The VLDB Journal (2014) 23:103–128
DOI 10.1007/s00778-013-0316-z

REGULAR PAPER

Sampling from repairs of conditional functional dependency violations

George Beskales · Ihab F. Ilyas · Lukasz Golab · Artur Galiullin

Received: 1 July 2012 / Revised: 8 March 2013 / Accepted: 9 April 2013 / Published online: 26 April 2013
© Springer-Verlag Berlin Heidelberg 2013

Abstract Violations of functional dependencies (FDs) and conditional functional dependencies (CFDs) are common in practice, often indicating deviations from the intended data semantics. These violations arise in many contexts such as data integration and Web data extraction. Resolving these violations is challenging for a variety of reasons, one of them being the exponential number of possible repairs. Most of the previous work has tackled this problem by producing a single repair that is nearly optimal with respect to some metric. In this paper, we propose a novel data cleaning approach that is not limited to finding a single repair, namely sampling from the space of possible repairs. We give several motivating scenarios where sampling from the space of CFD repairs is desirable, we propose a new class of useful repairs, and we present an algorithm that randomly samples from this space in an efficient way. We also show how to restrict the space of repairs based on constraints that reflect the accuracy of different parts of the database. We experimentally evaluate our algorithms against previous approaches to show the utility and efficiency of our approach.

Keywords Data cleaning · Conditional functional dependency · Integrity constraint violation · Sampling repairs

G. Beskales (✉) · I. F. Ilyas
Qatar Computing Research Institute, Doha, Qatar
e-mail: georgeanwar@gmail.com; gbeskales@qf.org.qa

I. F. Ilyas
e-mail: ikaldas@qf.org.qa

L. Golab · A. Galiullin
University of Waterloo, Waterloo, Canada
e-mail: lgolab@uwaterloo.ca

A. Galiullin
e-mail: agaliullin@uwaterloo.ca

1 Introduction

Data quality is a key requirement for effective data analysis and processing. In many situations, the quality of business and scientific data is impaired by various sources of noise (e.g., heterogeneity of data formats, imperfection of information extractors, and imprecision of data generating devices). This leads to data quality problems such as missing values [14,26], violated integrity constraints [4,9,23], and duplicate records [16,27]. These problems cost enterprises billions of dollars annually and may have unpredictable consequences in mission-critical tasks [15]. Databases that experience data quality problems are usually referred to as unclean/dirty databases. The process of data cleaning refers to detecting and correcting errors in the data. A great deal of effort has been directed to improving the effectiveness and efficiency of data cleaning.

Functional dependencies (FDs) can be thought of as integrity constraints that encode data semantics. In that sense, violations of FDs indicate deviations from the expected semantics, possibly caused by data quality problems. In practice, FDs tend to break after integrating heterogeneous data or extracting data from the Web. Even in a traditional database, unknown FDs may be hidden in a complex evolving schema, or the database administrator may choose not to enforce some FDs for various reasons. For example, Fig. 1 shows a database instance and a set of FDs, some of which are violated (e.g., tuples t_2 and t_3 violate `ZIP` $\rightarrow$ `City`, tuples t_2 and t_3 violate `Name` $\rightarrow$ `SSN, City`, and tuples t_1 and t_4 violate `ZIP` $\rightarrow$ `State, City`).

There may be many ways to modify a table so that it satisfies a set of FDs. One is to delete the offending tuples (ideally, delete the fewest possible such tuples) such that the remainder satisfies all the FDs [10,11]. For example, we can repair the relation instance in Fig. 1 by deleting t_1 and t_3. However,

Input Instance

	SSN	Name	City	State	ZIP
t_1	72163	John Smith	Chicago	IL	90101
t_2	87991	Mark Green	LA	CA	90065
t_3	87891	Mark Green	Los Angeles	CA	90065
t_4	23212	Mary Clarke	LA	CA	90101

Functional Dependencies:
SSN → Name, City, State, ZIP
Name → SSN, City, State, ZIP
ZIP → State, City

Repair 1

SSN	Name	City	State	ZIP
72163	John Smith	LA	CA	90101
87991	Mark Green	LA	CA	90065
87891	?	Los Angeles	CA	?
23212	Mary Clarke	LA	CA	90101

Repair 2

SSN	Name	City	State	ZIP
72163	John Smith	Chicago	IL	?
87891	Mark Green	LA	CA	90065
87891	Mark Green	LA	CA	90065
23212	Mary Clarke	LA	CA	90101

Fig. 1 An inconsistent database and possible repairs

deleting an entire tuple may result in loss of "clean" information iff only one of its attribute values is incorrect. Alternatively, we can modify selected attribute values (we do not consider adding new tuples as this would not fix any existing violations). For example, Fig. 1 shows two possible repairs obtained by modifying some attribute values; question marks indicate that an attribute value (to which we refer as a *cell*) can be modified to one of several values in order to satisfy the FDs.

One extension of FDs is conditional functional dependencies (CFDs), which are frequently used in the context of data cleaning [8,12,17]. A CFD consists of a template FD and a tableau that contains a set of patterns. The role of the patterns is restricting the scope of the template FD to a specific set of tuples in the database instance and/or restricting the right-hand-side (RHS) attribute of the template FD to a certain constant. Clearly, FDs represent a subclass of CFDs.

For instance, an example CFD defined on the database in Fig. 1 is (`City`→ `State`, (`'LA'`, `'CA'`)). This CFD indicates that FD `City`→`State` holds for tuples with `City = 'LA'`. Also, `State` must be equal to `'CA'` for these tuples. Another CFD is (`State, ZIP` → `City`, (`'MI'`,_,_)), which indicates that the FD `State, ZIP`→`City` holds for tuples with `State = 'MI'`.

In this paper, we present a novel approach to resolving violations of FDs and CFDs, which is to sample from the space of possible repairs. Our technique is complementary to existing data quality and cleaning tools, and, as we will show, it is useful in various practical situations.

1.1 Motivating examples

Independently of how we choose to repair constraint violations, different repair frameworks have appeared in previous work. One approach is to produce a single, nearly optimal repair, in terms of the number of deletions or attribute modifications (e.g., [9,12,23]). For instance, we might prefer `Repair 2` in Fig. 1 because it makes fewer modifications. The main shortcoming of this approach is that all other possible repairs, including other minimal repairs, are discarded.

A second approach—consistent query answering—computes answers to selected classes of queries that are valid in every "reasonable" repair [4,10,11,20,30,31]. In Fig. 1, a consistent answer of the query that selects all tuples with ZIP code 90101, with respect to the two illustrated repairs, is $\{t_4\}$. However, consistent query answering may produce empty results iff there are multiple ways of repairing the same tuple.

A third approach is to have a domain expert manually clean the data. Unfortunately, this approach does not scale well with the data size and requires constant attention from the expert.

We argue that one-shot cleaning algorithms and consistent query answering do not address the needs of at least the following applications.

Interactive data cleaning Consider an interactive data cleaning process, where several possible CFD repairs (of a whole table, a subset of a table, or a single tuple) are suggested to the user. The user may then perform some of the suggested repairs and request a new set of suggestions, in which previously performed repairs do not change. For example, in Fig. 1, `Repair 1` and `Repair 2` provide two alternatives for modifying each tuple in the database. A user might prefer changing t_1 according to `Repair 1` and prefer changing t_2 according to `Repair 2`. Note that this application is not tied to a specific query, so consistent answers are not suitable. Moreover, the application requires several suggested repairs, but not necessarily all possible repairs, to be generated at any given time. Hence, computing a single repair is not sufficient.

Uncertain query answering We can generalize the notion of consistent query answering to an approach that computes probabilistic query answers as though each possible repair was a possible world. Even if generating all repairs is intractable, computing a subset of the possible repairs may be sufficient to obtain meaningful answers. One example of such a framework is the Monte Carlo Database (MCDB) [22]. Again, computing a single repair or a consistent query answer is not sufficient for this application.

1.2 Challenges and contributions

Our motivating applications have the following requirements and challenges in common.

– Due to the exponential space of possible FD and CFD repairs, we may not be able to, or may not want to, generate all possible repairs. Instead, the challenge lies in finding a meaningful subset of repairs that is sufficiently large and can be generated in an efficient way.

- We need to ensure that the constraints that reflect the user's confidence in the data (e.g., specifying which cells must remain unchanged) are satisfied during the repairing process.

In this paper, we propose a novel data cleaning technique that accommodates our motivating applications and addresses the above challenges. Our approach is based on efficiently generating a sample from a meaningful repair space. Our contributions in this paper are as follows.

- We introduce a novel space of possible repairs, called cardinality-set-minimal, that combines the advantages of two existing spaces: set-minimal and cardinality-minimal.
- We give an efficient algorithm for generating a sample of cardinality-set-minimal repairs of FD violations. A major challenge here is the interplay among violations of FDs: repairing a tuple that violates one FD may introduce a new violation of another FD. Note that although existing heuristics for finding a single nearly optimal repair may be modified to generate multiple random repairs, they do not give any guarantees on the space of generated repairs (more details in Sect. 9).
- We introduce a mechanism that partitions the input instance into disjoint blocks that can be repaired independently in order to significantly improve the efficiency of repair sampling.
- We describe a modification of our approach that allows users to specify constraints on the set of cells that reflect the user's confidence in the accuracy of data. We use a confidence model that is different from previous work (e.g., [9,12,23]), where database tuples are associated with weights reflecting their accuracy (refer to Sects. 3, 5.3, and 9.2).
- We extend our sampling algorithm to generate repairs of CFD violations.

We also conduct an experimental study to show the scalability of our repair sampling technique.

The remainder of the paper is organized as follows. In Sect. 2, we describe the notation used in this paper. In Sect. 3, we define our space of possible repairs. In Sect. 4, we discuss repairing violations of a single FD. In Sect. 5, we introduce our approach to sample from the new space of possible repairs for violations of multiple FDs and we show how to enforce user-defined hard constraints. In Sect. 6, we improve the efficiency of the sampling algorithm by partitioning the data into separately repairable blocks. In Sect. 7, we extend our sampling algorithm to support CFDs. In Sect. 8, we present an experimental study of our sampling approach. In Sect. 9, we discuss related work and explain why previous data cleaning algorithms cannot be extended to generate a sample from a well-defined space of repairs. We conclude the paper with final remarks in Sect. 10.

2 Notation and definitions

Let R be a relation schema consisting of m attributes, denoted $(A_1, \ldots, A_m)$. We use the notation $A_1, A_2, \ldots, A_k$ to represent the union of the concatenated attributes (i.e, $\{A_1, A_2, \ldots, A_k\}$). Let $Dom(A)$ be the domain of an attribute A. We denote by I an instance of R consisting of n tuples, each of which belongs to the domain $Dom(A_1) \times \ldots \times Dom(A_m)$ and has a unique identifier. We denote by $TIDs(I)$ the identifiers of tuples in I and use the terms "tuple" and "tuple identifier" interchangeably. Let $ADom(A, I)$ be the *active domain* of attribute A in I, which is defined as the set of values of attribute A that appear in tuples of I (i.e., $ADom(A, I) = \Pi_A(I)$).

We refer to an attribute $A \in R$ of a tuple $t \in TIDs(I)$ as a *cell*. Each cell C is identified by a pair (t, A) consisting of the tuple $t \in TIDs(I)$ and the attribute $A \in R$. For a set of attributes $X \subseteq R$, we denote by (t, X) the set of cells $\{(t, A) : A \in X\}$. We denote by $CIDs(I) = \{(t, A) : t \in TIDs(I), A \in R\}$ the set of all cell identifiers in I. We denote by $I(t, A)$ the value of a cell (t, A) in an instance I. If the instance I is clear from the context, we write $t[A]$ instead of $I(t, A)$.

For two attribute sets $X, Y \subseteq R$, an FD $X \to Y$ holds on an instance I, denoted $I \models X \to Y$, iff for every two tuples t_1, t_2 in I such that $t_1[X] = t_2[X]$, $t_1[Y] = t_2[Y]$. The set of FDs defined over R is denoted as Σ. We assume that Σ is minimal and in canonical form [2]; each FD is in the form $X \to A$, where $X \subseteq R$ and $A \in R$. I is inconsistent with respect to Σ iff I violates at least one FD in Σ. For an FD $X \to A$, we refer to X as the left-hand-side (LHS) attributes, and we refer to A as the RHS attribute.

A recent generalization of FDs, named CFDs, has been proposed in [8]. CFDs are regular FDs that are defined only on a subset of tuples. A CFD is defined as a pair $(X \to A, t_c)$, where $X \to A$ is an FD, and t_c is a (pattern) tuple whose attributes are XA. Each attribute of t_c can be either a constant or an unnamed variable '_'. An instance tuple t *matches* pattern tuple t_c on X, written $t[X] \asymp t_c[X]$, iff$\forall B \in X(t_c[B] = t[B] \vee t_c[B] = _)$. CFDs are divided into two variants: variable CFDs, where $t_c[A] = _$, and constant CFDs, where $t_c[A]$ is a constant. A variable CFD $(X \to A, t_c)$ indicates that for any two tuples t_1, t_2, $t_1[X] = t_2[X] \asymp t_c[X] \to t_1[A] = t_2[A]$. A constant CFD $(X \to A, t_c)$ indicates that for each tuple t, $t[X] \asymp t_c[X] \to t[A] = t_c[A]$. Without loss of generality, we assume that for a constant CFD $(X \to A, t_c)$, all values in $t_p[X]$ are constant [17].

For example, consider a relation `Address (StreetNumber, StreetName, City, Country, PostalCode)`. A constant CFD defined over `Address` is (`PostalCode` → `City`, `(N2L3G1, Waterloo)`), which indicates that for all tuples with `PostalCode = N2L3G1`, `City` must be equal to `Waterloo`. An example of a variable CFD on relation `Address` is (`Country`, `PostalCode` → `StreetName`, (`UK`, _, _)), which indicates that for pairs of tuples with `Country = UK` and equal values of `PostalCode`, their `StreetName` values must be equal.

In general, there is a large number of FD repairs for a given database instance. In this paper, we use the notion of V-instances, which was introduced in [23], to concisely represent data instances. In V-instances, cells can be either set to constants or to variables that can be instantiated in a specific way.

Definition 1 (*V-instance*) Given a set of variables $\{v_1^A, v_2^A, \ldots\}$ for each attribute $A \in R$, a V-instance of R is an instance of R where each cell (t, A) can be assigned to either a constant in $Dom(A)$ or a variable from the set $\{v_1^A, v_2^A, \ldots\}$.

A V-instance I represents multiple ground (i.e., variable free) instances of R that can be obtained by assigning variables in I as follows. Each variable v_i^A in attribute A in I can be assigned to any value from $Dom(A)\backslash ADom(A, I)$ and such that no two distinct variables v_i^A and v_j^A have equal values.

Finding at least one ground instance for a V-instance is always possible iff the number of domain values for each attribute is larger than the number of tuples in the input database (e.g., in case of unbounded and high-cardinality domains). In this paper, we assume that all attributes satisfy this condition. Handling the case in which attribute domains are bounded and have low cardinality is left for future work.

The main use of variables in the context of repairing FD violations is representing unknown values that emerge from modifying the LHS attributes of a violated FD. In the remainder of the paper, we refer to a V-instance as simply an instance.

3 Spaces of possible repairs

A repair of an inconsistent instance I with respect to a set of FDs Σ is another instance I' that satisfies Σ. In general, violations of FDs can be repaired by either deleting erroneous tuples (e.g., [10,24]) or changing erroneous cells (e.g., [9,23]). In this paper, we only consider repairs obtained by modifying tuple attributes. An FD repair is formally defined as follows:

Definition 2 (*FD repair*) Given a set of FDs Σ defined over a relation R, and an instance I of R that does not satisfy Σ, a repair of I is another instance I' of R such that $I' \models \Sigma$, $TIDs(I) = TIDs(I')$ and $CIDs(I) = CIDs(I')$.

That is, a repair I' of an inconsistent instance I is an instance that satisfies Σ and has the same set of tuple and cell identifiers (i.e., tuples or columns are not added or deleted).

We denote by $Repairs(I, \Sigma)$ the set of all possible repairs of an instance I w.r.t. Σ. Let $\Delta(I, I')$ be the identifiers of cells that have different values in I and I', that is, $\Delta(I, I') = \{C \in CIDs(I) : I(C) \neq I'(C)\}$. For example, in Fig. 2, $\Delta(I, I_2) = \{(t_2, B), (t_3, B)\}$. Also, we denote by $\lambda(I, I')$ the set of changes made in I in order to obtain I', where each change is represented as a pair of a cell and the new value assigned to this cell in I'. Formally, $\lambda(I, I') = \{(C, x) : I(C) \neq I'(C) \wedge x = I'(C)\}$. For example, in Fig. 2, $\lambda(I, I_4) = \{((t_1, A), 7), ((t_1, B), 3)\}$.

It is useful to filter out repairs that are less likely to represent the actual clean database. A widely used criterion is the *minimality of changes* (e.g., [9–12,21,23]). Frequently used definitions for minimality of changes are described as follows.

Definition 3 (*Cardinality-minimal repair* [23]) A repair I' of I is cardinality-minimal iff there is no repair I'' of I such that $|\Delta(I, I'')| < |\Delta(I, I')|$.

That is, a repair I' of I is cardinality-minimal iff the number of changed cells in I' is the minimum across all repairs of I.

A weighted version of the cardinality-minimal repairs is used in [9,12]. Each cell C is associated with a weight in the range [0, 1], denoted $w(C)$, reflecting the confidence placed by user in the accuracy of C. Also, the distance between any two values x and y is measured using a distance function $dis(x, y)$. The cost of a repair I' of I is defined as follows.

$$cost(I, I') = \sum_{C \in \Delta(I, I')} w(C) \cdot dis\left(I(C), I'(C)\right) \tag{1}$$

Definition 4 (*Cost-minimal repair* [9]) A repair I' of I is cost-minimal iff there is no repair I'' of I such that $cost(I, I'') < cost(I, I')$.

Another definition of minimal repairs uses set-containment for describing minimality of changes.

Definition 5 (*Set-minimal repair* [4,24]) A repair I' of I is set-minimal iff there is no repair I'' of I such that $\lambda(I, I'') \subset \lambda(I, I')$.

That is, a repair I' of I is set-minimal if no strict subset of the changed cells in I' can be reverted to their original values in I without violating Σ. Note that we use the symbol $\subset$ to indicate strict (proper) subset (also written as $\subsetneq$ in other publications).

Previous approaches that generate a single repair of an inconsistent relation instance typically find a nearly optimal

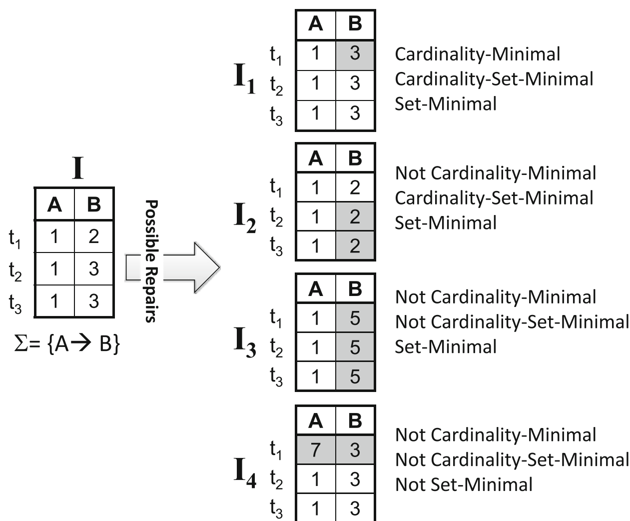

Fig. 2 Examples of various types of repairs

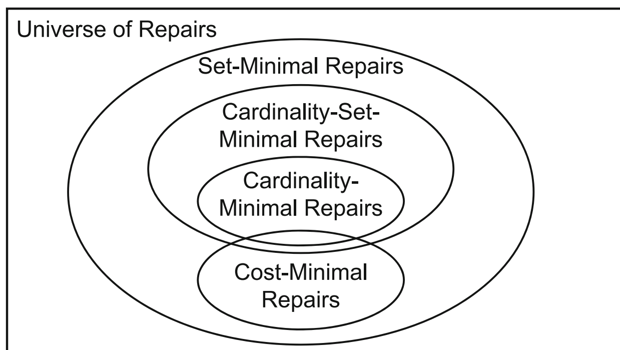

Fig. 3 The relationship between spaces of possible repairs

cost-minimal or cardinality-minimal repair (finding a cost-minimal or a cardinality-minimal repair is NP-hard [9,10,23]). In contrast, prior work on consistent query answering considers set-minimal repairs [11,21]. Repairs that are not set-minimal are not desirable since they involve unnecessary changes [4,11,24].

We introduce a novel space of repairs called cardinality-set-minimal. Our goal is to provide a space of repairs that can be sampled in an efficient manner and is neither too restrictive (e.g., cost-minimal or cardinality-minimal) nor too large (e.g., set-minimal).

Definition 6 (*Cardinality-set-minimal repair*) A repair I' of I is cardinality-set-minimal iff there is no repair I'' of I such that $\Delta(I, I'') \subset \Delta(I, I')$.

That is, a repair I' of I is cardinality-set-minimal iff no subset $\mathcal{C}$ of the changed cells in I' can be reverted to their original values in I without violating Σ, even if we allow modifying the cells in $\Delta(I, I')\backslash\mathcal{C}$ to other values.

In Fig. 2, we show various types of repairs of an instance I, with the changed cells grayed out. Repair I_1 is cardinality-minimal because no other repair has fewer changed cells. Repair I_1 is also cardinality-set-minimal and set-minimal. Repairs I_2 and I_3 are set-minimal because reverting any subset of the changed cells to the values in I will violate $A \rightarrow B$. On the other hand, I_3 is not cardinality-set-minimal (or cardinality-minimal) because reverting $t_2[B]$ and $t_3[B]$ back to 3 and changing $t_1[B]$ to 3 instead of 5 gives a repair of I, which is the same as I_1. Repair I_4 is not set-minimal because I_4 still satisfies $A \rightarrow B$ after reverting $t_1[A]$ to 1. The relationship among various minimal repairs is depicted in Fig. 3 and described in the following lemma.

Lemma 1 *The set of cardinality-minimal repairs is a subset of cardinality-set-minimal repairs. The set of cardinality-set-minimal repairs is a subset of set-minimal repairs. Finally, the set of cost-minimal repairs is a subset of set-minimal repairs if for each cell $C \in CIDs(I)$, $w(C) > 0$.*

Proof For any two repairs I' and I'' of I,

$\Delta(I, I'') \subset \Delta(I, I') \rightarrow |\Delta(I, I'')| < |\Delta(I, I')|$

This implies that for any repair I' of I,

$$\nexists I'' \in Repairs(I, \Sigma)\ \big(|\Delta(I, I'')| < |\Delta(I, I')|\big)$$
$$\rightarrow \nexists I'' \in Repairs(I, \Sigma)\ \big(\Delta(I, I'') \subset \Delta(I, I')\big)$$

Therefore, if I' is a cardinality-minimal repair, I' is cardinality-set-minimal.

Similarly, for any two repairs I' and I'' of I,

$$\lambda(I, I'') \subset \lambda(I, I') \leftrightarrow$$
$$\Delta(I, I'') \subset \Delta(I, I') \wedge \forall C \in \Delta(I, I'')\ \big(I''(C) = I'(C)\big)$$

and thus,

$$\lambda(I, I'') \subset \lambda(I, I') \rightarrow \Delta(I, I'') \subset \Delta(I, I')$$
$$\nexists I'' \in Repairs(I, \Sigma)\ \big(\Delta(I, I'') \subset \Delta(I, I')\big)$$
$$\rightarrow \nexists I'' \in Repairs(I, \Sigma)\ \big(\lambda(I, I'') \subset \lambda(I, I')\big)$$

Thus, if I' is a cardinality-set-minimal repair, I' is set-minimal as well.

Since dis is a distance function, for each two different values x and y, $dis(x, y) > 0$. Also, for each cell C, $w(C) > 0$. It follows:

$\lambda(I, I'') \subset \lambda(I, I') \rightarrow cost(I, I'') < cost(I, I')$

and,

$$\nexists I'' \in Repairs(I, \Sigma)\ \big(cost(I, I'') \subset cost(I, I')\big)$$
$$\rightarrow \nexists I'' \in Repairs(I, \Sigma)\ \big(\lambda(I, I'') \subset \lambda(I, I')\big)$$

In other words, if I' is a cost-minimal repair, I' is set-minimal as well. $\square$

In general, cost-minimal repairs are not necessarily cardinality-minimal or cardinality-set-minimal, and vice versa. However, for a constant weighting function w (i.e., all cells are equally trusted) and a constant distance function dis (i.e., the distance between any pair of values is the same), the set of cost-minimal repairs and the set of cardinality-minimal repairs coincide.

4 Repairing violations of a single FD

In this section, we discuss how to generate various types of repairs when Σ consists of a single FD.

First, we define the concept of *clean cells* and relate it to cardinality-minimal and cardinality-set-minimal repairs.

We define a clean set of cells $\mathcal{C} \subseteq CIDs(I)$ with respect to a set of FDs Σ as follows.

Definition 7 (*Clean cells*) A set of cells $\mathcal{C}$ in an instance I is clean iff there is at least one repair $I' \in Repairs(I, \Sigma)$ such that $\forall C \in \mathcal{C},\ I'(C) = I(C)$.

That is, a set of cells in an instance I is clean if their values in I can remain unchanged while obtaining a repair of I. For example, in Fig. 2, the sets $\{(t_1, A), (t_1, B), (t_2, A)\}$ and $\{(t_1, B), (t_2, A), (t_2, B)\}$ are clean, while the set $\{(t_1, A), (t_1, B), (t_2, A), (t_2, B)\}$ is not clean.

Definition 8 (*Maximum set of clean cells*) A set of cells $\mathcal{C}$ is a *maximum* clean set iff $\mathcal{C}$ is clean and no other clean set has greater cardinality than $\mathcal{C}$.

For example, $\{(t_1, A), (t_2, A), (t_2, B), (t_3, A), (t_3, B)\}$ and $\{(t_1, B), (t_2, A), (t_2, B), (t_3, A), (t_3, B)\}$ in Fig. 2 are maximum clean sets.

Definition 9 (*Maximal set of clean cells*) A set of cells $\mathcal{C}$ is a *maximal* clean set iff $\mathcal{C}$ is clean and no strict superset of $\mathcal{C}$ is clean.

For example, the sets $\{(t_1, A), (t_1, B), (t_2, A), (t_3, B)\}$ and $\{(t_1, A), (t_2, A), (t_2, B), (t_3, A), (t_3, B)\}$ in Fig. 2 are maximal clean sets.

In the following theorem, we establish the link between clean cells and cardinality-minimal and cardinality-set-minimal repairs.

Theorem 1 *Given an input instance I and a set of FDs Σ, a repair I' of I with respect to Σ is cardinality-set-minimal iff the set of unchanged cells in I' (i.e., $CIDs(I') \backslash \Delta(I, I')$) is a maximal clean set of cells. A repair I' of I with respect to Σ is cardinality-minimal iff the set of unchanged cells in I' is a maximum clean set.*

Proof We prove the "if" condition of the first statement as follows. Let $\mathcal{C} = CIDs(I') \backslash \Delta(I, I')$ be a maximal clean set of cells. We cannot add any cell to $\mathcal{C}$ without making $\mathcal{C}$ unclean. Based on the definition of clean cells (Definition 7), there does not exist any other repair of I that has a set of unchanged cells $\mathcal{C}'$ that is a strict superset of $\mathcal{C}$ (i.e., $\nexists I'' \in Repairs(I, \Sigma)(\Delta(I, I'') \subset \Delta(I, I'))$. Thus, I' is a cardinality-set-minimal repair.

We prove the "only if" condition of the first statement as follows. Let I' be a cardinality-set-minimal repair of I. The set $\mathcal{C} = CIDs(I') \backslash \Delta(I, I')$ is a clean set of cells because I' is a repair. Because I is cardinality-set-minimal, no cells in $\Delta(I, I')$ can be reverted back to their original values without violating Σ, even if we allow remodifying other changed cells (i.e., $\nexists I'' \in Repairs(I, \Sigma)(\Delta(I, I'') \subset \Delta(I, I'))$). It follows that we cannot extend $\mathcal{C}$ by adding one or more cells without violating the clean cells property. It follows that $\mathcal{C}$ is a maximal clean set of cells.

Similarly, we can prove the second statement in the theorem by replacing the set-containment minimality criterion with the set-cardinality-minimality criterion. □

In general, obtaining a maximum clean set of cells (and hence a cardinality-minimal repair) is NP-complete [23]. However, as we will show shortly, finding a maximum clean set of cells for a single FD can be done in PTIME. On the other hand, obtaining a maximal clean set of cells (and hence a cardinality-set-minimal repair) can still be done in PTIME for multiple FDs (Sect. 5).

Based on Theorem 1, we can generate a cardinality-set-minimal repair by obtaining a maximal clean set and modifying the remaining cells suitably. In this section, we assume there is a single FD in Σ, and we extend our algorithm to multiple FDs in Sect. 5. Assuming that Σ contains an FD $X \rightarrow A$, we describe in Algorithm 1 how to generate a maximal clean set of cells, namely $CleanSet$, and how to change the cells outside $CleanSet$ in order to obtain a cardinality-set-minimal repair.

Algorithm 1 is a randomized algorithm and thus invoking it multiple times gives a random sample of cardinality-set-minimal repairs. Moreover, it is possible to modify the algorithm to generate cardinality-minimal repairs (for a single FD) as we show later in this section.

Algorithm 1 `Gen-Repair-1FD`$(I, X \rightarrow A)$

1: $CleanSet \leftarrow \emptyset$
2: $I' \leftarrow I$
3: **for** each tuple $t \in TIDs(I)$ (based on a random iteration order) **do**
4:   **if** $\exists(t', XA) \subset CleanSet$ such that $I(t, X) = I(t', X)$ and $I(t, A) \neq I(t', A)$ **then**
5:     randomly select $k - 1$ cells from $(t, X) \cup \{(t, A)\}$ and add them to $CleanSet$, where k is the number of attributes in $X \cup \{A\}$
6:   **else**
7:     add (t, X) and (t, A) to $CleanSet$
8:   **end if**
9: **end for**
10: **for** each cell $(t, A_i) \in CIDs(I) \backslash CleanSet$ **do**
11:   **if** $A_i \in X$ (i.e., A_i is a left-hand-side attribute) **then**
12:     assign any value $c \in Dom(A_i)$ to $I'(t, A_i)$ such that $\nexists(t', XA) \subset CleanSet$ where $I'(t', X) = I'(t, X) \wedge I'(t', A) \neq I'(t, A)$
13:   **else**
14:     locate a tuple t' such that $(t', XA) \subset CleanSet$ and $I'(t', X) = I'(t, X)$
15:     $I'(t, A) \leftarrow I'(t', A)$
16:   **end if**
17:   add (t, A_i) to $CleanSet$
18: **end for**
19: **return** I'

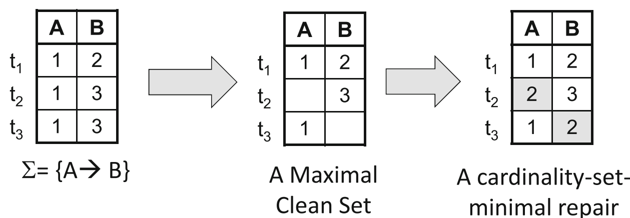

Fig. 4 An example of repairing violations of a single FD $A \rightarrow B$

In lines 3–9, the algorithm iterates over the tuples in I in random order and expands the set $CleanSet$ by inserting cells that are consistent with the cells already in $CleanSet$. In lines 10–18, unclean cells are updated accordingly to obtain a cardinality-set-minimal repair I'.

The set $CleanSet$ constructed in lines 3–9 represents a clean set of cells because (1) all complete tuples in the set $CleanSet$ are consistent w.r.t. $X \rightarrow A$, and (2) we can find an assignment to the cells that are not in $CleanSet$ such that $X \rightarrow A$ is not violated. That is, for each tuple t such that $(t, X) \in CleanSet$ and $(t, A) \notin CleanSet$, we set (t, A) to the value of A of the first tuple processed by the algorithm with attributes X equal to $t[X]$. Also, for each tuple t such that $(t, B) \notin CleanSet$, for an attribute $B \in X$, and $(t, A) \in CleanSet$, we assign (t, B) to a value from $Dom(B)$ that results in no violations of $X \rightarrow A$. Such a value must exist, assuming that $Dom(B)$ is unbounded.

The set $CleanSet$ is a *maximal* clean set since adding any cell (t, A_i) from $CIDs(I) \backslash CleanSet$ to $CleanSet$ results in a violation of $X \rightarrow A$. Thus, the constructed instance is a cardinality-set-minimal repair.

In order to generate a sample of cardinality-set-minimal repairs, we run Algorithm 1 multiple times. Each run will generate a maximal clean set (by iterating over each tuple in random order in line 3) and modify the remaining tuples to satisfy the FD.

In Fig. 4, we show an example of repairing a single FD $A \rightarrow B$. To generate a cardinality-set-minimal repair, we first obtain a maximal clean set of cells; say the algorithm chooses $\{(t_1, A), (t_1, B), (t_2, B), (t_3, A)\}$. Then, we obtain a cardinality-set-minimal repair by setting the value of the unclean cell (t_2, A) to 2 and (t_3, B) to 2.

Obtaining a *maximum* clean set of cells, and hence, a cardinality-minimal repair can be performed in PTIME when Σ contains a single FD. To do so, lines 3–9 in Algorithm 1 can be changed as follows to obtain a maximum clean set. For each group of tuples with the same value of attributes X, we first insert (t, XA) into $CleanSet$ for each tuple t whose attribute A is associated with the most frequent value of A across all tuples in the group. For the remaining tuples in the group, we insert randomly selected $k - 1$ cells per tuple, where $k = |X \cup \{A\}|$. Clearly, each one of these tuples must have at least one cell changed, which is the cell that is not inserted into $CleanSet$. Thus, it is not possible to obtain a clean set of cells with greater cardinality than $CleanSet$ (i.e., $CleanSet$ is a maximum clean set of cells). Updating the unclean cells in lines 10–18 remains unchanged.

For example, returning to Fig. 4, the value 3 is the most frequently occurring value of B for $A = 1$, so first we add (t_2, A), (t_2, B), (t_3, A), and (t_3, B) to $CleanSet$., Then, we consider (t_1, A) and (t_1, B) next. The latest cell inserted into $CleanSet$ is modified to resolve the violation. For example, if (t_1, A) is considered first in line 3, we will change the value of (t_1, B) from two to three in line 15. Otherwise, we will change the value of (t_1, A) from one to, say, two, in line 12.

5 Sampling possible repairs for multiple FDs

The results indicated by Theorem 1 carry over to the case of multiple FDs in Σ. However, obtaining a maximum clean set is now equivalent to obtaining a repair with the minimum number of cell changes, which is NP-hard [23]. Fortunately, generating a *maximal* clean set remains doable in PTIME, as we show in this section.

The sampling space should be neither too restrictive (and thus missing too many repairs) nor too large (and thus sampling repairs with very low probability of being correct). We argue that the cardinality-set-minimal space provides this balance, and we thus target sampling from this space.

Our sampling algorithm for multiple FDs is based on Theorem 1. We randomly pick a maximal clean set of cells $\mathcal{C}$, and then, we randomly change cells outside $\mathcal{C}$ in a way that satisfies Σ.

The organization of this section is as follows. First, we provide an algorithm to detect whether a set of cells is clean in Sect. 5.1, and we show how to generate a maximal clean set of cells. Then, we introduce a sampling algorithm in Sect. 5.2 that samples from the space of cardinality-set-minimal repairs. In Sect. 5.3, we define two forms of hard constraints that specify the user confidence in cells and we show how to enforce them.

5.1 Generating a maximal clean set of cells

In the following, we determine whether a set of cells is clean. We observe that it is not enough to verify that the cells in $\mathcal{C}$ do not violate any FDs, that is, checking that $\nexists (t_1, XA), (t_2, XA) \subset \mathcal{C}, X \rightarrow A \in \Sigma$ such that $(t_1[X] = t_2[X] \wedge t_1[A] \neq t_2[A])$). For example, consider Fig. 5, which shows a set of non-empty cells in an instance. Assume that we need to determine if the shown cells are clean. Although the shown cells do not directly violate any FD in Σ (i.e., we cannot find a pair of tuples that violates Σ), no repair may contain the current values of those cells regardless of the val-

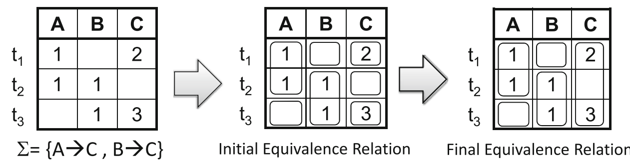

Fig. 5 Checking whether the six non-empty cells are clean

ues of the other cells. This is because $t_1[A] = t_2[A]$ implies $t_1[C] = t_2[C]$ (by $A \rightarrow C$) and $t_2[B] = t_3[B]$ implies that $t_2[C] = t_3[C]$ (by $B \rightarrow C$). Thus, $t_1[C]$, $t_2[C]$ and $t_3[C]$ have to be equal in any repair. However, $t_1[C] \neq t_3[C]$ in the shown instance.

To determine whether a set of cells $\mathcal{C}$ is clean, we need to capture the equality constraints over the cells in I that are imposed by the FDs in Σ. Then, we check for contradictions between these constraints and the values of cells in $\mathcal{C}$. We model equality constraints as an equivalence relation over cells in I, denoted $\mathcal{E}$. Equivalence relations have been used for the same purpose in previous data cleaning algorithms (e.g., [9,12]). We denote by $ec(\mathcal{E}, C_i)$ the equivalence class $E \in \mathcal{E}$ to which a cell C_i belongs according to $\mathcal{E}$. We denote by *merging* two equivalence classes in $\mathcal{E}$ replacing them by a new equivalence class that is equal to their union. Algorithm 2 builds the equivalence relation $\mathcal{E}$ given a set of cells $\mathcal{C}$ in an instance I.

Algorithm 2 `BuildEquivRel`$(\mathcal{C}, I, \Sigma)$

1: let $TIDs(\mathcal{C})$ be the set of tuple identifiers involved in $\mathcal{C}$, $\{t : (t, A) \in \mathcal{C}\}$
2: let $Attrs(\mathcal{C})$ be the set of attributes involved in $\mathcal{C}$, $\{A : (t, A) \in \mathcal{C}\}$
3: let $\mathcal{E}$ be an initial equivalence relation on the set $\{(t, A) : t \in TIDs(\mathcal{C}), A \in Attrs(\mathcal{C})\}$ such that the cells in $\mathcal{C}$ that belong to the same attribute and have equal values in I are in the same equivalence class, and all other cells outside $\mathcal{C}$ belong to separate (singleton) classes
4: **while** $\exists t_1, t_2 \in TIDs(\mathcal{C}), A \in Attrs(\mathcal{C}), X \subset Attrs(\mathcal{C}), X \rightarrow A \in \Sigma$ such that $\forall B \in X$ $(ec(\mathcal{E}, (t_1, B)) = ec(\mathcal{E}, (t_2, B)))$, and $ec(\mathcal{E}, (t_1, A)) \neq ec(\mathcal{E}, (t_2, A))$ **do**
5: merge the equivalence classes $ec(\mathcal{E}, (t_1, A))$ and $ec(\mathcal{E}, (t_2, A))$
6: **end while**
7: **return** $\mathcal{E}$

Figure 5 shows an example of the initial and final equivalence relations that are built by Algorithm 2. The equivalence class $\{(t_1, C), (t_2, C), (t_3, C)\}$ in the final equivalence relation indicates that these three cells must be equal in any repair in which the six non-empty cells are unchanged. This is clearly infeasible since (t_1, C) and (t_3, C) have different values, which means that the set of six non-empty cells is not clean.

In general, a set of cells $\mathcal{C}$ in I is clean with respect to Σ, denoted $isClean(\mathcal{C}, I, \Sigma, \mathcal{E})$, iff every two cells in $\mathcal{C}$ that belong to the same equivalence class in $\mathcal{E}$ have the same value in I.

Lemma 2 *Given a set of cells $\mathcal{C}$ in an instance I and a set of FDs Σ, let $\mathcal{E}$ be the outcome of procedure `BuildEquivRel`$(\mathcal{C}, I, \Sigma)$. $\mathcal{C}$ is clean iff $\forall C_i, C_j \in \mathcal{C}$, $ec(\mathcal{E}, C_i) = ec(\mathcal{E}, C_j)$ implies that $I(C_i) = I(C_j)$.*

Proof We prove the "only if" direction as follows. Let $\mathcal{C}$ be a clean set of cells in I, and let $\mathcal{I}$ be the non-empty subset of repairs of I whose cells in $\mathcal{C}$ are unchanged (i.e., $\forall I' \in \mathcal{I}$ $(\Delta(I, I') \cap \mathcal{C} = \emptyset)$). The proof consists of two steps. We first prove that each pair of cells in I belonging to the same equivalence class in $\mathcal{E}$ have equal values in every $I' \in \mathcal{I}$. Second, we prove that each pair of cells in $\mathcal{C}$ belonging to the same equivalence class in $\mathcal{E}$ have equal values in I.

We prove the first part as follows. Based on Algorithm 2, for every two cells (t_1, A) and (t_2, A) that belong to the same equivalence class, we have two possibilities:

– (t_1, A) and (t_2, A) were placed in the same equivalence class in line 3 in Algorithm 2. In other words, (t_1, A) and (t_2, A) belong to $\mathcal{C}$ and $I(t_1, A) = I(t_2, A)$, and thus, $I'(t_1, A) = I'(t_2, A)$ for all $I' \in \mathcal{I}$, or
– (t_1, A) and (t_2, A) were placed in the same equivalence class in line 5 in Algorithm 2. In this case, there must exist an FD $X \rightarrow A \in \Sigma$ such that for all $B \in X$, (t_1, B) and (t_2, B) belong to the same equivalence class. Recursively, we can prove that for all $B \in X$, $I'(t_1, B) = I'(t_2, B)$. The fact that $I' \models X \rightarrow A$ implies that $I'(t_1, A) = I'(t_2, A)$.

The second part of the proof is trivial since each cell in $\mathcal{C}$ has the same value in I and all repairs in $\mathcal{I}$, and thus $\forall C_i, C_j \in \mathcal{C}$ $(ec(\mathcal{E}, C_i) = ec(\mathcal{E}, C_j) \rightarrow I(C_i) = I(C_j))$.

We prove the "if" direction as follows. Consider the case where $\forall C_i, C_j \in \mathcal{C}$ $(ec(\mathcal{E}, C_i) = ec(\mathcal{E}, C_j) \rightarrow I(C_i) = I(C_j))$. We need to prove that the set $\mathcal{I}$ (i.e., the set of repairs of I that do not change cells in $\mathcal{C}$) is not empty. We construct one instance $I' \in \mathcal{I}$ as follows. We assign each cell in $\mathcal{C}$ in I' to the same value in I (i.e., $\forall C \in \mathcal{C}, I'(C) = I(C)$). We iterate over all other cells outside $\mathcal{C}$ in random order. Each cell that belongs to a singleton equivalence class in $\mathcal{E}$ or belongs to a tuple that is not mentioned in $\mathcal{C}$ is set to a new variable. For each cell C that belongs to a non-singleton equivalence class $E \in \mathcal{E}$ that includes at least one cell with an assigned value (call it x), we set $I'(C)$ to x. Finally, for each cell C that belongs to a non-singleton equivalence class $E \in \mathcal{E}$ whose cells are not assigned to any values yet, we assign C to a new variable.

Now, we show that the constructed instance I' is indeed a repair. The construction method of I' as well as the fact that $\forall C_i, C_j \in \mathcal{C}$ $(ec(\mathcal{E}, C_i) = ec(\mathcal{E}, C_j) \rightarrow I(C_i) = I(C_j))$ ensure that cells belonging to the same equivalence class in

$\mathcal{E}$ are assigned to the same value in I'. Also, all cells in I' that have equal values belong to the same equivalence class. This is true for cells assigned to variables in I' since we can only assign a cell C to an old variable v_i if C belongs to an equivalence class that contains another cell already assigned to v_i. On the other hand, a cell C can be assigned to a constant c only if C belongs to an equivalence class that contains one or more cells from $\mathcal{C}$ whose values are equal to c. Also, Algorithm 2 places all cells in $\mathcal{C}$ with the same value into the same equivalence class. It follows that cells in I' that are assigned to constants and have equal values are in the same equivalence classes.

For every two tuples $t_1, t_2 \in I'$ and for every FD $X \rightarrow A \in \Sigma$, $I'(t_1, X) = I'(t_2, X)$ implies that for all $B \in X$, (t_1, B) and (t_2, B) belong to the same equivalence class. Therefore, (t_1, A) and (t_2, A) must belong to the same equivalence class as well (refer to lines 4–6 in Algorithm 2), and thus $I'(t_1, A) = I'(t_2, A)$. This proves that $I' \models \Sigma$ and thus $\mathcal{I}$ is not empty (i.e., cells in $\mathcal{C}$ are clean). □

Next, we show how to randomly pick a maximal clean set of cells, given I and Σ. We describe our procedure in Algorithm 3.

Algorithm 3 `GetMaxCleanSet` (I, Σ)

1: Define a set $CleanSet$ and initialize it to $\emptyset$
2: **for** each cell $C \in CIDs(I)$ (based on a random iteration order) **do**
3: $\quad \mathcal{E} \leftarrow$ `BuildEquivRel`$(CleanSet \cup \{C\}, I, \Sigma)$
4: $\quad$ **if** $isClean(CleanSet \cup \{C\}, I, \Sigma, \mathcal{E})$ **then**
5: $\quad\quad CleanSet \leftarrow CleanSet \cup \{C\}$
6: $\quad$ **end if**
7: **end for**
8: **return** $CleanSet$

Algorithm 3 starts with an empty set of clean cells and attempts to add cells to the clean set, one cell at a time, in random order. The algorithm terminates when all the cells have been considered. We prove its correctness below.

Lemma 3 *The sets of cells returned by Algorithm 3 are maximal clean sets.*

Proof Given a set of clean cells returned by Algorithm 3, denoted $\mathcal{C}$, we need to prove that for any subset of $CIDs(I)\backslash\mathcal{C}$ (call it S), $\mathcal{C} \cup S$ is not clean.

First, we prove that if a set $\mathcal{C}$ is not clean, then any superset of $\mathcal{C}$ is not clean as well. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two sets of cells in an instance I such that $\mathcal{C}_1 \subset \mathcal{C}_2$. Let $\mathcal{E}_1$ (respectively, $\mathcal{E}_2$) be the outcome of `BuildEquivRel`$(\mathcal{C}_1, I, \Sigma)$ (respectively, `BuildEquivRel`$(\mathcal{C}_2, I, \Sigma)$). By analyzing Algorithm 2, we reach that each equivalence class in $\mathcal{E}_1$ must be contained in another equivalence class in $\mathcal{E}_2$. Therefore, if there exist two cells in $\mathcal{C}_1$ that belong to the same equivalence class in $\mathcal{E}_1$ and have different values in I (i.e., $\mathcal{C}_1$ is not clean), the two cells must belong to the same equivalence class in $\mathcal{E}_2$, which means that $\mathcal{C}_2$ is not clean as well.

Assume, to the contrary, that $\exists S \subset CIDs(I)\backslash\mathcal{C}$ such that $\mathcal{C} \cup S$ is clean. Clearly, every cell C in S has been rejected in line 4 in Algorithm 3, which means that $\mathcal{C}_s \cup \{C\}$ is not clean, where $\mathcal{C}_s$ is the subset of $\mathcal{C}$ that is constructed up to the point of rejecting C. The set $\mathcal{C} \cup S$ is a superset of $\mathcal{C}_s \cup \{C\}$. Therefore, $\mathcal{C} \cup S$ is not clean, a contradiction. □

Complexity analysis Let n be the number of tuples in the input instance I and m be the number of attributes in I. In Algorithm 2, the maximum number of merges of equivalence classes is less than the number of tuples that appear in $\mathcal{C}$ multiplied by the number of attributes that appear in $\mathcal{C}$. Each merge operation can be done in a constant time (for all practical database sizes) using the find-union algorithm [29]. Therefore, the complexity of Algorithm 2 is in $O(n \cdot m)$. Evaluating $isClean$ can be done in $O(n \cdot m)$ using a hash table structure. That is, all cells belonging to the same equivalence class are hashed to a unique bucket, and we associate each bucket with the values of the inserted cells so far. Upon insertion of each cell, we only need to compare the cell value to the bucket value to determine the cleanness of the cells. A straightforward implementation of Algorithm 3 has a complexity of $O(n^2 \cdot m^2)$.

5.2 Sampling cardinality-set-minimal repairs

In this section, we present a randomized algorithm for generating cardinality-set-minimal repairs (Algorithm 4). This algorithm is a generalized version of the procedure we described in the proof of Lemma 2.

The first step is constructing a maximal clean set of cells, denoted $MaxCleanCells$ (line 2). The algorithm iteratively cleans the cells outside $MaxCleanCells$ and adds them to a set called $Cleaned$. Initially, $Cleaned$ is equal to $MaxCleanCells$. In each iteration, the algorithm assigns a value to the current cell (t, A) such that $Cleaned \cup \{(t, A)\}$ becomes clean. Specifically, if (t, A) belongs to a non-singleton equivalence class in $\mathcal{E}$ that contains other cells previously inserted into $Cleaned$, the only choice is to set $I'(t, A)$ to the same value as the other cells in the equivalence class (lines 6, 7). Otherwise, we randomly choose one of the following three alternative values for $I't, A)$: (1) a constant that is randomly selected from $ADom(A, I)$, (2) a variable that is randomly selected from the set of variables previously used in attribute A in I', or (3) a new variable v_j^A (line 8). For the first and second alternatives, we need to make sure that the selected constant or variable makes the set $Cleaned \cup \{(t, A)\}$ clean. One simple approach is to keep picking a constant (similarly, a variable) at random until this condition is met. In the worst case, we can select up to n constants (similarly, n variables), where n is the number

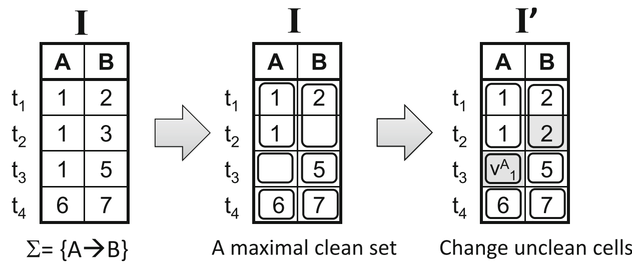

Fig. 6 An example of executing Algorithm 4

of tuples in the input instance. The third alternative, which is setting $I'(t, A)$ to a new variable, guarantees that the set $Cleaned \cup \{(t, A)\}$ becomes clean. In fact, enforcing the third alternative at every iteration reduces Algorithm 4 to the repairing algorithm described in the proof of Lemma 2. The algorithm terminates when all the cells have been added to $Cleaned$ and returns the resulting instance I'.

Algorithm 4 `Gen-Repair(`I, Σ`)`

1: $I' \leftarrow I$
2: $MaxCleanCells \leftarrow$ `GetMaxCleanSet`(I, Σ)
3: $Cleaned \leftarrow MaxCleanCells$
4: $\mathcal{E} \leftarrow$ `BuildEquivRel`$(Cleaned, I, \Sigma)$
5: **for** each $(t, A) \in CIDs(I) \backslash MaxCleanCells$ (based on a random iteration order) **do**
6: **if** (t, A) belongs to a non-singleton equivalence class in $\mathcal{E}$ that contains other cells in $Cleaned$ **then**
7: assign $I'(t, A)$ to the value (either a constant or a variable) of the other cells in $ec(\mathcal{E}, (t, A)) \cap Cleaned$
8: **else**
9: randomly set $I'(t, A)$ to one of three alternatives: a randomly selected constant from $ADom(A, I)$, a randomly selected variable v_i^A that was previously used in I', or a fresh variable v_j^A such that $Cleaned \cup \{(t, A)\}$ becomes clean
10: **end if**
11: $Cleaned \leftarrow Cleaned \cup \{(t, A)\}$
12: $\mathcal{E} \leftarrow$ `BuildEquivRel`$(Cleaned, I', \Sigma)$
13: **end for**
14: **return** I'

We show an example of executing Algorithm 4 in Fig. 6. The algorithm obtains a maximal clean set, which is shown as the middle relation, and changes the two unclean cells (t_2, B) and (t_3, A). Because (t_2, B) exists in the same equivalence class as (t_1, B), the algorithm assigns the value of (t_1, B) to (t_2, B). For the cell (t_3, A), the algorithm chooses to assign a fresh variable, v_1^A, to it.

In the following theorem, we prove the correctness of Algorithm 4.

Theorem 2 *Every instance that is generated by Algorithm 4 is a cardinality-set-minimal repair. Additionally, all cardinality-set-minimal repairs can be generated by Algorithm 4.*

Proof We need to prove the following points.

1. Every generated instance I' is a repair of I with respect to Σ.
2. Every generated repair I' is cardinality-set-minimal.
3. Every cardinality-set-minimal repair of I can be generated by the algorithm.

First, we prove that every instance I' generated by Algorithm 4 is a repair of I with respect to Σ. In other words, we need to show that all cells in a generated repair I' represent a clean set. Initially, the set $Cleaned = MaxCleanCells$ is clean with respect to Σ (based on Lemma 3). In each iteration, the algorithm adds a cell to $Cleaned$ and changes this cell to ensure that the resulting version of $Cleaned$ is clean as well. This is done by assigning a value to each newly added cell that satisfies the constraints represented by $\mathcal{E}$. Upon termination, all cells in I' are in $Cleaned$, which indicates that the resulting instance I' satisfies Σ.

Second, we prove that each generated repair is cardinality-set-minimal. The initial maximal clean set of cells, denoted $MaxCleanCells$, is not modified throughout the algorithm. Thus, the set of unchanged cells in any generated repair represents a maximal clean set of cells, which indicates that the generated repair is cardinality-set-minimal based on Theorem 1.

Third, we prove that every cardinality-set-minimal repair can be generated by Algorithm 4. Let I' be one such repair. First, note that I' consists of some maximal clean set of cells $\mathcal{C}$ (Theorem 1). This clean set will be used by Algorithm 4 if the random iteration order in line 2 of Algorithm 3 is such that all the cells in $\mathcal{C}$ are considered first. Next, note that regardless of the iteration order in which Algorithm 4 processes the remaining cells $CIDs(I') \backslash \mathcal{C}$ (line 5), each cell C processed in lines 6–10 can be assigned to the value in I' to make $Cleaned \cup \{C\}$ clean. Assuming otherwise implies that there exists a subset of cells in I' that is not clean, which contradicts the fact that I' is a repair. It follows that any cardinality-set-minimal repair I' can be generated by Algorithm 4. □

Complexity analysis Obtaining a maximal clean set of cells costs $O(n^2 \cdot m^2)$, where n denotes the number of tuples in I and m denotes the number of attributes. In Algorithm 4, the number of iterations is at most equal to the number of cells in I' (i.e., $n \cdot m$). In each iteration, Algorithm 2 is invoked to build the equivalence classes of cells in $Cleaned$. Additionally, the condition $isClean$ can be evaluated for all possible constants and variables that appear in the attribute A in I' (in the worst case). Hence, the complexity of each iteration is $O(m \cdot n^2)$, and the overall complexity of Algorithm 4 is $O(m^2 \cdot n^3)$. Note that if we restrict changing cells in line 9 to the third alternative only (i.e., assigning new variables to the cells), the complexity is reduced to $O(n^2 \cdot m^2)$. Additionally, we can reduce the runtime of the algorithm by not recomputing the equivalence relation from scratch in every iteration.

5.3 User-defined hard constraints

In this section, we modify our approach to generate a sample of repairs that is consistent with the user's confidence in the accuracy of the data. We discuss two possible ways to define the confidence of a cell, and we show how to modify our algorithm accordingly.

The first method to specify cell confidence is to provide a set of cells $\mathcal{T}$ that are completely trusted. Such cells are considered clean, and thus, the cleaning algorithm must keep their values unchanged. Since the cleaning algorithm cannot change cells in $\mathcal{T}$, we must first ensure that $\mathcal{T}$ itself is clean. To do so, we build the equivalence relationship $\mathcal{E}_{\mathcal{T}}$ over $\mathcal{T}$ using Algorithm 2 and check the value of $isClean(\mathcal{T}, I, \Sigma, \mathcal{E}_{\mathcal{T}})$. If $\mathcal{T}$ is unclean, we return an empty answer. Assuming that $\mathcal{T}$ is clean, we describe our modifications to the cleaning algorithm as follows. When creating a maximal clean set of cells using Algorithm 3, we insert the cells in $\mathcal{T}$ first into the set $CleanSet$ (i.e., we initialize $CleanSet$ to $\mathcal{T}$ in line 1 in the algorithm). The remainder of the algorithm remains unchanged. Finding a maximal clean set that is a superset of $\mathcal{T}$ is possible as long as $\mathcal{T}$ is clean. This modification produces repairs in which none of the cells in $\mathcal{T}$ are changed since Algorithm 4 does not change the cells in the maximal clean set generated by Algorithm 3.

The second way to specify confidence is using a partial order relation to indicate the relative trust between pairs of cells. Consider a strict partial order $\succ_c$ that is defined over $CID(I)$ such that $C_1 \succ_c C_2$ iff the user is more confident about the accuracy of C_1 than C_2. By definition, $\succ_c$ is antisymmetric, which prevents contradicting beliefs about the confidence of different cells (i.e., if $C_1 \succ_c C_2$, then $C_2 \not\succ_c C_1$).

Intuitively, if $C_1 \succ_c C_2$, we should prioritize changing C_2 over changing C_1. We formulate this requirement as follows.

Definition 10 ($\succ_c$*-compatible repair*) Given an instance I and a strict partial order relation $\succ_c$ defined over cells in I, let $LC(C_i)$ be defined as $\{C_j \in CIDs(I) : C_i \succ_c C_j\}$. A repair $Intuitively$ of I is $\succ_c$-compatible iff there does not exist a repair I'' of I such that $Reverted = \Delta(I, I')\backslash\Delta(I, I'')$ is non-empty and $\Delta(I, I'')\backslash\Delta(I, I') \subseteq \bigcup_{C \in Reverted} LC(C)$.

That is, a repair I' is $\succ_c$-compatible iff we cannot obtain another repair by reverting one or more changed cells in I' back to their original values while allowing changing cells in I' that are less confident than the reverted cells.

It is possible to modify Algorithm 4 to only sample from $\succ_c$-compatible repairs as follows. Let $\succ_c^*$ be a *linear extension* of $\succ_c$, which is a total order on cells in I such that for every C_i and C_j in $CIDs(I)$, if $C_i \succ_c C_j$, then $C_i \succ_c^* C_j$ [13]. In general, there exists multiple linear extensions for a given partial order. A random sample of linear extensions may be obtained using a (randomized) topological sorting algorithm (refer to [13] for more details). We modify Algorithm 3 by replacing the random selection of cells in line 2 by the order indicated by $\succ_c^*$, starting with the cell C such that $\nexists C_i \in CIDs(I)(C_i \succ_c^* C)$.

Lemma 4 *Given that cells in line* 2 *in Algorithm* 3 *are selected based on* $\succ_c^*$*, each repair* I' *of* I *generated by Algorithm* 4 *is a* $\succ_c$*-compatible repair.*

Proof Assume, for a contradiction, that a generated repair I' is not a $\succ_c$-compatible repair. It follows that there exits a repair I'' such that $Reverted = \Delta(I, I')\backslash\Delta(I, I'') \neq \emptyset$ and $\Delta(I, I'')\backslash\Delta(I, I') \subseteq \bigcup_{C \in Reverted} LC(C)$.

While building I', cells are processed in Algorithm 3 based on a total order $\succ_c^*$ that extends $\succ_c$. Every cell in $\Delta(I, I')\backslash\Delta(I, I'')$ fails the cleanness test in line 4 in Algorithm 3 (otherwise it would be in $CleanSet$ and not in $\Delta(I, I')$). Let C_1 be the cell in $\Delta(I, I')\backslash\Delta(I, I'')$ that first failed the cleanness test according to $\succ_c^*$. Let $\mathcal{C}_{pre}$ be the cells inserted into $CleanSet$ in Algorithm 3 before C_1. Because cells are processed according to $\succ_c^*$, cells in $LC(C_i)$ are processed after C_i for each $C_i \in \Delta(I, I')\backslash\Delta(I, I'')$. Thus, cells in $\bigcup_{C \in Reverted} LC(C)$ are processed after C_1 and $\mathcal{C}_{pre} \cap \bigcup_{C \in Reverted} LC(C) = \emptyset$. It follows that $\mathcal{C}_{pre} \cap (\Delta(I, I'')\backslash\Delta(I, I')) = \emptyset$. Since cells in $\mathcal{C}_{pre}$ are not changed in I' (i.e., $\mathcal{C}_{pre} \cap \Delta(I, I') = \emptyset$), $\mathcal{C}_{pre} \cap \Delta(I, I'') = \emptyset$ (i.e., cells in $\mathcal{C}_{pre}$ are not changed in I''). But $\mathcal{C}_{pre} \cup \{C_1\}$ is not clean and cannot remain completely unchanged in any repair of I. Thus, C_1 must be in $\Delta(I, I'')$, a contradiction. □

6 Block-wise repairing

In this section, we improve the efficiency of generating repairs by partitioning the input instance I into disjoint blocks, each of which represents a subset of cells in I, such that the blocks can be repaired independently. A similar idea has been previously used in the context in duplicate elimination where tuples are partitioned into blocks and each block is de-duplicated separately [5,25].

Partitioning an instance I into multiple disjoint blocks effectively splits a problem instance into a number of smaller instances, which results in a significant increase in performance. Also, such partitioning allows parallelization of the cleaning process (i.e., all blocks can be repaired in parallel). Furthermore, because subrepairs of individual blocks are independent, we effectively generate an exponentially larger number of repairs, which represents all possible combinations of subrepairs. That is, if instance I is partitioned into r blocks, and we generated k repairs for each partition, the sample size is effectively equal to k^r.

A simple strategy for partitioning I is to partition the attributes in R into multiple disjoint groups such that no FD in Σ spans more than one group of attributes (i.e., vertical

partitioning). However, this strategy has a limited impact on the performance as it fails to reduce the number of tuples in each partition, which is the main complexity factor.

In order to allow more aggressive partitioning of the input instance, where each block represents a set of cells, we have to restrict the values that can be assigned to cells in line 9 in Algorithm 4 to new variables (i.e., the third alternative). This restriction ensures that the modified cell (t, A) can never have a value equal to the value of any other cell (t', A) in other blocks. Thus, (t, A) cannot be a part of a violation of an FD that contains A in the LHS attributes. We refer to the modified versions of Algorithm 4 as Algorithm `Block-Gen-Repair`. Note that the modified version might miss some cardinality-set-minimal repairs as a result of restricting the new values of the changed cells, which might affect the quality of the generated sample of repairs.

Modifying line 9 in Algorithm 4 allows deleting line 12, which reconstructs the equivalence relation $\mathcal{E}$ after modifying each cell. The reason is that the changes performed in line 7 and the modified version of line 9 do not alter the equivalence relation $\mathcal{E}$. The only possible change to $\mathcal{E}$ in the original version of Algorithm 4 is caused by merging two equivalence classes due to changing a cell in line 7 to a constant or a variable that already exists in I' (splitting an equivalence class is not possible under any circumstances). This case is not possible after modifying line 9 as described.

In the following, we describe our partitioning algorithm. Let $\mathcal{E}_0$ be the equivalence relation that is constructed over all cells in I (i.e., $\texttt{BuildEquivRel}(CIDs(I), I, \Sigma)$). Relation $\mathcal{E}_0$ clusters cells into equivalence classes such that all pairs of cells that belong to the same attribute and might be assigned to the same constant throughout the execution of $\texttt{Block-Gen-Repair}(I, \Sigma)$ are in the same equivalence class (refer to the proof of Theorem 3). Cells of the same attribute that belong to different equivalence classes can never have equal values since we can only assign new variables to LHS cells and different variables cannot be assigned to the same constants (Definition 1). For example, Fig. 7 shows an instance I and the corresponding equivalence relation $\mathcal{E}_0$. Cells (t_1, C), (t_2, C), and (t_3, C) belong to the same equivalence class, which means that they may have equal values in some generated repairs. On the other hand, (t_1, B) and (t_2, B) belong to different equivalence classes, meaning that they can never have equal values.

We use the equivalence relation $\mathcal{E}_0$ to partition the input instance I such that any two tuples that belong to different blocks can never have equal values for the LHS attributes X, for all $X \to A \in \Sigma$ (details are in Algorithm 5). Thus, any violation of FDs throughout the course of repairing I cannot span more than one block. In other words, repairing every block separately results in a repair for the entire instance I.

In Fig. 7, we show an example of partitioning an instance. Initially, an equivalence relation $\mathcal{E}_0$ is constructed on the input instance by invoking $\texttt{BuildEquivRel}(CIDs(I), I, \Sigma)$. Each equivalence class is represented as a rectangle that surrounds the class members. We initially assign each cell to a separate block (i.e., cell (t_1, A) belongs to P_1, cell (t_2, A) belongs to P_2, and so on). For each FD $X \to A$, we locate tuples whose attributes X belong to the same equivalence classes and we merge the blocks of attributes XA of those tuples. For example, since the cells (t_1, A) and (t_2, A) belong to the same equivalence class and the FD $A \to C \in \Sigma$, we merge the blocks of (t_1, A), (t_2, A), (t_1, C), and (t_2, C). We continue the partitioning algorithm, and we return the final partitioning that is shown in the figure.

Algorithm 5 $\texttt{Partition}(I, \Sigma)$

```
1: E_0 ← BuildEquivRel(CIDs(I), I, Σ)
2: Initialize the set of blocks P such that each cell in I belongs to a separate block
3: for each X → A ∈ Σ do
4:   for each pair of tuples t_i, t_j ∈ I such that ∀B ∈ X, ec(E_0, (t_i, B)) = ec(E_0, (t_j, B)) do
5:     merge the blocks of the cells (t_i, XA) ∪ (t_j, XA)
6:   end for
7: end for
8: return P
```

We prove in Theorem 3 that the blocks generated by Algorithm 5 can be repaired separately using Algorithm `Block-Gen-Repair`.

Theorem 3 *Given a partitioning of cells in an instance I that is constructed by Algorithm $\texttt{Partition}(I, \Sigma)$, repairing the individual blocks of cells separately using Algorithm `Block-Gen-Repair` results in a repair of I.*

Proof Let $P_1, \ldots, P_r$ be the blocks of I generated by Algorithm 5, and let P_i' be a repair of P_i generated by Algorithm

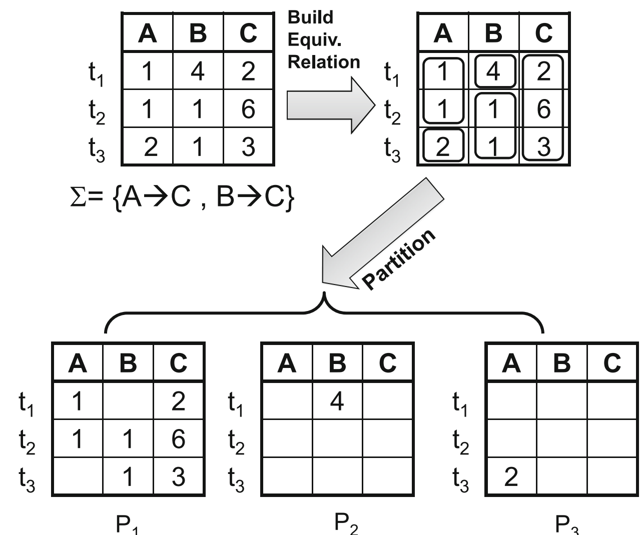

Fig. 7 An example of partitioning an instance

Block-Gen-Repair. We need to prove that the instance I' that represents the union of all blocks $P'_1, \ldots, P'_r$ satisfies Σ. We first give a road map of the proof as follows.

- Let $\mathcal{E}_0$ be the outcome of the procedure BuildEquivRel $(CIDs(I), I, \Sigma)$. We prove that for any two cells (t_1, A) and (t_2, A), if there exists any possible repair I' generated by Algorithm Block-Gen-Repair where $I'(t_1, A) = I'(t_2, A)$, (t_1, A), and (t_2, A) must be in the same equivalence class in $\mathcal{E}_0$.
- We prove that for each FD $X \rightarrow A \in \Sigma$, and for any two tuples t_1 and t_2, if $I'(t_1, B) = I'(t_2, B)$ for $B \in X$, then cells (t_1, X), (t_1, A), (t_2, X), and (t_2, A) are necessarily in the same block and thus $I' \models \Sigma$.

We first prove that, for any two cells (t_1, A) and (t_2, A), if $I'(t_1, A) = I'(t_2, A)$ for any repair I' generated by Algorithm Block-Gen-Repair, then (t_1, A) and (t_2, A) are in the same equivalence class in $\mathcal{E}_0$. If (t_1, A) and (t_2, A) have equal values in I, they belong to the same equivalence class due to the initial step in creating $\mathcal{E}_0$ (line 3 in Algorithm BuildEquivRel). Otherwise, (t_1, A) and (t_2, A) have different values in I, and at least one of them has been modified by Algorithm Block-Gen-Repair to have equal values in I'. After modifying line 9 in Algorithm 4, we only assign new variables to cells in line 9 (i.e., they cannot be equal to any other cell). Therefore, the changed cells (i.e., (t_1, A), (t_2, A), or both) must have been changed in line 7 in (modified) Algorithm 4. Thus, both cells have to belong to the same equivalence class E in the equivalence relation $\mathcal{E}$ created by the repairing algorithm in line 4. Because the original values of (t_1, A) and (t_2, A) in I are different, E must have been created based on an FD $X \rightarrow A \in \Sigma$ (refer to Algorithm 2). That is, there exists $X \rightarrow A \in \Sigma$ such that for all $B \in X$, (t_1, B), and (t_2, B) belong to the same equivalence class in $\mathcal{E}$ that is maintained by the repairing algorithm. Because any repair must satisfy the constraints imposed by $\mathcal{E}$, we deduce that for all $B \in X$, $I'(t_1, B) = I'(t_2, B)$. We recursively prove that for all $B \in X$, (t_1, B) and (t_2, B) belong to the same equivalence class in $\mathcal{E}_0$. Based on Algorithm BuildEquivRel and FD $X \rightarrow A$, (t_1, A) and (t_2, A) must be in the same equivalence class in $\mathcal{E}_0$ as well.

Now, we prove the second point. Given an FD $X \rightarrow A \in \Sigma$, let t_1 and t_2 be any tuples in I' such that $I'(t_1, B) = I'(t_2, B)$ for $B \in X$. For each attribute $B \in X$, if $I'(t_1, B) = I'(t_2, B)$, then (t_1, B) and (t_2, B) are in the same equivalence class in $\mathcal{E}_0$. Based on Algorithm 5, cells (t_1, X), (t_1, A), (t_2, X), and (t_2, A) must be in the same block. Because each block P'_i satisfies Σ, it follows that (t_1, X), (t_1, A), (t_2, X), and (t_2, A) satisfies $X \rightarrow A$. Thus, all pairs of tuples in I', which represents the union of all blocks $P'_1, \ldots, P'_r$, satisfy Σ. □

Complexity analysis The time complexity of Algorithm 5 is $O(n \cdot m)$, where n is the number of tuples in I and m is the number of attributes. Building the equivalence relation $\mathcal{E}_0$ is performed in $O(n \cdot m)$. Furthermore, there are at most $O(n \cdot m)$ merges done in lines 3–7 in Algorithm 5, each of which can be done in a constant time (for all practical database sizes) [29]. It follows that the overall complexity is $O(n \cdot m)$.

7 Repairing violations of CFDs

In this section, we extend our sampling algorithm to handle CFDs [8].

Let Σ be a set of CFDs $\{\varphi_1, \varphi_2, \ldots\}$ defined over a relation R. Each CFD φ_i consists of a pair $(X \rightarrow A, t_p)$ where $X \subseteq R$, $A \in R$ and t_p represents the pattern tuple of the CFD (recall Sect. 2). We assume that CFDs in Σ are consistent (i.e., there exists a non-empty database that satisfies them). We divide the task of extending our cleaning algorithm into three subtasks: (1) changing the procedure of detecting whether a set of cells is clean, (2) changing the sampling algorithm, and (3) changing the block-wise repair algorithm. In the following sections, we address each one of these tasks.

7.1 Extending the clean cells algorithm

The first step is to redefine the concept of V-instances for CFDs. In particular, the variables used in a V-instance can be only substituted by values in the attribute domains that do not appear in I or the CFD patterns. That is, a variable v_i^A of attribute A can only be assigned to a value from $Dom(A) \backslash ADom(A, I) \backslash \{t_p[A] : (X \rightarrow B, t_p) \in \Sigma \wedge A \in X\}$. Also, two variables cannot be assigned to the same value. Based on this interpretation of variables in a V-instance, a variable v_i^A cannot match any constant in a CFD pattern t_p; it can only match the unnamed variable '_'. This modification is important in order to guarantee that whenever a variable appears in a LHS attribute of a tuple for a certain CFD, this tuple cannot violate the CFD.

The role of the equivalence relation $\mathcal{E}$ that is introduced in Sect. 5.1 is capturing the equality constraints due to FDs in Σ. Constant CFDs impose different types of constraints: equating multiple cells to the constants defined in the CFD tableaux. Thus, we need to extend the equivalence relation by associating each equivalence class $E \in \mathcal{E}$ with a constant denoted $E.c$. If the cells of an equivalence class are not constrained to a specific constant, we set $E.c$ to the unnamed variable '_'.

In the following, we extend the procedure BuildEquiv Rel (Algorithm 2) to consider CFDs, resulting in procedure

BuildEquivRel_CFD described in Algorithm 6. We also modify the algorithm to allow early termination as soon as we detect contradicting constraints (e.g., when trying to merge two equivalence classes E_1 and E_2 where $E_1.c$ and $E_2.c$ are equal to different constants). In this case, the algorithm returns $\emptyset$, indicating that $\mathcal{C}$ is not clean. Otherwise, the algorithm returns the equivalence relation $\mathcal{E}$ indicating that $\mathcal{C}$ is clean.

The algorithm builds the initial equivalence relation $\mathcal{E}$ based on the values of cells in $\mathcal{C}$ similar to Algorithm 2. For each equivalence class E in $\mathcal{E}$, if E contains a cell from $\mathcal{C}$, we set $E.c$ to the value of this cell in I. Otherwise, we set $E.c$ to the unnamed variable '_'.

The algorithm repeatedly selects two tuples that violate a variable CFD, or a single tuple that violates a constant CFD. If no such tuples are found, the algorithm returns $\mathcal{E}$ and terminates. For each pair of tuples t_1 and t_2 violating a variable CFD $(X \rightarrow A, t_p)$, we merge the equivalence classes of the cells (t_1, A) and (t_2, A). If the values assigned to the merged equivalence classes do not match (i.e., two constants that are not equal), the algorithm returns $\emptyset$ and terminates (lines 9 and 14). Otherwise, the value of the resulting equivalence class E_{12} is set to the most restricted value of the merged classes (line 10). For a single tuple t_1 that violates a constant CFD $(X \rightarrow A, t_p)$, if $ec(\mathcal{E}, (t_1, A)).c$ is equal to the unnamed variable '_', we set the value of $ec(\mathcal{E}, (t_1, A)).c$ to $t_p[A]$ and we merge this equivalence class with other classes with the same assigned value. Otherwise, the algorithm returns $\emptyset$ and terminates.

We show in Fig. 8 an example of executing Algorithm 6. In Fig. 8a, we show the input CFDs and the set of cells that we need to check. In Fig. 8b, we show the initial equivalence relation $\mathcal{E}$ resulting from step 3 in Algorithm 6. Each equivalence class is shown as a black rectangle that surrounds its member cells, and the value of $E.c$ is shown in a solid black circle at the upper right corner of each rectangle. The resulting equivalence relation is shown in Fig. 8c. For example, cell (t_1, A) matches the LHS of CFD $(A \rightarrow B, (1, 1))$. Thus, we change the constant associated with the equivalence class of (t_1, B) to 1. We do the same for t_2, and we merge the equivalence classes of (t_1, B) and (t_2, B) since they are associated with the same constant, 1. Also, cells (t_3, A) and (t_4, A) match the LHS of CFD $(A \rightarrow B, (_, _))$, and thus, we merge the equivalence classes of (t_3, B) and (t_4, B), and we set the constant of the resulting equivalence class to 1. We merge the two equivalence classes $\{(t_1, B), (t_2, B)\}$ and $\{(t_3, B), (t_4, B)\}$ since their associated constants are equal. We continue the process of merging the equivalence classes, and we show the final result in Fig. 8c. The shown set of cells is clean since we can find a non-empty equivalence relation.

In the following lemma, we prove the correctness of Algorithm 6.

Algorithm 6 BuildEquivRel_CFD($\mathcal{C}, I, \Sigma$)

1: let $TIDs(\mathcal{C})$ be the set of tuples involved in $\mathcal{C}$, $\{t : (t, A) \in \mathcal{C}\}$
2: let $Attrs(\mathcal{C})$ be the set of attributes involved in $\mathcal{C}$, $\{A : (t, A) \in \mathcal{C}\}$
3: let $\mathcal{E}$ be an initial equivalence relation on the set $\{(t, A) : t \in TIDs(\mathcal{C}), A \in Attrs(\mathcal{C})\}$ such that cells in $\mathcal{C}$ that belong to the same attribute and have equal values in I are in the same equivalence class, and all other cells outside $\mathcal{C}$ belong to separate (singleton) classes
4: for all $E \in \mathcal{E}$, if E contains at least one cell from $\mathcal{C}$, set $E.c$ to the value of this cell in I. Otherwise, set $E.c$ to '_'
5: var_CFD_viols ← True
6: const_CFD_viols ← True
7: **while** var_CFD_viols = True or const_CFD_viols = True **do**
8: select two tuples $t_1, t_2 \in TIDs(\mathcal{C})$ such that there exists a variable CFD $(X \rightarrow A, t_p) \in \Sigma$ where $\forall B \in X$ $(ec(\mathcal{E}, (t_1, B)) = ec(\mathcal{E}, (t_2, B)) \wedge ec(\mathcal{E}, (t_1, B)).c \asymp t_p[B])$ and $ec(\mathcal{E}, (t_1, A)) \neq ec(\mathcal{E}, (t_2, A))$ (if no such tuples exist, set var_CFD_viols to False and skip to step 16)
9: **if** $ec(\mathcal{E}, (t_1, A)).c = _ \vee ec(\mathcal{E}, (t_2, A)).c = _ \vee ec(\mathcal{E}, (t_1, A)).c = ec(\mathcal{E}, (t_2, A)).c$ **then**
10: merge the equivalence classes $E_1 = ec(\mathcal{E}, (t_1, A))$ and $E_2 = ec(\mathcal{E}, (t_2, A))$ into one equivalence class E_{12}
11: set $E_{12}.c$ to the unnamed variable '_' if $E_1.c = E_2.c = _$, and to $E_1.c$ (respectively, $E_2.c$) if $E_1.c$ (respectively, $E_2.c$) is a constant
12: if $E_{12}.c$ is a constant and there exists another equivalence class E such that $E.c = E_{12}.c$, merge E_{12} and E
13: **else**
14: **return** $\emptyset$
15: **end if**
16: select a tuple $t_1 \in TIDs(\mathcal{C})$ such that there exists a constant CFD $(X \rightarrow A, t_p) \in \Sigma$ where $\forall B \in X$ $(ec(\mathcal{E}, (t_1, B)).c \asymp t_p[B])$ and $ec(\mathcal{E}, (t_1, A)).c \neq t_p[A]$ (if no such tuple exists, set const_CFD_viols to False and skip to step 23)
17: **if** $ec(\mathcal{E}, (t_1, A)).c = _$ **then**
18: $ec(\mathcal{E}, (t_1, A)).c \leftarrow t_p[A]$
19: merge $ec(\mathcal{E}, (t_1, A))$ with the equivalence class E of attribute A where $E.c = t_p[A]$ (if any)
20: **else**
21: **return** $\emptyset$
22: **end if**
23: **end while**
24: **return** $\mathcal{E}$

Lemma 5 *The set $\mathcal{C}$ in I is clean with respect to Σ iff procedure BuildEquivRel_CFD($\mathcal{C}, I, \Sigma$) returns an equivalence relation $\mathcal{E}$ that is not equal to $\emptyset$.*

Proof First, we prove the "if" condition as follows. Let $\mathcal{E} \neq \emptyset$ be the equivalence relation returned by BuildEquivRel_CFD. We need to prove that the set $\mathcal{C}$ is clean. We approach the proof by showing how to construct a repair I' of I such that the cells in $\mathcal{C}$ are unchanged.

We construct I' as follows. We assign each cell in $\mathcal{C}$ in I' to the same value in I (i.e., $\forall C \in \mathcal{C}, I'(C) = I(C)$). We iterate over all other cells outside $\mathcal{C}$ in random order. For each cell that belongs to an equivalence class E where $E.c$ is a constant, we set the cell value to $E.c$. In the case that $E.c = _$, if E is a singleton equivalence class or the cell belongs to a tuple that is not mentioned in $\mathcal{C}$, we set the cell value to

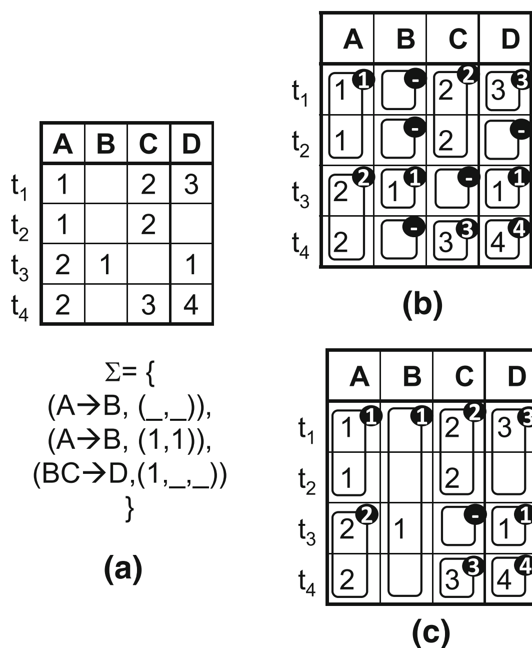

Fig. 8 An example of executing Algorithm 6: **a** Input CFDs and cells to be checked. **b** Initial equivalence relation. **c** Final equivalence relation

a new variable. If E is a non-singleton equivalence class that includes at least one cell with an already assigned value (call it x), we set $I'(C)$ to x. Finally, if E is a non-singleton equivalence class $E \in \mathcal{E}$ whose cells are not assigned to any values yet, we assign the cell value to a new variable.

Based on Algorithm 6 and our construction method of I', we list a number of facts as follows.

- For each $E \in \mathcal{E}$, $E.c$ is a constant iff E contains a cell from $\mathcal{C}$ (refer to line 3 in Algorithm 6). In this case, $E.c$ is equal to the value of that cell.
- For each pair of classes E_1 and E_2 in the same attribute, if $E_1.c$ and $E_2.c$ are constants, then $E_1.c \neq E_2.c$.
- If $E.c = _$, then all members in E are not in $\mathcal{C}$ and all members will be assigned to the same variable that is different from all other variables used in I' (based on our construction method).
- For any cells (t_1, A) and (t_2, A) that have equal values in I', both cells belong to the same equivalence class in $\mathcal{E}$.
- For any cells (t_1, A) and (t_2, A) that belong to the same equivalence class in $\mathcal{E}$, both cells have the same value in I'.

Now, we show that the constructed instance I' is indeed a repair. For every two tuples $t_1, t_2 \in TIDs(I')$ and for every variable CFD $(X \rightarrow A, t_p) \in \Sigma$, $t_1[X] = t_2[X] \asymp t_p[X]$ implies that for all $B \in X$, (t_1, B) and (t_2, B) belong to the same equivalence class E, and $E.c \asymp t_p[B]$. Based on Algorithm 6, (t_1, A) and (t_2, A) are placed in the same equivalence class and thus $I'(t_1, A) = I'(t_2, A)$.

For each tuple $t_1 \in TIDs(I')$ and for each constant CFD $(X \rightarrow A, t_p) \in \Sigma$, $t_1[X] = t_p[X]$ implies that $ec(\mathcal{E}, (t_1, B)).c = t_p[B]$ for all $B \in X$. Based on Algorithm 6, $ec(\mathcal{E}, (t_1, A)).c = t_p[A]$, and thus $I'(t_1, A) = t_p[A]$. This proves that $I' \models \Sigma$, and thus, cells in $\mathcal{C}$ are clean.

In the second part of the proof, we prove the "only if" condition. Let $\mathcal{C}$ be a set of cells that is clean. We need to show that `BuildEquivRel_CFD`$(\mathcal{C}, I, \Sigma)$ returns an equivalence relation $\mathcal{E}$ that is not equal to $\emptyset$. We first highlight the steps of the proof as follows.

- Because $\mathcal{C}$ is clean, there exists at least one repair of I, I', in which the cells in $\mathcal{C}$ are unchanged. We prove that for each intermediate non-singleton equivalence class E that is created by Algorithm 6, the member cells of E have equal values in I'. Also, for both singleton and non-singleton intermediate equivalence classes, if $E.c$ is a constant, all member cells of E in I' must be equal to this constant.
- Based on the first fact, we show that Algorithm 6 cannot return $\emptyset$.

We prove the first point as follows. The initial equivalence relation constructed in line 3 satisfies the fact mentioned in the first point: for each non-singleton equivalence class E, all members of E belong to $\mathcal{C}$ and $E.c$ is equal to the value of the member cells in I, which is equal to their value in I' as well. Also, for all equivalence classes with $E.c$ is a constant, the member cell(s) has/have to be in $\mathcal{C}$ and the value of the member cell(s) in I and I' must be equal to $E.c$.

For the equivalence classes that are subsequently created in Algorithm 6, the described fact still holds. Assume that this fact holds at a certain time of executing the algorithm. We show that this fact holds after creating a new equivalence class (i.e., when merging two existing classes or changing $E.c$ from '_' to a constant). Assume that a violation of a variable CFD $(X \rightarrow A, t_p) \in \Sigma$ is detected in line 8 involving two tuples t_1 and t_2. For $B \in X$, each pair of cells (t_1, B) and (t_2, B) belong to the same equivalence class E and $E.c \asymp t_p[B]$. Thus, $I'(t_1, B) = I'(t_2, B) \asymp t_p[B]$ for $B \in X$. The equivalence class resulting from merging $ec(\mathcal{E}, (t_1, A))$ and $ec(\mathcal{E}, (t_2, A))$ has member cells that have equal values in I' (i.e., $I'(t_1, A) = I'(t_2, A)$) because $I' \models \Sigma$.

Assume that we have a violation of constant CFD $(X \rightarrow A, t_p) \in \Sigma$ detected in line 16 involving tuple t_1. Without loss of generality, we assume that all values in $t_p[X]$ are constants [17]. Following our assumption, $ec(\mathcal{E}, (t_1, B)).c = I'(t_1, B) = t_p[B]$ for all $B \in X$. Thus, $I'(t_1, A) = t_p[A]$, which is consistent with the possible changing of $ec(\mathcal{E}, (t_1, A)).c$ to $t_p[A]$ and merging $ec(\mathcal{E}, (t_1, A))$ with any other equivalence class E' with $E'.c = t_p[A]$.

We prove the second point as follows. Assume, to the contrary, that Algorithm 6 returns $\emptyset$ in line 14. It follows that there

exist two tuples t_1 and t_2 that belong to an equivalence class E such that $E.c \asymp t_p[X]$ and thus $I'(t_1, X) = I'(t_2, X) \asymp$. Also, (t_1, A) belongs to an equivalence class E_1, and (t_2, A) belongs to a different equivalence class E_2 such that $E_1.c$ and $E_2.c$ are different constants. It follows that $I'(t_1, A) \neq I'(t_2, A)$, which cannot occur since $I' \models \Sigma$.

Assume, to the contrary, that Algorithm 6 returns $\emptyset$ in line 21. It follows that there exists a tuple t_1 that belongs to an equivalence class E such that $E.c$ is a constant and $E.c = I'(t_1, X) = t_p[X]$. Also, $t_1[A]$ belongs to an equivalence class E_1 such that $E_1.c$ is a constant and $E_1.c = I'(t_1, A) \neq t_p[A]$, which cannot occur as $I' \models \Sigma$. Thus, Algorithm 6 cannot return $\emptyset$. □

7.2 Extending the sampling algorithm

The algorithm that generates a maximal clean set of cells (Algorithm 3) remains unchanged; we only need to replace the condition $isClean(CleanSet \cup \{C\}, I, \Sigma, \mathcal{E})$ in line 4 with $\mathcal{E} \neq \emptyset$. The sampling algorithm, on the other hand, must take into consideration the additional constraints modeled by the extended equivalence classes. Algorithm 7 describes the modified sampling algorithm.

The main modification is inserting a new condition in line 8 to capture the case where $E.c$ is a constant. In this case, we must change the cell values in I' to this constant. The remainder of the algorithm is similar to Algorithm 4.

Algorithm 7 Gen-Repair-CFD(I, Σ)

1: $I' \leftarrow I$
2: $MaxCleanCells \leftarrow$ GetMaxCleanSet(I, Σ)
3: $Cleaned \leftarrow MaxCleanCells$
4: $\mathcal{E} \leftarrow$ BuildEquivRel_CFD($Cleaned$, I, Σ)
5: **for** each $(t, A) \in CIDs(I) \backslash MaxCleanCells$ (based on a random iteration order) **do**
6: **if** (t, A) belongs to a non-singleton equivalence class in $\mathcal{E}$ that contains other cells in $Cleaned$ **then**
7: set $I'(t, A)$ to the value (either a constant or a variable) of the other cells in $ec(\mathcal{E}, (t, A)) \cap Cleaned$
8: **else if** $ec(\mathcal{E}, (t, A)).c$ is a constant **then**
9: set $I'(t, A)$ to $ec(\mathcal{E}, (t, A)).c$
10: **else**
11: randomly set $I'(t, A)$ to one of three alternatives: a randomly selected constant from $ADom(A, I)$, a randomly selected variable v_i^A that was previously used in I', or a fresh variable v_j^A such that $Cleaned \cup \{(t, A)\}$ becomes clean
12: **end if**
13: $Cleaned \leftarrow Cleaned \cup \{(t, A)\}$
14: $\mathcal{E} \leftarrow$ BuildEquivRel_CFD($Cleaned$, I', Σ)
15: **end for**
16: **return** I'

We show an example of generating a repair using Algorithm 7 in Fig. 9. In Fig. 9a, we show the input instance and the set of CFDs. In Fig. 9b, we show a maximal set of clean cells that is obtained using Algorithm 3, which in turn uses

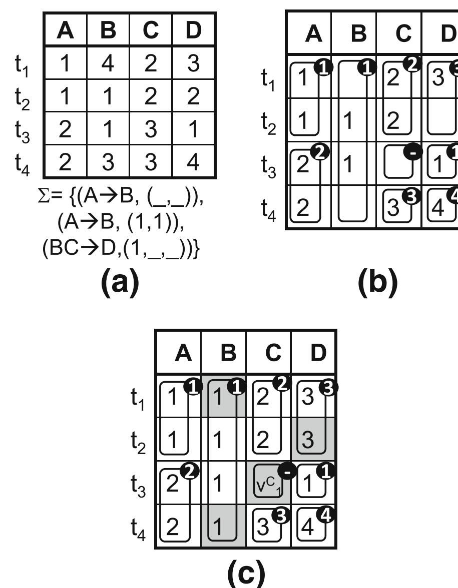

Fig. 9 An example of executing Algorithm 7: **a** Input CFDs and relation instance to be checked. **b** A maximal set of clean cells. **c** The final repair

Algorithm 6 for checking whether cells are clean. In Fig. 9c, we show the repair returned by Algorithm 7. Cells (t_1, B) and (t_4, B) belong to an equivalence class that is associated with a constant equal to 1. Thus, they are modified to the value 1. Cell (t_3, C) is associated with a singleton equivalence class that is not associated with a constant. Therefore, one possibility based on line 11 in Algorithm 7 is to assign (t_3, C) to a new variable.

We show in the following theorem that repairs generated by Algorithm 7 are cardinality-set-minimal.

Theorem 4 *Every instance that is generated by Algorithm 7 is a cardinality-set-minimal repair.*

Proof The proof is similar to the proof of Theorem 2. We prove that every instance I' generated by Algorithm 7 is a repair of I by showing that the set of cells in I' is clean. Initially, the set $Cleaned = MaxCleanCells$ is clean with respect to Σ (based on Lemmas 3 and 5). In each iteration, the algorithm adds a cell to $Cleaned$ and changes this cell to ensure that the resulting version of $Cleaned$ is clean as well. This is done by assigning a value to each newly added cell that satisfies the constraints represented by $\mathcal{E}$. Upon termination, all cells in I' are in $Cleaned$, which indicates that the resulting instance I' satisfies Σ.

Second, we prove that each generated repair is cardinality-set-minimal. The initial maximal clean set of cells, denoted $MaxCleanCells$, is not modified throughout the algorithm. Thus, the set of unchanged cells in any generated repair represents a maximal clean set of cells, which indicates that the generated repair is cardinality-set-minimal based on Theorem 1. □

Some cardinality-minimal repairs cannot be generated by Algorithm 7 due to the constraint that different variables in the generated repairs cannot be assigned to the same value (refer to Definition 1). It is possible to alter the definition of a V-instance to take into account all possible assignments to variables that result in a valid repair. However, we argue that the resulting substitution process will be too complicated for the end user to follow. Another alternative is to directly generate a ground instance using our sampling algorithm. In order to implement this alternative, we replace the three options in line 11 with only one option that assigns the cell (t, A) in I' to a value from the domain of attribute A, $Dom(A)$, such that the set $Cleaned \cup \{(t, A)\}$ is clean.

7.3 Extending block-wise repairing

For the block-wise sampling algorithm, we need to restrict the values assigned to unclean cells in line 11 in Algorithm 7 to the third alternative only (i.e., assigning a new variable to the unclean cell) for the same reasons described in Sect. 6. Also, it is safe to remove line 14 which updates the equivalence relation $\mathcal{E}$. We call the modified algorithm `Block-Gen-Repair-CFD`.

The core of Algorithm 5, which partitions a given instance w.r.t. a set of FDs, is the fact that cells in different equivalence classes in $\mathcal{E}_0$ cannot have equal values during the repairing process (Sect. 6). In the following, we show how to construct $\mathcal{E}_0$ for CFDs in Σ as described in Algorithm 8. The key idea is to detect all *possible* violations of CFDs and to modify $\mathcal{E}_0$ to reflect the possible ways of repairing the violations. Note that because $\mathcal{E}_0$ corresponds to all possible executions of the randomized cleaning algorithm, each equivalence class $E \in \mathcal{E}_0$ could be assigned to various constants (i.e., $E.c$ is not fixed for all possible executions). Therefore, we extend the equivalence relation $\mathcal{E}_0$ by associating each equivalence relation with a *set* of possible constants, denoted $E.S$.

Algorithm 8 builds $\mathcal{E}_0$ in a way that is similar to how Algorithm 6 builds $\mathcal{E}$. The key difference is that Algorithm 8 does not terminate when contradicting constraints are found (e.g., when trying to merge two equivalence classes that are associated with different constants). Algorithm 6 views the constraints defined by equivalence classes as *possible* constraints, and thus, it allows merging contradicting equivalence classes and associates the resulting equivalence classes with the union of the constants associated with the merged classes (lines 7, 8, 11, 12).

Algorithm 9 extends Algorithm 5 by checking for possible CFD violations instead of FD violations. This is achieved by extending the condition $ec(\mathcal{E}_0, (t_i, B)) = ec(\mathcal{E}_0, (t_j, B))$ in line 4 in Algorithm 5 to also check if the constant of $ec(\mathcal{E}_0, (t_i, B))$ can possibly match the LHS of the considered CFD. Also, we check whether each tuple t_i could violate a constant CFD $X \rightarrow A$ (lines 8–12). In this case, we merge the partitions of cells in (t_i, XA).

Algorithm 8 `Build_`$\mathcal{E}_0$`_CFD(`I, Σ`)`

1: let $\mathcal{E}_0$ be an initial equivalence relation on all cells in I such that cells belonging to the same attribute and having equal values in I are in the same equivalence class
2: for all $E \in \mathcal{E}_0$, set $E.S$ to $\{v\}$, where v is the value of the member cells in I
3: `var_CFD_viols` ← `True`
4: `const_CFD_viols` ← `True`
5: **while** `var_CFD_viols = True` or `const_CFD_viols = True` **do**
6: select any tuples $t_1, t_2 \in TIDs(I)$ such that there exists a variable CFD $(X \rightarrow A, t_p) \in \Sigma$ where $\forall B \in X$ $(ec(\mathcal{E}, (t_1, B)) = ec(\mathcal{E}, (t_2, B)) \wedge \exists c \in ec(\mathcal{E}, (t_1, B)).S(c \asymp t_p[B]))$ and $ec(\mathcal{E}, (t_1, A)) \neq ec(\mathcal{E}, (t_2, A))$ (if no such tuples exist, set `var_CFD_viols` to `False` and skip to step 9)
7: merge the equivalence classes $E_1 = ec(\mathcal{E}, (t_1, A))$ and $E_2 = ec(\mathcal{E}, (t_2, A))$ into one equivalence class E_{12}
8: $E_{12}.S \leftarrow E_1.S \cup E_2.S$
9: select any tuple $t_1 \in TIDs(I)$ such that there exists a constant CFD $(X \rightarrow A, t_p) \in \Sigma$ where $\forall B \in X$ $(\exists c \in ec(\mathcal{E}, (t_1, B)).S$ $(c \asymp t_p[B]))$ and $t_p[A] \notin ec(\mathcal{E}, (t_1, A)).S$ (if no such tuple exists, set `const_CFD_viols` to `False` and skip to step 12)
10: Add $t_p[A]$ to $ec(\mathcal{E}, (t_1, A)).S$
11: merge $E_1 = ec(\mathcal{E}, (t_1, A))$ with the equivalence classes E_2 of attribute A where $t_p[A] \in E_2.S$ (if any) resulting in E_{12}
12: $E_{12}.S \leftarrow E_1.S \cup E_2.S$
13: **end while**
14: **return** $\mathcal{E}_0$

We show in Fig. 10 an example of partitioning an instance using Algorithm 9, which depends on Algorithm 8 to build $\mathcal{E}_0$. In Fig. 10a, we show the input relation instance and the CFDs. In Fig. 10b, we show the initialization of $\mathcal{E}_0$ that is performed in lines 1, 2 in Algorithm 8. Figure 10c shows relation $\mathcal{E}_0$ resulting from Algorithm 8. For example, we merge the equivalence classes $\{(t_1, B)\}$ and $\{(t_2, B), (t_3, B)\}$ since $t_1[A]$ matches the LHS of CFD $(A \rightarrow B, (1, 1))$. The resulting equivalence class is associated with the constant set $\{1, 4\}$. The algorithm keeps merging the equivalence classes, and the final equivalence relation $\mathcal{E}_0$ is returned. In Fig. 10d, we show the cell partitioning resulting from Algorithm 9. The algorithm starts with singleton blocks and merges blocks that could possibly contain a violation throughout the execution of the cleaning algorithm. For example, cells (t_1, A) and (t_1, B) could possibly violate the constant CFD $(X \rightarrow A, (1, 1))$, and thus, they are placed in the same block. Also, cells (t_1, BCD) and (t_2, BCD) could violate the variable CFD $(BC \rightarrow D, (1, _, _))$, and thus, their blocks are merged together. The algorithm terminates when all possible violations are processed.

In the following, we prove the correctness of Algorithms 8 and 9.

Algorithm 9 `Partition_CFD(I, Σ)`

1: $\mathcal{E}_0$ = Build_$\mathcal{E}_0$_CFD(I, Σ)
2: Initialize the set of blocks $\mathcal{P}$ such that each cell in I belongs to a separate block
3: **for** each variable CFD $(X \to A, t_p) \in \Sigma$ **do**
4: **for** each pair of tuples $t_i, t_j \in I$ such that $\forall B \in X(ec(\mathcal{E}_0, (t_i, B)) = ec(\mathcal{E}_0, (t_j, B)) \wedge \exists c \in ec(\mathcal{E}_0, (t_i, B)).S(c \asymp t_p[B]))$ **do**
5: merge the blocks of the cells $(t_i, XA) \cup (t_j, XA)$
6: **end for**
7: **end for**
8: **for** each constant CFD $(X \to A, t_p) \in \Sigma$ **do**
9: **for** each tuple $t_i \in I$ such that $\forall B \in X(\exists c \in ec(\mathcal{E}_0, (t_i, B)).S(c \asymp t_p[B]))$ **do**
10: merge the blocks of the cells in (t_i, XA)
11: **end for**
12: **end for**
13: **return** $\mathcal{P}$

Lemma 6 *Given $\mathcal{E}_0$ constructed by procedure* Build_$\mathcal{E}_0$_CFD(I, Σ)*, every two cells that have equal values in at least one possible repair generated by Algorithm* Block-Gen-Repair-CFD(I, Σ) *are necessarily in the same equivalence class in $\mathcal{E}_0$. Also, any cell that is assigned to a constant c in at least one repair generated by Algorithm* Block-Gen-Repair-CFD(I, Σ) *is necessarily in an equivalence class E in $\mathcal{E}_0$ where $c \in E.S$.*

Proof The proof is divided into two steps as follows.

– We prove that if there exists any possible repair I' generated by Algorithm Block-Gen-Repair-CFD in which (t_1, A) and (t_2, A) have the same value, (t_1, A) and (t_2, A) belong to the same equivalence class in $\mathcal{E}$ that is created in line 4 in Algorithm 7. Also, if a cell (t_1, A) is assigned to a constant c in I', then (t_1, A) belongs to an equivalence class E in $\mathcal{E}$ where $E.c = c$.
– We prove that for each equivalence class $E_1 \in \mathcal{E}$, there exists an equivalence class $E_2 \in \mathcal{E}_0$ such that $E_1 \subseteq E_2$. Also, if $E_1.c$ is a constant, then $E_1.c \in E_2.S$.

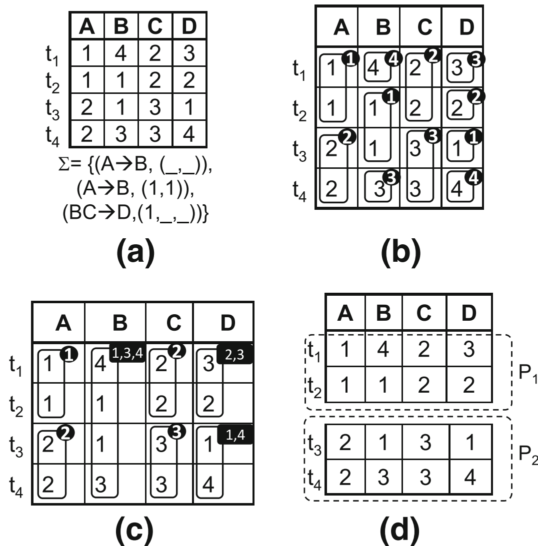

Fig. 10 An example of partitioning an instance: **a** Input CFDs and relation instance. **b** Initialization of $\mathcal{E}_0$. **c** Final shape of $\mathcal{E}_0$. **d** Partitioned instance

We prove the first point as follows. Given a specific maximal clean set of cells, the repair generated by Algorithm Block-Gen-Repair-CFD is identical to the repair constructed in the proof of Lemma 5 (modulo the randomization resulting from random selection of cells to be processed next). Thus, the five facts mentioned in the proof of Lemma 5 hold on any repair generated by Algorithm Block-Gen-Repair-CFD as well. It follows that if (t_1, A) and (t_2, A) have the same value in I', they belong to the same equivalence class in $\mathcal{E}$. Also, if $I'(t_1, A)$ is a constant, then (t_1, A) belongs to an equivalence class E in $\mathcal{E}$ such that $E.c = I'(t_1, A)$.

We prove the second point as follows. Let I_c be an instance where each cell in $MaxCleanCells$ has the same value as in I, and all other cells are assigned to unique variables. The equivalence relations $\mathcal{E}_0$ = Build_$\mathcal{E}_0$_CFD(I_c, Σ) and $\mathcal{E}$ = BuildEquivRel_CFD($MaxCleanCells$, I, Σ) are identical. That is, for each $E_1 \in \mathcal{E}$, there exists $E_2 \in \mathcal{E}_0$ such that $E_1 = E_2$ and $((E_1.c = _ \wedge E_2.S = \{v_i\}) \vee \{E_1.c\} = E_2.S)$. This is because the initial equivalence relations are equal (compare line 1 in Algorithm 8 and line 3 in Algorithm 6), and subsequent operations are identical. Now, assume that we replace one variable in I_c with the value found in I. Clearly, no equivalence class in $\mathcal{E}_0$ will be split; only merging equivalence classes could occur. Also, for each equivalence class E, Algorithm 8 can only replace a variable in $E.S$ with a constant and possibly expand the set of constants in $E.S$. By repeating this process, we conclude that the equivalence classes in Build_$\mathcal{E}_0$_CFD(I_c, Σ) are contained in equivalence classes in Build_$\mathcal{E}_0$_CFD(I, Σ), and for each $E_1 \in$ Build_$\mathcal{E}_0$_CFD(I_c, Σ), there exists $E_2 \in$ Build_$\mathcal{E}_0$_CFD(I, Σ) such that $E_1.S \subseteq E_2.S$, which completes the proof. □

Theorem 5 *Given a partitioning of cells in an instance I that is constructed by Algorithm* Partition_CFD(I, Σ)*, repairing the individual blocks of cells separately using Algorithm* Block-Gen-Repair-CFD *results in a repair of I.*

Proof We need to prove that for each variable CFD $(X \to A, t_p) \in \Sigma$, and for any two tuples t_1 and t_2, if $I'(t_1, B) = I'(t_2, B) \asymp t_p[B]$ for all $B \in X$, then cells (t_1, X), (t_1, A), (t_2, X), and (t_2, A) are necessarily in the same block. Also, for each constant CFD $(X \to A, t_p) \in \Sigma$ and for each tuple t_1, if $I'(t_1, B) = t_p[B]$ for all $B \in X$, the cells (t_1, X) and (t_1, A) are in the same block.

Given a variable CFD $(X \to A, t_p) \in \Sigma$, let t_1 and t_2 be any tuples in I' such that $I'(t_1, B) = I'(t_2, B) \asymp t_p[B]$

for all $B \in X$. For each attribute $B \in X$, since $I'(t_1, B) = I'(t_2, B)$, (t_1, B), and (t_2, B) are in the same equivalence class in $\mathcal{E}_0$ based on Lemma 6. Based on Algorithm 9, cells (t_1, X), (t_1, A), (t_2, X), and (t_2, A) must be in the same block.

Given a constant CFD $(X \rightarrow A, t_p) \in \Sigma$, let t_1 be any tuple in I' such that $I'(t_1, B) = t_p[B]$ for all $B \in X$. For each attribute $B \in X$, since $I'(t_1, B)$ is a constant, (t_1, B) is in an equivalence class E in $\mathcal{E}_0$ such that $I'(t_1, B) = E.c$ based on Lemma 6. Based on Algorithm 9, cells (t_1, X), (t_1, A) must be in the same block. □

8 Experimental study

In this section, we present an experimental evaluation of our approach. The goal of our experiments is twofold. First, we show that the proposed algorithms can efficiently generate random repairs. Second, we use our repair generator to study the correlation between the number of changes in a repair and the quality of the repair.

For completeness, we implemented three previous approaches that deterministically repair FD/CFD violations [9,12,23]. The goal of these approaches is to obtain a single repair that is cost-minimal, which is an NP-hard problem. Since obtaining a cardinality-set-minimal repair can be done in PTIME, the running time of these algorithms is not directly comparable to that of our algorithm. However, we report the running times to put the performance of our algorithms into perspective. For example, these results can show how many repairs can be produced by our algorithm in the time taken to generate one repair using one of the previous algorithms.

8.1 Setup

All experiments were conducted on a SunFire X4100 server with a Dual Core 2.2 GHz processor, and 8 GB of RAM. All computations are executed in memory. We used both synthetic and real data sets. The synthetic data is generated by a modified version of the UIS database generator [1]. This program produces a mailing list that has the following schema: `RecordID, SSN, FirstName, MiddleInit, LastName, StNumber, StAddr, Apt, City, State, ZIP`. The following FDs are defined on the schema:

- `SSN` → `FirstName, MiddleInit, LastName, StNumber, StAddr, Apt, City, State, ZIP`
- `FirstName, MiddleInit, LastName` → `SSN, StNumber, StAddr, Apt, City, State, ZIP`
- `ZIP` → `City, State`

We used the following constant CFDs (we omit the CFD tableaux for brevity; the tableau of each CFD is set to the most frequent 10 patterns).

- `LastName` → `StNumber`
- `LastName` → `StAddr`
- `LastName` → `Apt`

Also, we used the following variable CFDs (we omit the CFD tableaux for brevity; the LHS attributes in the tableaux are set to the most frequent 10 values where the FD holds).

- `LastName` → `City`
- `LastName` → `State`
- `LastName` → `ZIP`

We chose these CFDs based on the intuition that married couples, or families in general, with the same last name would have the same address.

The UIS data generator was originally created to construct mailing lists that have duplicate records. We modified it to generate two instances: a clean instance I_c and another instance I_d that is obtained by marking random perturbations to cells in I_c. These perturbations include modifying characters in attributes, swapping the first and last names, and replacing SSNs with all zeros to indicate missing values. To control the amount of perturbation, we use a parameter P_{pert} that represents the probability of modifying a tuple $t \in I_c$ by altering one or more attributes. The default value for P_{pert} is 5 %. Note that not every cell modification results in an CFD violation (e.g., changing attributes that are not mentioned in any CFD, or changing a LHS attribute to a unique value).

The real data set we used consists of the census-income data [1], which is part of the UC Irvine Machine Learning Repository. We select 10,000 tuples from this data set to evaluate the quality of the generated repairs. The data set has 42 attributes. In the following, we briefly describe the attributes that appear in the FDs and CFDs we used.

- `region_of_previous_residence`: indicates the previous region of residence (if applicable). Possible values are Not in universe, South, Northeast, West, Midwest, Abroad.
- `migration_code_change_in_reg`: indicates whether the region of residence has changed. Possible values are Not in universe, Non-mover, Same county, Different county same state, Different state same division, Abroad, Different region, Different division same region.

[1] http://archive.ics.uci.edu/ml/datasets/Census-Income+(KDD).

- migration_code_move_within_reg: indicates whether a person has changed his/her residence within the same region. Possible values are Not in universe, Non-mover, Same county, Different county same state, Different state in West, Abroad, Different state in Midwest, Different state in South, Different state in Northeast.
- migration_code_change_in_msa: indicates whether a person has changed his/her Metropolitan Statistical Area of residence. Possible values are Not in universe, Non-mover, MSA to MSA, NonMSA to nonMSA, MSA to nonMSA, NonMSA to MSA, Abroad to MSA, Not identifiable, Abroad to nonMSA.
- live_in_this_house_1_year_ago: Indicates whether a person lived in the same house for more than one year. Possible values are Not in universe under 1 year, Yes, No.
- occupation_code: a code representing the person's occupation. Total number of codes is 47.
- major_occupation_code: a code representing the coarse-grained occupation of a person. There are 15 possible codes in total.
- industry_code: a code representing the detailed industry classification to which a person is associated. In total, there are 52 possible values.
- major_industry_code: a code representing the coarse-grained classification of the industry. There are 24 possible codes.
- education: indicates the education level of a person.
- income: indicates the total income of a person.

The FDs are chosen from a discovered set that approximately held on our 10,000 tuple data sample, based on having a reasonable number of LHS attributes. The used FDs are as follows.

- region_of_previous_residence, migration_code_change_in_reg → migration_code_move_within_reg
- occupation_code → major_occupation_code
- migration_code_change_in_msa → live_in_this_house_1_year_ago

We also used the following CFDs.

- A constant CFD with FD template being education → income.
- A variable CFD with FD template being industry_code → major_industry_code.

The unconditional versions of these CFDs approximately hold for the database instance. We populate the tableau of each CFD with the most frequent 10 patterns.

We use five approaches to clean the instance I_d that are described as follows.

- Sampling: This approach implements Algorithm 7 for repairing FDs and CFDs. One optimization we introduced in our implementation is obtaining the equivalence relation $\mathcal{E}$ in an incremental way by updating $\mathcal{E}$ every time a cell is inserted instead of recomputing $\mathcal{E}$ from scratch.
- Block-wise: This approach partitions the input instance using Algorithm 5 into disjoint blocks, and then uses Algorithm Block-Gen-Repair-CFD (the modified version of Algorithm 7 for repairing each individual block.
- Vertex Cover [23]: This approach is based on modeling CFD violations as hyper-edges and using an approximate minimum vertex cover of the resulting hyper-graph to find a repair with a small number of changes.
- Greedy-RHS [9]: This approach repeatedly picks the FD violation that is cheapest to repair, based on the cost function described in Sect. 3, and fixes it. Modifications are only performed to the RHS attributes of the violated FDs.
- Greedy [12]: This approach extends the algorithm in [9] to repair violations of CFDs. This algorithm could possibly change the RHS and/or the LHS attributes of violated CFDs.

The cost model that is used for implementing Algorithms Greedy and Greedy-RHS rely on a constant function, dis, that returns 1 for all pairs of different values and 0 for equal values (we found that more sophisticated metrics such as the Damerau–Levenshtein (DL) distance did not improve the repairing quality). Each tuple is associated with a confidence weight that is inversely proportional to the number of violations each tuple is involved in.

8.2 Performance analysis

We used the synthetic data set for evaluating the performance of the cleaning algorithms due to ability to manipulate the data size and the number of errors. In Fig. 11, we show the running time for generating one repair for violations of FDs only. We first fixed the perturbation probability at 5 % and used different numbers of tuples in the database. In the second experiment, we fixed the data size at 5,000 tuples and used various perturbation probabilities. We report the average runtime for generating five repairs. For Algorithm Block-wise, the cost of the initial partitioning of the input instance is amortized across the generated repairs.

Algorithm Block-wise provides the best scalability, followed by Algorithm Sampling. More specifically, Algorithm Block-wise is more than one order of magnitude

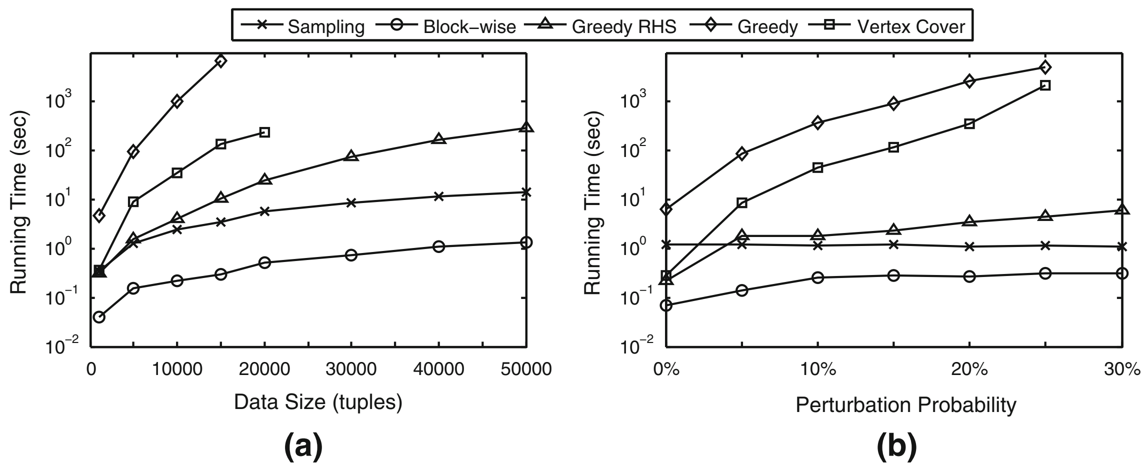

Fig. 11 The running time for generating a repair of FD violations: **a** Against various data sizes. **b** Against various perturbation probabilities

faster than Algorithm `Sampling`, two orders of magnitude faster than `Greedy-RHS` (for sizes $\geq$30,000 tuples), and three orders of magnitudes faster than `Vertex Cover`. The memory requirements of Algorithm `Vertex Cover` grow quickly as the number of violations increase due to the large number of the hyper-edges in the initial hypergraph (e.g., 2.2 million hyper-edges when the input instance contains 15,000 tuples). Algorithm `Greedy` is slower than `Greedy-RHS` since it considers changing both RHS and LHS cells.

The running time of our sampling approach is almost linear in the number of generated repairs (i.e., the sample size) because the running time for generating a random repair is almost constant.

Figure 11 depicts the running time of the algorithms for various levels of errors in the input instance, which is captured by parameter P_{pert}. The performance of Algorithm `Sampling` is constant for all perturbation levels. The reason is that most of the running time is consumed in obtaining a maximal clean set of cells by iterating over all cells in the database, which is constant in this experiment. The running time of Algorithm `Block-wise` is almost constant as well (after perturbation probability of 10 %). The running time of Algorithm `Vertex Cover` grows rapidly with the perturbation probability because larger numbers of violations result in larger conflict graphs. Most of the running time consists of building these conflict graphs.

Figure 12 shows the running times of the algorithms when the set of constraints Σ contains the FDs and the CFDs described in Sect. 8.1. The performance of each algorithm is almost the same as in the case of having only FDs in Σ (cf. Figure 11).

We conclude that the number of repairs that can be sampled by our algorithm is in the order of hundreds, given the same amount of time that is required to generate a single repair by previous algorithms.

8.3 Relationship between the number of changes and repair quality

In this section, we use our repair sampling algorithm to study the correlation between the number of changes in a repair and the quality of the repair, given that the ground truth is available. The goal of this study is to verify that repairs which make the fewest changes do not always have the highest quality, which justifies adopting cardinality-set-minimality.

We use the precision (i.e., the percentage of correct data changes) of the performed changes with respect to the given ground truth as a quality metric. Note that we do not report the recall (i.e., the percentage of errors that have been corrected) because some errors in the data set do not lead to violations of FDs (e.g., typos in the first name).

We use the clean instance I_c as the ground truth to assess the quality of a given repair I_r. First, we show how to count the number of correct changes in I_r. We denote by $CC(I_r)$ the set of cells that has been correctly fixed in I_r.

$$CC(I_r)=\{C \in \Delta(I_d, I_r) : I_r(C)=I_c(C) \wedge I_d(C) \neq I_c(C)\}$$

Replacing an incorrect value of a cell in I_d (with respect to I_c) with a variable can be considered as a *partially correct change*. We denote by $CV(I_r)$ the set of cells that are partially corrected.

$$CV(I_r) = \{C \in \Delta(I_d, I_r) : I_r(C) \; is \; a \; variable \wedge I_d(C) \neq I_c(C)\}$$

We define the number of correct changes as the sum of the cardinality of $CC(I_r)$ and a fraction (0.5 in our experiments) of the cardinality of $CV(I_r)$. We compute the precision of a repair I_r as the ratio between the number of correct changes in I_r and the total number of changes in I_r.

We measured the precision of the repairs generated by all approaches. However, for clarity of presentation, we omit the

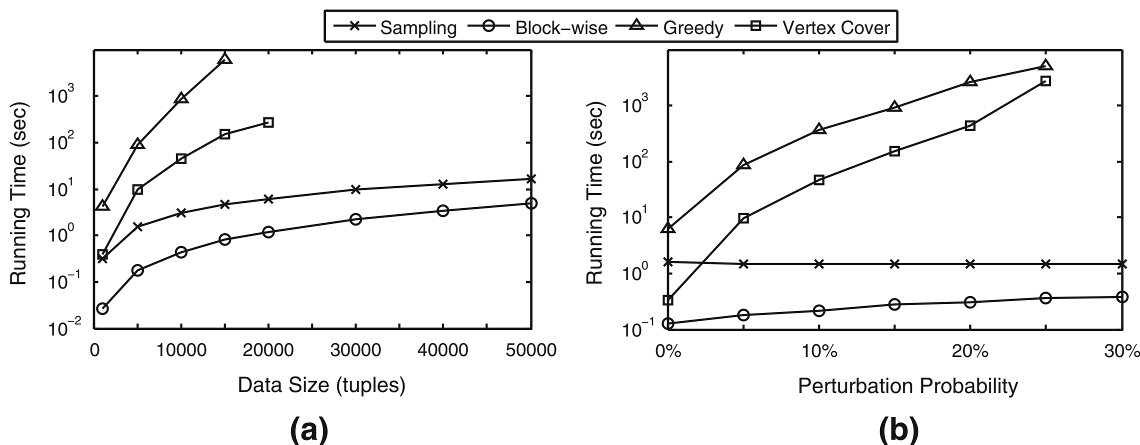

Fig. 12 The running time for generating a repair of CFD violations: **a** Against various data sizes. **b** Against various perturbation probabilities

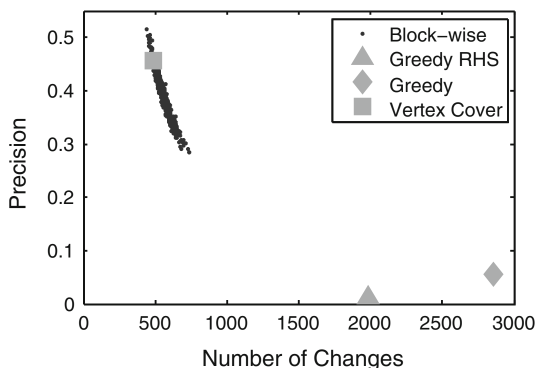

Fig. 13 Precision of the generated repairs of FD violations in synthetic data set

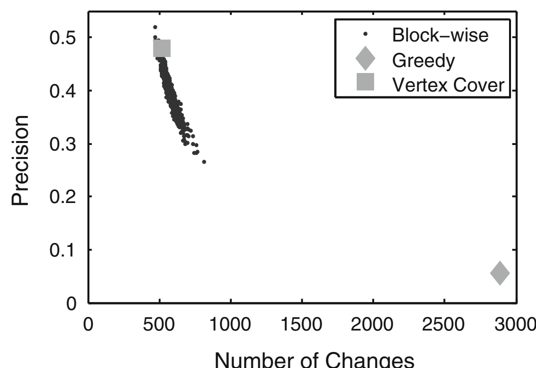

Fig. 14 Precision of the generated repairs of CFD violations in synthetic data set

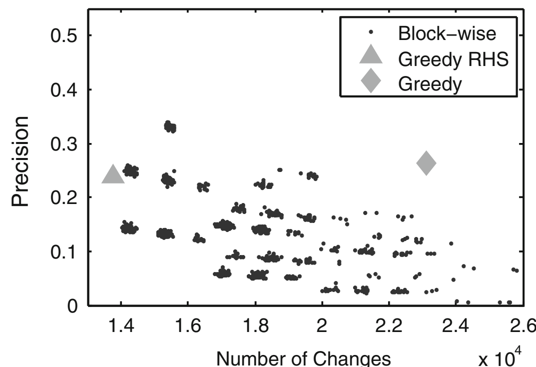

Fig. 15 Precision of the generated repairs of FD violations in real data set

results of Algorithm `Sampling` and we only describe these results in our discussion. Figures 13 and 14 show the quality results for the case of FDs and CFDs, respectively, using the synthetic data set (Algorithm `Greedy-RHS` does not support CFDs). The input instance consists of 5,000 tuples, and the parameter P_{pert} is set to 5 %. Figures 15 and 16 show the precision for the case of FDs and CFDs, respectively, using the real data set. Algorithm `Vertex Cover` ran out of memory in the real data set, and thus, it is not shown in Figs. 15 and 16. In all cases, Algorithms `Sampling` and `Block-wise` were executed 500 times (due to the randomness of the generating process) , while Algorithms `Vertex Cover` and `Greedy-RHS` (when applicable) were executed once.

The main observation is that the quality of repairs, represented by precision, is not always correlated to the number of changes in repairs. For example, in the real data set, the best repair is not the one with the minimum number of changes. On the other hand, we observe a strong correlation between precision and the number of changes in the synthetic data sets (Figs. 13 and 14). The main conclusion is that cardinality-minimality is not always a trustworthy quality criterion. Relying on minimality of changes heavily depends on the characteristics of the data (e.g., the distribution of attribute values, the causes of data errors, and the amount of redundancy in the data).

We found that the precision of the repairs generated by Algorithm `Sampling` is much lower than the repairs gen-

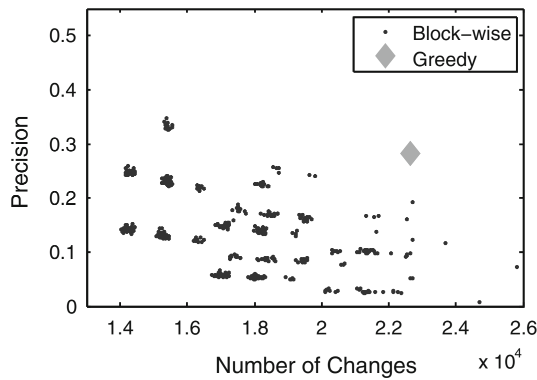

Fig. 16 Precision of the generated repairs of CFD violations in real data set

erated by Algorithm `Block-wise` (<0.1 in average). The reason is that the former algorithm can assign constant values to modified cells in line 9, which are very unlikely to be equal to the correct values. Algorithm `Block-wise` avoids this by always assigning a variable to represent a range of possible values.

In Fig. 13, Algorithm `Greedy-RHS` has lower precision than the other algorithms. The main reason is that this algorithm performs changes only to the RHS attributes of FDs. Thus, errors in LHS attributes of FDs are always fixed in the wrong way. For example, missing SSNs are usually replaced by all zeros. Algorithm `Greedy-RHS` changes all attributes of tuples with missing SSNs to the same value instead of replacing missing SSNs with variables.

Algorithm `Vertex Cover` provides a relatively high precision compared to other approaches. The reason is that Algorithm `Vertex Cover` uses an *approximate* minimum vertex cover to decide which cells should be changed, which results in a relatively small number of changes.

Algorithm `Greedy` provides a relatively high precision in the real data set. However, it does not perform as well in the case of synthetic data. The reason is that, for most incorrectly changed cells, the new incorrect values had higher frequency than the correct values. Also, in the cases where both incorrect and correct values had the same frequency (usually = 1), there was a very small difference in the tuples' weights to prefer the incorrect value over the correct value (i.e., the tuple of the correct value is involved in more violations).

9 Related work

9.1 Single repair cleaning algorithms

In the context of repairing the violations of FDs and other integrity constraints, the most popular approach has been to obtain a single repair that is as close as possible to the input database instance based on some distance function

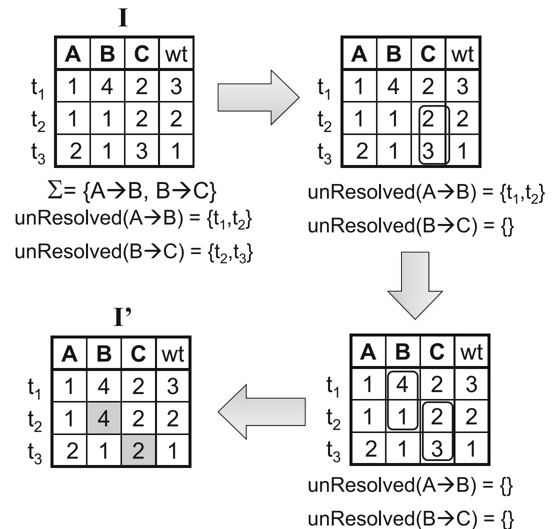

Fig. 17 An example of a repair generated by the algorithm in [9] that is not set-minimal

(e.g., [9,12,23]). We adopt a different approach of generating a sample of repairs that are not necessary cost-minimal through a randomized cleaning algorithm.

Single repair cleaning algorithms cannot be easily altered to *efficiently* generate a large number of random repairs from a well-defined space of repairs. Furthermore, randomizing single repair algorithms by simply replacing deterministic decisions (e.g., which violation to resolve first and how to resolve each violation) by randomized decisions may lead to an ad-hoc sampling space that is difficult to define in a non-procedural way. Also, the resulting sampling space might contain repairs that are unlikely to be correct (e.g., repairs that are not set-minimal) or might miss interesting repairs (e.g., cardinality-minimal repairs). For example, the algorithm in [9] can produce repairs that are not set-minimal while the algorithm in [23] could miss some cardinality-minimal repairs. In the following, we give examples to illustrate these two cases.

First, we show that the algorithm introduced in [9] may generate repairs that are not set-minimal (i.e., contain unnecessary changes). The algorithm repairs an input instance by repeatedly searching for tuples that violate FDs in Σ and selecting a tuple t that violates an FD $X \rightarrow A$ and can be resolved in the cheapest way. Then, the algorithm merges the equivalence classes of (t, A) and the attribute A of other tuples with the same values of X. For example, in Fig. 17, we show an instance I that violates $\Sigma = \{A \rightarrow B, B \rightarrow C\}$. Attribute wt reflects the user confidence about the accuracy of each tuple in I. Initially, the sets $unResolved(A \rightarrow B)$ and $unResolved(B \rightarrow C)$ contain the tuples that violate $A \rightarrow B$ and $B \rightarrow C$, respectively. In the first step, the algorithm selects either t_2 or t_3 from $unResolved(B \rightarrow C)$ to repair since they are the cheapest tuples to resolve. The

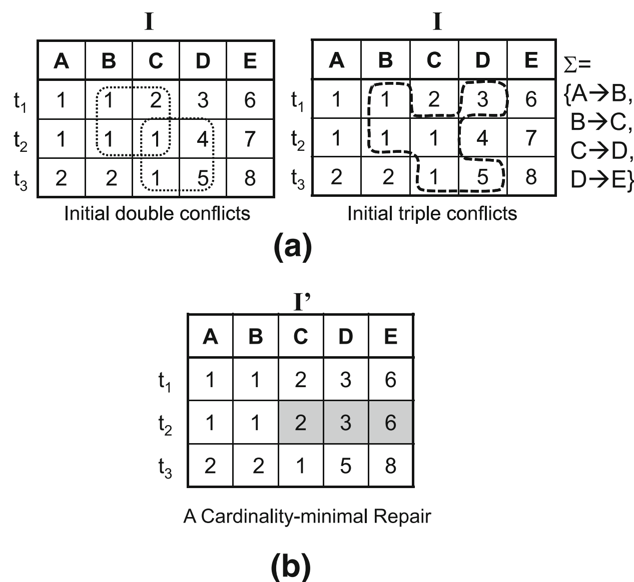

Fig. 18 An example of a cardinality-minimal repair that cannot be generated by the algorithm in [23]

algorithm repairs this violation by merging the equivalence classes of (t_2, C) and (t_3, C). In the second step, the algorithm selects either t_1 or t_2 from $unResolved(A \rightarrow B)$ to repair by merging the equivalence classes of (t_1, B) and (t_2, B). Finally, a repair I' is obtained by replacing each cell in an equivalence class with a "target" value of the equivalence class that has the minimum cost. For example, (t_2, B) is assigned to 4, and (t_3, C) is assigned to 2. Modified cells are shaded in the figure. The resulting repair I' is not set-minimal because the cell (t_3, C) can be reverted to its value in I without violating any FD.

We show that some cardinality-minimal repairs cannot be generated by the approach presented in [23]. We illustrate this fact using the example in Fig. 18. In Fig. 18a, we show the hyper-edges (also called double and triple conflicts) that exist in the initial conflict graph. The algorithm in [23] can only change a cell (t, B) if it appears in the initial conflict graph, or there exists an FD $X \rightarrow A \in \Sigma$ such that $B \in X$ and (t, A) appears in the initial conflict graph. It follows that the cell (t_2, E) in Fig. 18 can never be changed by the algorithm. Therefore, the cardinality-minimal repair I' that is shown in Fig. 18b cannot be generated by the algorithm in [23].

A related problem is checking whether a given instance is a repair of the input instance. For example, in [3], the authors studied the complexity of deciding whether a specific instance is a minimal repair (under several semantics of minimality) for larger classes of integrity constraints. New definitions of minimality are also proposed in [3] such as *component cardinality repairs*, which generalize cardinality-minimal repairs of database instances that consist of multiple relations (i.e., components) in the sense that repairs that are Pareto-optimal w.r.t. the number of changes in each component are considered as possible repairs, even if the total number of changes is not minimal. Such generalizations are not useful in the class of FDs because individual relations are repaired independently, and thus, component cardinality repairs are equivalent to cardinality-minimal repairs.

9.2 Data cleaning based on user confidence in data

Multiple papers have addressed the problem of having different degrees of trust in different pieces of the data. One possibility is to associate each tuple (or cell) with a weight reflecting the user's confidence (e.g., [9,12,23]). The cost of the repairs takes into consideration the weights of the changed cells, as discussed in Sect. 3.

Another line of research aims at obtaining fixes to dirty data that are completely trusted (e.g., [18,19]. This is achieved by linking the input dirty data to master data, which is completely trusted, through a set of editing rules.

In this paper, we propose a different confidence model in which the user either identifies a set of cells that are trusted (and hence should not be changed at all) or specifies his or her preferences about the relative order in which cells can be changed (if necessary). In both cases, the algorithm generates a sample of possible repairs, all of which satisfy these rules.

9.3 Query answering using multiple possible repairs

A related research topic is consistent query answering, which aims at obtaining query results that are true in every possible repair (for some definition of a possible repair). Approaches that provide consistent query answers perform query rewriting (e.g., [4,20]) or construct a condensed representation of all repairs that allows obtaining consistent answers [30,31]. Usually, a restricted class of queries can be answered efficiently while harder classes are answered using approximate methods (e.g., [24]).

In our work, we provide a sample of possible repairs, which could be used for answering queries over uncertain data. Recent work has addressed the problem of modeling data uncertainty in a compact way to allow efficient query answering (e.g., [6,28]). One of the related works is the MCDB [22], which allows answering user queries efficiently using a sample of possible realizations (i.e., possible worlds) of a database by avoiding redundant computations. For example, query planning is performed only once for the entire set of possible repairs. We envision integrating our repair-generating algorithms with MCDB to facilitate efficient query answering. This approach extends the concept of consistent query answering to allow almost certain answers, that is, those which appear in a high percentage of the sampled repairs, but not necessarily all of them.

In [21], the authors proposed a model of possible repairs of inconsistent databases, which can be used for answering queries in a probabilistic way. The proposed model imposes a strong constraint on the defined FDs. Specifically, any attribute that appears in the RHS of an FD cannot appear in the left-hand side of another FD. In this setting, FD violations can be repaired independently and cardinality-minimal repairs can be obtained in PTIME. Our solutions apply to arbitrary sets of FDs/CFDs.

9.4 Previous repair sampling algorithms

In [7], we presented our initial efforts toward sampling from the space of cardinality-set-minimal repairs. However, the initial algorithm samples from a superset of the cardinality-set-minimal repairs. That is, some of the generated repairs could violate cardinality-set-minimality (although all repairs are guaranteed to be set-minimal). The reason is that the algorithm presented in [7] obtains a clean set of cells that is not necessarily maximal and then modifies the remaining cells whenever necessary to obtain a repair. In this paper, we rectified this shortcoming by first obtaining a maximal clean set of cells and only changing the cells that are not in such set. This guarantees that we sample exactly from the space of cardinality-set-minimal repairs (Theorem 1).

Also, we describe in this paper how to extend our algorithms in [7] to support CFDs. Finally, we introduced a number of optimizations in our implementation of the algorithms, resulting in significant improvements in their performance (Sect. 8).

10 Conclusion

In this paper, we presented a new technique for constraint-based database repair, in which we generate a sample of possible repairs. We described a realization of this technique in the context of FDs and CFDs, given a novel repair space that combines the features of two well-known existing spaces: set and cardinality minimal repairs. We also extended our sampling algorithm to allow user-defined hard constraints that specify a set of cells that must remain unchanged during the repairing process. Experimental results indicate that partitioning the input instance into blocks that can be repaired independently results in orders of magnitude performance gains.

An important direction for future work is to design new repair sampling algorithms for broader classes of integrity constraints such as denial constraints and matching dependencies. Another point to be pursued in future work is how to assign confidence values to the generated repairs that reflect their expected quality.

References

1. UIS data generator, http://www.cs.utexas.edu/users/ml/riddle
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
3. Afrati, F.N., Kolaitis, P.G.: Repair checking in inconsistent databases: algorithms and complexity. In: ICDT, pp. 31–41 (2009)
4. Arenas, M., Bertossi, L.E., Chomicki, J.: Consistent query answers in inconsistent databases. In: PODS, pp. 68–79 (1999)
5. Baxter, R., Christen, P., Churches, T.: A comparison of fast blocking methods for record linkage. In ACM SIGKDD (2003)
6. Benjelloun, O., Sarma, A.D., Halevy, A.Y., Widom, J.: ULDBs: Databases with uncertainty and lineage. In: VLDB, pp. 953–964 (2006)
7. Beskales, G., Ilyas, I.F., Golab, L.: Sampling the repairs of functional dependency violations under hard constraints. PVLDB **3**(1), 197–207 (2010)
8. Bohannon, P., Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Conditional functional dependencies for data cleaning. In: ICDE, pp. 746–755 (2007)
9. Bohannon, P., Flaster, M., Fan, W., Rastogi, R.: A cost-based model and effective heuristic for repairing constraints by value modification. In: SIGMOD, pp. 143–154 (2005)
10. Chomicki, J., Marcinkowski, J.: Minimal-change integrity maintenance using tuple deletions. Inf. Comput. **197**(1/2), 90–121 (2005)
11. Chomicki, J., Marcinkowski, J., Staworko, S.: Computing consistent query answers using conflict hypergraphs. In: CIKM, pp. 417–426 (2004)
12. Cong, G., Fan, W., Geerts, F., Jia, X., Ma, S.: Improving data quality: consistency and accuracy. In: VLDB, pp. 315–326 (2007)
13. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press and McGraw-Hill, Cambridge and New York (2001)
14. Yuan, Y.C.: Multiple imputation for missing data: Concepts and new development. In: The 25th Annual SAS Users Group International Conference (2002)
15. Eckerson, W.W.: Data quality and the bottom line: Achieving business success through a commitment to high quality data. TDWI Report Series (2002)
16. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate record detection: a survey. IEEE Trans. Knowl. Data Eng. **19**(1), 1–16 (2007)
17. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Conditional functional dependencies for capturing data inconsistencies. ACM Trans. Database Syst. **33**(2), 6:1–6:48 (2008)
18. Fan, W., Li, J., Ma, S., Tang, N., Yu, W.: Towards certain fixes with editing rules and master data. PVLDB **3**(1), 173–184 (2010)
19. Fan, W., Li, J., Ma, S., Tang, N., Yu, W.: Towards certain fixes with editing rules and master data. VLDB J. **21**(2), 213–238 (2012)
20. Fuxman, A., Miller, R.J.: First-order query rewriting for inconsistent databases. J. Comput. Syst. Sci. **73**(4), 610–635 (2007)
21. Greco, S., Molinaro, C.: Approximate probabilistic query answering over inconsistent databases. In: ER, pp. 311–325 (2008)
22. Jampani, R., Xu, F., Wu, M., Perez, L.L., Jermaine, C.M., Haas, P.J.: MCDB: a monte carlo approach to managing uncertain data. In: SIGMOD Conference, pp. 687–700 (2008)
23. Kolahi, S., Lakshmanan, L.V.S.: On approximating optimum repairs for functional dependency violations. In: ICDT, pp. 53–62 (2009)
24. Lopatenko, A., Bertossi, L.E.: Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. In: ICDT, pp. 179–193 (2007)
25. McCallum, A., Nigam, K., Ungar, L.H.: Efficient clustering of high-dimensional data sets with application to reference matching. In: KDD, pp. 169–178 (2000)

26. Müller, H., Freytag, J.C.: Problems, Methods and Challenges in Comprehensive Data Cleansing. Technical Report HUB-IB-164, Humboldt-Universität zu Berlin, Institut für Informatik (2003)
27. Mulry, M.H., Bean, S.L., Bauder, D.M., Wagner, D., Mule, T., Petroni., R.J.: Evaluation of estimates of census duplication using administrative records information. J. Off. Stat. **22**(4), 655–679 (2006)
28. Sen, P., Deshpande, A.: Representing and querying correlated tuples in probabilistic databases. In: ICDE, pp. 596–605 (2007)
29. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM **22**(2), 215–225 (1975)
30. Wijsen, J.: Condensed representation of database repairs for consistent query answering. In: ICDT, pp. 375–390 (2003)
31. Wijsen, J.: Database repairing using updates. ACM Trans. Database Syst. **30**(3), 722–768 (2005)