Vietnam J Comput Sci (2017) 4:3–12
DOI 10.1007/s40595-015-0052-y

CrossMark

REGULAR PAPER

Inferring the cause of errors for a scalable, accurate, and complete constraint-based data cleansing

Ayako Hoshino[1] · Hiroki Nakayama[1] · Chihiro Ito[2] · Kyota Kanno[1] · Kenshi Nishimura[2]

Received: 20 October 2015 / Accepted: 28 October 2015 / Published online: 26 November 2015
© The Author(s) 2015. This article is published with open access at Springerlink.com

Abstract In real-world dirty data, errors are often not randomly distributed. Rather, they tend to occur only under certain conditions, such as when the transaction is handled by a certain operator, or the weather is rainy. Leveraging such common conditions, or "cause conditions", the proposed data-cleansing algorithm resolves multi-tuple conflicts with high speed, achieves higher completeness, and runs with high accuracy in realistic settings. We first present complexity analyses of the problem, pointing out two subproblems that are NP-complete. We then introduce, for each subproblem, heuristics that work in sub-polynomial time. We also raise the issue that some previous studies overlook the notion of repair-completeness, which means, having less number of unsolved conflicts in the resulting repairs. The proposed method is capable of obtaining a complete repair if we are allowed to preprocess the input set of constraints. The algorithms are tested with three sets of data and rules. The experiments show that, compared to the state-of-the-art methods for conditional functional dependencies-based and FD-based data cleansing, the proposed algorithm scales better with respect to the data size, is the only method that outputs complete repairs, and is more accurate especially when the error distribution is skewed.

Keywords Data cleansing · Conditional functional dependencies · Cause of errors · Multi-tuple violations

✉ Ayako Hoshino
ayako.hoshino@gmail.com; a-hoshino@cj.jp.nec.com

Hiroki Nakayama
h-nakayama@cj.jp.nec.com

Chihiro Ito
c-ito@az.jp.nec.com

Kyota Kanno
k-kanno@ah.jp.nec.com

Kenshi Nishimura
k-nishimura@az.jp.nec.com

[1] NEC Knowledge Discovery Research Laboratories, 1753, Shimonumabe, Nakahara-ku, Kawasaki, Kanagawa 211-8666, Japan

[2] NEC System Integration, Services & Engineering Operations Unit, 5-7-1 Shiba, Minato-ku, Tokyo, 108-8001, Japan

1 Introduction

Data cleansing is a crucial step in data integration. As more data are made available, this task has gained a considerable attention both in business and research. One promising approach is constraint-based data cleansing, which is based on traditional functional dependencies (FDs) or recently proposed conditional functional dependencies (CFDs) [1]. Below are examples of an FD, a variable CFD and a constant CFD.

$$\phi_1 : \text{companyID, employeeID} \rightarrow \text{personname}$$
$$\phi_2 : \text{companyID, deskaddress} \rightarrow \text{employeeID}, (001, _ \,||\, _)$$
$$\phi_3 : \text{companyID, personname} \rightarrow \text{deskaddress}, (001, \text{“AliceB.''} \,||\, \text{“F12} - \text{S} - 123\text{''})$$

Each constraint expresses regularity in the data. The first constraint ϕ_1 is an example of FD, indicating "the company ID and employee ID determine individual names". The second constraint ϕ_2 is an example of CFD, indicating "when company ID is 001, desk address determines employee ID". The third constraint designates "when the company ID is 001 and the person name is Alice B., the desk address is F12-S-123". Such constraints can be used to detect errors in the data, as well as to determine the correct values.

Data can have multi-tuple conflicts with an FD or a variable CFD. For example, an erroneous tuple t_k: (company ID, employee ID, person name) = (001, A123, "John") will cause a *conflict*, when there is another tuple t_1: (001, A123, "Paul") and thus, the two tuples *violate* constraint ϕ_1. Also, an erroneous tuple t_k: (company ID, desk address, employee ID) = (001, F12-North345, A123) will cause a *conflict*, when there is another tuple t_2: (001, F12-North345, A456) and thus, the two tuples *violate* constraint ϕ_2.

We propose a data-cleansing method that addresses the problem of discovering a common cause of errors. By discovering such a condition, we can both cut down the search space and obtain a more accurate repair. While CFD is an addition to FD in that it specifies the subset of data where a constraint holds, our proposed "cause condition" specifies the subset of the data that contains conflict-causing errors. The discovery of cause condition is much more difficult as we will prove, but often needed in real-world data-cleansing.

This paper is organized as follows. Section 2 summarizes the previous research on constraint-based data cleansing. Section 3 describes our method for discovering error conditions and generating a repair. Section 4 presents our experimental evaluation. Finally, Sect. 5 concludes our paper.

2 Related work

With the recent appearance of conditional functional dependencies (CFD) [1], constraint-based data cleansing is experiencing a revival. Numerous methods have already been proposed on CFD-based data cleansing [2–8]. Prior to CFD, there had been data cleansing with FDs [9–12], and Association Rules (ARs) [13], but here we focus on the methods that have been applied to CFD.

The cleansing algorithm of Cong et al.'s BatchRepair and IncRepair is, just like their predecessor [14], a cost-based algorithm where the optimal repair is chosen based on the editing cost from the original data [2], measured in Levenstein's edit distance [15], or measured by the number of tuples to be removed [16]. Note that all cost-based methods follow "majority policy". Beskales et al. proposed a sampling-based method that generates repairs from among repairs with minimal changes [7], which can also be categorized as a cost-based method.

Chiang and Miller [3] proposed a method for deriving CFDs that almost hold (i.e., X → A holds on D, allowing some exception tuples), filtering these *approximate CFDs* using an interest measure, and then detecting dirty data values. In their definition, dirty values are infrequent right hand side (RHS) values within the set of left hand side (LHS) matched tuples, which makes them also follow the "majority policy". Notably, only values on RHS are the target of error detection.

Fan et al. [4,5] proposed a highly accurate method that uses hand-crafted editing rules and a human-validated certain region of the table. The correction may not always follow the majority policy, but preparing editing rules and a certain region requires human input, which is often not available in reality.

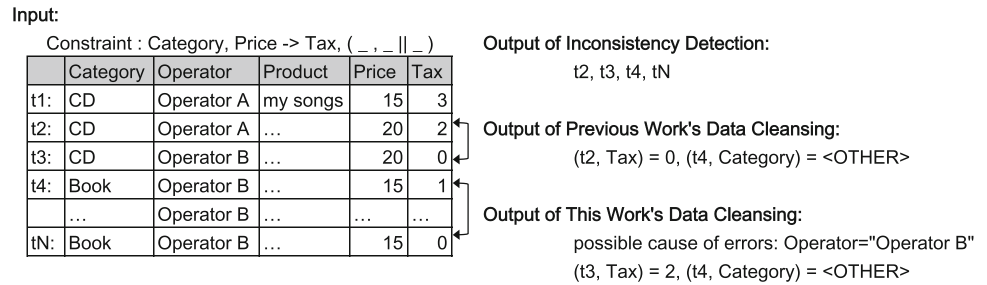
In our study, we develop a method called CondRepair that identifies erroneous tuples when there are conflicts with FD or CFD rules. It relies on neither the majority RHS nor edit distance-based cost metrics, which do not work when the differences (in frequency or in cost) among candidate fixes are not significant. It determines the wrong values based on the common cause of errors. Although the idea of leveraging the patterns among errors has been explored for other types of data cleansing [17], to the best of our knowledge, the idea has never been applied with (C)FDs or ARs. In the experimental evaluation, we use error injection with both uniform and skewed error distribution whose design is based on our observation on the real errors. In this work, we also raise the issue that some previous studies overlook the notion of repair-completeness, which means, having less number of unsolved conflicts in the resulting repairs. The proposed method is capable of obtaining a complete repair if we are allowed to preprocess the input set of constraints.

3 Data cleansing based on error conditions

In this section, we first introduce the problem of CFD-based data cleansing, and highlight some of its difficulties by showing the class of problems its subproblems belong to. We then describe our proposed method, which consists of two steps: (1) finding the cause conditions and (2) generating a repair.

3.1 Problem description

The problem of CFD-based data cleansing is defined as follows. Let D be the input relation and Σ be a set of CFD rules. We assume D consists of one table. Let D_{repr} be an output relation of the cleansing process. It is required that $D_{\text{repr}} \models \Sigma$, which is, there is no violation in D w.r.t. any rules in Σ. Let A, B, and C denote attributes, and X, Y and Z denote sets of attributes. Each tuple in D has a tuple ID so that $D = \{t_1, t_2, \ldots, t_N\}$, where N is the total number of tuples in D. We refer to a cell in a tuple as c. Figure 1 shows an illustrative example of an input rule set (consisting of one rule), data with conflicts and possible outputs of previous methods and ours.



	Category	Operator	Product	Price	Tax
t1:	CD	Operator A	my songs	15	3
t2:	CD	Operator A	…	20	2
t3:	CD	Operator B	…	20	0
t4:	Book	Operator B	…	15	1
	…	Operator B	…	…	…
tN:	Book	Operator B	…	15	0

Fig. 1 Example constraint, data, and outputs of cleansing algorithms. *Arrows* on the *right side* of the table show conflicting tuples

We now examine two subproblems that belong to the class of problems that are known to be difficult to solve.

Theorem 1 *Selecting the tuples to be corrected among the ones in conflicts includes an NP-complete problem.*

Proof Here, we try to solve the conflicts that involve more tuples first, i.e., we do not take a naive approach such as modifying all tuples that are involved in a conflict. Let the set of tuples that are involved in multi-tuple conflicts be D_{doubt}, and the set of conflicts between tuples $t_i, t_j \in D_{\text{doubt}}$ be f_{ij}. The problem of finding tuples to be corrected is selecting the subset D'_{doubt} of D_{doubt} at least one of whose members have conflict with all the remaining tuples in set $D_{\text{doubt}} \backslash D'_{\text{doubt}}$. There is a polynomial time projection from the set D_{doubt} to the set of vertices V and from conflicts f_{ij} to the set of edges E of a graph (V, E). Hence, the problem can be reduced to the dominating set problem of a graph (V, E), which is known to be NP-complete. A naive solution for this problem takes computation order of $O(2^n n)$ and will be intractable as the number of tuples in D_{doubt} increases. □

Note that there are some exceptional cases where conflicts between two tuples remain. These conflicts should be solved after resolving the ones with more tuples. Also note that we have not yet mentioned which cells of the tuples to be corrected. Still, the problem of selecting tuples to be corrected is already NP-hard.

Secondly, we show that, even after we select D'_{doubt}, the problem of finding a common condition among those tuples is difficult.

Theorem 2 *The problem of discovering a common condition among the tuples in D'_{doubt} is NP-complete.*

Proof As pointed out by Zaki and Ogihara, the problem of finding a common itemset among the tuples is NP-complete, since it can be reduced to the problem of clique discovery in a bipartite graph [18]. Similarly, the problem of finding a common set of attribute values among a set of tuples can be reduced to a bipartite clique problem as follows. Assume we have a graph $G = (U, V, E)$, where G is a bipartite graph consisting of parts U and V, and edges E. A set of tuples can be projected to U and a set of attribute values to V, and the existence of a tuple containing a set of attribute values to an edge in E. A clique in a bipartite graph is equivalent of a set of attribute values that is common in a set of tuples. Then, the problem of finding a common attribute values is reduced to the problem of finding a clique in a bipartite graph G. This problem is known to be NP-complete and, for instance, finding the largest clique requires computation $O(|U||V|)$. □

Definition 1 An LHS value sets $\{S_{\phi,1}, S_{\phi,2}, \ldots, S_{\phi,k}\}$ is defined for a CFD rule $\phi \in \Sigma$, which is a set of tuple sets $S_\phi = \{\{t_{1,1}, t_{1,2}, \ldots, t_{1,N1}\}, \{t_{2,1}, t_{2,2}, \ldots, t_{2,N2}\}, \ldots\}$ where each tuple within each set (or, LHS value set) matches the condition of the rule ϕ, and all tuples within LHS value set have the same values on the LHS of the rule ϕ, namely $t_{k,i}[\phi.\texttt{LHS}] = t_{k,j}[\phi.\texttt{LHS}]$ holds for all $i, j \in \{1, 2, \ldots, N_k\}$ for all k tuple sets. (We denote the LHS attribute set as $\phi.\texttt{LHS}$, and the values of attribute set A as $t[A]$.)

Definition 2 A doubt tuple is a tuple in conflict w.r.t. a rule in Σ, namely $\{t \mid \exists t' \in D \wedge t, t' \in S_{\phi,k} \wedge t[\phi.\texttt{RHS}] \neq t'[\phi.\texttt{RHS}]\}$. We call a set of doubt tuples $D_{\text{disagree},\phi,k}$, which is the kth tuple set in S_ϕ where any pair of the member tuples disagree on the RHS of ϕ.

Definition 3 Cause attribute $(\mathbf{Z}, \mathbf{v})$ is defined as a set of attribute values that is common to the tuples to be corrected in D.

3.2 Finding the cause conditions

The condition we try to discover is defined as a form of a triplet $\mathbf{Z}$, $\mathbf{v}$, $\kappa(\mathbf{Z} = \mathbf{v}, L =$“doubt”), which are a set of attributes, their values, and an agreement metrics which evaluates co-occurrence between a condition and a “doubt” label. We first identify the cause of error $\mathbf{Z}$ (hereafter called *cause attributes*) among $\texttt{attr}(D)$, using a sample of data D_{sample}.

We treat a tuple as a basic unit for probability computation. We label tuples in D either “clean” or “doubt”, using a set of labels for tuples $L = \{$“clean”, “doubt”$\}$, where $L(t) =$

"clean" when tuple t is not in conflict and $L(t)$ = "doubt" when in conflict with any rule in Σ. In Fig. 1, doubt tuples are t_2, t_3, t_4 and t_N, and all the rest are clean tuples.

When determining the error attributes, CondRepair uses the agreement statistics Kappa which indicates co-occurrence between the doubt tuples and candidate cause conditions. The Kappa statistics κ is defined as follows.

The Kappa agreement statistics: $\kappa(\mathbf{Z} = \mathbf{v}, L = \text{"doubt"}) = \frac{P_{\text{actual}} - P_{\text{coincidental}}}{1 - P_{\text{coincidental}}}$ where $P_{\text{actual}} = \frac{|\{t|t[\mathbf{Z}]=\mathbf{v} \wedge L(t)=\text{"doubt"}\}|}{|D_{\text{sample}}|}$ and $P_{\text{coincidental}} = \left(\frac{|\{t|t[\mathbf{Z}]=\mathbf{v}\}|}{|D_{\text{sample}}|}\right)\left(\frac{|\{t|L(t)=\text{"doubt"}\}|}{|D_{\text{sample}}|}\right)$.

The meaning of Kappa index is, basically, the difference between the rate of actual co-occurrence and the rate of theoretical co-occurrence. The value is normalized by the negation of coincidental co-occurrences. Therefore, when the probability of coincidental co-occurrence is higher, κ will be higher.

We now describe some heuristics introduced in response to the subproblem described in Theorem 2. We could have discovered a common condition among a set of doubt tuples in an a priori-like manner ([19]). However, the a priori algorithm is known to be still computationally expensive especially with data of high arity. Hence, we developed a more efficient inference method for cause discovery using the Kappa index. The virtue of Kappa index is, it evaluates different attribute values with a single viewpoint, a co-occurrence with doubt tuples. We obtain attribute values in a single list in the order of Kappa value and seek if there is a composed cause condition $(\mathbf{Z} \cup A, \mathbf{v} + v)$ that has higher Kappa than a simpler condition $(\mathbf{Z}, \mathbf{v})$.

3.3 Generating a repair

When tuples to be corrected are determined, it is fairly straightforward to generate a repair. We use *equivalence classes* proposed by Bohannon et al., based on the description given by Beskales et al. [1,7]. Equivalence class is a useful data structure to repair data with multiple rules. It groups cells into sets within which all member cells should have the same value when the cleansing is completed, delaying decision on the exact value the set will have.

Use of equivalence classes assures a complete repair, i.e., a repair with no remaining violation. However, as Bohannon et al. have noted as *collision* problem, it often generates excessively large equivalence sets by applying repeated merges with multiple rules. For example, an equivalence class with cell $t[B]$ is merged with the one with cell $t'[B]$ based on $\phi_1 : A \rightarrow B$, then an equivalence class with cell $t[C]$ is merged with the one with $t'[C]$, based on another rule $\phi_2 : B \rightarrow C$, and so on. We make some modifications to Beskales et al.'s equivalence class. First, we do not make equivalence classes where there is no conflict, whereas Beskales' method first merges all cells that have the same values. Secondly, we merge equivalence classes not based on their LHS equivalence classes, but simply based on the values on the LHS attributes. In order to achieve a complete repair, we impose some limitations on the order and the involving attributes of rules in the input rule, so that $\Sigma^{<^{\texttt{rules}}} = \{\phi \mid \forall A \in \phi.\texttt{LHS}, A \notin \phi'.\texttt{RHS}, \forall \phi' <^{\texttt{rules}} \phi\}$, which means Σ is a list of rules sorted in the order $<^{\texttt{rules}}$ where any attribute on LHS of rule ϕ is not included in the RHS of any preceding rule ϕ'.

During the repair generation, a cell's value $t[B]$ is changed to another tuple's value $t'[B]$ where there is a *cleaner tuple*, which is a tuple that is assigned with the lowest probability of being erroneous among the ones in the same equivalence class, t' within the equivalence class.

Equivalence classes cannot produce corrections for constant CFDs. So, constant CFDs are treated separately by changing any of the cell in LHS to a special symbol `OTHER_VALUE`, which indicates a value not in the domain of the attribute and defined not to match any value (thus, `OTHER_VALUE` $\neq$ `OTHER_VALUE`). The specific value can be filled in by a human in the process of verification, which is out of the scope of this paper. We now provide a step by step description of the algorithm CondRepair (Algorithm 1).

Algorithm 1 CondRepair

```
Input: D, Σ, n(sample size), m(a threshold to limit the size of S)
Output: D_repr ⊨ Σ
 1: D_repr := D
 2: take D_sample, a sample of size n from D_repr

    //label each tuple
 3: for each φ ∈ Σ do
 4:   D_disagree,φ := {S_φ,k | t, t' ∈ S_φ,k ∧ S_φ,k ⊆ D_sample ∧ t[φ.RHS] ≠ t'[φ.RHS]}
 5:   if |S_φ,k| ≤ m where S_φ,k has a conflict then
 6:     label(t) := "doubt" for all t ∈ S_φ,k

    //infer the cause condition
 7: for each (A, v) ∈ D_disagree, a set of tuples whose labels are "doubt" do
 8:   calculate κ(A = v, L = "doubt")
 9: for each (A, v) in the descending order of kappa do
10:   if A ∉ Z and κ(Z, v) < κ(Z ∪ A, v + v) then
11:     (Z, v) := (Z ∪ A, v + v) else break

    // fix LHS of constant CFDs
12: for each t ⊭ φ ∈ Σ, t ∈ D_repr, where φ is a constant CFD, do
13:   t[B] = OTHER_VALUE for any attribute B ∈ φ.LHS

    // build equivalence classes
14: BuildEquivRel(D_repr, Σ)

    //fix values
15: for each e ∈ E, where e is a equivalence class and E is a set of all equivalence classes, do
16:   c* := a cell under (Z, v) which has the smallest κ in E
17:   for each c ∈ e do
18:     val(c) := val(c*)
19: return D_repr
```

The inputs for the algorithm are D, the data to be cleaned, Σ, a set of CFD rules, n, a size of sample data, and m, a maximum size of LHS value sets to be used to infer the cause attribute. The output is the D_{repr}, which conforms to all rules in Σ.

The algorithm first takes a sample of size n from D_{repr} (Line 2). It then label the tuples as "clean" or "doubt", depending on the existence of violation with rules in Σ. If a tuple t violates a rule in Σ and if the size of the kth LHS value set (described as $|D_{\text{disagree},\phi,\text{k}}|$) is equal to or smaller than the parameter m, it is labeled as "doubt", otherwise the tuple is treated as "clean" (lines 3–6).

Lines 7–11 perform a discovery of the cause condition. The algorithm calculates the kappa agreement statistics of the doubt tuples and all attribute value pairs (A, v)s appearing in D_{disagree}, which is a union of all $D_{\text{disagree},\phi,\text{k}}$s. It then, looks at the list of attribute value pairs (A, v)s in the descending order of kappa and joins an (A, v) if it has larger kappa when combined with the preceding condition. If a combined condition does not exceed the preceding condition in kappa, it stops looking further in the list.

Lines 12–13 make corrections on the violations to constant CFDs. The algorithm turns any of the attribute values on LHS of the rule to an `OTHER_VALUE`, which resolves violation without producing a new conflict.

Line 14 builds equivalence classes on D_{repr}, with the modifications described in Sect. 3.3.

Lines 15–18 fix values in each equivalence class by turning them to the ones of the cells which have the smallest κ within the equivalence classes. The comparison of kappa values canbe done when building equivalence class, so this step consists only of producing corrections on cells that have higher κ than other cells in the equivalence classes.

4 Experiments

The proposed algorithm, along with two previous methods, is tested in terms of its scalability and accuracy in detecting and correcting error cells with different degrees of error skewness. The algorithms are implemented in Java™and all experiments are conducted on a Linux CentOS with 16-Core Xeon E5-2450 (2.1 GHz) and 128-GB memory. We describe experimental settings (Sects. 4.1, 4.2, 4.3, 4.4) followed by some key results (Sects. 4.5, 4.6). Results are examined from the aspects of completeness of repairs (Sect. 4.7) and correctness of repair values (Sect. 4.8).

4.1 Datasets

We used three datasets, two of which are from the UCI machine learning database: (1) Wisconsin Breast Cancer (WBC) and (2) US Census dataset (Adult). WBC is a numeric

data and US Census contains mostly nominal values. The third dataset is DBGen [20], a synthetic data set obtained from the authors of a previous algorithm [7].

4.2 The previous algorithms

Two previous algorithms were used for comparison, which are IncRepair by Cong et al. [2] and FDRepair by Beskales et al. [7]. See "Appendices 1 and 2" for algorithm descriptions. IncRepair ("Appendix 1") is an incremental algorithm that cleans ΔD when ΔD has been added to a dataset D so $\Delta D \cup D$ satisfies Σ. Note that it performs with a better scalability than its batch counterpart (BatchRepair) without loss of accuracy. We used IncRepair so that it treated all the data as ΔD as done by Cong et al. IncRepair was re-implemented using the same basic utility classes as CondRepair for data IO and for CFD validations.

FDRepair ("Appendix 2") is a repair algorithm based on Beskales et al.'s proposed notion of cardinality-set-minimal repairs. It employs the equivalence classes originally proposed by Bohannon et al. [14] and attempts to rectify the *collision* problem that we described in Sect. 3.3 by reverting the values to the original ones where it does not cause a violation. FDRepair is a sampling algorithm that produces samples from the cardinality-set-minimal repairs. We used a Java implementation of FDRepair obtained from the author.

4.3 Input rules

The CFD rules we used was extracted from WBC and Adult datasets before error injection using a FastCFD algorithm with the support threshold set to 10 % of the number of tuples in input data, where 10 % is a popularly used value in previous work. The number of CFD rules can be excessive, and rules with a large number of attributes on LHS are often not useful in data cleansing, so we have limited the size of LHS to at most four. For FDRepair, we used FDs which were included in the result of the CFD discovery, and for the other two algorithms, we used the same number of randomly sampled CFD rules. As a result, 35 rules from WBC and Adult have been extracted. For the dataset DBGen, we used the same 18 FD rules as used by Beskales et al. [7].

4.4 Injected errors

We injected errors by turning an attribute value of a random tuple (t, A) into the value of the same attribute of another randomly chosen tuple (t', A). In effect, this can cause multiple cells with originally different values to have the same value, or multiple cells with originally the same value to have different values. When (t, A)'s value was the same as that of (t', A), `OTHER_VALUE` was inserted in the selected cell. The default error rate was set to 0.05 (i.e., the number of errors injected is 5 % of the total number of tuples).

For experiments with error skewness, we injected the errors that follow the probability $P(\mathrm{Err}_t)$, or the probability of tuple t includes an error, as follows:

$$P(\mathrm{Err}_t) = \begin{cases} \epsilon s / |\{t \mid t[\mathbf{Z}] = \mathbf{v}\}| & (t[\mathbf{Z}] = \mathbf{v}) \\ \epsilon(1.0 - s) / |\{t \mid t[\mathbf{Z}] \neq \mathbf{v}\}| & (\text{otherwise}) \end{cases}$$

where ϵ is the overall error rate in the dataset and $s (0 \leq s \leq 1)$ is the skewness factor denoting the proportion of errors that occur under specified condition $(\mathbf{Z}, \mathbf{v})$. When $s \gg |\{t \mid t[\mathbf{Z}] = \mathbf{v}\}| / |\{t \mid t \in D\}|$ holds, if the cell is within the specified area $\mathbf{Z} = \mathbf{v}$, the cell is erroneous for the specified probability, otherwise the cell can still be erroneous, but for a much smaller probability.

4.5 Scalability

We first look at the runtime of the repair algorithms as the input data size increases. Figures 2, 3 and 4 describe the results with each dataset (the average of 10 iterations). IncRepair performed the fastest and looked the most scalable with WBC, but clearly did not seem to scale well with the two larger datasets. We stopped the runs where it took too long to complete. The result shows that IncRepair's exploration of all values in the domain of attribute C to find a fix that sat-

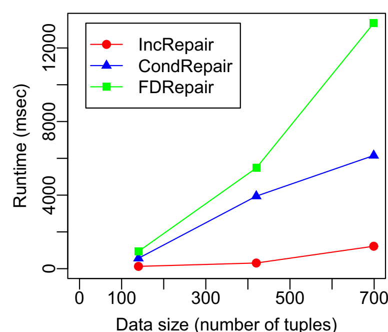


Fig. 2 Scalability (WBC)

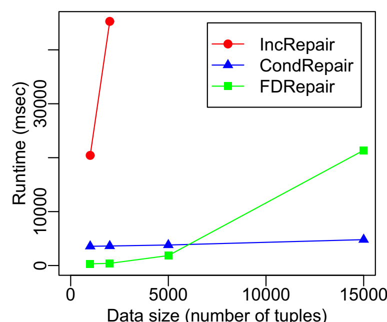


Fig. 3 Scalability (DBGen)

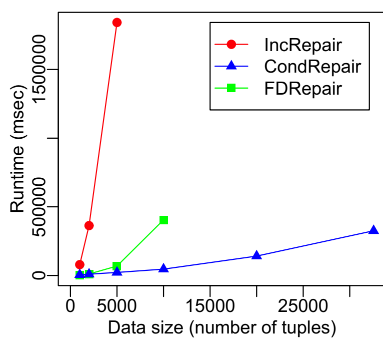


Fig. 4 Scalability (Adult)

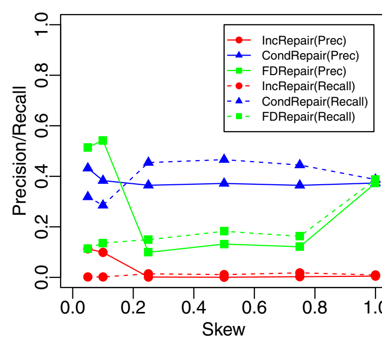


Fig. 5 Accuracy (WBC)

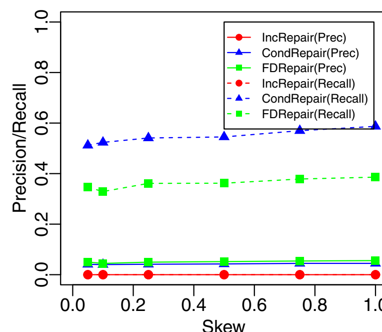


Fig. 6 Accuracy (DBGen)

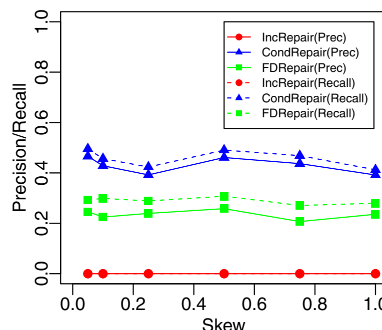


Fig. 7 Accuracy (Adult)

isfies Σ is prohibitively expensive with a large dataset with unbounded attribute domains. FDRepair's runtime, as shown in the original paper, is at least quadratic, and looks sensitive even when the data consist of limited number of different values as with WBC. We think that the result is due to the algorithm's high cost for validating and reverting the candidate corrections. As opposed to the two previous algorithms, we observe that CondRepair's runtime is closer to linear with the data size.

4.6 Effect of error skewness

We then change the error skewness using the aforementioned error distribution model. Overall error rate was fixed to 0.05. A set of tuples with a predetermined condition is called "high error area" or HIGH, which is defined as HIGH $= \{t \mid t[\mathbf{Z}] = \mathbf{v}\}$ where $\mathbf{v}$ is a set of values selected from the domains of attributes in $\mathbf{Z}$. For $(\mathbf{Z}, \mathbf{v})$, we used a single attribute and $\mathbf{v}$ was selected so the number of tuples with $t[\mathbf{Z} = \mathbf{v}]$ is closest to 10 % of the input tuples. Cause conditions clump_thickness = "10" for WBC, state = `NULL` for DBGen, and occupation = "Adm-clerical" for Adult were used throughout the iterations. The parameter s varied from 0.05, 0.1 (no skew) to 1.0 (extremely skewed, where all errors occur under condition $\mathbf{Z} = \mathbf{v}$, but no errors occur in other areas).

As noted in some of the previous work, an injected error does not always violate a CFD rule, which may lead to a low recall. To separate out the problem of errors' not being detected by the input rules, we measure the performances with the score metrics defined as follows:

$$\text{Precision} = \frac{\#\ \text{correctly detected errors}}{\#\ \text{detected errors}},$$

$$\text{Recall} = \frac{\#\ \text{correctly detected errors}}{\#\ \text{conflict} - \text{introducing errors}}.$$

Figures 5, 6 and 7 show the accuracy (precision and recall) with different degrees of skewness (average of 20 iterations). The precision and recall scores are calculated on the cell-base count of errors.

With all datasets, CondRepair exceeded, in most of the skewness settings, the other two algorithms both in precision and recall. CondRepair's score ranged 0.4–0.6 with all data sets with an exception for the precision for DBGen, where all the other algorithm's scores were low as well. FDRepair performed with equal to or higher scores than CondRepair only with WBC and where there is little or no skewness. IncRepair's scores were especially low. We think this is because the type of errors used could not be correctly detected by the edit distance metrics.

In summary, CondRepair's runtime is nearly linear with the data size and its accuracy surpassed that of IncRepair and FDRepair with all tested datasets when the skewness is larger than 0.1.

Table 1 Correctness of error detection and correction when $s = 0.25$

Dataset	WBC (2 × 699 lines)						DBGen (2 × 1000 lines)						Adult (2 × 1000 lines)					
# ICV	38.1/46.7						90.5						24.2/33.65					
Algorithm	TDP	TDR	DP	DR	CP	CR	TDP	TDR	DP	DR	CP	CR	TDP	TDR	DP	DR	CP	CR
IncRepair	0.07	**0.57**	0.00	0.01	0.00	0.00	0.14	**0.93**	0.00	0.00	0.00	0.00	0.31	0.02	0.00	0.00	0.00	0.00
FDRepair	0.43	0.53	0.10	0.15	0.09	0.14	0.16	0.72	**0.05**	0.36	0.04	0.27	0.66	0.30	0.24	0.29	0.16	0.19
CondRepair	**0.47**	0.38	**0.36**	**0.45**	**0.36**	**0.45**	**0.18**	0.63	0.04	**0.54**	**0.04**	**0.50**	**0.69**	**0.67**	**0.39**	**0.42**	**0.22**	**0.24**

The figures of the best performing algorithms are shown in bold
ICV injected and caused violation for CFDs/FDs, *TDP* tuple-wise detection precision, *TDR* tuple-wise detection recall, *DP* cell-wise detection precision, *DR* cell-wise detection recall, *CP* correction precision, *CR* correction recall

4.7 Result analysis 1: completeness of repairs

To test the completeness of repairs, we have used a basic function for CFD validation implemented separately from the one used in CondRepair and IncRepair. The basic function followed the original definition of CFD described in Bohannon et al. [1], by naively matching LHS value sets and RHS values for all pairs of tuples for all input rules. CondRepair produced repairs without a violation with WBC and Adult datasets, and left on average 43.2 tuples with violation with DBGen, because the input FD set contained rules that lead to the *collision* problem. It should be noted that if we are allowed to impose an order restriction described in Sect. 3.3 on the input rules, there should be no remaining errors also with DBGen. There were on average 308.5 tuples with remaining violations with IncRepair, 534.6 with FDRepair (WBC), 271.2 with IncRepair and 268.4 with FDRepair (1 K tuples of DBGen), and 4.8 with IncRepair and 236.6 with FDRepair (1 K tuples of Adult), when the error rate 0.05 and with no skew.

Here are some possible reasons why the two previous algorithms could not produce complete repairs. IncRepair, when multiple rules in Σ cannot be satisfied at once by changing the focused k values within the tuple, the tuple is left with violations. In the case of FDRepair, apparently, the algorithm does not create equivalence classes for singleton values. Leaving singleton values as they are can result in remaining violations because singleton values on a rule's RHS attribute can cause a conflict. In fact, FDRepair generated much less number of corrections than the number of conflict-introducing errors.

4.8 Result analysis 2: correctness of repair values

We look at precision and recall of corrections, which are the rate of correctly fixed errors over all fixes made and the rate of correctly fixed errors over all conflict-introducing errors, respectively. Table 1 summarizes numbers of injected errors, conflict-introducing errors, and scores for tuple-wise detection, cell-wise detection (same as shown in Figs. 5, 3, 7), and scores for correction (skewness 0.25, a weak skew). Simply injecting errors to the datasets did not produce sufficient number of conflicts, so we repeated all tuples in the datasets so each tuple appears twice, which leads to sufficient number of conflicts for evaluation.

The scores of CondRepair were the highest except for tuple-wise detection with WBC and DBGen and for cell-wise detection precision with DBGen. IncRepair's tuple-wise detection was higher with the two cases with tuple-wise detection, but notably, its cell-wise detection scores and correction scores were zero or very low. FDRepair performed much better than reported in the original paper, but did not exceed in the scores for correction of CondRepair. CondRepair were able to find 67 % tuples with conflict-introducing errors with 69 % precision, which looks promising for a practical use. The scores for correction are not so high as we can leave the machine an important data to cleanse, but the algorithm can be used to suggest human users possible corrections.

5 Conclusions

We have proposed a data-repairing technique that discovers and leverages the common cause of errors. Our method CondRepair, achieved a nearly linear scalability and accuracy of error detection that is higher than previous methods. Further directions include (1) a closer observation of real-world data-cleansing work and incorporation of observed characteristics of errors in experimental settings, (2) an incremental version of the algorithm, and (3) interaction with a human user to efficiently achieve an optimal repair.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

Appendix 1: IncRepair (Cong et al. [2])

Algorithm 2 summarizes our understanding the algorithm IncRepair.

Algorithm 2 IncRepair

Input: $D, \Delta D, \Sigma, \mathcal{O}, k$
Output: ΔD_{repr}, such that $D \cup \Delta D_{\text{repr}} \models \Sigma$

```
1: ΔD_repr := ΔD
2: sort ts in ΔD_repr by order O
3: for each t in ΔD_repr do
4:   C := {}
5:   for each C ∈ attr(D) do
6:     V := {}
7:     for each v_j ∈ dom(C) ∪ NULL do
8:       t'_j := t; t'_j[C] := v_j; C := {C ∪ C}; V := V ∪ t'_j
9:       if k < |C| then
10:        break
11:      end if
12:    end for
13:    for each φ ∈ Σ(C ∪ C) do
14:      if t'_j ⊭ φ then
15:        V := V − {t'_j}
16:      end if
17:    end for
18:    v* := arg min_{t'∈V} costfix(t[C], t'[C]); t[C] := v*
19:  end for
20: end for
21: return ΔD_repr
```

IncRepair takes as an input ΔD, Σ, and $\mathcal{O}$, where ΔD is a batch of data added to D and Σ is a set of rules, and $\mathcal{O}$ is an order of tuples in ΔD. As we described in Sect. 4, we use the whole D as the input ΔD, and Σ is the same as CondRepair.

IncRepair linearly processes tuples in ΔD applying fixes that have the least `costfix` among the candidates t'_js that satisfy the rules. It produces all values v in $\mathtt{dom}(C)$, where C is the attribute to fix. Since the algorithm did not specify the order of Cs, we used an arbitrary order. Note that the algorithm works linearly with attributes within a tuple, producing a set of candidates t'_js that satisfies $\Sigma(\mathcal{C} \cup C) = \{\phi : X \rightarrow A \mid (X \cup A) \subseteq (\mathcal{C} \cup C)\}$, where $\mathcal{C}$ is the attribute that has already been fixed and C is the attribute on which the algorithm has just produced the candidates. The algorithm proceeds to obtain the t'_j with the least `costfix` and reaches at the last C in $\mathtt{attr}(D)$ with t'_j that satisfies all rules in Σ.

There is a parameter k, which specifies the size of the attribute set C, or the number of values changed at once. In the experiment, we set k to one. According to Cong et al., IncRepair achieves a good accuracy with $k = 1$ or 2. As we increase parameter k, the algorithm's computational cost increases exponentially, since it generates as many candidates fixes vs as the combination of $\mathtt{dom}(C)$s.

The cost function `costfix` is defined as follows:

$$\mathtt{costfix}(v, v') = \begin{cases} \frac{\mathtt{dist}(v,v')}{\max(|v|,|v'|)} & (v' \neq \mathrm{NULL}) \\ \infty & (v' = \mathrm{NULL}) \end{cases}$$

where $\mathtt{dist}(v, v)$ is Levenstein's edit distance, and $|v|$ is the number of characters in value v. We have added the case of $v' = \mathtt{NULL}$, so NULL is selected when all values in $\mathtt{dom}(C)$ fail to satisfy $\Sigma(\mathcal{C} \cup C)$.

What is notable about this algorithm is the calculation order when the number of different values increases as the data size. It will be (N^3), where N is the number of tuples in the input batch ΔD.

Proof The loop that starts at line three contains another loop over $\mathtt{dom}(C) \cup \{\mathtt{NULL}\}$ (starting at line 7), where $|\mathtt{dom}(C)| \propto N$. Within this second loop, when ϕ_k is a variable CFD, the algorithm does a validity check that requires as many as N times string comparisons. Thus, the computational cost of IncRepair is $O(N^3)$. □

Appendix 2: Cardinality-set-minimal repair (Beskales et al. [7])

Motivated by the same problem as described in Sect. 3.1, Beskales et al. has proposed a method to obtain repairs that satisfy their proposed cardinality-set-minimal [7]. In their claims, cardinality-set-minimal precludes repairs with unwanted redundancy from among the exponential universe of repairs, thereby achieving high quality repairs. We briefly describe the notion of cardinality-set-minimal and their proposed algorithm.

Cardinality-set-repair is an intermediate set between the previously proposed cardinality-minimal repair and the set-minimal repair. Given a relation instance I, Beskales et al.'s cardinality-set-minimal repair is defined as follows:

Definition 4 (*Cardinality-Set-Minimal Repair* [7]) 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 cardinality-set-minimal-repair is a repair achieved with the minimal set of cells (i.e., reverting any changed cells to the original values in I causes a violation, while other cells are not necessarily the same values as in I'). With this, they aim at striking a balance between the "fewest changes" metric of cardinality-minimality and the "necessary changes" criterion of set-minimality. Their algorithm GenRepair (referred to as FDRepair in Sect. 4), simply random-samples from the space of cardinality-set-minimal repair, as shown in Algorithm 3.

Algorithm 3 GenRepair (FDRepair)

Input: I, Σ
Output: I'
1: CleanCells := ϕ
2: **while** CleanCells $\neq$ all cells in I **do**
3: Insert a random cell $t[A]$ to CleanCells
4: E := BuildEquivRel(CleanCells, I', Σ)
5: **if** IsClean(CleanCells, I', E) = false **then**
6: E := BuildEquivRel(CleanCells$\setminus\{t[A]\}$, I', Σ)
7: try changing the value of $t[A]$ to the one of other clean cells in the equiv. class
8: try changing the value of $t[A]$ to a randomly selected constant or variable
9: try changing the value of $t[A]$ to a new variable v'
10: **end if**
11: **end while**
12: return I'

Algorithm 4 BuildEquivRel

Input: D, Σ
Output: E
1: **for** each attribute $A \in \mathtt{attr}(R)$ **do**
2: **for** each cell $c \in A$ **do**
//e: an equivalence class; E: the equivalence relation
3: $e_{\mathtt{val}(c)} \Leftarrow e_{\mathtt{val}(c)} \cup c$; $E \Leftarrow E \cup e_{\mathtt{val}(c)}$
4: **end for**
5: **end for**
6: **while** each rule $\phi \in \Sigma$ where $\exists t \in D \not\models \phi$ **do**
7: LHSValGroup $\Leftarrow \{\forall e \mid e \in$ LHS value$\}$
8: **for** each equiv. class pair $e_1, e_2 \in$ LHSValGroup **do**
9: **for** each equiv. class pair $e_3, e_4 \in \{\forall e \mid e \in$ RHS$\}$ **do**
10: $e_3 \Leftarrow e_3 \cup e_4$; $e_4 \Leftarrow \mathtt{NULL}$
11: **end for**
12: **end for**
13: **end while**
14: return E

References

1. Bohannon, P., Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Conditional functional dependencies for data cleaning. In: ICDE, pp. 746–755 (2007)
2. Cong, G., Fan, W., Geerts, F., Jia, X., Ma, S.: Improving data quality: consistency and accuracy. In: VLDB, pp. 315–326 (2007)
3. Chiang, F., Miller, R.J.: Discovering data quality rules. PVLDB **1**(1), 1166–1177 (2008)
4. 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)
5. Fan, W., Geerts, F.: Capturing missing tuples and missing values. In: PODS, pp. 169–178 (2010)
6. Yeh, P.Z., Puri, C.A.: Discovering conditional functional dependencies to detect data inconsistencies. In: Proceedings of the Fifth International Workshop on Quality in Databases at VLDB2010 (2010)
7. Beskales, G., Ilyas, I.F., Golab, L.: Sampling the repairs of functional dependency violations under hard constraints. VLDB Endowment, vol 3(1–2), pp. 197–207 (2010)
8. Fan, W., Li, J., Ma, S., Tang, N., Yu, W.: Interaction between record matching and data repairing. In: SIGMOD Conference, pp. 469–480 (2011)
9. Bertossi, L., Bravo, L., Franconi, E., Lopatenko, A.: The complexity and approximation of fixing numerical attributes in databases under integrity constraints. Inf. Syst. **33**(4–5), 407–434 (2008)
10. Chomicki, J., Marcinkowski, J.: Minimal-change integrity maintenance using tuple deletions. Inf. Comput. **2005** (2005)
11. Kolahi, S., Lakshmanan, L.V.S.: On approximating optimum repairs for functional dependency violations. In: Proceedings of the 12th International Conference on Database Theory, series ICDT '09, pp. 53–62. ACM, New York (2009)
12. Chandel, A., Koudas, N., Pu, K.Q., Srivastava, D.: Fast identification of relational constraint violations. In: International Conference on Data Engineering, pp. 776–785 (2007)
13. Zhang, B., Tang, X., Wei, W., Zhang, M.: A data cleaning method based on association rules. In: International Conference on Intelligent Systems and Knowledge Engineering, ISKE (2007)
14. Bohannon, P., Flaster, M., Fan, W., Rastogi, R.: A cost-based model and effective heuristic for repairing constraints by value modification. In: SIGMOD Conference, pp. 143–154 (2005)
15. Damerau, F.J.: A technique for computer detection and correction of spelling errors. Commun. ACM **7**(3), 171–176 (1964)
16. Golab, L., Karloff, H.J., Korn, F., Srivastava, D., Yu, B.: On generating near-optimal tableaux for conditional functional dependencies. PVLDB **1**(1), 376–390 (2008)
17. Berti-Equille, L., Dasu, T., Srivastava, D.: Discovery of complex glitch patterns: a novel approach to quantitative data cleaning. In: ICDE, pp. 733–744 (2011)
18. Zaki, M.J., Ogihara, M.: Theoretical foundations of association rules. In: 3rd ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (1998)
19. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. In: SIGMOD Record, vol. 22(2), pp. 207–216 (1993). doi:10.1145/170036.170072
20. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly generating billion-record synthetic databases. In: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, series SIGMOD '94, pp. 243–252. ACM, New York (1994). doi:10.1145/191839.191886