphi = ((X, X_m) \to (B, B_m),\ t_p[X_p])$ in Σ such that (a) $X \in Z$ and $t_c[X]$ consists of only constants, (b) $t_p[X_p] \approx t_c[X_p]$, and (c) there is a master tuple $t_m \in D_m$ with $t_m[X_m] = t_c[X]$.

Both conditions are checkable in PTIME, and hence, so is the coverage problem. □

Proof of Corollary 3.6

(I) For the Z-validating problem and the Z-counting problem, a close look at the proofs in Theorems 3.3 reveals that those proofs also work for this special case.

(II) The NP-hardness of the Z-minimum problem is shown by reduction from the minimum node cover problem, which is NP-complete [16].

A node cover in a graph $G(V, E)$ is a subset $V' \subseteq V$ such that for each edge (u, v) of the graph, at least one of u and v belongs to the set V'. Given a graph $G(V, E)$ and a positive integer $K \leq |V|$, the node cover problem asks whether there exists a node cover V' in G having $|V'| \leq K$.

Consider an instance $\text{VC} = (G(V, E), K)$ of the node cover problem, where $V = \{v_1, \ldots, v_{|V|}\}$ and $E = \{e_1, \ldots, e_{|E|}\}$. We construct schemas (R, R_m) and a set Σ of eRs such that there is a solution to VC iff there is a solution to the constructed minimum Z instance.

Algorithm validateW

Input: W $= (Z_1, Z_2 \parallel Z_m)$, a set Σ of eRs on schemas (R, R_m), and a reduced master relation D_s of R_m.

Output: true if W is valid, or false otherwise.

1. $t := \emptyset$; /* t is an R tuple */
2. **for** each master tuple t_m in D_s **do**
3. $\quad t[Z_2] := t_m[Z_m]$; $t[R \setminus Z_2] := (_, \ldots, _)$;
4. $\quad$**for** each φ in Σ having $X \subseteq Z_2$ and $B \notin (Z_1Z_2)$ **do**
   $\quad$/* Here $\varphi = ((X, X_m) \to (B, B_m),\ t_p[X_p])$ */
5. $\quad\quad$**if** $t[X_p] \approx t_p[X_p]$ **and** $t'_m[X_m] = t[X]$ $(t'_m \in D_s)$
6. $\quad\quad$**then** $t[B] := t'_m[B_m]$;
7. $\quad$**if** $t[R \setminus Z_1]$ contains only constants **then return** true;
8. **return** false.

Figure 5: Algorithm validateW

(III) The approximation hardness of the Z-minimum problem is shown by an approximation preserving reduction [28] from the minimum set cover problem, a minor modification of the one for the minimum key problem [2]. □

Proof of Corollary 3.7

The reductions given in the proofs of Theorem 3.1 and Theorem 3.2 both use a *fixed* master relation, which have five attributes and three master tuples. As a result, the coNP lower bounds remain intact for the consistency and coverage problems when the master relation D_m is fixed. □

Proof of Proposition 4.1

Consider a clique $\mathcal{C} = \{v_1, \ldots, v_k\}$ in G, where for each $i \in [1, k]$, $v_i = (\varphi_i, t_{m_{j_i}})$. Let $\Sigma_\mathcal{C}$ be the set of eRs in the clique $\mathcal{C}$. By the definition of the compatible graph, we have the following: (a) $\text{LHS}(\Sigma_\mathcal{C}) \cap \text{RHS}(\Sigma_\mathcal{C}) = \emptyset$; and (b) $|\text{RHS}(\Sigma_\mathcal{C})| = |\mathcal{C}| = k$, *i.e.*, the number of attributes in $\text{RHS}(\Sigma_\mathcal{C})$ is equal to the number of nodes in $\mathcal{C}$.

Let $Z = R \setminus \text{RHS}(\Sigma_\mathcal{C})$, and t_c be a tuple with attributes in Z such that (a) $t_c[\text{LHS}(\Sigma_\mathcal{C})] = t_{j_1} \bowtie \ldots \bowtie t_{j_k}[\text{LHS}(\Sigma_\mathcal{C})]$, where for each $i \in [1, k]$, $t_{j_i}[X_iB_i] = t_{m_{j_i}}[X_{m_i}B_{m_i}]$, and (b) $t_c[A] =$ '_' for all the remaining attributes $A \in Z$.

Since there are no conflict tuples in the compatible graph, for all R tuples $t \rightleftharpoons t_c$, the eR-tuple pairs in the clique $\mathcal{C}$ guarantee that the tuple t has a certain fix.

Hence, we can derive a certain region for (Σ, D_m) from each clique in the compatible graph G. □

Proof of Theorem 4.2

By reduction from the 3SAT problem to the Z-validating problem, where (a) for each attribute $B \in (R \setminus Z)$, there is exactly one eR φ with $\text{LHS}(\varphi) = B$, and (b) the eRs form a clique in the compressed compatible graph. The reduction in the proof of Theorem 3.3 is such a reduction. From this it follows that the Z-validating problem for a clique in a compressed graph G^c is NP-complete. □

Proof of Proposition 4.3

We show this by giving algorithm validateW, shown in Fig. 5. We first show that the algorithm runs in $O(|\Sigma||D_s|\log|D_s|)$ time, and then verify its correctness.

(I) We first show the algorithm is in $O(|\Sigma||D_s|\log|D_s|)$ time.

1. For each eR $\varphi \in \Sigma$ such that $\text{LHS}(\varphi) \in Z_2$ and $\text{RHS}(\varphi) \notin (Z_1Z_2)$, we first build a *hash* index based on $\text{LHS}_m(\varphi)$ for master tuples in D_s. This takes $O(|\Sigma||D_s|\log|D_s|)$ time, and this part is not shown in the pseudo-code.

2. There are at most $O(|D_s||\Sigma|)$ loops (lines 2–7). Each innermost loop takes $O(\log |D_s|)$ time (lines 5–6). Hence in total it takes $O(|\Sigma||D_s|\log |D_s|)$ time.

Putting these together, it is easy to see that the algorithm indeed runs in $O(|\Sigma||D_s|\log |D_s|)$ time.

(II) We now show the correctness of the algorithm.

That is, given $W = (Z_1, Z_2 \parallel Z_m)$, there is a non-empty pattern tableau T_c such that (Z_1Z_2, T_c) is a certain region for (Σ, D_m) iff algorithm validateW returns true. Recall that we only focus on direct fixes (Section 3).

First, assume that the algorithm returns 'yes'. Then there exists a master tuple t_m that makes $t[R \setminus Z_1]$ contain only constants. It suffices to show that $(Z, t_m[Z_m])$ is a certain region. Indeed, this is because (a) the algorithm guarantees that for all tuples t of R, if $t[Z] = t_m[Z_m]$, then $t[A]$ is a constant for all attributes $A \in (R \setminus Z_1Z_2)$ (line 6), and (b) there are no conflict tuples in D_s.

Conversely, assume that W is valid. That is, there exists a master tuple t_m in D_s such that $(Z_1Z_2, \{t_c\})$ is a certain region for (Σ, D_s), where $t_c[Z_1]$ consists of '_' only and $t_c[Z_2] = t_m[Z_m]$.

For the master tuple t_m, $t[R \setminus Z_1]$ must contain only constants, and the algorithm returns true. □

Details of Algorithm compCRegions

Algorithm compCRegions is presented in Fig. 6. It first computes a *reduced* master relation D_s (line 1), and builds the compressed compatible graph G^c of (Σ, D_s) (line 2). It then invokes Procedure findCliques to find up to K maximal cliques in G^c (line 3). These cliques are converted into certain regions by Procedure cvrtClique (line 5). Finally, it constructs M by merging certain regions with the same Z (lines 6-7). Here M is guaranteed nonempty since (a) every graph has at least one maximal clique, and (b) cvrtClique finds a certain region from each clique (see below).

Procedure findCliques is presented following the algorithm given in [21] for the ease of understanding. However, we used the algorithm in [24] in the experiments. These algorithms output a maximal clique in $O(|V||E|)$ time for a graph $G(V, E)$, in a lexicographical order of the nodes. Procedure findCliques first generates a total order for eRs in Σ (lines 1–3). Then it recursively generates K maximal cliques (lines 4–12). This part is a simulation of the algorithm in [21], which outputs maximal independent sets. This takes $O(K|\Sigma|^3)$ time in total. The correctness of Procedure findCliques is ensured by that of the algorithm in [21].

Procedure findCliques makes use of the methods of [21, 24] to find maximal cliques. Those methods have proven effective and efficient in practice. Indeed, the algorithm of [24] can find about 100,000 maximal cliques per second on sparse graphs (http://research.nii.ac.jp/∼uno/code/mace.htm).

Given a clique $\mathcal{C}$, Procedure cvrtClique derives a set of certain regions, using the heuristic given in Section 4.1. It first extracts Z_2 and Z_m from the set $\Sigma_{\mathcal{C}}$ of eRs in $\mathcal{C}$ (line 1). For each master tuple t_m, it then identifies a set Z_1Z_2 of attributes and a pattern $t[Z_1Z_2]$ such that $(Z_1Z_2, \{t[Z_1Z_2]\})$ forms a certain region (lines 2–7). The rational behind this includes: (a) no conflict tuples are in D_s, and (b) for any $B \in (R \setminus Z_2)$, $t[B]$ is a constant taken from D_s (line 7).

Example 7.1: Consider the master relation D_m in Fig. 1(b), and $\Sigma' = \{\varphi_{(\mathsf{FN},2)}, \varphi_{(\mathsf{LN},2)}, \varphi_{(\mathsf{AC},1)}, \varphi_{(\mathsf{str},1)}, \varphi_{(\mathsf{city},1)}, \varphi_4\}$ consisting of eRs derived from φ_1, φ_2 and φ_4 of Example 2.1 by, *e.g.,* instantiating B_1 with AC, str and city.

The algorithm first builds a compressed graph $G^c(V^c, E^c)$ such that $V^c = \Sigma'$, and there is an edge in E^c for all node pairs except for node pairs $(\varphi_{(\mathsf{AC},1)}, \varphi_4)$ and $(\varphi_{(\mathsf{city},1)}, \varphi_4)$.

For $K = 2$, findCliques finds two cliques $\mathcal{C}_1 = \{\varphi_{(\mathsf{FN},2)}, \varphi_{(\mathsf{LN},2)}, \varphi_{(\mathsf{AC},1)}, \varphi_{(\mathsf{str},1)}, \varphi_{(\mathsf{city},1)}\}$ and $\mathcal{C}_2 = \{\varphi_{(\mathsf{FN},2)}, \varphi_{(\mathsf{LN},2)}, \varphi_{(\mathsf{str},1)}, \varphi_4\}$, by checking eRs following their order in Σ'.

The algorithm returns two certain regions (Z_1, T_{c_1}) and (Z_2, T_{c_2}), where $Z_1 = (\mathsf{zip}, \mathsf{phn}, \mathsf{type}, \mathsf{item})$ with $T_{c_1} = \{t_{1,1}, t_{1,2}\}$ and $Z_2 = (\mathsf{zip}, \mathsf{phn}, \mathsf{AC}, \mathsf{type}, \mathsf{item})$ with $T_{c_2} = \{t_{2,1}, t_{2,2}\}$. For each $i \in [1, 2]$, (a) $t_{1,i}[\mathsf{type}, \mathsf{item}] = (_, _)$, $t_{1,i}[\mathsf{zip}, \mathsf{phn}] = s_i[\mathsf{zip}, \mathsf{Mphn}]$; and (b) $t_{2,i}[\mathsf{type}, \mathsf{item}] = (_, _)$, $t_{2,i}[\mathsf{zip}, \mathsf{phn}, \mathsf{AC}] = s_i[\mathsf{zip}, \mathsf{Mphn}, \mathsf{AC}]$ for s_i of Fig. 1(b). □

Correctness and complexity. The algorithm guarantees to return a nonempty set M of certain regions, by Propositions 4.1 and 4.3. It is in $O(|\Sigma|^2|D_m|\log |D_m| + K|\Sigma|^3 + K|\Sigma||D_m|\log |D_m|)$ time: it takes $O(|\Sigma|^2|D_m|\log |D_m|)$ time to build a compressed compatible graph (lines 1-2), $O(K|\Sigma|^3)$ time to find cliques (line 3), and $O(K|\Sigma||D_m|$

Algorithm compCRegions

Input: A number K, a set Σ of eRs defined on (R, R_m), and a master relation D_m of R_m.
Output: An array M of regions (Z, T_c).

1. Compute D_s from D_m by removing conflict tuples;
2. Build the compressed compatible graph G^c for (Σ, D_s);
3. $M := \emptyset$; $\Gamma :=$ findCliques(K, G^c);
4. **for** each clique $\mathcal{C}$ in Γ **do**
5. $S :=$ cvrtClique $(\mathcal{C}, \Sigma, D_s)$;
6. **for** each (Z, t_c) in S **do**
7. $M[Z] := (Z, M[Z].T_c \cup \{t_c\})$;
8. **return** M.

Procedure findCliques

Input: Number K, a compressed compatible graph $G^c(V^c, E^c)$.
Output: A set Γ of at most K maximal cliques.

1. **let** $H[1..h]$ be the list of attributes in $\text{RHS}(V^c)$;
   /* Sorted in decreasing order *w.r.t.* attribute confidence */
2. **let** $\Sigma[i]$ $(i \in [1, h])$ be the list of eRs φ with $\text{RHS}(\varphi) = H[i]$;
   /* Partition eRs (nodes) *w.r.t.* their RHS attributes */
3. Let $\mathcal{O}$ be a total order on V^c, and $\mathcal{O}(\varphi) < \mathcal{O}(\varphi')$ for $\varphi \in \Sigma[i]$, $\varphi' \in \Sigma[j]$ ($i < j$);
4. Greedily find a maximal clique $\mathcal{C}$;
5. $\Gamma := \emptyset$; Que.push $(\mathcal{C})$; /* Que is a priority queue */
6. **while** $(|\Gamma| < K$ **and** there are changes in Que) **do**
7. $\mathcal{C} :=$ Que.pop (); $\Gamma := \Gamma \cup \{\mathcal{C}\}$;
8. **for** all $\varphi_2 \in \overline{\mathcal{N}}(\varphi_1)$ having $\varphi_1 \in \mathcal{C}$ **and** $\mathcal{O}(\varphi_1) < \mathcal{O}(\varphi_2)$ **do**
   /* $\overline{\mathcal{N}}(v)$ is the non-neighbors of node $v \in V^c$ */
9. $V_j := \{ \varphi \mid \varphi \in V^c$ **and** $\mathcal{O}(\varphi) \le \mathcal{O}(\varphi_2)\}$;
10. $\mathcal{C}_j := (\mathcal{C} \cap V_j \setminus \overline{\mathcal{N}}(\varphi_2)) \cup \{\varphi_2\}$;
11. **if** $\mathcal{C}_j$ is a maximal clique in the subgraph $G^c[V_j]$
12. **then** Enlarge $\mathcal{C}_j$ to a maximal clique of G^c; Que.push $(\mathcal{C}_j)$;
13. **return** Γ.

Procedure cvrtClique

Input: A clique $\mathcal{C}$ in the graph G^c, a set Σ of eRs on schemas (R, R_m), and a reduced master relation D_s of R_m.
Output: A set S of (Z, t_c) pairs.

1. $S := \emptyset$; $Z_2 := \text{LHS}(\Sigma_{\mathcal{C}})$; $Z_m := \text{LHS}_m(\Sigma_{\mathcal{C}})$;
2. **for** each master tuple t_m in D_s **do**
3. $t[Z_2] := t_m[Z_m]$; $t[R \setminus Z_2] := (_, \ldots, _)$; /* t is an R tuple */
4. **for** each φ in Σ with $X \subseteq Z_2$ and $B \in \text{RHS}(\Sigma_{\mathcal{C}})$ **do**
   /* Here $\varphi = ((X, X_m) \to (B, B_m), t_p[X_p])$ */
5. **if** $t[X_p] \approx t_p[X_p]$ **and** $t'_m[X_m] = t[X]$ $(t'_m \in D_s)$
6. **then** $t[B] := t'_m[B_m]$;
7. $S := S \cup \{(Z_1Z_2, t[Z_1Z_2])\}$; /* $t[Z_1]$ contains exactly '_' */
8. **return** S;

Figure 6: A Graph-based Algorithm

$\log|D_m|$) time to derive certain regions from the cliques (lines 4-7). In practice, $|\Sigma|$ and K are often small. We verify its effectiveness and efficiency in Section 5. □

Additional Materials for the Experimental Study

We present more details on the datasets, the eRs that we designed for each data set, and the algorithm IncRep.

Datasets and editing rules.

(1) HOSP *data.* The data is maintained by the U.S. Department of Health & Human Services, and comes from hospitals that have agreed to submit quality information for Hospital Compare to make it public.

There are three tables: HOSP, HOSP_MSR_XWLK, and STATE_MSR_AVG, where (a) HOSP records the hospital information, including id (provider number, its ID), hName (hospital name), phn (phone number), ST (state), zip (ZIP code), and address; (b) HOSP_MSR_XWLK records the score of each measurement on each hospital in HOSP, *e.g.,* mName (measure name), mCode (measure code), and Score (the score of the measurement for this hospital); and (c) STATE_MSR_AVG records the average score of each measurement on hospitals in all US states, *e.g.,* ST (state), mName (measure name), sAvg (state average, the average score of all the hospitals in this state).

We created a big table by joining the three tables with *natural join*, among which we chose 19 attributes as the schema of both the master relation R_m and the relation R. We designed 37 eRs in total for the HOSP data. Five important ones are listed as follows.

φ_1 : ((zip, zip) → (ST, ST), t_{p1}[zip] = $(\overline{\text{nil}})$);
φ_2 : ((phn, phn) → (zip, zip), t_{p2}[phn] = $(\overline{\text{nil}})$);
φ_3 : (((mCode, ST), (mCode, ST)) → (sAvg,sAvg), $t_{p3} = ()$);
φ_4 : (((id, mCode), (id, mCode)) → (Score,Score), $t_{p4} = ()$);
φ_5 : ((id, id) → (hName,hName), $t_{p5} = ()$).

(2) DBLP *data.* The DBLP service is well known for providing bibliographic information on major computer science journals and conferences. We first transformed the XML-formatted data into relational data. We then created a big table by joining the *inproceedings* data (conference papers) with the *proceedings* data (conferences) on the crossref attribute (a foreign key). Besides, we also included the homepage info (hp) for authors, which was joined by the homepage entries in the DBLP data.

From the big table, we chose 12 attributes as the schema of both the master relation R_m and the relation R, including ptitle (paper title), a1 (the first author), a2 (the second author), hp1 (the homepage of a1), hp2 (the homepage of a2), btitle (book title), and publisher.

We designed 16 eRs for the DBLP data, shown below.

φ_1 : ((a1, a1) → (hp1, hp1), t_{p1}[a1] = $(\overline{\text{nil}})$);
φ_2 : ((a2, a1) → (hp2, hp1), t_{p2}[a2] = $(\overline{\text{nil}})$);
φ_3 : ((a2, a2) → (hp2, hp2), t_{p3}[2] = $(\overline{\text{nil}})$);
φ_4 : ((a1, a2) → (hp1,hp2), t_{p4}[a2] = $(\overline{\text{nil}})$);
φ_5 : (((type, btitle, year), (type, btitle, year)) → (A, A), t_{p5}[type] = ('*conference*'));
φ_6 : (((type, crossref), (type, crossref) → (B, B), t_{p6}[type] = ('*conference*'));
φ_7 : (((type, a1, a2, title, pages), (type, a1, a2, title, pages)) → (C, C), t_{p7}[type] = ('*conference*')).

Here the attributes A, B and C range over the sets {isbn, publisher, crossref}, {btitle, year, isbn, publisher} and {isbn, publisher, year, btitle, crossref}, respectively.

Observe that in eRs φ_2 and φ_4, the attributes are mapped to different attributes. That is, even when the master relation R_m and the relation R share the same schema, some eRs still could not be syntactically expressed as CFDs, not to mention their semantics.

(3) TPC-H *data.* The TPC Benchmark^{TM}H (TPC-H) is a benchmark for decision support systems. We created a big table by joining eight tables based on their foreign keys. The schema of both the master relation R_m and the relation R is the same as the one of the big table consisting of 58 attributes, *e.g.,* okey (order key), pkey (part key), num (line number), tprice (order total price), ckey (customer key), and skey (supplier key).

We designed 55 eRs all with empty pattern tuples. Since the data was the result of joining eight tables on foreign keys, we designed all eRs based on the foreign key attributes. We selectively report four eRs in the following.

φ_1 : (((okey, pkey), (okey, pkey)) → (num, num), $t_{p1} = ()$);
φ_2 : ((okey, okey) → (tprice, tprice), $t_{p2} = ()$);
φ_3 : ((ckey, ckey) → (name, name), $t_{p3} = ()$);
φ_4 : ((skey, skey) → (address, address), $t_{p4} = ()$).

Adding noise. In the attribute level experiments, we added noises to the three datasets. The noise rate is defined as the ratio of (# of dirty attributes)/(# of total attributes). For each attribute that the noise was introduced, we kept the edit distance between the dirty value and the clean value less or equal than 3.

Algorithm IncRep. We implemented the incremental repairing algorithm IncRep in [9] to compare with the method proposed in this paper. Below we simply illustrate the algorithm IncRep (please see [9] for more details).

Taking as input a clean database D, a set ΔD of (possibly dirty) updates, a set Σ of CFDs, and an ordering O on ΔD, it works as follows. It first initializes the repair *Repr* with the current clean database D. It then invokes a procedure called TupleResolve to repair each tuple t in ΔD according to the given order O, and adds the local repair $Repr_t$ of t to *Repr* before moving to the next tuple. Once all the tuples in ΔD are processed, the final repair is returned as *Repr*. The key characteristics of IncRep are (i) the repair grows at each step, providing in this way more information that can be used when cleaning the next tuple, and (ii) the data in D is not modified since it is assumed to be clean.

For IncRep, we adopted the cost model presented in [9] based on the edit distance. For two values in the same domain, the cost model is defined as:

cost $(v, v') = w(t, A)\cdot$ dis $(v, v')/\max(|v|, |v'|)$, where $w(t, A)$ is a *weight* in the range $[0, 1]$ associated with each attribute A of each tuple t in the dataset D.

For the cost of changing a tuple from t to t', we used the sum of cost $(t[A], t[A'])$ for each A in the schema of R, *i.e.,* cost $(t, t') = \Sigma_{A\in R}$ cost $(t[A], t'[A])$.

More specifically, in these experiments, we designed the CFDs based on the eRs that we have. Since the D_m and R have the same schemas in all datasets, we can easily design the corresponding CFDs from the eRs.

During the repair process, we enumerated one R tuple a time as ΔD. We then enlarged D to *Repr*, and repeated the process until all tuples were repaired. Because each time there was only one tuple in ΔD, we did not need to deal with the ordering O problem in IncRep.