Improving Data Quality: Consistency and Accuracy

Gao Cong[1] Wenfei Fan[2,3] Floris Geerts[2,4,5] Xibei Jia[2] Shuai Ma[2]
[1]Microsoft Research Asia [2]University of Edinburgh [4]Hasselt University
[3]Bell Laboratories [5]transnational Univ. Limburg
gaocong@microsoft.com {wenfei@inf, fgeerts@inf, x.jia@sms, sma1@inf}.ed.ac.uk

Abstract

Two central criteria for data quality are consistency and accuracy. Inconsistencies and errors in a database often emerge as violations of integrity constraints. Given a dirty database D, one needs automated methods to make it *consistent*, *i.e.*, find a repair D' that satisfies the constraints and "minimally" differs from D. Equally important is to ensure that the automatically-generated repair D' is *accurate*, or makes sense, *i.e.*, D' differs from the "correct" data within a predefined bound. This paper studies effective methods for improving both data consistency and accuracy. We employ a class of *conditional functional dependencies* (CFDs) proposed in [6] to specify the consistency of the data, which are able to capture inconsistencies and errors beyond what their traditional counterparts can catch. To improve the consistency of the data, we propose two algorithms: one for automatically computing a repair D' that satisfies a given set of CFDs, and the other for incrementally finding a repair in response to updates to a clean database. We show that both problems are intractable. Although our algorithms are necessarily heuristic, we experimentally verify that the methods are effective and efficient. Moreover, we develop a statistical method that guarantees that the repairs found by the algorithms are *accurate above a predefined rate* without incurring excessive user interaction.

1. Introduction

Real-world data is often dirty, *i.e.*, containing inconsistencies, conflicts and errors. A recent survey [31] reveals that enterprises typically expect data error rates of approximately 1%–5%. The consequences of dirty data may be severe. For example, it is reported [12] that wrong price data in retail databases alone costs US consumers $2.5 billion annually. With this comes the need for effective methods to improve the quality of data, or to clean data.

Inconsistencies, errors and conflicts in a database often emerge as violations of integrity constraints [2, 29]. A central problem for data cleaning is how to make the data *consistent*: given a dirty database D, we want to minimally *edit* the data in D such that it satisfies certain constraints. In other words, we want to find a *repair* of D, *i.e.*, a database Repr that satisfies the constraints and is as close to the original D as possible. This is the data cleaning approach that US national statistical agencies, among others, have been practicing for decades [13, 35]. Manually editing the data is unrealistic when the database D is large. Indeed, manually cleaning a set of census data could easily take months by dozens of clerks [35]. This highlights the need for automated methods to find a repair of D.

VLDB '07, September 23-28, 2007, Vienna, Austria.

In practice one also wants *incremental* methods to improve the consistency of the data: given a clean database D that satisfies a set Σ of constraints, and updates ΔD on the database D, it is to find a repair ΔD_{Repr} of ΔD such that $D \oplus \Delta D_{\mathsf{Repr}}$ satisfies Σ (we use $\oplus$ to denote the application of updates). This is often advantageous to *batch* methods that compute a repair Repr of $D \oplus \Delta D$ starting from scratch instead of finding a typically much smaller ΔD_{Repr}.

Another important problem for data cleaning is how to guarantee that a repair is *accurate*, or makes sense. Although an automatically-generated repair Repr ($\mathsf{Repr} = D \oplus \Delta D_{\mathsf{Repr}}$ in the incremental case) satisfies the constraints, it may contain edits to the original D that are not what the user wants. To ensure that Repr cannot go too wrong, assume that D_{opt} is the "correct"repair of D. We want Repr to be as close to D_{opt} as possible by guaranteeing that $|\mathsf{dif}(\mathsf{Repr}, D_{\mathsf{opt}})|/|D_{\mathsf{opt}}|$ is within a predefined bound ϵ. Here dif counts the attribute-level differences between two databases.

There has been a host of work on data cleaning (*e.g.*, [2, 5, 25, 10, 14, 34]). However, to develop practical data-cleaning tools there is much more to be done. First, the previous work often models the consistency of data using traditional dependencies, *e.g.*, functional dependencies (FDs). Traditional FDs were developed mainly for schema design, but are often inadequate for data cleaning. This calls for the use of constraints particularly developed for data cleaning that are able to catch more inconsistencies than traditional dependencies [29]. Second, few algorithms have been developed for automatically finding repairs, and even less incremental methods are in place. Third, none of the previous automated methods provides performance guarantee for the *accuracy* of the repairs found. These are illustrated by the example below.

Example 1.1: A company maintains a relation of sale records:

$$\mathsf{order}(\mathsf{id}, \mathsf{name}, \mathsf{AC}, \mathsf{PR}, \mathsf{PN}, \mathsf{STR}, \mathsf{CT}, \mathsf{ST}, \mathsf{zip}).$$

Each order tuple contains information about an item sold (a unique item id, name and price PR), and the phone number (area code AC, phone number PN) and address of the customer who purchased the item (street STR, city CT, state ST). An example database D is shown in Fig. 1(a) (the wt rows will be elaborated on later).

Traditional FDs on the order database include:

fd_1: $[\mathsf{AC}, \mathsf{PN}] \rightarrow [\mathsf{STR}, \mathsf{CT}, \mathsf{ST}]$ fd_2: $[\mathsf{zip}] \rightarrow [\mathsf{CT}, \mathsf{ST}]$
fd_3: $[\mathsf{id}] \rightarrow [\mathsf{name}, \mathsf{PR}]$ fd_4: $[\mathsf{CT}, \mathsf{STR}] \rightarrow [\mathsf{zip}]$

That is, the phone number of a customer uniquely determines her address, and the zip code determines the city; in addition id uniquely determines the name and PR of the item sold, and the city and street uniquely determine the zip code.

Although the database of Fig. 1(a) satisfies these FDs, the data is not clean: tuples t_3 and t_4 indicate that when the area code is 212, the city could be PHI in PA, which is not the case in real life.

Such inconsistencies can be captured by *conditional functional dependencies* (CFDs) introduced in [6]. For example, Fig. 1(b) shows two CFDs φ_1 and φ_2. CFD φ_1 extends FD fd_1 by including a *pattern tableau* T_1; it asserts that for any two order tuples, if they have the same area code 212 (resp. 610, 215) and PN, then they must have the same STR, CT, ST and moreover, the city and

	id	name	PR	AC	PN	STR	CT	ST	zip
t_1:	a23	H. Porter	17.99	215	8983490	Walnut	PHI	PA	19014
wt	(1)	(0.5)	(0.5)	(0.5)	(0.5)	(0.8)	(0.8)	(0.8)	(0.8)
t_2:	a23	H. Porter	17.99	610	3456789	Spruce	PHI	PA	19014
wt	(1)	(0.5)	(0.5)	(0.5)	(0.5)	(0.6)	(0.6)	(0.6)	(0.6)
t_3:	a12	J. Denver	7.94	212	3345677	Canel	PHI	PA	10012
wt	(1)	(0.9)	(0.9)	(0.9)	(0.9)	(0.6)	(0.1)	(0.1)	(0.8)
t_4:	a89	Snow White	18.99	212	5674322	Broad	PHI	PA	10012
wt	(1)	(0.6)	(0.5)	(0.9)	(0.9)	(0.1)	(0.6)	(0.6)	(0.9)

(a) Example order data

$\varphi_1 = ([\mathsf{AC}, \mathsf{PN}] \rightarrow [\mathsf{STR}, \mathsf{CT}, \mathsf{ST}], T_1)$

	AC	PN	STR	CT	ST
	_	_	_	_	_
T_1:	212	_	_	NYC	NY
	610	_	_	PHI	PA
	215	_	_	PHI	PA

$\varphi_2 = ([\mathsf{zip}] \rightarrow [\mathsf{CT}, \mathsf{ST}], T_2)$

	zip	CT	ST
	_	_	_
T_2:	10012	NYC	NY
	19014	PHI	PA

(b) Example CFDs

Figure 1: Example data and CFDs

state must be NYC and NY (resp. PHI and PA), respectively, regardless of what values PN, STR have (intuitively '_' indicates "don't care"). It enforces bindings of semantically related values: each tuple in T_1 specifies a constraint that only applies to tuples satisfying a certain pattern, rather than to the entire relation like fd_1. For example, the constraint specified by the second tuple in T_1 only applies to tuples with AC = 212. Similarly, CFD φ_2 extends FD fd_2. Note that CFDs φ_1 and φ_2 cannot be expressed as traditional FDs since they specify patterns with *data values*. In contrast, standard FDs are a special case of CFDs [6].

The database of Fig. 1(a) does not satisfy these CFDs. Indeed, tuple t_3 violates φ_1 since $t_3[\mathsf{AC}] = 212$ but $t_3[\mathsf{CT}, \mathsf{ST}] \neq$ (NYC, NY); it also violates φ_2: although $t_3[\mathsf{zip}] = 10012$, $t_3[\mathsf{CT}, \mathsf{ST}] \neq$ (NYC, NY). Similarly, t_4 also violates φ_1 and φ_2.

To make the database D *consistent*, one may want to edit t_3 and t_4 such that $t_3[\mathsf{CT}, \mathsf{ST}] = t_4[\mathsf{CT}, \mathsf{ST}] =$ (NYC, NY), as suggested by CFDs φ_1 and φ_2. In other words, a repair Repr of D consists of tuples t_1, t_2 and t_3, t_4 updated as above. A central task of data cleaning is to develop automated methods to find such repairs.

Now suppose that one wants to inserts a tuple t_5 into Repr, where $t_5[\mathsf{AC}, \mathsf{PN}, \mathsf{CT}, \mathsf{ST}, \mathsf{zip}]$ = (215, 8983490, NYC, NY, 10012). Then t_5 and t_1 violate fd_1: while they agree on AC, PN, they have different CT, ST. The objective of *incremental* data cleaning is to automatically and minimally update t_5 such that Repr and the updated t_5 satisfy all the CFDs and FDs given above. This is nontrivial: a naive approach to updating t_5 may lead to an infinite process. Indeed, one might want to change $t_5[\mathsf{CT}, \mathsf{ST}]$ to (PHI, PA) as suggested by CFD φ_1. However, the updated t_5 now violates CFD φ_2: $t_5[\mathsf{zip}] = 10012$ but $t_5[\mathsf{CT}, \mathsf{ST}]$ is not (NYC, NY). Now if we change $t_5[\mathsf{CT}, \mathsf{ST}]$ back to (NYC, NY) as suggested by φ_2, we are back to the original t_5 and again need to resolve the violation of φ_1.

A possible fix might be by changing $t_5[\mathsf{CT}, \mathsf{ST}, \mathsf{zip}]$ to (PHI, PA, 19014). While Repr and this edited t_5 indeed satisfy all the constraints, this change may not be *accurate*: the correct edit could be letting $t_5[\mathsf{AC}] = 212$ while keeping the rest of t_5 unchanged. Improving the *accuracy* of the data aims to guarantee that the repairs found are as close to the correct data as possible. □

Contributions. We present a data-cleaning framework that supports automated methods for finding repairs of databases, and for incrementally finding repairs in response to database updates. It also supports a statistical method that guarantees that the repairs found by our algorithms are accurate. As opposed to previous work on data cleaning, our methods are based on CFDs introduced in [6], rather than traditional dependencies. As we have seen above, CFDs are able to capture inconsistencies beyond what standard FDs can detect. Furthermore, CFDs commonly arise in practice. In data integration, for example, FDs that hold on individual sources will hold only conditionally, and thus become CFDs, on the integrated data.

Our first contribution is an algorithm for finding repairs of databases based on CFDs. As shown in [5], the problem of finding a quality repair is NP-complete even for a fixed set of traditional FDs. We show that this problem remains intractable for CFDs, and that FD-based repairing algorithms may not even terminate when applied to CFDs. To this end we adopt the cost model of [5] that incorporates both the accuracy of the data and edit distance. Based on the cost model, we extend the FD-based repairing heuristic introduced in [5] such that it is guaranteed to terminate and find quality repairs when working on CFDs. To our knowledge no prior work has considered repairing algorithms based on CFDs.

Our second contribution consists of complexity bounds and an effective algorithm for incrementally finding repairs. We show that the problem for incrementally finding quality repairs does not make our lives easier: it is also NP-complete. In light of this we develop an efficient heuristic algorithm for finding repairs in response to updates, namely, deletions or insertions of a group of tuples. This algorithm can also be used to find repairs of a dirty database.

Our third contribution is a statistical method to improve the accuracy of the repairs found by our algorithms. On one hand, in order to ensure that the repairs meet the expectation of the user, it is necessary to involve domain experts to inspect the repairs. On the other hand, it is too costly to manually check each editing when dealing with a large dataset. In response to this we develop a sampling method that, by involving the user to inspect and edit samples of manageable size, guarantees that the accurate rates of the repairs found are above a *predefined bound* with a *high confidence*.

Our fourth contribution is an experimental study of our proposed cleaning algorithms. We evaluate the accuracy and scalability of our methods with real data scraped from the Web. We find that CFDs are able to catch inconsistencies that traditional FDs fail to detect, and that our repairing and incremental repairing algorithms efficiently find accurate candidate repairs for large datasets.

Our conclusion is that CFDs and the proposed algorithms are a promising tool for cleaning real-world data. To our knowledge, our algorithms are the first automated methods for finding repairs and incrementally finding repairs based on conditional constraints. Furthermore, no prior work has studied methods for guaranteeing the accuracy of repairs without incurring excessive manual efforts.

2. Conditional Functional Dependencies

In this section we review conditional functional dependencies (CFDs) proposed in [6].

For a relation schema R, let $\mathsf{attr}(R)$ denote its set of attributes. The domain of an attribute A is denoted by $\mathsf{dom}(A)$. Given a database instance D over R, the active domain of an attribute A is denoted by $\mathsf{adom}(A, D)$; it consists of all the constants in $\mathsf{dom}(A)$ that appear as the A-attribute of a tuple in D.

In this paper we consider relation schemas consisting of a single relation R only. However, our repairing methods are applicable to general relation schemas by repairing each relation in isolation. This is possible since CFDs address a single relation only.

φ_3 = (order:[id] → [name, PR], T_3), and T_3 is

id	name	PR
_	_	_

φ_4 = (order:[CT, STR] → [zip], T_4), where T_4 is

CT	STR	zip
_	_	_

Figure 2: Standard FDs expressed as CFDs

CFD. A CFD ϕ on relation R is a pair $(R : X \rightarrow Y,\ T_p)$, where (1) X and Y are subsets of $\mathsf{attr}(R)$; (2) $R : X \rightarrow Y$ is a standard FD, referred to as the FD *embedded in* ϕ; (3) T_p is a tableau with all attributes in X and Y, referred to as the *pattern tableau* of ϕ, where for each A in X or Y, and each *pattern tuple* $t_p \in T_p$, $t_p[A]$ is either a constant 'a' in $\mathsf{dom}(A)$, or an unnamed variable '_'.

If A appears in both X and Y, we use $t_p[A_L]$ and $t_p[A_R]$ in the tableau T_p to distinguish the occurrence of the A attribute in X and Y, respectively. We denote X as $\mathsf{LHS}(\phi)$ and Y as $\mathsf{RHS}(\phi)$.

Example 2.1: Constraints φ_1 and φ_2 given in Fig. 1(b) are CFDs. In φ_1, for example, X (*i.e.*, $\mathsf{LHS}(\varphi_1)$) is {AC, PN}, Y (*i.e.*, $\mathsf{RHS}(\varphi_1)$) is {STR,CT,ST}, the standard FD embedded in φ_1 is [AC, PN] → [STR,CT, ST], and the pattern tableau is T_1 (we separate the LHS and RHS attributes in a pattern tuple with '‖'). Each pattern tuple in T_1 expresses a constraint. For instance, the first tuple of t_1 expresses the standard FD fd_1.

In fact all the constraints we have encountered so far can be expressed as CFDs. Indeed, the first pattern tuple of φ_2 expresses fd_2, and the CFDs given in Fig. 2 specifies fd_3 (φ_3) and fd_4 (φ_4). □

Observe the following. (1) A standard FD $R : X \rightarrow Y$ is a special case of the CFD $(R : X \rightarrow Y,\ T_p)$ in which T_p consists of a single pattern tuple solely containing '_'. See, for instance, Fig. 2. (2) The pattern tableau T_p of a CFD ϕ refines the standard FD embedded in ϕ by enforcing the binding of semantically related data values. In general, the FD embedded in ϕ may not hold on the entire relation; it holds only on tuples matching the pattern tuples.

Semantics. To give the precise semantics of CFDs, we first define an order $\asymp$ on data values and '_': $\eta_1 \asymp \eta_2$ if either $\eta_1 = \eta_2$, or η_1 is a data value 'a' and η_2 is '_'. The order $\asymp$ naturally extends to tuples, *e.g.*, (Walnut, NYC, NY) $\asymp$ (_, NYC, NY) but (Walnut, NYC, NY) $\not\asymp$ (_, PHI, _). We say that a tuple t_1 *matches* t_2 if $t_1 \asymp t_2$.

A relation instance D of R *satisfies* the CFD $\phi = (R : X \rightarrow Y,\ T_p)$, denoted by $D \models \phi$, iff for *each pair* of tuples t_1, t_2 in D, and for *each* tuple t_p in the pattern tableau T_p, if $t_1[X] = t_2[X] \asymp t_p[X]$, then $t_1[Y] = t_2[Y] \asymp t_p[Y]$. That is, if $t_1[X]$ and $t_2[X]$ are equal and match the pattern $t_p[X]$, then $t_1[Y]$ and $t_2[Y]$ must also be equal to each other and match the pattern $t_p[Y]$.

Example 2.2: The order table in Fig. 1 satisfies φ_3, φ_4 of Fig. 2. However, as remarked in Example 1.1, each of t_3, t_4 does not satisfy, *i.e.*, *violates*, CFDs φ_1, φ_2 of Fig. 1(b). Indeed, consider t_p = (212, _ ‖ _, NYC, NY) in T_1. Although $t_3[\mathsf{AC}, \mathsf{PN}] = t_3[\mathsf{AC}, \mathsf{PN}] \asymp t_p[\mathsf{AC}, \mathsf{PN}]$, we have that $t_3[\mathsf{STR}, \mathsf{CT}, \mathsf{ST}] \not\asymp t_p[\mathsf{STR}, \mathsf{CT}, \mathsf{ST}]$. This tells us that while a violation of a standard FD requires *two* tuples, a *single* tuple may violate a CFD. □

We say that a database D *satisfies* a set Σ of CFDs, denoted by $D \models \Sigma$, if $D \models \varphi$ for *each* $\varphi \in \Sigma$. Moreover, we say that D is *consistent with respect to* Σ if $D \models \Sigma$; otherwise we call D *inconsistent* or *dirty*.

Observe that pattern tableaus in CFDs are quite different from Codd tables, variable tables and conditional tables, which have been traditionally used in the context of incomplete information [22, 18]. The key difference is that each of these tables represents possibly infinitely many relation instances, one instance for each instantiation of variables. No instance represented by these table formalisms can include two tuples that result from different instantiations of a table tuple. In contrast, a pattern tableau is used to constrain–as part of a CFD–a *single* relation instance, which can contain any number of tuples that are all instantiations of the same pattern tuple via different valuations of the unnamed variables '_'.

Normal form. From the semantics of CFDs we immediately obtain a *normal form* of CFDs: Given a set Σ of CFDs, we may assume that each CFD $\phi \in \Sigma$ is of the form $\phi = (R : X \rightarrow A, t_p)$, where $A \in \mathsf{attr}(R)$ and t_p is a single pattern tuple. For ease of exposition we assume that CFDs are given in the normal form.

Satisfiability. To clean data based on CFDs we need to make sure that the CFDs are satisfiable, or make sense. The *satisfiability problem* is to determine, given a set Σ of CFDs, whether or not there exists a (non-empty) database D such that $D \models \Sigma$. While this problem is trivial for traditional FDs, *i.e.*, any set of FDs is satisfiable, this is no longer true for CFDs. Indeed, it has been shown that this problem is intractable in general [6]. However, when the database schema is fixed, satisfiability of CFDs can be decided in PTIME. In the sequel we consider satisfiable CFDs only.

3. A Framework for Data Cleaning

We have seen that CFDs are capable of capturing more *inconsistencies* than traditional FDs. The next question is how to resolve these violations and hence improve data consistency? Moreover, as there may exist (possibly infinitely) many repairs, which candidate repair should be chosen? Furthermore, how can one tell whether a repair is accurate or not? In this section we answer these questions, state the problems we will tackle, and present an overview of our data-cleaning framework.

3.1 Violations and Repair Operations

We first formalize the notion of violations, which helps us decide how "dirty" a data tuple is. We then discuss edit operations to resolve the violations.

Consider a database D and a set Σ of CFDs. For each tuple t in D, the *number of violations* incurred by t, denoted by $\mathsf{vio}(t)$, is computed as follows. Initially $\mathsf{vio}(t)$ is set to 0.

(1) For each CFD $\phi = (R : X \rightarrow A,\ t_p)$ in Σ, if $t[X] \asymp t_p[X]$ but $t[A] \not\asymp t_p[A]$, we say that t *violates* ϕ, and increment $\mathsf{vio}(t)$ by 1. This may occur when $t_p[A]$ is a constant.

(2) For each CFD $\phi = (R : X \rightarrow A,\ t_p)$ in Σ, if $t[X] \asymp t_p[X]$ and $t[A] \asymp t_p[A]$, then for *each* tuple t' in D such that $t[X] = t'[X] \asymp t_p[A]$ but $t[A] \neq t'[A]$, we say that t *violates* ϕ *with* t', and add 1 to $\mathsf{vio}(t)$. We can w.l.o.g. assume that $t_p[A]$ = '_' since otherwise the violation is already covered by case (1) above

For a subset C of D, the number of violations in C is defined to be the sum of $\mathsf{vio}(t)$ for all t in C, denoted by $\mathsf{vio}(C)$.

A repair Repr of a database D *w.r.t.* a set Σ of CFDs is a database that (i) satisfies Σ, *i.e.*, Repr $\models \Sigma$, and (ii) is obtained from D by means of a set of *repair operations*.

We consider *attribute value modifications* as repair operations, along the same lines as [5, 14, 24, 34]. Note that tuple insertions do not lead to repairs when CFDs (or FDs) are concerned, and that tuple deletions can be mimicked by attribute value modifications.

When we modify the A-attribute of a tuple t in the database D, we either draw its value from $\mathsf{adom}(A, D)$, *i.e.*, the set of A-attribute values occurring in D, or use the special value null when necessary. That is, we do *not invent* new values. We pick null if the value of an attribute is *unknown* or *uncertain*. To simplify the discussion we assume that one can keep track of a given tuple t in D during the repair process despite that the value of t may change (this can be achieved by *e.g.*, using a temporary unique tuple id).

Attribute value modifications are sufficient to resolve CFD violations: If a tuple t violates a CFD $\phi = (R : X \rightarrow A,\ t_p)$ (case 1 above), we *resolve the* CFD *violation* by either modifying the values of the $\mathsf{RHS}(\phi)$ attribute such that $t[A] \asymp t_p[A]$, or changing the values of some $\mathsf{LHS}(\phi)$ attributes such that $t[X] \not\asymp t_p[X]$. If t violates ϕ with another tuple t' (case 2 above), we either modify $t[A]$ (resp. $t'[A]$) such that $t[A] = t'[A]$, or change $t[X]$ (resp. $t'[X]$) such that $t[X] \not\asymp t_p[X]$ (resp. $t'[X] \not\asymp t_p[X]$) or $t[X] \neq t'[X]$.

Remarks. (1) We adopt the *simple* semantics of the SQL standard [23] for null: $t_1[X] = t_2[X]$ evaluates to true if *either one* of them contains null. (2) In contrast, when matching a data tuple t and a pattern tuple t_p, $t[X] \asymp t_p[X]$ is false if $t[X]$ contains null, *i.e.,* CFDs only apply to those tuples that precisely match a pattern tuple, which does not contain null. (3) In case some attributes are non-nullable, we use SET DEFAULT to reset attributes values to their default value. The semantics of the matching operator is redefined accordingly. For convenience, we assume that all attributes are nullable. (4) A tuple can be "deleted" via value modifications by setting null to all of its attributes.

3.2 Cost Model

As a violation may be resolved in more than one way, an immediate question is which one to choose? One might be tempted to pick the one that incurs least repair operations. While such a repair is close to the original data, it may not be accurate.

We would like to make the decision based on both the accuracy of the attribute values to be modified, and the "closeness" of the new value to the original value. Following the practice of US national statistical agencies [13, 35], we assume that a *weight* in the range $[0, 1]$ is associated with each attribute A of each tuple t in the dataset D, denoted by $w(t, A)$ (see the wt rows in Fig. 1(a)). The weight reflects the confidence of the *accuracy* placed by the user in the *attribute* $t[A]$, and can be propagated via data provenance analysis in data transformations. Given this, we extend the cost model of [5] to provide a guidance for how to choose a repair.

For two values v, v' in the same domain, we assume that a *distance function* $\mathsf{dis}(v, v')$ is in place, with lower values indicating greater similarity. In our implementation, we simply adopt the Damerau-Levenshtein (DL) metric [16], which is defined as the minimum number of single-character insertions, deletions and substitutions required to transform v to v'. The cost of changing the value of an attribute $t[A]$ from v to v' is defined to be:

$$\mathsf{cost}(v, v') = w(t, A) \cdot \mathsf{dis}(v, v')/\mathsf{max}(|v|, |v'|),$$

Intuitively, the more accurate the original $t[A]$ value v is and more distant the new value v' is from v, the higher the cost of this change. We use $\mathsf{dis}(v, v')/\mathsf{max}(|v|, |v'|)$ to measure the similarity of v and v' to ensure that longer strings with 1-character difference are closer than shorter strings with 1-character difference.

The cost of changing the value of an R-tuple t to t' is the sum of $\mathsf{cost}(t[A], t'[A])$ for each $A \in \mathsf{attr}(R)$ for which the value of $t[A]$ is modified. The cost of a repair Repr of D, denoted $\mathsf{cost}(\mathsf{Repr}, D)$ is the sum of the costs of modifying tuples in D.

Example 3.1: Recall from Example 1.1 that tuple t_3 violates CFDs φ_1, φ_2 given in Fig. 1(b). There are at least two alternative methods to resolve the violations: changing (1) $t_3[\mathsf{CT}, \mathsf{ST}]$ to (NYC, NY), or (2) $t_3[\mathsf{zip}]$ to 19014 and $t_3[\mathsf{AC}]$ to 215. The costs of these repairs are 3/3 * 0.1 + 3/3 * 0.1 = 0.2 and 1/3 * 0.9 + 2/5 * 0.8 = 0.6, respectively, in favor of option (1). Indeed, although option (1) involves more editing than option (2), it may be more reasonable since the weights of $t_3[\mathsf{CT}, \mathsf{ST}]$ indicate that these attributes are less trustable and thus are good candidates to change. □

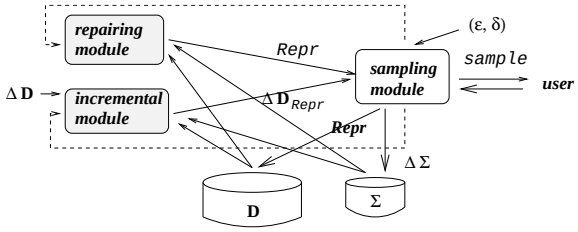

Figure 3: Data cleaning framework

Remarks. (1) Although the cost model incorporates the weight information, our cleaning algorithms to be given shortly do not necessarily rely on this. In the absence of the weight information, our algorithms set $w(t, A)$ to 1 for each attribute A of each tuple t. In this case our algorithms use the number of violations $\mathsf{vio}(t)$ to guide repairing process, and our experimental results show that the algorithms work well even when the weight information is not available. (2) Other similarity metrics (see, *e.g.,* [11]) can also be used instead of the DL metric in our model.

3.3 A Data Cleaning Framework: Overview

The *repairing* problem is stated as follows: given a set Σ of CFDs over a schema R and a database instance D of R, it is to compute a repair Repr of D such that $\mathsf{Repr} \models \Sigma$ and $\mathsf{cost}(\mathsf{Repr}, D)$ is minimum. That is, we want *automated* methods to find a repair *consistent w.r.t.* Σ by modifying D. Intuitively, the smaller $\mathsf{cost}(\mathsf{Repr}, D)$ is, the more accurate and closer to the original data Repr is.

We also study the *incremental repairing problem*: suppose that the database D is consistent, *i.e.,* $D \models \Sigma$. Given updates ΔD to D, we want to find a repair ΔD_{Repr} of ΔD such that $D \oplus \Delta D_{\mathsf{Repr}} \models \Sigma$ and $\mathsf{cost}(\Delta D_{\mathsf{Repr}}, \Delta D)$ is minimum. Since small ΔD often incurs a small number of CFD violations, and because D is clean and thus should not be updated, it is more reasonable and more efficient to compute ΔD_{Repr} than computing a repair Repr of $D \oplus \Delta D$ starting from scratch. We consider *group updates*: ΔD is a set of tuples to be inserted or deleted. For any deletions ΔD, the tuples can be simply removed from D without causing any CFD violation. Thus we need only to consider tuple insertion.

To assess the accuracy of repairs, assume a correct repair D_{opt} of D, perhaps worked out manually by domain experts. We say that a repair is *accurate w.r.t.* a *predefined bound* ϵ at a *predefined confidence level* δ, if the ratio $|\mathsf{dif}(\mathsf{Repr}, D_{\mathsf{opt}})|/|D_{\mathsf{opt}}|$ is within the bound ϵ at the confident level δ.

In practice it is unrealistic to manually find D_{opt} or involve domain experts to inspect the entire Repr when the dataset is large. To this end we employ a semi-automated and interactive approach: we let the user inspect small samples, and edit the sample data as well as input CFDs if necessary; leveraging the user input, we invoke our automated (incremental) repairing methods to revise repairs.

Putting these together, we develop a framework for data cleaning as shown in Fig. 3. The framework consists of three modules. (a) The repairing module takes as input a database D and a set Σ of CFDs. It *automatically* finds a candidate repair Repr. (b) The incremental repairing module takes updates ΔD as additional input, and *automatically* finds repair ΔD_{Repr}. (c) The output repairs of these two modules are sent to the sampling module, which also takes as input accuracy bound and confidence (ϵ, δ). The sampling module generates a sample and lets the user inspect it. The user feedback – both changes $\Delta\Sigma$ to the CFDs and changes to the sample data – is recorded. If the accuracy is below the predefined bound, the repairing or incremental repairing module is invoked again based on the user feedback. The process may continue until an accurate enough repair is recommended to the user. In the next three sections, we present algorithms and methods for supporting these modules.

4. An Algorithm for Finding Repairs

We now present an algorithm for the repairing module, which *automatically* finds a candidate repair for an inconsistent database.

It is nontrivial to find a quality repair. As shown in [5], the repairing problem is already NP-complete for standard FDs even when the relational schema and FDs are fixed (*i.e.,* the intractability is the data complexity). We show that for CFDs the problem remains NP-complete, *i.e.,* CFDs do not add to the complexity of this problem.

Corollary 4.1: *The repairing problem for* CFDs *is* NP*-complete, even for a fixed database schema and a fixed set of* CFDs. □

This tells us that practical automated methods for this problem have to be heuristic. Worse, although CFDs do not increase the worst-case complexity, previous methods for repairing FDs no longer work on CFDs. Indeed, while it suffices to resolve FD violations by only editing values of attributes in the RHS of FDs [5], this strategy may not terminate on CFDs, as shown by the next example.

Example 4.1: Recall CFDs φ_1, φ_2 from Fig 1(b). As illustrated in Example 1.1, tuples t_1, t_5 violate φ_1. While this violation can be resolved by changing the value (NYC, NY) of the $\mathsf{RHS}(\varphi_1)$ attributes $t_5[\mathsf{CT}, \mathsf{ST}]$, to the values $t_1[\mathsf{CT}, \mathsf{ST}]$, this introduces a violation of φ_2. This can no longer be resolved by changing the value of the $\mathsf{RHS}(\varphi_2)$ attributes $t_5[\mathsf{CT}, \mathsf{ST}]$ back to (NYC, NY) as suggested by φ_2, since otherwise we are back to the original t_5, have to resolve the violation of φ_1 again, and end up with an infinite process. □

To cope with this we present a repair algorithm, BATCHREPAIR, which is a nontrivial extension of the algorithm for FDs proposed in [5]. It extends the notion of equivalence classes of [5], and it guarantees to terminate and finds a repair *w.r.t.* CFDs.

4.1 Resolving CFD Violations

We first revise the notion of equivalence classes explored in [5], and then present our strategy for repairing CFDs.

Equivalence classes. An *equivalence class* consists of pairs of the form (t, A), where t identifies a tuple in which A is an attribute. In a database D, each tuple t and each attribute A in t have an associated equivalence class, denoted by $\mathsf{eq}(t, A)$.

In a repair we will assign a unique *target value* to each equivalence class E, denoted by $\mathsf{targ}(E)$. That is, for all $(t, A) \in E$, $t[A]$ has the same value $\mathsf{targ}(E)$. The target value $\mathsf{targ}(E)$ can be either '_', a constant a, or null, where '_' indicates that $\mathsf{targ}(E)$ is not yet fixed, and null means that $\mathsf{targ}(E)$ is uncertain due to conflict. To resolve CFD violations we may "upgrade" $\mathsf{targ}(E)$ from '_' to a constant a, or from a to null, but not the other way around. In particular, we *do not* change $\mathsf{targ}(E)$ from one constant to another.

Intuitively, we resolve CFD violations by *merging* or *modifying* the target values of equivalence classes. Consider a CFD $\phi = (R : X \to A,\ t_p)$. For any pair of tuples t_1 and t_2 in D, if $t_1[X] = t_2[X] \asymp t_p[X]$, then (t_1, A) and (t_2, A) should belong to the *same* equivalence class and eventually, $t_p[A] = \mathsf{targ}(E)$. If $(t_1, A) \neq (t_2, A)$, we may be able to resolve the violation by merging $\mathsf{eq}(t_1, A)$ and $\mathsf{eq}(t_2, A)$ into one. By using equivalence classes, we separate the decision of which attribute values should be equal from the decision of what value should be assigned to the equivalence class. We defer the assignment of $\mathsf{targ}(E)$ as much as possible to reduce poor local decisions, such as changing the value of $t_5[\mathsf{CT}, \mathsf{ST}]$ in Example 4.1.

We use $\mathcal{E}$ to keep track of the current set of equivalence classes in a database D. Initially, $\mathcal{E}$ consists of $\mathsf{eq}(t, A)$ for all tuples t in D and all attribute A in t, where $\mathsf{eq}(t, A)$ starts with a single pair (t, A), with $\mathsf{targ}(\mathsf{eq}(t, A))$ = _.

Procedure CFD-RESOLVE. Leveraging equivalence classes, we present the main idea of our strategy for resolving CFD violations, which is done by procedure CFD-RESOLVE, a key component of algorithm BATCHREPAIR.

Procedure CFD-RESOLVE takes as input a pair (t, A) and a CFD $\varphi = (R : X \to A, t_p)$, where t violates φ. Recall from Section 3.1 that t may violate φ if $t[X] \asymp t_p[X]$ and in addition, either (1) $t[A] \not\asymp t_p[A]$ and $t_p[A]$ is a constant a; or (2) there exists another tuple t' such that $t'[X] = t[X]$ but $t'[A] \neq t[A]$, where $t_p[A] =$ _. The procedure resolves the violation as follows.

(1) $t[A] \not\asymp t_p[A]$ and $t_p[A] = a$. There are two cases to consider.

(1.1) If $\mathsf{targ}(\mathsf{eq}(t, A))$ = '_', *i.e.,* the target value of $\mathsf{eq}(t, A)$ is not yet fixed, we resolve this by simply letting $\mathsf{targ}(\mathsf{eq}(t, A)) := a$.

(1.2) Otherwise $\mathsf{targ}(\mathsf{eq}(t, A))$ is either a distinct constant b, or null for which we know that the value cannot be made certain. In this case we have to change the value of some $\mathsf{LHS}(\varphi)$ attribute of t, a situation that does not arise when repairing traditional FDs.

More specifically, we look at each attribute $B_i \in X$ such that $\mathsf{targ}(\mathsf{eq}(t, B_i))$ is '_', *i.e.,* not yet fixed. If no such B_i exists, we cannot resolve the conflict with a certain value. Thus we pick B_i such that the sum of weights of attributes in $\mathsf{eq}(t, B_i)$ is minimal, and change $\mathsf{targ}(\mathsf{eq}(t, B_i))$ to null. If there exists B_i with $\mathsf{targ}(\mathsf{eq}(t, B_i)) =$ _, we pick such a B_i and a value v such that $\mathsf{cost}(\mathsf{eq}(t[B_i]), v)$ is minimum, and let $\mathsf{targ}(\mathsf{eq}(t, B_i)) := v$. The value v is picked by a procedure FINDV, which we shall discuss shortly, along with the definition of $\mathsf{cost}(\mathsf{eq}(t[B_i]), v)$.

Example 4.2: Continuing with Example 4.1, suppose that we want to resolve the violation of φ_2 caused by tuple t_5. If $\mathsf{targ}(\mathsf{eq}(t_5, \mathsf{CT}))$ and $\mathsf{targ}(\mathsf{eq}(t_5, \mathsf{ST}))$ are '_', we can resolve this by simply letting them to be NYC and NY, respectively. However, if these target values were already set to PHI and PA when, *e.g.,* resolving the violation of φ_1 caused by t_5 and t_1, we can no longer change these target values of the $\mathsf{RHS}(\varphi_2)$ attributes. Hence, we have to change the value of the $\mathsf{LHS}(\varphi_2)$ attribute $t_5[\mathsf{zip}]$. Now procedure FINDV may set $\mathsf{targ}(\mathsf{eq}(t_5, \mathsf{zip}))$ to 19014. If, however, $\mathsf{targ}(\mathsf{eq}(t_5, \mathsf{zip}))$ was already given another constant, we set it to null since there is no certain value to resolve the violation. □

(2) t violates φ with another tuple t'. We consider the following cases. Suppose that $\mathsf{targ}(\mathsf{eq}(t, A)) = \eta$ and $\mathsf{targ}(\mathsf{eq}(t', A)) = \eta'$.

(2.1) Neither η nor η' is null, and at least one of them is '_'. In this case the violation is resolved by *merging* $\mathsf{eq}(t, A)$ and $\mathsf{eq}(t', A)$ into one. We remark that this step is identical to the resolution step for FDs presented in [5]. In fact this is the *only* operation required to resolve all FD violations. For CFDs, more needs to be done. We let $\mathsf{targ}(\mathsf{eq}(t, A))$ be '_' if both η and and η' are '_'; if one of them is a constant c, we let $\mathsf{targ}(\mathsf{eq}(t, A))$ be c.

(2.2) η' and η' are distinct constants c, c', respectively. Like case (1.2) above, this inconsistency cannot be resolved by changing $\mathsf{RHS}(\varphi)$ attributes, and we have to resolve this by changing some $\mathsf{LHS}(\varphi)$ attribute of either t or t', along the same lines as case (1.2).

(2.3) At least one of η and η' is null. Assume that it is η. Then $t[A]$ will be given null as its value. By the simple semantics of null, $t[A] = \mathsf{targ}(\mathsf{eq}(t', A))$ no matter what value $\mathsf{targ}(\mathsf{eq}(t', A))$ will eventually take. In other words, the violation is already resolved.

Example 4.3: Consider again the setting of Example 4.1, and suppose that we want to resolve the violation of φ_1 caused by t_5 and t_1. If the target values of $\mathsf{eq}(t_5, \mathsf{CT})$ and $\mathsf{eq}(t_5, \mathsf{ST})$ (resp. $\mathsf{eq}(t_1, \mathsf{CT})$ and $\mathsf{eq}(t_1, \mathsf{ST})$) are '_', and none of them is null, we can resolve the violation by simply merging $\mathsf{eq}(t_5, \mathsf{CT})$ and $\mathsf{eq}(t_1, \mathsf{CT})$ and by merging $\mathsf{eq}(t_5, \mathsf{ST})$ and $\mathsf{eq}(t_1, \mathsf{ST})$. In the presence of conflicting target values, *e.g.,* when $\mathsf{eq}(t_5, \mathsf{CT})$ and $\mathsf{eq}(t_1, \mathsf{CT})$ have distinct

Procedure BATCHREPAIR(D, Σ)

Input: A set Σ of CFDs, and a database D.
Output: A repair Repr of D.

1. $\mathcal{E} := \{\{(t, A)\} \mid t \in R, A \in att(R)\}$;
2. **for** each $E \in \mathcal{E}$ **do** /* initializing $\mathsf{targ}(E)$ */
3. $\mathsf{targ}(E) := \_$;
4. Initialize Dirty_Tuples;
5. **while** Dirty_Tuples $\neq \emptyset$
6. $(t, B, v, \varphi) :=$ PICKNEXT();
7. Repr := CFD-RESOLVE(t, B, v, φ);
8. Update Dirty_Tuples;
9. **if** Dirty_Tuples $= \emptyset$ **then**
10. **for** each $E \in \mathcal{E}$ **do**
11. **if** $\mathsf{targ}(E) = \_$ **then** /* instantiating _ */
12. $\mathsf{targ}(E)$:= a constant with the least cost;
13. Update Dirty_Tuples;
14. **for** each $E \in \mathcal{E}$ and each $(t, A) \in E$ **do**
15. $t[A] := \mathsf{targ}(E)$; /* updating D to obtain Repr
16. **return** D.

Figure 4: Algorithm BATCHREPAIR

constant target values, we have to change the target value of the $\mathsf{LHS}(\varphi_1)$ attributes of either t_1 or t_5, *i.e.,* the target value of one of $\mathsf{eq}(t_5, \mathsf{AC})$, $\mathsf{eq}(t_5, \mathsf{PN})$, $\mathsf{eq}(t_1, \mathsf{AC})$ or $\mathsf{eq}(t_1, \mathsf{PN})$. □

4.2 Batch Repair Algorithm

We now present algorithm BATCHREPAIR. In addition to the set $\mathcal{E}$ of equivalence classes, the algorithm keeps track of violations of CFDs. As we have seen in Example 4.1, a repair may generate *new violations*. Therefore, we maintain for each CFD $\varphi \in \Sigma$ a set Dirty_Tuples(φ) of tuples that (possibly) violate φ. We update these sets after each resolution of a violation. More precisely, suppose that a violation of φ caused by t is resolved by updating $\mathsf{eq}(t, A)$. Then for each tuple t', if $(t', A) \in \mathsf{eq}(t, A)$, and for each $\psi = (R : X \to C, t_p)$, if $A \in X \cup \{C\}$, we add t' to Dirty_Tuples(ψ). We then remove t from Dirty_Tuples(φ). In this way Dirty_Tuples always contain all *potentially unresolved* tuples.

The algorithm is shown in Fig. 4. We start with initialization of the set $\mathcal{E}$ of equivalence classes and Dirty_Tuples (lines 1-4). Next, as long as there are dirty tuples (loop on line 5) we greedily look for the "best" next repair. More specifically, the procedure PICKNEXT loops over each CFD $\varphi \in \Sigma$ and its violating tuple t; it identifies which pair (φ, t) incurs the least cost to repair (line 6). The algorithm then resolves t for φ (line 7), resulting in a modified set of equivalence classes, by invoking procedure CFD-RESOLVE. It then updates the set of dirty tuples (line 8) before finding the next best repair. If no more dirty tuples are unresolved (line 9), then for each equivalence class $E \in \mathcal{E}$ with $\mathsf{targ}(E) = \_$, it finds a constant value with the least cost to instantiate $\mathsf{targ}(E)$ (lines 10-12). That is, *ultimately* all equivalence classes will have either a constant value or null. This instantiation may introduce new violations, and thus Dirty_Tuples should be maintained (line 13). After the loop, we create a repair Repr by editing the original database D by using the target values of equivalence classes (lines 14-15).

The most expensive and elaborate procedure is PICKNEXT (see Fig. 5). It finds the next tuple t and CFD φ to be resolved. More specifically, for each CFD φ and its unresolved tuple t, PICKNEXT first decides for which attribute B of t it can update $\mathsf{eq}(t, B)$ to resolve the violation (line 3), following the analysis described in Section 4.1. After B is fixed, it finds a set S of tuples that agree with t on all the attributes in φ except B (line 4). The idea is that we may pick a target value v for $\mathsf{eq}(t, B)$ from the B-attribute values of the tuples in S (line 5). It then analyzes the cost of repairing the violation using v (lines 6-7), where $\mathsf{Cost}(t, B, v)$ is defined to be $\sum_{(t', C) \in \mathsf{eq}(t, B)} w(t', C) \cdot \mathsf{cost}(v, t'[C])$. It returns (t, B, v) with

Procedure PICKNEXT()

1. BestCost := ∞;
2. **for** each CFD $\varphi = (R : X \to A,\ t_p)$, $t \in$ Dirty_Tuples(φ) **do**
3. decide an attribute B in t to update $\mathsf{eq}(t, B)$;
4. $S := \{t' \in R \mid t'[X \cup \{A\} \setminus \{B\}] = t[X \cup \{A\} \setminus \{B\}]\}$;
5. $v :=$ FINDV(t, B, S, φ);
6. **if** $\mathsf{Cost}(t, B, v) <$ BestCost **then**
7. BestFix := (t, B, v, φ); BestCost := $\mathsf{Cost}(t, B, v)$;
8. **return** BestFix;

Figure 5: procedure PICKNEXT

the least cost (line 8).

It remains to show how the value v is picked. Given t, B and φ, procedure FINDV (not shown) aims to select semantically-related values by first using values in CFDs. If this is not possible, a value is selected from values appearing in related tuples. Moreover, by the definition of Cost the optimal value is selected in a similar way as in the most-common-value strategy. More precisely, FINDV checks whether $B = A$. If so, v is already determined by either $t_p[A]$ (case (1.1) in Section 4.1) or the target values of $\mathsf{eq}(t, A)$ and $\mathsf{eq}(t', A)$ (t' is the tuple with which t violates φ, case (2.1)). Otherwise, *i.e.,* if $B \in \mathsf{LHS}(\varphi)$, it inspects $\mathsf{targ}(\mathsf{eq}(t_1, B))$ for all $t_1 \in S$, and finds v with the least $\mathsf{Cost}(t, B, v)$ such that $v \neq t[B]$. The motivation for picking v from S is to find a semantically-related value, identified by the pattern $t[X \cup \{A\} \setminus \{B\}]$. If such v does not exist, it lets $v :=$ null.

Example 4.4: Returning to Example 4.2, suppose now that the target values of $(\mathsf{eq}(t_5, \mathsf{CT}), \mathsf{eq}(t_5, \mathsf{ST}))$ are (PHI, PA). To resolve the violation of φ_2 caused by t_5, we decide to change the target value of $t_5[\mathsf{zip}]$. Procedure PICKNEXT finds $S = \{t_1, t_2, t_3, t_4\}$, *i.e.,* S consists of all tuples t' with (PHI, PA) as the target value of $(\mathsf{eq}(t', \mathsf{CT}), \mathsf{eq}(t', \mathsf{ST}))$, Now Procedure FINDV attempts to choose v from the target values of $\mathsf{eq}(t', \mathsf{zip})$ for $t' \in S$. There are two such values: 19014 and 10012. It decides to pick 19014 since it is the only one that differs from $t_5[B]$. If S were empty or $\mathsf{targ}(\mathsf{eq}(t_5, \mathsf{zip}))$ already had a constant, it assigns null to v. □

Upon receiving (t, B, v, φ) from PICKNEXT, procedure CFD-RESOLVE in algorithm BATCHREPAIR merges or update the target values of equivalence classes to resolve the violation of φ caused by t, as described in Section 4.1.

Correctness. Clearly at each step of algorithm BATCHREPAIR, a CFD violation is resolved. However, each step can also introduce new violations as illustrated in Example 4.1; moreover, a tuple t can appear as a violation multiple times. Nevertheless, BATCHREPAIR always terminates and generates a repair.

Theorem 4.2: *Given any database D and any set Σ of* CFDs, BATCHREPAIR *terminates and finds a repair* Repr $\models \Sigma$ *for D.* □

Proof sketch: At each step either the total number N of equivalence classes is reduced or the number H of those classes that are assigned a constant or null is increased. Let k be the number of (t, A) pairs in D. Since $N \leq k$ and $H \leq 3 \cdot k$ (the target value of $\mathsf{eq}(t, A)$ can only be '_', a constant, or null), BATCHREPAIR necessarily terminates. Furthermore, since the algorithm proceeds until no more dirty tuples exist, it always finds a repair of D. □

5. An Incremental Repairing Algorithm

In this section we present the algorithm underlying the incremental module of our framework shown in Fig 3, which tackles the *incremental repairing problem*. As remarked in Section 3.3, it suffices to consider ΔD consisting of insertions only, as deletions never cause any inconsistencies.

Procedure INCREPAIR($D, \Delta D, \Sigma, \mathcal{O}$)

Input: A clean database D, a set Σ of CFDs, a set of updates ΔD, and an ordering $\mathcal{O}$ on ΔD.
Output: A repair Repr of $D \oplus \Delta D$ such that $D \subseteq$ Repr.

1. Repr := D;
2. **for** each t in ΔD in the given order $\mathcal{O}$ **do**
3. $\quad$ Repr_t := TUPLERESOLVE (t, Repr, Σ);
4. $\quad$ Repr := Repr $\cup \{\mathsf{Repr}_t\}$;
5. **return** Repr.

Figure 6: Algorithm INCREPAIR

One might think that the incremental repairing problem is simpler than its batch (non-incremental) counterpart. Unfortunately it is not the case. Indeed, since the repairing problem (see Section 3.3) can be seen as an instance of the incremental repairing problem (indeed, just consider the case that $D = \emptyset$), we immediately obtain the following corollary from Theorem 4.1.

Corollary 5.1: *The incremental repairing problem for* CFDs *is* NP*-complete, even for a fixed schema and a fixed set of* FDs. □

Therefore, we again have to rely on heuristics in the incremental setting. We first develop a heuristic in Section 5.1 and then present optimization techniques to improve the algorithm in Section 5.2. Finally, we show in Section 5.3 that the incremental algorithm in fact provides an alternative method for the repairing problem.

5.1 Incremental Algorithm and Local Repairing Problem

Given a set of updates ΔD, Corollary 5.1 tells us that it is beyond reach in practice to find an optimal ΔD_{Repr}. Furthermore, we cannot directly apply the algorithm developed for the repairing problem to finding ΔD_{Repr} since we cannot prevent it from updating the clean D. Following the approach commonly used in repairing census data [13, 35], we repair the tuples in ΔD *one at a time* following some ordering $\mathcal{O}$ on these tuples. We assume that $\mathcal{O}$ is given but will provide various orderings in Section 5.2.

Therefore, the key problem is to find, given a clean database D, a tuple t to be inserted into D, and a set Σ of CFDs, a repair Repr_t of t of minimum cost such that $D \cup \{\mathsf{Repr}_t\}$ is a repair. We refer to this as the *local repairing problem*.

Algorithm INCREPAIR. The overall driver of our incremental repairing algorithm is presented in Fig. 6. Taking as input a database D, a set ΔD of updates, a set Σ of CFDs, and an ordering $\mathcal{O}$ on ΔD, it does the following. It first initializes the repair Repr with the current clean database D (line 1). It then invokes a procedure called TUPLERESOLVE (line 3) to repair each tuple t in ΔD according to the given order $\mathcal{O}$ (line 2), and adds the local repair Repr_t of t to Repr (line 4) before moving to the next tuple. Once all tuples in ΔD are processed, the final repair is reported (line 5).

The key characteristics of INCREPAIR are (i) that the repair grows at each step, providing in this way more information that we can use to clean the next tuple, and (ii) that the data in D is not modified since it is assumed to be clean already.

Algorithm TUPLERESOLVE. The core of the INCREPAIR algorithm is the procedure TUPLERESOLVE that aims to solve the local repairing problem. One might think that the local repairing problem would make our lives easier. However, the result below tells us that it is not the case.

Theorem 5.2: *The local repairing problem is* NP*-complete. Moreover, it remains intractable if one considers standard* FDs *only.* □

Proof sketch: The NP-hardness is verified by reduction from the distance-SAT problem, which is NP-complete [3]. That is to determine, given a propositional logic formula ϕ, an initial truth assignment ρ_1, and a constant k, whether there exists a truth assignment ρ_2 that satisfies ϕ and differs from ρ_1 in at most k variables. □

Procedure TUPLERESOLVE(t, Repr, Σ)

Input: A tuple t to repair, the current repair Repr, and a set Σ of CFDs.
Output: A repair Repr_t of t such that Repr $\cup \{\mathsf{Repr}_t\} \models \Sigma$.

1. $\mathcal{C} := \emptyset$; $\mathsf{Repr}_t := t$;
2. **while** $\mathsf{attr}(R) \neq \mathcal{C}$ **do**
3. $\quad$ cost := ∞;
4. $\quad$ **for** each $C \in [\mathsf{attr}(R) \setminus \mathcal{C}]_k$ **do**
5. $\quad\quad$ $\mathcal{V} := \{\hat{v} \mid \mathsf{Repr} \cup \{repr_t[C/\hat{v}]\} \models \Sigma(C \cup \mathcal{C})\}$;
6. $\quad\quad$ $\hat{v} := \arg\_\min_{\hat{v} \in \mathcal{V}} \mathsf{costfix}(C, \hat{v})$;
7. $\quad\quad$ **if** $\mathsf{costfix}(C, \hat{v}) <$ cost **then**
8. $\quad\quad\quad$ cost := $\mathsf{costfix}(C, \hat{v})$; BestFix:=$(C, \hat{v})$;
9. $\quad$ $\mathcal{C} := \mathcal{C} \cup C$; $\mathsf{Repr}_t := \mathsf{Repr}_t[C/\hat{t}]$;
10. **return** Repr_t.

Figure 7: Algorithm TUPLERESOLVE

Theorem 5.2 shows that finding the optimal repair Repr_t of t is infeasible in practice. Indeed, the naive approach, namely, enumerating all possible repairs and then selecting the one with the minimal cost, is clearly not an option in case that the number of attributes or the size of the active domains is large.

In light of this intractability, procedure TUPLERESOLVE is based on a *greedy* approach. As shown in Fig. 7, it takes as input a single tuple t to be inserted, the current repair Repr, and a set Σ of CFDs, and returns a repair Repr_t of t such that $\mathsf{Repr} \cup \{\mathsf{Repr}_t\} \models \Sigma$.

Before we explain TUPLERESOLVE in more detail, we need some notation. For a fixed integer $k > 0$ and a set of attributes $X \subseteq \mathsf{attr}(R)$ we denote by $[X]_k$ the set of all subsets of X of size k. For a tuple t, a set $C \in [X]_k$ and $\bar{v} = (v_1, \ldots, v_k)$, where $v_i \in \mathsf{adom}(D, A_i) \cup \{\mathsf{null}\}$ for each $A_i \in C$, we denote by $t[C/\bar{v}]$ the tuple obtained by replacing $t[A_i]$ by v_i for each $A_i \in C$ and leaving the other attributes unchanged. Finally, for a set Σ of CFDs and a set $X \subseteq \mathsf{attr}(R)$, we denote by $\Sigma(X)$ the set of CFDs in Σ of the form $(R : Y \to A, t_p)$ with $Y \cup \{A\} \subseteq X$.

We explain how procedure TUPLERESOLVE works in an inductive way. In a nutshell, it greedily finds the "best" sets of attributes of t to modify in order to create a repair. More specifically, for a fixed $k > 0$ it first finds the "best" $C_1 \in [\mathsf{attr}(R)]_k$ (lines 4–9) and attribute values $\hat{v} = (v_1, \ldots, v_k)$ for the attributes in C_1 such that

(i) v_i is in $\mathsf{adom}(\mathsf{Repr}, A_i) \cup \{\mathsf{null}\}$ (line 5);

(ii) $\mathsf{Repr} \cup \{t[C_1/\hat{v}]\}$ satisfies all CFDs in $\Sigma(C_1)$ (line 5); and

(iii) the cost $\mathsf{costfix}(C_1, \hat{v}) = \mathsf{cost}(t, t[C_1/\hat{v}]) \times \mathsf{vio}(t[C_1/\hat{v}])$ is minimal (lines 6–8).

In other words, the predefined parameter k limits the number of possible repairs that we consider. Our experiments show that for $k = 1, 2$ we are already able to obtain good results. We denote the set of all k-tuples $\bar{v}$ satisfying (i) and (ii) by $\mathcal{V}$ (line 5). Once TUPLERESOLVE finds C_1 and $\hat{v}$, C_1 is added to $\mathcal{C}$ and t is replaced by $t_1 = t[C_1/\hat{v}]$ (line 9). Furthermore, TUPLERESOLVE will *never* backtrack and modify t_1 for the attributes in C_1 again.

Suppose that TUPLERESOLVE already selected n best pairwise disjoint sets $C_1, \ldots, C_n$ in $[\mathsf{attr}(R)]_k$ and k-tuples $\hat{v}_1, \ldots, \hat{v}_n$ such that for $t_n = t_{n-1}[C_n/\hat{v}_n]$, we have that $\mathsf{Repr} \cup \{t_n\} \models \Sigma(\mathcal{C})$, where $\mathcal{C} = C_1 \cup \cdots \cup C_{n-1}$. That is, t_n is the current (almost) repair for t. If $\mathsf{attr}(R) = \mathcal{C}$ then clearly t_n is a real repair of t and TUPLERESOLVE will output $\mathsf{Repr}_t = t_n$ (line 2, line 10). Otherwise, TUPLERESOLVE finds the next best set C_{n+1} in $[\mathsf{attr}(R) \setminus \mathcal{C}]_k$ and finds a k-tuple $\hat{v}_{n+1}$ satisfying the same conditions (i)–(iii) as above *except* that the repair $t_{n+1} = t_n[C_{n+1}/\hat{v}_{n+1}]$ must satisfy $\Sigma(C_{n+1} \cup \mathcal{C})$. Again, the set C_{n+1} is then added to $\mathcal{C}$ and the current (almost) repair is set to t_{n+1}. The procedure TUPLERESOLVE keeps selecting such sets of attributes and values until $\mathsf{attr}(R)$ is completely covered.

It is important that $\bar{v}$ is allowed to contain null values (see property (i)). Indeed, this is needed for guaranteeing the existence of k-tuples $\bar{v}$ satisfying property (ii) as the next example illustrates.

Example 5.1: Consider t_5 in Example 1.1 and suppose that $k = 2$. Suppose that TUPLERESOLVE already fixed all attributes except CT and ST. In fact, no attribute values in t_5 are changed since the violated CFDs involve the two non-fixed attributes. In order for TUPLERESOLVE to repair t_5 it needs to find a tuple $\hat{v} = (v_1, v_2)$ for $C = \{\mathsf{CT}, \mathsf{ST}\}$ such that $t_5[C/\hat{v}]$ satisfies both φ_1 and φ_2. As observed in Example 1.1 no such $\hat{v}$ exists when we only consider values in the active domains. Thus the only possible $\hat{v}$ here is (null, null). In contrast, Example 1.1 shows that C={CT, ST, zip} for $k = 3$, and $\hat{v}$=(PHI, PA, 19014) provides a repair for t_5. □

Correctness. The termination of INCREPAIR follows from the fact that (i) each tuple in ΔD is treated only once; and (ii) each attribute is modified at most once by TUPLERESOLVE. Moreover, TUPLERESOLVE always generates a repair for each tuple in ΔD.

Theorem 5.3: *Given a database D, a set Σ of* CFDs *and update ΔD,* INCREPAIR *always terminates and finds a repair ΔD_{Repr} such that $D \oplus \Delta D_{\mathsf{Repr}} \models \Sigma$, regardless of the ordering $\mathcal{O}$.* □

5.2 Ordering for Processing Tuples and Optimizations

While the ordering $\mathcal{O}$ for processing tuples has no impact on the termination of an INCREPAIR process, it does make a difference when it comes to repairing performance and the accuracy of the repair. We next study various orderings, based on which we develop (and experiment with) variants of the INCREPAIR algorithm.

Theorem 4.1 tells us that it is beyond reach in practice to find an ordering that leads to an optimal repair. Thus we propose and experiment with the following orderings.

Linear-scan ordering. A naive approach is to adopt an arbitrary linear-scan order for $\mathcal{O}$, with the benefit that it incurs no extra cost. We refer to INCREPAIR based on this as L-INCREPAIR.

A greedy algorithm based on violations. This algorithm, referred to as V-INCREPAIR, is based on the *number of violations* $\mathsf{vio}(t)$ of each tuple t, which is defined in Section 3.1. A tuple $t \in D$ might cause multiple violations of constraints in Σ. Intuitively, the less $\mathsf{vio}(t)$ is, the more accurate t is and the less costly to repair it. Algorithm V-INCREPAIR repairs tuples in the *increasing* order of $\mathsf{vio}(t)$ so that accurate tuples are included in Repr early, and based on them we resolve violations of "less accurate" tuples.

A greedy algorithm based on weights. Another approach is based on the weight $\mathsf{wt}(t)$ of a tuple t (recall the definition of $\mathsf{wt}(t)$ from Section 3.2). Intuitively, the larger $\mathsf{wt}(t)$ is, the more accurate t is. We develop a variant of INCREPAIR, referred to as W-INCREPAIR, which processes tuples based on the *decreasing* order of $\mathsf{wt}(t)$ to reduce the cost and improve the quality of repairs found.

We next present optimizations adopted by our algorithm.

Optimization. The main computational cost of INCREPAIR lies in the procedure TUPLERESOLVE. Indeed, there one needs to (i) consider all possible subsets C of attributes of size k; (ii) for each such C compute the set $\mathcal{V}$ consisting of all possible k-tuples $\bar{v}$ on the attributes in C that satisfy the relevant CFDs; and (iii) obtain from $\mathcal{V}$ the tuple $\hat{v}$ that has minimal cost with $t[C]$ (Fig 7, lines 5–6). To do these tasks efficiently we leverage the use of indices.

LHS-indices. For each CFD $(R : X \to A,\ t_p)$ in Σ we build an index $\mathcal{I}$ for the embedded FD $X \to A$. The index consists of pairs $\langle \mathsf{key}, \mathsf{it} \rangle$ where key uniquely identifies item it in $\mathcal{I}$ and is constructed as follows: if $t_p[A] = a$, then we simply add $\langle t_p[X], a \rangle$ to $\mathcal{I}$; if $t_p[A] =\_$, then we add for each tuple $t' \in \mathsf{Repr}$ such that $t'[X] \asymp t_p[X]$ the pair $\langle t''[X], t''[A] \rangle$ to $\mathcal{I}$. Observe that because Repr is clean, such keys provide indeed a unique identifier.

Now, given a tuple t' and a fixed set of attributes $\mathcal{C}$, we can efficiently determine whether or not a candidate repair $t'' = t'[C/\bar{v}]$ violates a CFD $(R : X \to A,\ t_p)$ in $\Sigma(\mathcal{C} \cup C)$ by (i) searching the index for φ using $t''[X]$ as key; and (ii) testing whether $t''[A]$ matches the returned item. Doing this for all CFDs allows us to compute the number of violations of a candidate repair efficiently.

Finally, these indices are dynamically updated when repairs are added to Repr using standard update mechanisms.

Cost-based indices. We arrange the values of $\mathsf{adom}(\mathsf{Repr}, A)$ for each attribute A in a tree structure, by using a hierarchical agglomerative clustering method [20]. In the tree, "similar" values are grouped together based on the DL metric. Suppose for the moment that we are considering a single attribute A only and want to range over $\mathsf{adom}(\mathsf{Repr}, A)$ such that values are considered in decreasing similarity to a given attribute value $t[A]$. We then simply iterate over $\mathsf{adom}(\mathsf{Repr}, A)$ by first searching for $t[A]$, starting from the root, and then moving to its child cluster that is closest to $t[A]$ in terms of the DL metric. This process then continues until we find a value modification for $t[A]$ that satisfies the requirements given in TUPLERESOLVE. If no suitable candidate can be found, we simply use null. In case of multiple attributes (recall that TUPLERESOLVE tries to find k-tuples), we range over the individual trees in a nested way until a suitable candidate tuple is found. Again, we introduce null whenever no suitable attribute value can be found.

5.3 Applying INCREPAIR in the Non-incremental Setting

Algorithm INCREPAIR can also be used in the non-incremental setting. Indeed, given a dirty database D' one can first extract a maximal consistent set of tuples D from D' and then simply apply INCREPAIR to D and $\Delta D = D' \setminus D$. However, computing such a maximal set of tuples might be too hard in practice:

Proposition 5.4: *It is* NP*-hard to find, given a dataset D' and a set Σ of* CFDs*, a maximal subset C of D' such that $C \models \Sigma$.* □

Proof sketch: This is verified by reduction from the independent set problem, which is NP-complete (cf. [17]). □

Greedy algorithms do provide some approximation guarantees [7] for finding such a set C. However, unless for each CFD $\varphi \in \Sigma$ the number of tuples that violate φ with another tuple is bounded by a small constant, the approximation factor grows with the size of the database [19]. A simpler approach is to compute the set C' of tuples that do not violate *any* constraint in Σ. This clearly does not gives us a maximal set of tuples but as shown in [6] it can be efficiently computed using SQL queries. Moreover, in practice one can often expect this set to be fairly large. Indeed, the typical error rate of real-world data in enterprises is 1%–5% [31].

6. Statistical Methods for Improving Accuracy

In this section we present the third part of the cleaning framework shown in Fig. 3, *i.e.,* the *sampling module*. The repairing algorithms BATCHREPAIR and INCREPAIR both return a repair Repr that satisfies the CFDs in Σ, *i.e.,* consistent *w.r.t.* the given CFDs. However, certain value changes in Repr, which were automatically generated, may not be what the user wants. Referring to Examples 1.1 and 5.1, INCREPAIR (for $k = 3$) resolves the φ_5 by modifying t_5 in the attributes CT, ST and zip, while the user may have wanted to modify t_5[AC] only. This concerns the *accuracy* of the repair, rather than its consistency.

As remarked in Section 3.3, it is unrealistic to consult the user for every change. To improve the accuracy without incurring excessive human efforts, we propose a sampling process. The procedure SAMPLING (not shown) involves the user to inspect and edit *a*

sample of Repr rather than the entire Repr. This procedure ensures that for candidate repairs found by the repairing algorithms, their *estimated inaccuracy rate, i.e.,* $|\mathsf{dif}(\mathsf{Repr}, D_{\mathsf{opt}})|/|D_{\mathsf{opt}}|$, is below a predefined bound ϵ with high confidence δ.

Given a repair Repr and predefined ϵ and δ, procedure SAMPLING works as follows: (1) it draws a sample S from Repr and lets the user inspect S; (2) based on the user feedback and ϵ, it computes a *test statistic* z; and finally (3) it compares z with the critical value z_α at confidence level δ, which is obtained via normal distribution (see, *e.g.,* [1]), where $\alpha = 1 - \delta$. If $z \leq -z_\alpha$, then it rejects the null hypothesis that the proportion of inaccurate data in Repr is above the given ϵ value, and Repr is returned as a candidate repair. Otherwise it recruits the user to edit *both* the sample S and CFDs in Σ. This user interaction may trigger new violations after which the repairing algorithm and sampling process are invoked again, based on the possibly *user-revised* set Σ of CFDs and database.

The objective of SAMPLING is twofold: (i) It involves the users to check whether the repair is accurate enough to meet their expectation on the data quality; and (ii) it allows the repairing algorithms to "learn" from the user interaction and improve the next round of cleaning process. In particular, the user may enter new CFDs based on new semantic bindings of related values.

We next outline methods for drawing a sample and for computing the statistic test. We also discuss the size of the samples required to guarantee with high probability that the inaccuracy ratio is below the predefined ϵ threshold.

Sampling methods. A naive approach is to use uniform random sampling techniques. However, the tuples drawn in this way may not sufficiently *represent* those that were modified by the repairing algorithm, which are the tuples that we would like the user to check since they have a higher likelihood to be inaccurate. This motivates us to employ the stratified sampling method [1].

The idea is to partition the tuples in Repr into multiple strata and draw certain number of tuples from each strata, giving priority to strata that are likely to be inaccurate. More specifically, suppose that we want to draw a sample of k tuples. We partition Repr into m strata $P_1, \ldots, P_m$ with $m < k$. For $i \in [1, m]$, the stratum P_i consists of those tuples t' in Repr such that t' was obtained by the repairing algorithm by modifying a tuple t in the original dataset D with $\mathsf{vio}(t) \geq v_i$, where $\mathsf{vio}(t)$ is the number of violations of t (Section 3.1), and v_i is a fixed threshold. Alternatively, instead of using $\mathsf{vio}(t)$ one can use $\mathsf{cost}(t', t)$ to partition the data set.

We also assume predefined thresholds $\xi_1, \ldots, \xi_m$ such that $\sum_{i\in[1,m]} \xi_i = 1$ and $\xi_i \leq \xi_{i+1}$. Then we draw $\xi_i \cdot k$ many tuples from the stratum P_i. In this way we give a larger coefficient ξ_i to the stratum P_i, and thus draw more tuples from P_i, if tuples in P_i are more likely to be inaccurate. We draw tuples from each P_i by leveraging a widely used algorithm (*e.g.,* [33]) that scans the data in one pass and uses constant space, and let S consist of tuples drawn from all strata.

Statistical Test. Let random variable X denote the number of inaccurate tuples in a sample. Because the probability of having an inaccurate tuple in the sample is proportional to the size of that sample, the variable X obeys a Binomial distribution, which is commonly computed via its normal approximation (provided that the sample size is large enough). Thus we can compute the test statistic by $z = (\hat{p} - \epsilon)/(\sqrt{\frac{\epsilon(1-\epsilon)}{k}})$, where $\hat{p}$ is the inaccuracy rate in a specific sample, ϵ is the predefined inaccuracy rate and k is the sample size. As mentioned earlier, we compare the test statistics z with the critical value z_α at confidence level δ. If $z \leq -z_\alpha$, we can conclude that the inaccuracy rate of Repr is below ϵ with probability δ.

The remaining question is how to compute the inaccuracy rate $\hat{p}$ for a specific sample S. First, we let the user inspect and mark the tuples that fall short of the expectation. From the user feedback we get, for each $i \in [1, m]$, a number e_i, which is the number of inaccurate tuples in those tuples drawn from stratum P_i. The weighted inaccuracy rate $\hat{p}$ of the sample S is computed by: $\hat{p} = (\sum_{i\in[1,m]} e_i \cdot s_i)/(\sum_{i\in[1,m]} |P_i| \cdot s_i)$, where $s_i = |P_i|/(\xi_i \cdot k)$.

Sample size. We next discuss the choice of the size k for the sample S. In general, the lower the inaccurate rate of Repr is, the larger the sample is required. Intuitively, this is because in order for inaccurate tuples to appear in the sample, a large enough sample needs to be taken. A theoretical prediction for sampling size can be derived using Chernoff bounds [1], as follows.

Theorem 6.1: *For a random sample S of size k and a constant c, if $k > \frac{c}{\epsilon} + \frac{1}{\epsilon}\mathsf{ln}(\frac{1}{1-\delta}) + \frac{1}{\epsilon}\sqrt{(\mathsf{ln}(\frac{1}{1-\delta}))^2 + 2 \cdot c \cdot \mathsf{ln}(\frac{1}{1-\delta})}$, then $P[X < c] < 1 - \delta$ holds,* i.e., *the probability that at least c many inaccurate tuples appear in the sample S is no less than δ.* □

Proof sketch: The Chernoff bounds [1] state that for any positive constant $0 \leq \eta \leq 1$, we have $P[X < (1-\eta)k\epsilon] \leq e^{\frac{-k\eta\epsilon^2}{2}}$. By rewriting $P[X < c]$ to $P[X < (1-(1-c/(k\epsilon)))k\epsilon]$, and applying the Chernoff bound result to $P[X < (1-(1-c/(k\epsilon)))k\epsilon] < 1-\delta$, we get the inequality stated in the theorem. □

7. Experimental Evaluation

In this section, we present an experimental study of our repairing algorithms. We investigate the repair quality, scalability, and sensitivity to error rate and types of violations for both BATCHREPAIR and INCREPAIR.

7.1 Experimental Setting

Our experiments were conducted on an Apple Xserve with 2.3GHz PowerPC dual CPU and 4GB of memory; of those, at most 2GB could be used by our system. We used a commercial DBMS on the same machine.

Data and constraints. Our experiments used an extension of the relation shown in Fig. 1. Specifically, its schema models a company's sales records and includes 4 additional attributes, namely, the country of the customer CTY, the tax rate of the item VAT, the title TT and quantity of the item QTT. To populate this table, we scraped real-life data from AMAZON and other websites, and generated datasets of various sizes, ranging from 10k to 300k tuples.

Our set Σ consists of 7 CFDs: 5 taken from Fig. 1 and Fig. 2, together with two new cyclic CFDs.

We included 300–5,000 tuples in the pattern tableaus of these CFDs, enforcing patterns of semantically related values which we identified through analyzing the real data. Note that the set of constraints is fairly large since each pattern tuple is in fact a constraint.

We first populated the table such that the initial datasets are consistent with all the CFDs in Σ. We refer to this "correct" data as D_{opt}. We then introduced noise to attributes in D_{opt} such that each "dirty" tuple violates at least one or more CFDs. To add noise to an attribute, we randomly changed it either to a new value which is close in terms of DL metric (distance between 1 and 6) or to an existing value taken from another tuple. Such "dirty" dataset is referred to as D. We used a parameter ρ ranging from 1% to 10% for the noise rate.

Moreover, in accordance to the cost model defined in Section 3.2 we set weights to the attributes of tuples in D in the following way. Suppose that t is a tuple in D, then we say that A is a "clean" attribute for t if the corresponding tuple t' in D_{opt} agrees with t on attribute A; otherwise we call A "dirty" for t. For dirty attributes

in t, we randomly assign a weight $w(t, A)$ in $[0, a]$; for clean attributes we randomly select a weight $w(t, A)$ in $[b, 1]$. This is based on the assumption that a clean attribute usually has a slightly higher weight than a dirty attribute. In the experiments, we set $a = 0.6$ and $b = 0.5$. We also studied the case when no weight information was available, by setting the weights to 1 for all attributes.

Algorithms. We have implemented prototypes of BATCHREPAIR and all three variants of INCREPAIR, *i.e.,* L-INCREPAIR, V-INCREPAIR and W-INCREPAIR, all in Java. We did not experiment with algorithm SAMPLING because we could easily find out the inaccuracy rate in a repair Repr by comparing the clean data and the repair, since we started with the clean data.

In the experiments we used INCREPAIR to repair the entire data set, as described in Section 5.3, except in one occasion (Fig. 12). That is, L-INCREPAIR, V-INCREPAIR and W-INCREPAIR were applied to non-incremental setting except for Fig. 12.

Measuring repair quality. There is no benchmark algorithm available for repairing CFDs. While each repair Repr of the database D found by our algorithms satisfies all the CFDs (this follows from the correctness of our algorithms), it still may contain two types of errors: (a) the noises that are not fixed, and (b) the new noises introduced in the repairing process. Although it is important to distinguish these two types of errors, the metrics used in previous data cleaning work often considers the first type of errors while ignoring the second type. For example, [5] measures *the percentage of error corrected*, which does not distinguish these two types of errors.

To measure these two types of errors, we used the notions of *Precision* and *Recall*, which are widely used in information retrieval and many other areas. *Precision* is the ratio of the number of correctly repaired noises to the number of changes made by the repairing algorithm. It measures the repair correctness. *Recall* is the ratio of the number of correctly repaired noises to the total number of noises. It measures repair completeness. For a dirty dataset D and a Repr found by our algorithms, we compute the number of noises by $\mathsf{dif}(D, D_{\mathsf{opt}})$ (recall that we know D_{opt}). The number of changes made by the repairing algorithm is $\mathsf{dif}(D, \mathsf{Repr})$ and the number of noises correctly repaired is $\mathsf{dif}(D, \mathsf{Repr}) - \mathsf{dif}(D_{\mathsf{opt}}, \mathsf{Repr})$. Note that our algorithm may change some values to null. If such a value before the change is correct, we count the null as an error; otherwise, we treat it as a correction.

7.2 Experimental Results

We now report our findings concerning the accuracy (Precision/Recall) of our algorithms, their scalability in terms of the size of the data, noise rates, and types of violations, and show the efficacy of CFDs vs. FDs in repairing data.

Efficacy of CFDs vs. FDs. We first show that CFDs are indeed more effective than FDs in repairing dirty data. In Fig. 8, we ran BATCHREPAIR on a dataset of 60K tuples and varied the noise rate ρ between 2% to 10%. The upper two curves report the accuracy for our set of CFDs. The lower two curves show the accuracy for the embedded FDs (*i.e.,* the CFDs in which the pattern tableau consists of a single pattern of wildcards only). Figure 8 shows that patterns improved significantly the accuracy of the repair.

Quality of the repair. We evaluated the data quality of our repairing algorithms. We show the accuracy in terms of *Precision* (Fig. 9) and *Recall* (Fig. 10) of all our algorithms, *i.e.,* BATCHREPAIR, L-INCREPAIR, V-INCREPAIR and W-INCREPAIR. In these experiments, we varied the noise rate ρ from 1% to 10%. The total database size was fixed at 60K tuples.

Our experiments show that V-INCREPAIR and W-INCREPAIR consistently outperform L-INCREPAIR, while W-INCREPAIR performs slightly better than V-INCREPAIR. The accuracy of W-INCREPAIR is influenced by the quality of the weights, *i.e.,* the choice of a and b. The good performance of V-INCREPAIR is consistent with the expectation that a tuple which has less violations is more likely be a correct tuple. Indeed, algorithm V-INCREPAIR first repairs tuples that are more likely to be correct, which will provide more reliable information when cleaning less accurate dirty tuples subsequently. A similar argument holds for the good accuracy of W-INCREPAIR. Moreover, the running times (Fig. 13) of L-INCREPAIR and W-INCREPAIR are similar and slightly better than V-INCREPAIR. Therefore, the improved quality of the latter two algorithms *does not* come at a price, in terms of time.

Also in Fig. 9 and Fig. 10 we show the accuracy of the repair given by BATCHREPAIR. Although BATCHREPAIR and INCREPAIR are different in nature, the quality of the repairs provided by them is comparable. Note also that the *Precision* and *Recall* decrease slightly with the increase of noise rate, as expected. The values of *Recall* are relatively high, which means that our algorithms can repair most of the errors. *Precision* shows that new noises were introduced when repairing these errors.

In the following, when reporting on the INCREPAIR algorithm we always used V-INCREPAIR, as it consistently gave good results for a wide range of (a, b)-values.

In Fig. 14 we verify our intuition that CFDs with a constant in their RHS are more informative during the repairing than those with a variable RHS. In this experiment we fixed the size of the data to 60K tuples and varied the percentage of violations for constant CFDs *w.r.t.* violations for variable CFDs from 20% to 80%. As can be seen, an increasing number of constant CFD violations enabled both BATCHREPAIR and INCREPAIR to achieve higher accuracy.

Scalability. In the following experiments we investigate the scalability of our algorithms. In Fig. 11 we show the scalability of BATCHREPAIR. As described in Section 4, the overall complexity is governed by the procedure PICKNEXT. We found in our experiments that without any further optimization, BATCHREPAIR runs very slow. Therefore, we applied some additional optimizations based on the dependency graph of the CFDs, which help PICKNEXT to select the next CFD to repair. As Fig. 11 shows, the optimized BATCHREPAIR scales very well for database sizes varying from 60K to 300K tuples. The noise rate was fixed at 5%.

The effectiveness of INCREPAIR, when used in the *incremental* setting, is reported in Fig. 12. We started from a clean database consisting of 60K tuples and inserted 10 to 70 dirty tuples. It shows that INCREPAIR significantly outperforms BATCHREPAIR in this incremental setting, with comparable accuracy (see Figs. 9 and 10). Observe that the running time of INCREPAIR increases faster than that of BATCHREPAIR.

The scalability of all our algorithms with respect to noise rate is shown in Fig. 13. We fixed the data size to 60K tuples and varied the noise rate from 1% to 10%. All algorithms require more time when the data has more noise, as expected. An interesting observation is that BATCHREPAIR is less sensitive to the noise rate because it can repair many tuples simultaneously.

In Fig. 15 we show that the presence of violations for variable CFDs has a negative effect on the time performance of both BATCHREPAIR and INCREPAIR. This is not surprising since such violations involve multiple tuples.

Summary. Our experimental results demonstrate both the effectiveness and efficiency of our repairing algorithms. (1) We find that all of our repairing algorithms, even the worst-performed L-INCREPAIR, improve the quality of the data. (2) All of our algorithms scale well with the database size. (3) Algorithms BATCHREPAIR and V-INCREPAIR provide repairs that have comparable accuracy. (4) Repair quality decreases when the noise rate increases

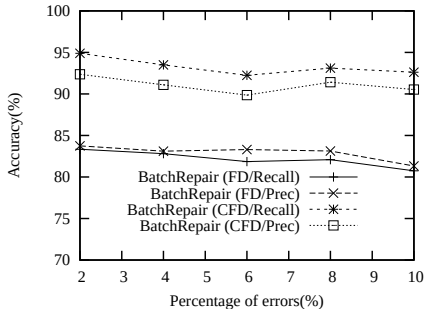

Figure 8: Efficacy of CFDs vs. FDs

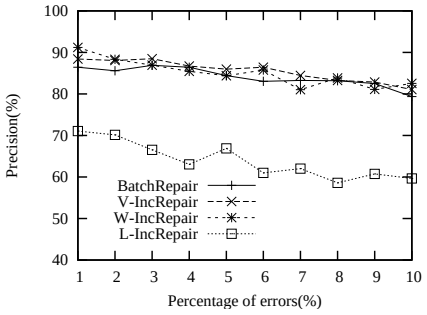

Figure 9: Precision vs. noise rate

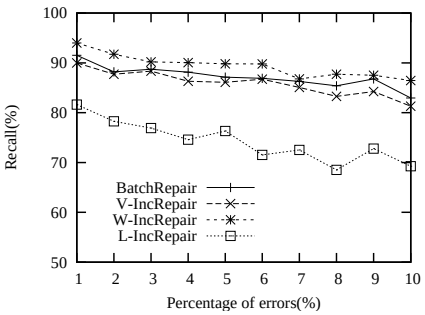

Figure 10: Recall vs. noise rate

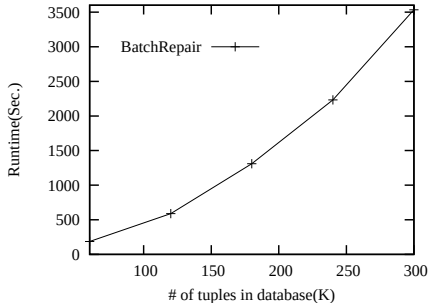

Figure 11: Scalability of BATCHREPAIR

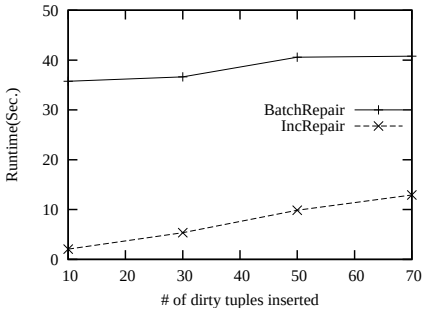

Figure 12: Scalability of INCREPAIR

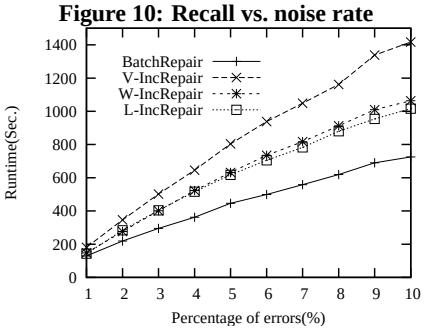

Figure 13: Scalability vs. noise rate

for all of the algorithms. (5) If violations are mainly caused by constant CFDs, then the algorithms run more efficiently and provide more accurate results. (6) While our algorithms correctly fix noises, they may also introduce new noises. This is an issue not yet well studied by previous work.

8. Related Work

A variety of constraint formalisms have been proposed [6, 4, 8, 26, 27]. Except for [6], these formalisms have not been applied in the context of data cleaning. CFDs are proposed in [6], which studies satisfiability and implication analyses of CFDs, and gives SQL techniques for detecting inconsistencies using CFDs. However, it does not propose cleaning methods. Constraints of [8], also referred to as conditional functional dependencies, and their extension known as constrained dependencies of [26], also restrict an FD to hold on a subset of a relation. However, they cannot express even CFDs. More expressive are constraint-generating dependencies (CGDs) of [4] and constrained tuple-generating dependencies (CTGDs) of [27]. While both CGDs and CTGDs can express CFDs, this expressive power comes with the price of high complexity.

Research on constraint-based data cleaning has mostly focused on two topics introduced in [2]: *repair* is to find another database that is consistent and minimally differs from the original database (*e.g.*, [2, 5, 25, 9, 10, 14]); and *consistent query answer* is to find an answer to a given query in every repair of the original database (*e.g.*, [2, 10, 24, 34]). Most earlier work (except [5, 9, 14, 34]) considers traditional full and denial dependencies, which subsume FDs, but do not consider patterns defined with data values. Beyond traditional dependencies, logic programming is studied in [9, 14] for fixing census data. A tableau representation of full dependencies with data values is studied in [34], which focuses on condensed representation of repairs and consistent query answers.

Closest to our work is [5]. Here, a cost model and repairing algorithms are developed for standard FDs and INDs. Our cost model (Section 3.2) is an extension of the one proposed in [5], by allowing weights to be associated with attributes rather than with tuples. As remarked earlier, repairing CFDs is far more intriguing than standard FDs. Our batch repairing algorithm (Section 4) is a nontrivial extension of the algorithms of [5] in that both are based on equivalence classes of tuple attributes, but the algorithms of [5] may not terminate on CFDs. Incremental repairing and sampling for improving data accuracy (Sections 5 and 6) are not considered in [5].

Value modifications as repair operations are used in [13, 14, 34, 5, 25, 24]. A method for cleaning census data, based on reduction to MWSC, was proposed in [13] and has been being used by US national statistical agents [35]. Our heuristic REPAIR-CFD is inspired by [13], but differs from it in that [13, 35] deal with editing rules on individual records among which there is no interaction, whereas modifying a single tuple may lead to violations CFDs by multiple other tuples. The repair algorithms of [25] are essentially an extension of the method of [13] for restricted denial constraints. As remarked earlier, [34, 24] focus on consistent query answer rather than repair. [14] employs logic programming to clean census data and is quite different from the techniques developed in this work.

There has been a host of work on the merge-purge problem (*e.g.*, [15, 21, 28]) for the elimination of *approximate duplicates*. As observed in [5], it is possible to model many cases of this problem in terms of FDs and INDs repair. As shown in Section 5.2, clustering techniques developed for merge-purge have immediate applications in constraint-based data cleaning. There have also been commercial ETL (extraction, transformation, loading) tools, in which a large portion of the cleaning work has still to be done manually or by low-level programs (see [29] for a comprehensive survey).

Related to this work are also the AJAX, Potter's Wheel and ARKTOS systems. AJAX [15] proposes a declarative language for specifying data cleaning operations (duplicate elimination) during data transformations. Potter's Wheel [30] is an interactive data cleaning system, which supports a sliding-window interface, and combines data transformations and error detection (syntax and irregularities). ARKTOS [32] is an ETL tool that detects inconsistencies based on basic keys, foreign keys and uniqueness constraints, etc., but it makes little effort to remove the detected errors. While a constraint repair facility will logically become part of the cleaning process supported by these tools and systems, we are not aware of analogous functionality currently in any of the systems mentioned.

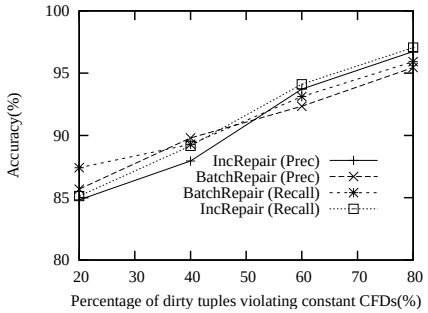

Figure 14: Accuracy vs. percentage of constant CFD violations

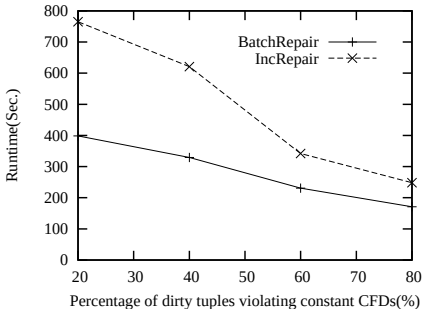

Figure 15: Time vs percentage of constant CFD violations

9. Conclusions

We have proposed a framework for improving data quality, based on CFDs. We have shown that the problem for finding optimal repairs and the problem for incrementally finding optimal repairs are both NP-complete. In light of these intractability results, we have developed heuristic algorithms for both problems, and experimentally verified their effectiveness and efficiency in improving the consistency of the data. To improve the accuracy of the data, we have proposed a statistical method that guarantees to find a repair above a predefined accuracy rate with a high confidence. To our knowledge, this work is among the first treatments of both consistency and accuracy, and is the first effort to (incrementally) clean data based on conditional constraints. We expect that CFDs and data-cleaning methods based on CFDs will yield a promising tool for improving the quality of real-life data.

Several extensions are targeted for future work. First, to effectively clean real-life data, it is often necessary to consider both CFDs and inclusion dependencies [5]. We are investigating effective methods for improving the consistency and accuracy of the data based on both CFDs and inclusion dependencies. Second, we are studying effective methods to automatically discover useful CFDs from real-life data. Finally, we exploring conditional constraints beyond CFDs.

Acknowledgments. Wenfei Fan is supported in part by EPSRC GR/S63205/01, GR/T27433/01, EP/E029213/1 and BBSRC BB/D006473/1. Floris Geerts is a postdoctoral researcher of the FWO Vlaanderen and is supported in part by EPSRC GR/S63205/01.

10. References

[1] N. Alon and J. H. Spencer. *"The Probabilistic Method"*. John Wiley Inc., 1992.

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

[3] O. Bailleux and P. Marquis. DISTANCE-SAT: Complexity and algorithms. In *AAAI/IAAI*, 1999.

[4] M. Baudinet, J. Chomicki, and P. Wolper. Constraint-Generating Dependencies. *JCSS*, 59(1):94–115, 1999.

[5] 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.

[6] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *ICDE*, 2007.

[7] R. Boppana and M. M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. *BIT*, 32(2):180–196, 1992.

[8] P. D. Bra and J. Paredaens. Conditional dependencies for horizontal decompositions. In *Colloquium on Automata, Languages and Programming*, 1983.

[9] R. Bruni and A. Sassano. Errors detection and correction in large scale data collecting. In *IDA*, 2001.

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

[11] W. Cohen, P. Ravikumar, and S. Feinberg. A comparison of string-distance metrics for name-matching tasks. In *IIWeb*, 2003.

[12] L. English. Plain English on data quality: Information quality management: The next frontier. *DM Review Magazine*, April 2000.

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

[14] E. Franconi, A. L. Palma, N. Leone, S. Perri, and F. Scarcello. Census data repair: a challenging application of disjunctive logic programming. In *LPAR*, 2001.

[15] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. AJAX: An extensible data cleaning tool. In *SIGMOD*, 2001.

[16] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Declarative data cleaning: Language, model and algorithms. In *VLDB*, 2001.

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

[18] G. Grahne. *The Problem of Incomplete Information in Relational Databases*. Springer, 1991.

[19] M. Halldórsson and J. Radhakrishnan. Greed is good: approximating independent sets in sparse and bounded-degree graphs. In *STOC*, 1994.

[20] J. Han and M. Kamber. *"Data Mining: Concepts and Techniques"*. Morgan Kaufmann Publishers, 2006.

[21] M. A. Hernandez and S. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[22] T. Imieliński and W. L. Jr. Incomplete information in relational databases. *JACM*, 31(4):761–791, 1984.

[23] International Standard ISO/IEC 9075-2:2003(E). Information technology: Database languages, SQL Part 2 (Foundation, 2nd edition), 2003.

[24] A. Lopatenko and L. Bertossi. Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. In *ICDT*, 2007.

[25] A. Lopatenko and L. Bravo. Efficient approximation algorithms for repairing inconsistent databases. In *ICDE*, 2007.

[26] M. J. Maher. Constrained dependencies. *Theoretical Computer Science*, 173(1):113–149, 1997.

[27] M. J. Maher and D. Srivastava. Chasing Constrained Tuple-Generating Dependencies. In *PODS*, 1996.

[28] A. Monge. Matching algorithm within a duplicate detection system. *IEEE Data Engineering Bulletin*, 23(4), 2000.

[29] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4), 2000.

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

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

[32] P. Vassiliadis, Z. Vagena, S. Skiadopoulos, N. Karayannidis, and T. Sellis. ARKTOS: towards the modeling, design, control and execution of ETL processes. *Inf. Syst.*, 8:537–561, 2001.

[33] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1), 1985.

[34] J. Wijsen. Condensed representation of database repairs for consistent query answering. In *ICDT*, 2003.

[35] W. E. Winkler. Methods for evaluating and creating data quality. *Inf. Syst.*, 29(7):531–550, 2004.