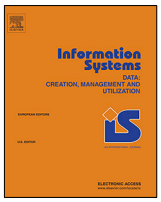
Information Systems 114 (2023) 102158

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is



Guided conditional functional dependency discovery

Sijia Jiang [a,b], Zijing Tan [a,b,*], Jiawei Wang [a,b], Zhikang Wang [a,b], Shuai Ma [c]

[a] School of Computer Science, Fudan University, China
[b] Shanghai Key Laboratory of Data Science, China
[c] SKLSDE Lab, Beihang University, China

ARTICLE INFO

Article history:
Received 8 June 2022
Received in revised form 6 October 2022
Accepted 8 December 2022
Available online 17 December 2022
Recommended by Gottfried Vossen

Keywords:
Conditional functional dependency
Dependency discovery
Metadata

ABSTRACT

Conditional functional dependencies (CFDs) generalize functional dependencies and lend themselves to wide applicability. CFDs on data are usually unknown and too costly to be designed manually. To this end, CFD discovery methods are studied for discovering hidden CFDs from data. In the setting of data cleaning, only a small number of CFDs are used to detect and repair errors, while common CFD discovery methods find all CFDs (approximately) holding on data, and an expensive post-processing step is further required for selecting those relevant ones. In this paper, we present an approach to discover CFDs that can detect errors in data, guided by a small set of erroneous attribute values labeled by users. (1) We present a method that consists of several modules of data sampling, CFD discovery and refinement guided by the user labeling and data re-sampling guided by the discovered CFDs, working in an iterative way. (2) We present novel efficient techniques to facilitate our approach, aiming at identifying CFDs useful for cleaning and reducing user interactions. (3) We conduct extensive experimental evaluations to verify our approach, against the state-of-the-art CFD discovery algorithms with or without user interactions.

1. Introduction

Data quality management is one of the most important aspects of data management, and data cleaning methods, *e.g.,* [1–11], are at the core of data quality management. Most of the data cleaning methods employ integrity constraints, *a.k.a.* dependencies, to state specifications that data should satisfy, and resolve violations by updating data to satisfy the constraints. To obtain the dependencies used in data cleaning and avoid the labor-intensive process of designing dependencies manually, (approximate) dependency discovery techniques are actively studied (see, *e.g.,* [12,13] for surveys), aiming at finding dependencies that (approximately) hold on data.

Conditional functional dependencies (CFDs) [14] generalize functional dependencies (FDs). CFDs can express dependencies that hold only on a subset of data, and are hence more applicable in practice and widely used in data cleaning tasks [15–17]. To find CFDs hidden in data, CFD discovery techniques [18–21] are extensively studied. However, we find their limitations hinder them from efficiently finding CFDs useful for data cleaning.

Traditional CFD discovery methods [18,19,21] discover *all* CFDs that (approximately) hold on data, which usually leads to a very large result set. Since data cleaning requires a (small) set of semantically meaningful CFDs and usually only users can make judgement on the usefulness of CFDs, an additional post-processing step is needed for choosing relevant CFDs among all the discovered ones. This step is necessarily hard and tedious even by domain experts. An alternative is to involve user interactions at early states. Intuitively, this can help discover meaningful CFDs, and also avoid the cost of discovering all CFDs. Explain [20] makes the first effort toward this. It asks users to manually identify and clean a set of dirty tuples only at the beginning, and then automatically discovers a single CFD that is assumed to "explain" all the user modifications. However, we argue it is very difficult for users to manually clean data, and even this is possible, it is almost impossible that all user cleaning behaviors are related to one CFD since users typically have no idea of the constraints "underlying" data.

In this paper, we present a novel solution to CFD discovery, referred to as *guided* CFD discovery. Compared with existing works, our approach has the following advantages. (a) Guided by a small number of user labeling, we identify a set of CFDs useful for data cleaning, rather than a single CFD or all CFDs holding on data. (b) We adopt the idea of *active learning* in user labeling. Users are asked to *label* erroneous attributes on a small set of tuples automatically selected by our system, instead of manually identifying and cleaning erroneous attribute values from all tuples.

* Corresponding author at: School of Computer Science, Fudan University, China.
E-mail address: zjtan@fudan.edu.cn (Z. Tan).

https://doi.org/10.1016/j.is.2022.102158
0306-4379/

Contributions. (1) We propose a novel approach to CFD discovery, leveraging a small number of user labeling (Section 4). We first sample a set of tuples, and users are asked to label tuples as being *true* or *false*, and label erroneous attributes on the *false* tuples. We then discover a set of CFDs concerning user's labeling, and re-sample tuples by considering the discovered CFDs. Users are again asked to label the newly sampled tuples, and the set of discovered CFDs is refined based on user's new labeling. Our approach works in an iterative way, until a set of CFDs that can detect errors is discovered.
(2) We develop a set of techniques for sampling initial tuples, discovering CFDs concerning user's labeling, re-sampling tuples guided by the discovered CFDs, and refining or generating new CFDs according to user's labeling (Section 5). Our aim is to find the CFDs that facilitate data cleaning, and to help users label errors in a convenient way by minimizing the number of required labeling.
(3) Using both real-life and synthetic data, we conduct an experimental study to verify the effectiveness and efficiency of our approach, against the state-of-the-art techniques for CFD discovery with or without user interactions (Section 6). The results show that our approach can effectively discover a set of CFDs for detecting errors, with limited user effort in labeling tuples.

2. Related work

In this section, we categorize related work as follows.

(Conditional) FD discovery without user interaction. (Conditional) FD discovery methods without user interaction are extensively studied in the literature. See [22] for a comparison of several methods. Different from FD discoveries, *e.g.,* [23–27], nontrivial extensions are required for CFD discoveries [18,19,21]. The methods that discover all (C)FDs, however, usually suffer from the huge number of the discovered (C)FDs.

Some methods aim for a more *informative* result set. Ranking functions are used to help select relevant FDs [24], in a post-processing step after all FDs are discovered. Some *top* FDs according to an entropy-based measure are discovered [28], and FDs are also measured with *the reliable fraction of information* [29]. The mutual information from multivariate and mixed-type data is studied [30], and the mutual information is further improved by using uniform priors and smoothing techniques [31]. The relationship between FD discovery and the solution of a sparse regression problem is also studied [32].

However, these approaches cannot guarantee that the discovered FDs can detect errors in data. This work is different, since interaction is involved to discover CFDs that can detect errors, guided by user's labeling.

(Conditional) FD discovery with user interaction. To our best knowledge, user interaction is only considered in [20] for (C)FD discovery. Specifically, users are asked to manually identify and clean dirty tuples, and only one CFD is discovered based on the assumption that all user modifications are related to it. We argue that this assumption is too strong and users can hardly manually identify incorrect attribute values from all tuples or provide correct values for them. We adopt a practical setting, where users are only required to label true or false values on the tuples presented to them, and a set of CFDs is discovered.

User interaction in detecting and repairing errors. The method of [33] tries to identify as many errors as possible, within a budget of the number of user interactions, where the interaction can be one of the validations of FDs, attribute values, or tuples. This paper aims to discover CFDs useful for data cleaning, guided by user's labeling. Another approach [16] repairs data by interacting with the user to confirm a set of SQL update queries. As noted in [20], such query can be encoded as a *constant* CFD. We consider discovering both *constant* and *variable* CFDs (defined in Section 3), resulting in a much larger search space. The user feedback is incorporated into data repairing [17], with a given set of CFDs as data quality rules. This work is different and indeed complementary to [17], by discovering CFDs to be used in data repairing.

Recently, an error detection system is proposed [34]. It generates a feature vector for each attribute value, by collecting the output of a set of error detection algorithms. It then samples tuples for users to label, and trains a classification model for each column based on the feature vectors and user labels. The trained models are used to predict the correctness of remaining attribute values. Based on [34], an error correction system is then developed [35]. This paper only employs CFDs to identify errors, without the need of configuring a set of different error detection algorithms. We aim for a set of CFDs, guided by user's labeling. The discovered CFDs have clear semantics and can serve as metadata, in addition to their usage in data cleaning.

Table 1
Example: a relation about customers.

TID	CC	AC	PN	NM	STR	CT	ZIP	LABEL
t_1	01	908	1111111	Mike	Tree Ave.	~~LA~~	07974	×
t_2	01	908	1111111	Jack	Tree Ave.	~~LA~~	07974	×
t_3	01	908	1111111	Rick	Tree Ave.	~~GLA~~	07974	×
t_4	01	212	2222222	Joe	3rd Str.	NYC	01202	√
t_5	01	908	2222222	Jim	3rd Str.	MH	07975	√
t_6	44	131	3333333	Ben	High Str.	EDI	EH4 1DT	√
t_7	44	131	4444444	Ian	High Str.	EDI	EH4 1DT	√
t_8	44	908	4444444	Ian	Port Pl	MH	W1B1JH	√
t_9	~~44~~	908	5555555	Sean	3rd Str.	MH	07975	×

3. Preliminaries

In this section, we review some basic notations of CFDs and CFD discovery.

$R(A_1, \ldots, A_n)$ denotes a relational schema with attributes $A_1, \ldots, A_n$, and $dom(A)$ denotes the domain of attribute A. We use r to denote a relation instance of schema R, s, t to denote tuples in r, and $t[A]$ to denote the value of attribute A in t.

Conditional functional dependency (CFD) [14,20]. A CFD φ is a pair $(X \rightarrow A, t_p)$, where (1) X is a set of attributes and A is an attribute of schema R; (2) $X \rightarrow A$ is a traditional FD, called the *embedded* FD; and (3) t_p is called a *pattern tuple*, whose attributes are from $X \cup \{A\}$. For each $B \in X \cup \{A\}$, $t_p[B]$ is either a constant value from $dom(B)$, or an unnamed variable '_'. Note that in this paper we consider *normal CFDs* with a single attribute on the RHS; obviously, this does not lose generality.

A CFD φ is called a *constant* CFD if $t_p[B]$ is a constant for all $B \in X \cup \{A\}$, otherwise φ is called a *variable* CFD (it is assumed that $t_p[A]$ must be '_' in a variable CFD). A constant CFD is indeed an association rule, and a variable CFD is a traditional FD if $t_p[B]$ is '_' for all $B \in X \cup \{A\}$.

The semantics of a CFD $\varphi = (X \rightarrow A, t_p)$ on an instance r is defined as follows. (1) A tuple $t \in r$ is said to *match* a pattern tuple t_p in attributes X, denoted by $t[X] \asymp t_p[X]$, if for all $B \in X$, either $t_p[B] =$ '_', or $t[B] = t_p[B]$. Intuitively, this implies that φ can be applied to t. (2) A tuple t *violates* a variable CFD $\varphi = (X \rightarrow A, t_p)$ if $t[X] \asymp t_p[X]$ and there exists another tuple $t' \in r$ such that $t[X] = t'[X]$ and $t[A] \neq t'[A]$. A tuple t *violates* a constant CFD $\varphi = (X \rightarrow A, t_p)$, if $t[X] = t_p[X]$ and $t[A] \neq t_p[A]$. If there are no violations of φ on r, then φ holds on r, written as $r \models \varphi$.

Example 1. In Table 1, we show a relation of customer data (neglect the attribute LABEL for now). We give some CFDs as follows.

(1) $\varphi_1 = ([\text{CC}, \text{AC}] \rightarrow \text{CT}, (01, 908 \parallel \text{MH}))$ is a constant CFD, where $t_p[\text{CC}] =$ "01", $t_p[\text{AC}] =$ "908", $t_p[\text{CT}] =$ "MH". It states that for a tuple t, if $t[\text{CC}] =$ "01" and $t[\text{AC}] =$ "908", then $t[\text{CT}]$ must be "MH".
(2) $\varphi_2 = ([\text{CC}, \text{AC}] \rightarrow \text{CT}, (\_, \_ \parallel \_))$ is a traditional FD. Any two tuples s, t that have the same values on both CC and AC, also have the same value on CT.

Since there are erroneous values (in red), φ_1 and φ_2 do not hold. For example, it can be seen that t_1 violates φ_1, and that t_1, t_5 violate φ_2. □

CFD discovery. To cope with dirty data in practice, CFD discovery algorithms [18,19,21] discover CFDs approximately holding on data with some exceptions. There are three commonly adopted measures.
(1) The *support* of $\varphi = (X \rightarrow A, t_p)$ in r, denoted by $\text{supp}(\varphi, r)$, is defined as the number of tuples $t \in r$ such that $t[X] \asymp t_p[X]$, *i.e.,* the number of tuples that φ can be applied to.
(2) The *confidence* of a CFD φ in r, denoted by $\text{conf}(\varphi, r)$, is computed as $1 - \frac{|r'|}{\text{supp}(\varphi, r)}$, where $r' \subseteq r$ is a minimal subset such that $r \setminus r' \models \varphi$. Intuitively, $|r'|$ is the minimum number of tuples that must be removed such that φ is satisfied. The *confidence* measures the degree of satisfaction of φ.
(3) The *frequency* of a constant value c on attribute A in instance r, is the ratio of the number of tuples with c on A to the number of total tuples in r.

CFDs for data cleaning. CFD discovery algorithms [18,19,21] find all CFDs whose *support* $\geq \theta$, *confidence* $\geq \mu$, and all constants used in any pattern tuple have a *frequency* $\geq \varepsilon$, where θ, μ, ε are given thresholds.

For discovered CFDs with a *confidence* in the range of $[\mu, 1)$, several tuples from r violate them. Since these CFDs hold on most of the data, the violating tuples are assumed to contain errors and should be cleaned. However, many violations of CFDs, maybe most, are not really related to errors in data. Hence, there is no guarantee that the discovered CFDs are valuable in capturing data errors and facilitating data cleaning.

Different from the previous works, our goal is to identify CFDs that can detect errors on data. In the sequel, we develop novel techniques for this purpose.

4. Overview of our approach

In this section, we briefly explain every module underlying our approach (Fig. 1), followed by an illustrative example.
(1) We start with the *Initial Sampling Module* (Section 5.1), aiming to automatically sample an initial set of tuples for users to label. Since we do not have any user interactions yet, this module samples data only based on the characteristic of data, and organizes sampled data for ease of labeling. The output of this module is some tuple pairs and tuples in each pair have the same values on several attributes. As will be illustrated in Example 2, this helps users label tuples and facilitates CFD discovery.
(2) The sampled data are labeled by users and then used as the input of the *Initial CFD Discovery Module* (Section 5.2). The goal of this module is to discover an initial set of CFDs. The discovered CFDs are put into the *Rule Repository* $\mathcal{R}$, while the failed CFD candidates are put into the *Trash Repository* $\mathcal{T}$ (Section 5.3). The CFDs discovered at this stage are usually too *general* to have enough *confidence* on the whole instance, and will be refined (specialized).
(3) The *Re-Sampling Module* (Section 5.4) samples more tuples for labeling, so as to refine CFDs from rule repository $\mathcal{R}$. It considers a set of carefully selected top-$\mathcal{K}$ CFDs from $\mathcal{R}$, and samples the most likely violating tuples, *i.e.,* tuples that violate the most number of CFDs discovered so far. It also prefers to sample *diversified* tuples, for providing good coverage of the original data.
(4) After users label the newly sampled tuples, the *CFD Refinement Module* (Section 5.5) is called with the updated labeled data. The goal is to update $\mathcal{R}$ by refining existing CFDs and introducing new CFDs, in response to newly labeled errors. $\mathcal{T}$ is updated as well. After several iterations of re-sampling and refinement, the discovery terminates when a set of desirable CFDs is found.

Example 2 (*Example 1 continued*)**.** The errors in Table 1 are related to the CFD $([\text{CC}, \text{AC}] \rightarrow \text{CT}, (\_, \_ \parallel \_))$. We show how this CFD can be properly discovered by our approach.
(1) We sample some pairs, say (t_1, t_3) and (t_1, t_5). It can be seen that the two tuples in a pair have the same values on several attributes. Intuitively, variable CFDs can only be violated by tuples having the same values on all the left hand side (LHS) attributes, and it is easier to label tuples in pair than a single tuple, since each tuple in the pair serves as a reference to the other.
(2) Assume users have perfect knowledge. They label t_1, t_3 as *false* tuples, $t_1[CT]$, $t_3[CT]$ as *false* values, and t_5 as a *true* tuple (*false* is denoted by red in Table 1). Some *general* CFDs, *e.g.,* $(\text{CC} \rightarrow \text{CT}, (\_ \parallel \_))$, $(\text{AC} \rightarrow \text{CT}, (\_ \parallel \_))$, $(\text{CT} \rightarrow \text{CC}, (\_ \parallel \_))$ are discovered by the initial CFD discovery module. These CFDs have attribute CT, *i.e.,* the attribute containing *false* values, on the LHS or right-hand-side (RHS). These CFDs pass the validation on the sampled data. However, they are too general to (approximately) hold on the original data and will be further refined, *i.e.,* the *confidence* of them is low.
(3) The re-sampling module may sample (t_4, t_5) that violates $(\text{CC} \rightarrow \text{CT}, (\_ \parallel \_))$. If this pair is found to be a *false negative* after user's labeling, then it will help refine $(\text{CC} \rightarrow \text{CT}, (\_ \parallel \_))$. We are *less* likely to sample (t_2, t_3), although it violates $(\text{CC} \rightarrow \text{CT}, (\_ \parallel \_))$ and $(\text{AC} \rightarrow \text{CT}, (\_ \parallel \_))$. The reason is that (t_1, t_3) and (t_2, t_3) have the same values on the same attribute set, *i.e.,* {CC, AC, PN, STR, ZIP }. Since (t_1, t_3) is already sampled, (t_2, t_3) is usually not considered for better *diversification*. As detailed in Sections 5.1 and 5.4, a parameter is used to favor the diversification.
(4) Users label the sampled pair (t_4, t_5) as a pair of *true* tuples. This helps refine $(\text{CC} \rightarrow \text{CT}, (\_ \parallel \_))$ to be, *e.g.,* $([\text{CC}, \text{AC}] \rightarrow \text{CT}, (\_, \_ \parallel \_))$, $([\text{CC}, \text{STR}] \rightarrow \text{CT}, (\_, \text{High Str.} \parallel \_))$ and $([\text{CC}, \text{NM}] \rightarrow \text{CT}, (\_, \_ \parallel \_))$. As will be studied in Section 5.3, (a) $([\text{CC}, \text{AC}] \rightarrow \text{CT}, (\_, \_ \parallel \_))$ is of high *rank* in the rule repository $\mathcal{R}$. (b) For $([\text{CC}, \text{STR}] \rightarrow \text{CT}, (\_, \text{High Str.} \parallel \_))$, there are no violations *w.r.t.* it, but it only applies to t_6, t_7 and hence has relatively low *support*. (c) $([\text{CC}, \text{NM}] \rightarrow \text{CT}, (\_, \_ \parallel \_))$ will be put into the trash repository $\mathcal{T}$, if (t_7, t_8) is sampled later and labeled by users. □

5. Details of our approach

In this section, we discuss every module in detail.

5.1. Initial sampling module

This module samples tuples from the original instance r, *without* interacting with users. The sampled tuples are organized in pairs, and two tuples in a pair have the same values on several attributes. The reason for this strategy is two-fold. (a) Violations of variable CFDs are incurred by two tuples having the same values on all LHS attributes; and (b) it is much easier for users to identify erroneous values from a pair of similar tuples than a single tuple.

Algorithm. Algorithm 1 illustrates the initial sampling module. It takes as input the instance r, and outputs a set $\mathcal{D}$ of tuple pairs.

Recall that we aim to sample tuples having the same values on several attributes. We first select the set *bestAttrs* of attributes for

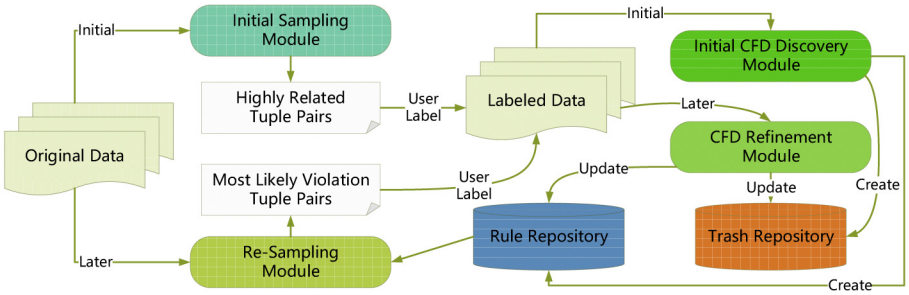


Fig. 1. The workflow of our system.

Algorithm 1: InitialSamplingModule

Input: all tuples from the original instance r
Output: a set $\mathcal{D}$ of tuple pairs
1 build auxiliary data structures *Plis* and *PliRecords*;
2 *bestAttrs* ← the set of attributes such that tuples in r have many distinct values on these attributes;
3 **for** *len* = |*bestAttrs* | − 1; *len* ≠ 0; −−*len* **do**
4 *combi* ← *combination* (*bestAttrs*, *len*);
5 *randomAttrSets* ← *randomSample* (*combi*);
6 **foreach** *attrSet in randomAttrSets* **do**
7 *eqClass* ← the set of tuples with the same values on all attributes from *attrSet*;
8 $\mathcal{D}$ ← $\mathcal{D}$ ∪ no more than λ tuple pairs from *eqClass*;
9 **if** $\mathcal{D}$.*size* ⩾ *the required sample size* **then**
10 **return**;

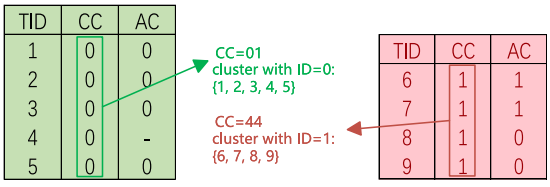


Fig. 2. *PliRecords* of attribute CC and AC.

this purpose, after building the auxiliary data structures (Lines 1–2). We favor an attribute A if there are many distinct values from r on A. Such attributes can be efficiently identified leveraging the auxiliary structures, as will be illustrated shortly. We then combine attributes in *bestAttrs* to facilitate the next step. We start from attribute sets with as many attributes as possible, and gradually decrease the number of attributes (Lines 3 to 5). For each attribute combination (set) *attrSet*, we get the set *eqClass* of tuples that agree on values of all attributes from *attrSet* (Line 7). We find the benefit of sampling tuple pairs with the same values on an attribute set decreases as the number of pairs increases. Hence, a threshold λ is used as the upper bound of the number of tuples pairs for each attribute set (Line 8). We adopt an additional optimization to reduce the number of sampled tuples (not shown in Algorithm 1). When sampling a pair for one attribute set, we favor tuples that are already sampled for other attribute set(s). For example, suppose (t_i, t_j) is sampled. If possible, for another attribute set we prefer to sample a pair also with t_i (t_j). The basic intuition is that the reuse of the same tuple reduces the size of initial sampling and hence the number of tuples to be labeled, without affecting effectiveness of the sampling.

We give more details of our implementation (data structures).

Data structures. We adopt the data structures of *position list indexes* (*Plis*) and *PliRecords* (line 1) to encode origin data in a compact way [24–26]. A *Pli* π_X is a set of clusters on attribute set X, and tuples in the same cluster have the same value on X. Along the same lines as previous works, it is safe to discard clusters with only one tuple. The compact representations of all tuples are *PliRecords*, in which each tuple's value on attribute set X is the cluster ID from *Pli* π_X. Recall *Plis* and *PliRecords* are very memory-efficient, and the original dataset is not needed any more after building *Plis* and *PliRecords*. We use the same data structures of *Plis* and *PliRecords* as former works [20,24–26], since the data structures have already been well studied and proven efficient, and are not the major concern of this paper.

Algorithm 2: *bestAttrs*

Input: all tuples from the original instance r
1 *Plis* ← *buildPlis* (*r*);
2 *Plis* ← *sort* (*Plis*, Descending);
3 *PliRecords* ← *createPliRecords* (*Plis*);
4 *bestAttrs* ← {*attrs* \ *attrs* with few clusters | *attrs* ∈ *Plis* };

Example 3. For Table 1, we have $\pi_{\{AC\}}$ = {{1, 2, 3, 5, 8, 9}, {6, 7}} and $\pi_{\{CC\}}$ = {{1, 2, 3, 4, 5}, {6, 7, 8, 9}}. We show *PliRecords* of attributes CC and AC in Fig. 2. We have no value in AC of tuple t_4, since the cluster containing t_4 has only one tuple and is discarded. □

We illustrate the usage of *Plis* and *PliRecords* with a detailed analysis of Algorithm 1. To select the set *bestAttrs* of attributes (line 2), we sort attributes according to the number of clusters in *Plis*, and discard attributes with very few clusters (shown in Algorithm 2). For each attribute set *attrSet*, we get the set *eqClass* of tuples that agree on values of all attributes from *attrSet* (line 7). Leveraging *PliRecords*, clusters on multiple attributes can be computed in $O(|r|)$, where $|r|$ is the number of tuples in r (shown in Algorithm 3).

Example 4 (*Example* 3 *continued*).. We illustrate the running of Algorithm 3, for computing $\pi_{\{CC,AC\}}$ from $\pi_{\{CC\}}$. Consider *lastEqClass* = $\pi_{\{CC\}}$ = {{1, 2, 3, 4, 5}, {6, 7, 8, 9}} (Line 1), and *cluster* = $\pi_{\{CC=44\}}$ = {6, 7, 8, 9} (Line 4). We partition tuples from {6, 7, 8, 9} by their values on AC, leveraging a key–value pair *clusterMap* whose key is tuple's cluster ID from $\pi_{\{AC\}}$. Refer to Fig. 2. We have *clusterMap* [0] = {8, 9} and *clusterMap* [1] = {6, 7}, *i.e.*, $\pi_{\{CC=44,AC=908\}}$ = {8, 9} and $\pi_{\{CC=44,AC=131\}}$ = {6, 7} (Lines 5–9). □

Remark. (1) Algorithm 1 has a worst-case complexity of $O(|r| \cdot 2^k)$ by building *Plis* and *PliRecords* with hashing in $O(|r|)$, where k is the size of *bestAttrs*. (2) All further manipulations of data leverage *PliRecords*, without visiting original data. The original tuples are only presented to users for labeling.

Algorithm 3: *eqClass*

Input: *Plis*, attribute set *attrSet*
Output: equivalence class *eqClass* for *attrSet*
1 *lastEqClass* ← *Plis.at* (*attrSet* [0]);
2 **for** *i = 1; i ≠ attrSet.size; ++i* **do**
3 *nowEqClass* ← ∅;
4 **foreach** *cluster in lastEqClass* **do**
5 **foreach** *recordID in cluster* **do**
6 *pliID* ← *PliRecords* [*recordID*][*attrSet* [*i*]];
7 *clusterMap* [*pliID*].*add* (*recordID*);
8 **foreach** *splitClu in clusterMap* **do**
9 **if** *splitClu.size* ≥ *2* **then** *nowEqClass.add* (*splitClu*) ;
10 *lastEqClass* ← *nowEqClass*;
11 *eqClass* ← *lastEqClass*;

5.2. Initial CFD discovery module

The initial CFD discovery module takes as input the sampled set $\mathcal{D}$. Leveraging user's labeling on $\mathcal{D}$, it discovers an initial set of CFDs.

User's labeling. User interaction is employed to guide CFD discovery in our system. A set of tuple pairs is automatically selected and presented to users, and users are asked to label it. Specifically, (a) if an attribute value is found to be incorrect, then users should label the attribute value and tuple with the value as *false*; and (b) if a tuple does not have any incorrect values, then users should label the tuple as *true*. Note that users are *not* asked to provide correct values. Our aim here is to reduce the burden on users. Indeed, labeling false values does not always imply that the correct values are known by the users. In real-life dirty data, some false values can be labeled even if the correct values are unknown. They can be typos, *e.g.,* "femle" for sex (users know this is an error even if they do not know whether "female" is the correct value), placeholder values, *e.g.,* "XXX" for name, and outliers, *e.g.,* "999" for age.

Tuples in $\mathcal{D}$ are presented to users in pairs. After user's labeling, tuple pairs can be categorized into three types:
(1) *Type*-{*true*, *true* }: we refer to a pair as a *tt-type* pair, if the two tuples in it are both labeled as *true* tuples. Since users are assumed to have perfect knowledge, we should guarantee that tuples from *tt-type* pairs *never* violate any discovered CFDs.
(2) *Type*-{*true*, *false* }: we refer to a pair as a *tf-type* pair, if one tuple is labeled as a *true* tuple, while the other is a *false* one. Intuitively, the *false* value labeled in the *false* tuple is very likely to be the same as the value on the same attribute of the *true* tuple. We consider this factor when generating candidate CFDs.
(3) *Type*-{*false*, *false* }: if both tuples in a pair are *false* tuples, then there are further three cases as follows.

(a) if the two tuples have the same incorrect value on the same attribute, then we refer to this tuple pair as a *ff-SS-type* pair (the same attribute with the same error value). Note that variable CFDs may fail to detect errors on these two tuples, but constant CFDs can. For example, both t_1 and t_2 have the same incorrect value "LA" on attribute CT. The variable CFD ([CC, AC] → CT, (_, _ ∥ _)) is not violated by t_1, t_2, while the constant CFD ([CC, AC] → CT, (01, 908 ∥ MH)) can detect this error.

(b) if the two tuples have different incorrect values on the same attribute, then we call this pair a *ff-SD-type* pair (the same attribute with different error values). The error can be identified by variable and constant CFDs. For example on attribute CT, the error value of t_1 is "LA", but that of t_3 is "GLA". We see that ([CC, AC] → CT, (_, _ ∥ _)) and ([CC, AC] → CT, (01, 908 ∥ MH)) can detect this error.

(c) if the two tuples have error values on different attributes, then we call this pair a *ff-D-type* pair (different attributes). We treat a *ff-D-type* pair as two *tf-type* pairs.

Note that it is possible for a pair of false tuples to be classified in different ways with respect to different attributes. We aim to discover CFDs for detecting errors, leveraging users's labeling. We first present several definitions.

Attributes concerning user's labeling. For a pair (t_i, t_j), we identify two sets of attributes according to user's labeling.
(1) $S^{err}[t_i, t_j]$ is a set of attributes on which *false* values are labeled.
(2) $S^{free}[t_i, t_j]$ is a set of attributes, such that (a) $S^{free}[t_i, t_j] \wedge S^{err}[t_i, t_j] = \emptyset$; and (b) $t_i[A] = t_j[A]$ for all $A \in S^{free}[t_i, t_j]$, *i.e.,* t_i, t_j have the same values on all attributes from $S^{free}[t_i, t_j]$.

To enable constants in the pattern tuples of CFDs, attributes in $S^{err}[t_i, t_j]$ and $S^{free}[t_i, t_j]$ can be associated with frequent values from instance r.[1] The type of (t_i, t_j) is considered when constants are introduced to $S^{err}[t_i, t_j]$ and $S^{free}[t_i, t_j]$, as illustrated below.

Example 5 (*Example 1 continued.*)**.** Suppose we sample tuple pairs (t_1, t_3), (t_1, t_5) and (t_1, t_9), and users label *false* values on attributes CC and CT, as shown in Table 1.
(1) (t_1, t_5) is a *tf-type* pair. We have $S^{err}[t_1, t_5] = \{\text{CT, CT} = \text{"MH"}\}$. As noted earlier, it is very likely that the *false* tuple t_1 has the same value as the *true* tuple t_5 on CT. Hence we only consider the frequent value "MH" rather than all frequent values. We have $S^{free}[t_1, t_5] = \{\text{CC, CC} = \text{"01"}, \text{AC, AC} = \text{"908"}\}$, since t_1, t_5 have the same values on CC, AC with values "01" and "908", respectively.
(2) (t_1, t_3) is a *ff-SD-type* pair. $S^{err}[t_1, t_3] = \{\text{CT, CT} = \text{"MH"}, \text{CT} = \text{"EDI"}\}$, since neither "LA" nor "GLA" is a correct value on CT for t_1 (t_3). $S^{free}[t_1, t_3] = \{\text{CC, CC} = \text{"01"}, \text{AC, AC} = \text{"908"}, \text{PN, PN} = \text{"1111111"}, \text{STR, STR} = \text{"Tree Ave."}, \text{ZIP, ZIP} = \text{"07974"}\}$.
(3) (t_1, t_9) is a *ff-D-type* pair. By treating it as two *tf-type* pairs, $S^{err}[t_1, t_9]$ is the union of $\{\text{CC, CC} = \text{"01"}\}$ and $\{\text{CT, CT} = \text{"MH"}\}$. □

Based on $S^{err}[t_i, t_j]$ and $S^{free}[t_i, t_j]$, we present our method to generate candidate CFDs.

$S^{err}[t_i, t_j]$ **as RHS attribute.** Each attribute from $S^{err}[t_i, t_j]$ is used as the RHS attribute of CFDs one by one, and combinations of attributes from $S^{free}[t_i, t_j]$ are used as LHS attributes. We start from a single attribute on the LHS, and more attributes from $S^{free}[t_i, t_j]$ are added to the LHS if necessary.

$S^{err}[t_i, t_j]$ **as LHS attribute.** One attribute from $S^{free}[t_i, t_j]$ is used as the RHS of CFDs, and attributes from $S^{err}[t_i, t_j]$, $S^{free}[t_i, t_j]$ are used on the LHS. We start from CFDs with only one attribute from $S^{err}[t_i, t_j]$ on the LHS, and more attributes are gradually added to the LHS if necessary.

Algorithm. Algorithm 4 illustrates the initial CFD discovery module. It is conducted on $\mathcal{D}_t$, the "shuffled" version of $\mathcal{D}$ with user's labeling. That is, $\mathcal{D}_t$ is a set of tuples, containing exactly the same tuples as $\mathcal{D}$. Algorithm 4 stores the discovered candidate CFDs (resp. candidates that fail in the validation) in the rule repository $\mathcal{R}$ (resp. trash repository $\mathcal{T}$). It enumerates tuple pair (t_i, t_j) from $\mathcal{D}_t$ with at least one *false* tuple from $\mathcal{D}$, and considers $S^{err}[t_i, t_j]$ and $S^{free}[t_i, t_j]$. It first generates the most general candidate CFDs. More attributes are then added to the LHS for more specified candidates if former ones fail in the validation (lines 2–5 and lines 6–13), following the approach to generating candidates stated above. A parameter *maxLength* is used in *appendAttr* to limit the maximum number of LHS attributes. Function *Validation* is at the core of Algorithm 4, which is called to check the validity of a CFD candidate.

Validation. We check the validity of a candidate CFD φ as follows.
(1) Query the trash repository $\mathcal{T}$, to verify that φ did not fail in

[1] We assume all frequent values are collected in the pre-processing step.

Algorithm 4: InitialCFDDiscoveryModule

```
Input: Labeled data D_t, Rule and Trash Repository R, T;
tp⁻ ← all tuple pairs from D_t with false tuple(s) from D ;
for (t_i, t_j) in tp⁻ do
    foreach rhs in S^err[t_i, t_j] do
        S_temp ← S^free[t_i, t_j];
        appendAttr (S_temp, maxLength, null, rhs);
    foreach rhs in S^free[t_i, t_j] do
        S_temp ← S^free[t_i, t_j] \ rhs;
        foreach a_lhs in S^err[t_i, t_j] do
            if Validation (a_lhs → rhs, D_t) then
                R.add (a_lhs → rhs);
                break ;
        T.add (a_lhs → rhs);
        appendAttr (S_temp, maxLength − 1, a_lhs, rhs);
Function appendAttr (S_temp, maxLength, init_lhs, rhs)
for len = 1; len ≠ maxLength; ++len do
    lhsList ← combination(S_temp, len);
    delSet ← ∅;
    foreach lhs in lhsList do
        rule ← (lhs ∪ init_lhs → rhs);
        if Validation (rule, D_t) then
            R.add (rule);
            delSet.add (attributes in lhs);
        else
            T.add (rule);
    S_temp.remove(attributes in delSet);
```

the validation in former rounds. We give more details of $\mathcal{T}$ in Section 5.3. (2) Check φ on $\mathcal{D}_t$, against all tuple pairs from $\mathcal{D}_t$ (a single tuple suffices for constant CFDs). If φ is violated, then make sure in the violation there are some *false* values labeled by users. That is, φ must hold on the partial data without *false* values, since we assume that users have perfect knowledge.

Example 6 (*Example 3 continued.*)**.** Assume we generate a candidate $\varphi_1 = (\text{AC} \rightarrow \text{CT}, (908 \parallel \text{EDI}))$, using CT = "EDI" from $S^{err}[t_1, t_3]$ on the RHS. We see t_5 can be used to invalidate φ_1 since $t_5[\text{AC}] =$ "908" and $t_5[\text{CT}] =$ "MH", and then φ_1 will be put into the trash repository $\mathcal{T}$. It can be seen that t_9 can also be used for this purpose. Although t_9 is a *false* tuple, the correct values in t_9 suffice.

After more attributes from $S^{free}[t_1, t_3]$ are added to the LHS, φ_1 will be refined to, *e.g.*, $\varphi_2 = ([\text{CC}, \text{AC}] \rightarrow \text{CT}, (01, 908 \parallel \text{EDI}))$ and $\varphi_3 = ([\text{AC}, \text{PN}] \rightarrow \text{CT}, (908, 1111111 \parallel \text{EDI}))$. We see that φ_2 is again invalidated by t_5. In contrast, φ_3 is *not* invalidated by t_1 or t_3, since the *false* values on CT are involved in the violations. □

Remark. (1) Algorithm 4 discovers candidate CFDs with attributes concerning user's labeling, on the labeled data $\mathcal{D}_t$. It has a complexity of $O(|\mathcal{D}_t| \cdot 2^m)$, where $|\mathcal{D}_t|$ is the number of tuples in $\mathcal{D}_t$, and m is the size of $S^{err}[t_i, t_j] \cup S^{free}[t_i, t_j]$ for all $t_i, t_j \in \mathcal{D}_t$.

(2) Candidate CFDs can be generated by using attributes only from $S^{free}[t_i, t_j]$ ($t_i, t_j \in \mathcal{D}_t$), even if there is no *false* tuple in the initial sampling. Algorithm 4 can be easily modified for this, still with the same complexity. The discovered CFDs will be put into the trash repository $\mathcal{T}$ later if we find they *cannot* detect errors (Section 5.3), or they can be refined after the re-sampling module (Section 5.4).

5.3. Rule repository and trash repository

We maintain rule repository $\mathcal{R}$ and trash repository $\mathcal{T}$. $\mathcal{R}$ is used to store candidate CFDs discovered so far, while $\mathcal{T}$ is used to store *failed* candidates. The two repositories are updated in the initial discovery (Algorithm 4) and in each round of CFD refinement (Section 5.5).

After initial discovery and refinement, for candidates in $\mathcal{R}$ we compute their *support* and *confidence* on the whole instance r. That is, we check on r the *validity* of CFD candidates that are discovered on the sampled data. The computation of *support* and *confidence* requires to enumerate candidates from $\mathcal{R}$, with a cost linear in $|r|$ for each candidate leveraging *PliRecords*. We then further update $\mathcal{R}$ and $\mathcal{T}$ to facilitate the next round of re-sampling as follows.

Trash repository $\mathcal{T}$. We move some candidates from $\mathcal{R}$ to $\mathcal{T}$, such that $\mathcal{T}$ stores *failed* candidates that (a) have *confidence* of 100% in r, or (b) have *support* less than the threshold θ in r, or (c) failed in the validations (Section 5.2). This is because (a) a candidate CFD of 100% *confidence* is not useful for data cleaning since there are no violations *w.r.t.* it, and (b) the *support* of a CFD never increases when more attributes (with constants) are added to the LHS. Besides each failed candidate, we also add all of the more general CFDs to $\mathcal{T}$. For example, if the CFD $([\text{CC}, \text{CT}] \rightarrow \text{AC}, (01, \text{MH} \parallel 908))$ is added into $\mathcal{T}$, then all CFDs more general than it, *e.g.*, $(\text{CT} \rightarrow \text{AC}, (\text{MH} \parallel 908))$ and $(\text{CC} \rightarrow \text{AC}, (01 \parallel 908))$, are also added into $\mathcal{T}$. In our discovery process, we can directly discard a candidate if it is found in $\mathcal{T}$.

Top-$\mathcal{K}$ strategy. After moving some failed candidates to $\mathcal{T}$, remaining ones in $\mathcal{R}$ are candidate CFDs that will be refined through the re-sampling and refinement modules. It can be costly to re-sample tuples for all candidates in $\mathcal{R}$. To this end, we adopt a strategy based on top-$\mathcal{K}$ results: we rank CFDs in $\mathcal{R}$, only re-sample tuples for the top-$\mathcal{K}$ CFDs, and terminate the discovery if the *confidence* and *support* of each of the top-$\mathcal{K}$ CFDs are above the given thresholds. This is consistent with our goal of finding a small set of CFDs for detecting errors. It is worth mentioning that the top-$\mathcal{K}$ CFDs in $\mathcal{R}$ are *dynamic*: the ranks are required to be recomputed after each round of re-sampling and CFD refinement.

Rank CFDs in $\mathcal{R}$. We rank CFDs φ in $\mathcal{R}$ (a) first in ascending order of the number of attributes, (b) then in descending order of the number of attributes with *false* values, and (c) finally in descending order of $vsupp(\varphi, \mathcal{D}_t)$, where $vsupp(\varphi, \mathcal{D}_t) = \sum_{t_i, t_j \in \mathcal{D}_t} vsupp(\varphi, (t_i, t_j))$ and $vsupp(\varphi, (t_i, t_j)) = 1$ if all attributes (with constants) from φ are in $S^{err}[t_i, t_j] \cup S^{free}[t_i, t_j]$, and 0 otherwise. Intuitively, we prefer to perform re-sampling and refinement for CFDs that have fewer attributes, contain more attributes concerning *false* values, and can be applied to more tuple pairs in $\mathcal{D}_t$.

5.4. Re-sampling module

The goal of re-sampling module is to re-sample tuples for refining top-$\mathcal{K}$ CFDs from $\mathcal{R}$. The top-$\mathcal{K}$ CFDs whose *confidence* in the instance r is smaller than the given threshold μ are too general. Therefore, some tuples *without* incorrect values may violate the CFDs; the tuples are *false negative* tuples *w.r.t.* the CFDs. By sampling *false negative* tuples from r and asking users to label them, too general CFDs can be refined. After several rounds of re-sampling and CFD refinement, candidate CFDs in $\mathcal{R}$ will converge to final ones.

We again prefer a small set of tuples for labeling. Hence, the intuition is to sample tuples that violate the most number of top-$\mathcal{K}$ CFDs from $\mathcal{R}$; we can refine several CFDs if they are *true* tuples labeled by users. We also consider the *diversification* of attribute sets, along the same line as initial sampling.

Algorithm. Algorithm 5 enriches $\mathcal{D}$ by re-sampling tuples from r. It enumerates top-$\mathcal{K}$ CFDs φ from $\mathcal{R}$. For each φ whose *confidence* is smaller than the threshold μ, it samples tuple pairs such that

Algorithm 5: Re-SamplingModule

Input: Rule and Trash Repository $\mathcal{R}$ and $\mathcal{T}$, and Sampled set $\mathcal{D}$;
1 $tpSet \leftarrow \emptyset$, $sampleCount \leftarrow 0$;
2 **foreach** *top-$\mathcal{K}$ CFD φ from $\mathcal{R}$* **do**
3 **if** *the confidence of φ < threshold μ* **then**
4 $tpSet \leftarrow tpSet \cup$ several pairs of tuples that violate φ ;
5 sort tuple pair tp from $tpSet$ in descending order by $vio(tp)$, where $vio(tp)$ is the number of top-$\mathcal{K}$ CFDs violated by tp;
6 **foreach** *tp in tpSet* **do**
7 **if** *$\mathcal{D} \cup \{tp\}$ is diversified enough w.r.t. threshold λ* **then**
8 $\mathcal{D} \leftarrow \mathcal{D} \cup \{tp\}$;
9 $sampleCount \leftarrow sampleCount + 1$;
10 **if** *$sampleCount \geqslant$ the required size* **then**
11 **break** ;

the two tuples in a pair have the same values on the LHS but different values on the RHS of φ, *i.e.,* a pair of tuples violating φ (Lines 3–4). A parameter is used to limit the number of sampled tuple pairs for each such φ. It collects tuple pairs in the set *tpSet*, and then sorts them by the number of violated top-$\mathcal{K}$ CFDs in descending order (line 5). This favors tuple pairs that violate more CFDs. When adding tuple pairs into $\mathcal{D}$, it additionally considers the diversification of attribute sets, by using the threshold λ (the same as Algorithm 1) to restrict the number of tuple pairs in $\mathcal{D}$ on the same attribute set (Line 7). The re-sampling continues until the required number of tuples are collected (Lines 9–11).

Remark. Algorithm 5 enumerates $\mathcal{K}$ CFDs and has a complexity of $O(|r|)$ for each CFD, leveraging *PliRecords*.

5.5. CFD refinement module

After re-sampling tuples, the CFD refinement module is called to refine CFDs in $\mathcal{R}$ and add new CFDs to $\mathcal{R}$ if necessary. The two tuples in a re-sampled pair violate CFDs in $\mathcal{R}$, and are assumed to contain errors. If they are found to be *true* when labeled, we can refine CFDs violated by them. We can use not only *tt-type* pairs for this purpose, but also *tf-type* and *ff-type* pairs in some cases. This is because a CFD only concerns some attributes of a tuple. When a tuple pair contains *false* values, we can still use it to refine a CFD if the CFD is violated by the pair and only *true* values in the pair are concerned.

Example 7. For the pair (t_1, t_5), there is a *false* value on CT. However, this pair can still be used to refine ([CC, AC] $\rightarrow$ NM, (_, _ ‖ _)). The reason is that the pair violates the CFD, and all values concerning the CFD are *true*. That is, (t_1, t_5) still forms a *false negative* for ([CC, AC] $\rightarrow$ NM, (_, _ ‖ _)). □

Refine existing CFDs. If a *false negative* tuple pair, say (t_i, t_j), is found for a CFD, then we refine the CFD by including more attributes (with constants) on the LHS. The refinement strategy is different from that of the initial discovery module; those in $S^{free}[t_i, t_j]$ are not used. Indeed, it is easy to see the CFD is still violated by the pair if attributes (with constants) from $S^{free}[t_i, t_j]$ are added to its LHS, *i.e.,* (t_i, t_j) remains a *false negative w.r.t.* the refined CFD.

Example 8. As shown in Example 2 (3), (t_4, t_5) is a *false negative* for the CFD (CC $\rightarrow$ CT, (_ ‖ _)). After user's labeling, we have $S^{err}[t_4, t_5] = \emptyset$ and $S^{free}[t_4, t_5]$ = {CC, CC = "01", PN, PN = "2222222", STR, STR = "3rd Str."}. When more attributes are required to refine the CFD, those from $S^{free}[t_4, t_5]$ cannot be used. Instead, we can use AC, NM, ZIP (with frequent constants). We can also use CC = "44" to generate the refined CFD (CC $\rightarrow$ CT, (44 ‖ _)), but later it will be invalidated by, *e.g.,* (t_7, t_8). □

Table 2
Statistics of datasets.

Dataset	#Tuples	#Attrs	Threshold θ of *support*
Abalone	8,354	9	10%
Adult	48,842	11	1%
Soccer	100,000	10	10%
SP500	245,148	7	1%

Generate new CFDs. To discover CFDs that can detect all the errors, after re-sampling and labeling, we combine (a) each new *false* tuple with each of the previous *true* (resp. *false*) tuple to form a *tf-type* (resp. *ff-type*) pair, and (b) each new *true* tuple with each of the previous *false* tuple to form a *tf-type* pair. This is necessary since after re-sampling, *false* values can occur on some attributes that are free of *false* values before. A discovery process similar to the initial CFD discovery is then carried out to generate new CFDs. Since both $\mathcal{T}$ and the labeled dataset keep growing in each round, the new process should consider the latest version of them.

Remark. Rule repository $\mathcal{R}$ and trash repository $\mathcal{T}$ are updated in the CFD refinement module. Our approach terminates if we have top-$\mathcal{K}$ CFDs with *confidence* and *support* above the given thresholds, otherwise it starts the next round of re-sampling and refinement.

6. Experimental evaluations

In this section, we present an experimental study to compare our method with the state-of-the-art CFD discovery methods, and to analyze the performance of our algorithms in detail.

6.1. Experimental setting

Datasets. We test four datasets that are also used in [20]. Abalone and Adult are from the UCI Repository (http://archive.ics.uci.edu/ml/), SP500 is real-life data about stock trading from [36], and Soccer is synthetic data generated by BART [37] (http://www.db.unibas.it/projects/bart/). The characteristics of each dataset are in Table 2. Recall that previous works [15–17] on data cleaning with CFDs mostly concern categorical attributes but not numerical attributes.

Algorithms. We implement our method, referred to as GCFD (code is available online from https://github.com/jariver/GCFD), and obtain implementations of Explain [20], CTane [18], Itemset-First and FD-First [21] (available online from https://bit.ly/2yFNksO and https://bit.ly/2MYzjcH). Recall that [20] discovers a single CFD to "explain" all user modifications, while CTane, Itemset-First and FD-First discover all CFDs that approximately hold. All the algorithms are implemented in C++.

Noise introduction. There are no "golden" CFDs for evaluating the effectiveness of CFD discoveries. Along the same lines as [20,33], we use BART [37] to introduce noises (errors) to attribute values, and evaluate the effectiveness of discovered CFDs in terms of their abilities to detect the attribute values with introduced noises. BART takes a clean dataset and a set of CFDs as input, and produces a dataset with violations of the given CFDs, controlled by two parameters: (a) the number of attributes containing noises, and (b) the percentage of noise values. On each dataset, we use 3 (resp. 5, 10) CFDs for introducing noises (the CFDs are given in the Appendix for reference), and 80% (resp. 90%, 100%) of the attributes are concerned in the 3 (resp. 5, 10) CFDs. For each CFD, we use η = 0.1%/0.5%/1%/2% as the percentage of noises, where 20%–30% (resp. 70%–80%) of the noises are on the

Table 3
Statistics of user's labeling.

Dataset	#CFDs	#InitLabel	#InitFalse	#TotalLabel	#TotalFalse
Abalone	3	20	9.25	102.25	20.25
	5	20	9.75	97.5	15.75
	10	20	10.33	80.67	17.33
Adult	3	20	0	102	26.25
	5	19.5	0	96.25	21.25
	10	20	3.33	82.67	27.33
Soccer	3	19.5	9.5	82	19.25
	5	19.25	9.75	93	22.25
	10	19.33	9	86.67	26
SP500	3	20	10.75	61	15.75
	5	20	10.25	60.5	23.25
	10	20	11	64	13.67

LHS (resp. RHS) attributes of the CFD. Since noises introduced for some CFDs can be on the same tuple, we experimentally find that up to 10% of the tuples are modified by BART in our experiments.

Parameters. Along the same setting as [20], we set the *confidence* threshold $\mu = 1 - \eta$, where η is the percentage of introduced noises in BART. The threshold θ of *support* is shown in Table 2, and all constants in pattern tuples of CFDs have *frequency* $\varepsilon \geq \theta$. For GCFD, we discover top-30 CFDs by default, and set the threshold λ (Algorithms 1 and 5) as 2, to enable diversification in sampling. To favor the efficiency of CTane, Itemset-First and FD-First, we set an upper bound of 5 on the number of LHS attributes in discovered CFDs.

In our experimental evaluations, we simulate tuple labeling by software since the ground truth values are known. This does not affect the experimental results concerning the effectiveness of our approach. We use a machine with an Intel Core i5 Processor (2.9GHZ), 16 GB of memory and Ubuntu 18.04. We run each experiment 3 times and report the average.

6.2. Experimental results

Exp-1: User's labeling in GCFD. In Table 3, the column of #CFDs shows the number of CFDs to introduce noises. We vary η and report the average results of labeling. The results tell us the following.

(1) The #InitLabel and #InitFalse columns show the numbers of labeled tuples and *false* values in the initial sampling. Since we always sample pairs of tuples having the same values on several (many) attributes, we see there is a high probability that some tuples with noises are sampled. The only exception is found on Adult; no *false* value is labeled on the initial sampling when 3 or 5 CFDs are used. However, the results verify that we can still discover CFDs for detecting errors in this case, since we can refine the initially discovered CFDs and (or) discover new ones by re-sampling tuples with noises.

(2) The #TotalLabel and #TotalFalse columns show the total numbers of labeled tuples and *false* values during the discovery. We see only a very small number of tuples are labeled, *e.g.,* only 60+ tuples on SP500 of more than 245K tuples. Better, recall that the labeled tuples are automatically selected by our system. We also see the total number of labeled tuples *does not* increase with the number of CFDs (#CFDs). This justifies our approach to re-sampling tuple pairs that violate more CFDs, so as to simultaneously refine several CFDs.

(3) In Table 4, we run experiments by varying the number of tuples of each dataset (25% and 50% of all tuples). The results show that the number of labeled tuples is not sensitive to the instance size. For example, the number of labels increases by only 40% for a fourfold increase in tuples on Abalone, and even less on other datasets. In the columns of #Tuples, #FalseTuples and #Noises, we show the number of tuples, the number of tuples with noise attribute values and the number of noise attribute values, respectively. The value in #Noises is larger than that in #FalseTuples, since one tuple may contain several noise attribute values. More specifically, we find one tuple can contain up to 3 noise attribute values in this set of experiments. The value in #TotalLabel (resp. #TotalFalse) is much smaller than that in #Tuples (resp. #FalseTuples) on large datasets. Only a very small proportion of (false) tuples are required to be labeled, which justifies the efficiency of our approach.

We do not report the number of labeling in Explain [20], which would be too subjective since users are required to manually identify and correct erroneous tuples concerning one CFD.

Exp-2: Effectiveness of GCFD. We evaluate the effectiveness of CFD discoveries, aiming for a set of CFDs that can detect noises introduced to attribute values, in the form of CFD violations. We use f-measure $F_\beta = \frac{(\beta^2+1)\cdot prec\cdot recall}{\beta^2\cdot prec+recall}$, where *prec* is the ratio of the correctly detected attribute values with noises to all values involved in violations of the discovered CFDs, and *recall* is the ratio of values with noises detectable by the discovered CFDs to values with noises detectable by the original 10 CFDs for introducing noises.[2] Note we aim for CFDs with better abilities to detect errors, while judging CFDs based on other measures (given below) is a departure from our goal.

CTane discovers all CFDs approximately holding, so do Itemset-First and FD-First. This results in a much larger number of CFDs than GCFD by orders of magnitude (shown in Fig. 3). All the CFDs discovered by CTane, however, have very low *prec*, since most CFD violations do not concern the introduced noises. To favor the f-measure values of CTane, we perform additional post processings to select CFDs from the result of CTane, with different measures: (a) *confidence*, (b) *support*, and (c) *succinctness*. Succinctness is a common criterion for ranking discovered dependencies [36,38]. It favors CFDs with fewer LHS attributes, *i.e.,* more general CFDs. We use the top-100 CFDs instead of the top-30 CFDs for CTane, because the f-measure values of the top-30 CFDs of CTane are still very low. The results in Table 5 tell us the following.

(1) GCFD significantly outperforms all other methods. This justifies the idea of this work, to enable an effective CFD discovery leveraging user interactions. Specifically, Explain suffers from very low *recall* since it discovers one CFD. CTane(succ) has good *recall* but very low *prec* (not shown); it discovers general CFDs incurring too many *false negatives.* CTane(supp) has better *prec* but worse *recall* than CTane(succ). CTane(conf) performs the worst.

(2) As β increases, the f-measure of GCFD increases. Indeed, GCFD has very high *recall* (the top-30 CFDs include nearly all the CFDs for introducing noises) and relatively low *prec* (17%–63% on different datasets). This is reasonable since all CFDs have relatively low *prec*, except the ones for introducing noises. Therefore, a larger β favors the f-measure of GCFD. Similar trends are found on different versions of CTane.

(3) All methods have the worst performance on SP500. There are no frequent values in SP500 and hence all CFDs on SP500 are actually FDs. This justifies CFDs are better than FDs in detecting errors.

The discovery algorithms of CFDs are data-driven mining approaches with no guarantee that the discovered CFDs are genuine,

[2] We use BART to introduce noises by changing attribute values involved in CFDs with some fixed probability (the percentage of noises η). The noises are random noises, such as typos, duplicated values, and outliers. Some of the noises do not lead to CFD violations, and are hence not detectable even by the CFDs used for introducing noises.

Table 4
Number of Labels with varying number of tuples.

Dataset	#CFDs	%Tuples	#TotalLabel	#TotalFalse	#Tuples	#FalseTuples	#Noises
Abalone	10	25	56.25	12.25	2,089	106	109
		50	68.33	14.5	4,177	212	217
		100	80.67	17.33	8,354	425	434
Adult	10	25	75.5	24.25	12,210	576	591
		50	77.33	25	24,421	1,152	1,183
		100	82.67	27.33	48,842	2,305	2,366
Soccer	10	25	81.75	24.67	25,000	1,195	1,240
		50	84.33	25.33	50,000	2,389	2,480
		100	86.67	26	100,000	4,778	4,960
SP500	10	25	50.33	13	61,287	2,708	2,798
		50	55.25	13.33	122,574	5,413	5,596
		100	64	13.67	245,148	10,827	11,192

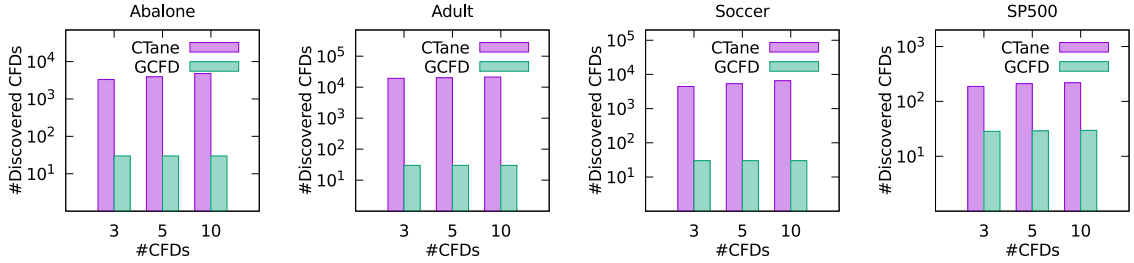


Fig. 3. The number of discovered CFDs.

Table 5
Effectiveness of different methods.

Dataset	β	CTane			Explain	GCFD
		supp	*conf*	*succ*		
Abalone	2	40.8%	0.1%	11.6%	10.9%	82.2%
	3	48.2%	0.1%	20.3%	9.8%	88.5%
	4	52.1%	0.1%	29.5%	9.4%	91.4%
Adult	2	12.1%	1.7%	16.7%	6.9%	60.7%
	3	12.4%	1.7%	22.7%	6.2%	70.7%
	4	12.5%	1.6%	26.6%	5.9%	75.1%
Soccer	2	39.0%	0.3%	23.5%	8.3%	81.9%
	3	48.1%	0.4%	31.6%	7.5%	85.1%
	4	53.2%	0.4%	36.9%	7.2%	86.5%
SP500	2	10.0%	1.0%	9.6%	7.7%	48.0%
	3	17.8%	1.9%	17.2%	6.9%	62.9%
	4	26.3%	3.0%	25.6%	6.7%	72.2%

and hence a post-processing step may be required to select CFDs. However, compared with asking a human to manually design CFDs, discovering candidate CFDs and then asking a human to select genuine ones is more practical in terms of the required human effort, especially for CFDs with multiple LHS attributes and constant values. Compared with the previous discovery methods, our approach can identify a smaller set of CFDs with better f-measure values. Therefore, it is much easier to further select CFDs from the result set of our method if necessary.

Exp-3: Efficiency of GCFD. We show running times of all the methods in Fig. 4. On each dataset, we vary the number of CFDs to introduce noises in different figures, and in each figure vary the percentage of introduced noises. Note the results should not be considered as a comparison of the running times of different methods, since GCFD discovers top-$\mathcal{K}$ CFDs instead of all CFDs and the labeling time of GCFD is not included.

We see the following from the results. (1) GCFD (excluding the labeling time) is very efficient. The efficiency is due to (a) we combine CFD discovery on the sampled data (initial CFD discovery and refinement) with CFD validation on the whole instance (confidence and support computation in rule repository $\mathcal{R}$); and (b) we aim for top-$\mathcal{K}$ CFDs that can detect errors instead of all CFDs. (2) Neither the number of CFDs for introducing noises nor the percentage of noises has evident effect on the efficiency of GCFD. Similar results are found on other methods in most cases.

The time for labeling is not reported, which could be too subjective. Note that (a) we only label a small number of tuples; (b) the time of user interactions in [20] is not included either, and [20] places a heavier burden on users than us; and (c) it takes substantially additional cost to choose meaningful CFDs in CTane, Itemset-First and FD-First.

Exp-4: Running time details of our method. In Table 6, we show the times for initial sampling (InitSamp) and initial discovery (InitDis), the rounds of re-sampling and CFD refinement (Rounds), and the average times of re-sampling (Re-samp) and refinement (Refine) in each round. All times are reported in millisecond (ms).

We see the following. (1) Leveraging the data structure of *PliRecords*, sampling processes are very fast. (2) CFD discoveries usually take a longer time than sampling, but are also very efficient. (3) Each round of re-sampling and refinement takes less than 1 s on Abalone and Soccer, and only 2–5 s on Adult and SP500. This implies a very quick response to user's labeling.

Exp-5: The impact of $\mathcal{K}$ and λ. In this set of experiments, we study the impact of the parameters $\mathcal{K}$ and λ. Recall that in GCFD we discover top-$\mathcal{K}$ CFDs with $\mathcal{K} = 30$ by default, and set $\lambda = 2$ by default (Algorithms 1 and 5). We still use 10 CFDs to introduce noises and $\beta = 2$ for f-measure. We vary $\mathcal{K}$ and λ, and report the number of labeled tuples (#Labels) and f-measure in Fig. 5. When $\lambda = \infty$, it implies that we do not restrict the number of tuple pairs on the same attribute set. We see the following.

(1) The change of $\mathcal{K}$ usually slightly affects #Labels when $\mathcal{K}$ increases from 10 to 50. The increase of #Labels becomes evident only when $\mathcal{K} = 100$.

(2) For f-measure, top-30 CFDs deliver good results, while top-10 or top-20 CFDs do not perform well. Although top-50 and top-100 CFDs can slightly beat top-30 CFDs in terms of f-measure on most datasets, recall that usually more tuples are required to be labeled for top-50 and especially for top-100 CFDs. The poor

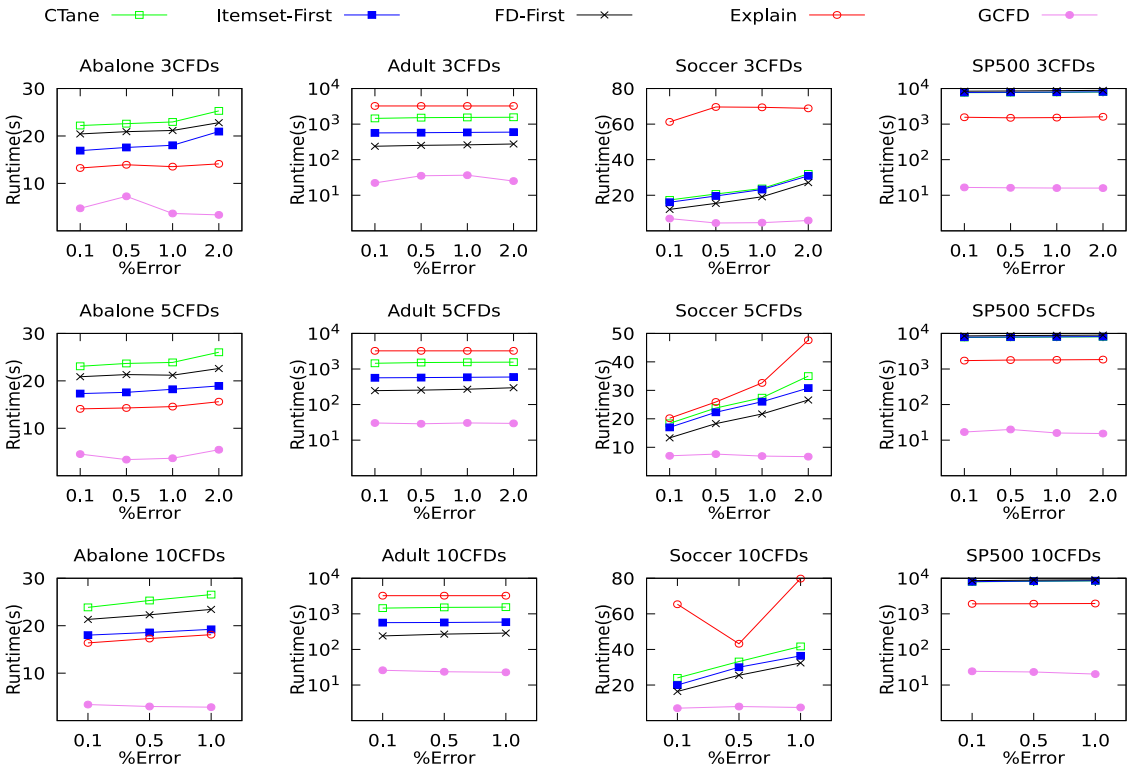


Fig. 4. Running time of all the methods.

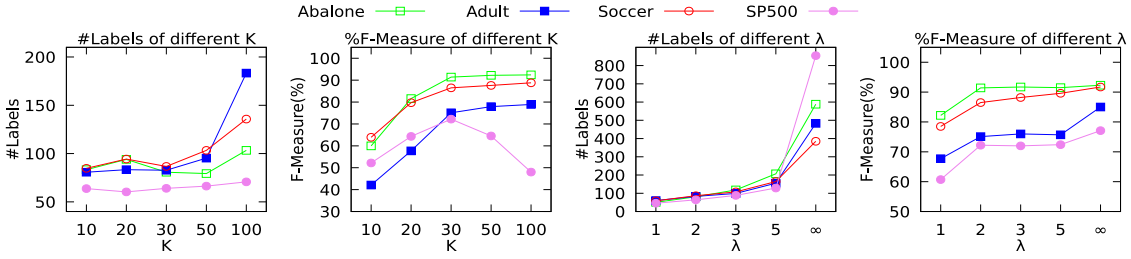


Fig. 5. The impact of $\mathcal{K}$ and λ.

Table 6
Running time of each module.

Dataset	#CFDs	InitSamp	InitDis	Rounds	Re-samp	Refine
Abalone	3	154 (ms)	735	14.75	58	251
	5	156	826	15.33	44	221
	10	157	742	12.22	43	187
Adult	3	1363	2949	19.17	104	2067
	5	1446	3286	17.50	101	1454
	10	1574	3351	14.44	98	2562
Soccer	3	1399	393	20.17	39	161
	5	1432	560	24.25	40	193
	10	1460	1064	23.00	40	221
SP500	3	3779	9482	10.00	296	975
	5	3791	13578	8.50	308	1251
	10	3966	12081	10.67	3034	1450

performance of top-50 and top-100 CFDs on SP500 is due to very low *prec* with too many discovered CFDs.

(3) #Labels significantly increases as λ increases. Although more labels always help improve f-measure, we cannot afford too many manual labels. By setting $\lambda = 2$, we strike a balance between effectiveness and efficiency.

Table 7
CFDs used in Abalone.

(whole, viscera, rings = 10) → height
(diameter, height, whole, whucked) → rings
(length, whole, shell, rings = 9) → height
(sex=M, whole, whucked, viscera) → shell
(whole, whucked, shell) → height
(whole, viscera, shell) → height
(diameter, whucked, viscera, shell) → height
(diameter, height, whole, whucked) → shell
(height, whucked, viscera, shell) → sex
(height, whole, whucked, shell) → sex

7. Conclusion

We have proposed an approach to CFD discovery, for identifying CFDs that can detect errors in data and minimizing the number of required labeling. We have developed a set of techniques underlying our approach, and conducted experimental evaluations to verify the effectiveness.

We intend to study more dependency discoveries with user interactions, *e.g.,* denial constraints [2] and order dependencies [39,40]. We also intend to adapt our approach to the distributed setting for dealing with very large datasets, along the same lines

Table 8
CFDs used in Adult.

(maritalstatus = Never-married, education = HS-grad, age = 18-21) → income = LessThan50K
(relationship = Husband, education = Some-college) → sex = Male
(relationship = Husband, income = MoreThan50K) → sex = Male
(relationship = Husband, age =>50) → marital-status = Married-civ-spouse
(relationship = Wife, country = United-States, age = 31–50) → sex = Female
(Marital-status = Married-civ-spouse, Education = Prof-school, Sex = Male) → Relationship = Husband
(age =<18) → income = LessThan50K
(marital-status = Never-married, sex = Male, education = Some-college, age = 18–21) → income = LessThan50K
(workclass = ?) → occupation = ?
(occupation = ?, age =>50) → workclass =?

Table 9
CFDs used in Soccer.

(stadium = King_Power_Stadium, position, surname, season) → team
(city, birthplace, season = 2013) → position
(position, surname, city = Solna) → season
(surname, birthplace, season) → position
(surname, city, season = 2013) → position
(position, surname) → name
(name, season) → position
(name, season) → birthplace
(name, season) → surname
(city, season = 2015) → team

Table 10
CFDs used in SP500.

(High, Low, Volume) → Close
(Symbol, Close, Volume) → High
(Open, Close, Volume) → High
(Date, Open, Volume) → High
(Date, Low, Volume) → High
(Date, High, Volume) → Low
(Open, High, Volume) → Date
(High, Low, Volume) → Date
(Date, High, Volume) → Symbol
(Open, High, Volume) → Symbol

as [41–43]. The dynamic scenario also deserves further investigation, to perform dynamic CFD discovery in response to the changes of data. Dynamic discovery techniques have been recently developed for FDs without user interactions [44,45].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

code is available online from https://github.com/jariver/GCFD

Acknowledgments

We thank anonymous reviewers for their valuable suggestions. This work is supported by National Key Research and Development Program of China 2018YFB1700403 and National Natural Science Foundation of China 62172102, 61925203.

Appendix

CFDs used in the experiments. In Table 7, Table 8, Table 9 and Table 10, we show the CFDs that are used to introduce noises on the tested datasets. We use shorthands in the tables. As an example, the expression (whole, viscera, rings = 10) → height is used to denote the CFD ([whole, viscera, rings] → height, (_, _, 10 ‖ _)). The ratios between constant CFDs and variable CFDs vary on different datasets. For example, all the CFDs used on Adult are constant CFDs. The numbers of the LHS attributes of the used CFDs are in the range of [1, 4]. Please note that in the dataset Adult (Table 8), "?" is a value for attributes workclass and occupation, so are "<18" and ">50" for the attribute age.

References

[1] X. Chu, I.F. Ilyas, S. Krishnan, J. Wang, Data cleaning: Overview and emerging challenges, in: SIGMOD, 2016.
[2] X. Chu, I.F. Ilyas, P. Papotti, Holistic data cleaning: Putting violations into context, in: ICDE, 2013.
[3] M. Dallachiesa, A. Ebaid, A. Eldawy, A.K. Elmagarmid, I.F. Ilyas, M. Ouzzani, N. Tang, Nadeef: a commodity data cleaning system, in: SIGMOD, 2013.
[4] F. Geerts, G. Mecca, P. Papotti, D. Santoro, The LLUNATIC data-cleaning framework, PVLDB 6 (9) (2013).
[5] C. He, Z. Tan, Q. Chen, C. Sha, Repair diversification: A new approach for data repairing, Inform. Sci. 346–347 (2016) 90–105.
[6] I.F. Ilyas, X. Chu, Trends in cleaning relational data: Consistency and deduplication, Found. Trends Databases 5 (4) (2015) 281–393.
[7] I.F. Ilyas, X. Chu, Data Cleaning, ACM, 2019.
[8] N. Prokoshyna, J. Szlichta, F. Chiang, R.J. Miller, D. Srivastava, Combining quantitative and logical data cleaning, Proc. VLDB Endow. 9 (4) (2015) 300–311.
[9] A.A. Qahtan, N. Tang, M. Ouzzani, Y. Cao, M. Stonebraker, Pattern functional dependencies for data cleaning, PVLDB 13 (5) (2020) 684–697.
[10] J. Rammelaere, F. Geerts, Cleaning data with forbidden itemsets, IEEE Trans. Knowl. Data Eng. 32 (8) (2020) 1489–1501.
[11] M. Volkovs, F. Chiang, J. Szlichta, R.J. Miller, Continuous data cleaning, in: ICDE, 2014.
[12] Z. Abedjan, L. Golab, F. Naumann, T. Papenbrock, Data profiling, in: Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2018.
[13] S. Song, F. Gao, R. Huang, C. Wang, Data dependencies extended for variety and veracity: A family tree, IEEE Trans. Knowl. Data Eng. 34 (10) (2022) 4717–4736.
[14] W. Fan, F. Geerts, X. Jia, A. Kementsietsidis, Conditional functional dependencies for capturing data inconsistencies, TODS 33 (2) (2008) 6:1–6:48.
[15] G. Cong, W. Fan, F. Geerts, X. Jia, S. Ma, Improving data quality: Consistency and accuracy, in: VLDB, 2007.
[16] J. He, E. Veltri, D. Santoro, G. Li, G. Mecca, P. Papotti, N. Tang, Interactive and deterministic data cleaning, in: SIGMOD, 2016.
[17] M. Yakout, A.K. Elmagarmid, J. Neville, M. Ouzzani, I.F. Ilyas, Guided data repair, PVLDB 4 (5) (2011) 279–289.
[18] W. Fan, F. Geerts, J. Li, M. Xiong, Discovering conditional functional dependencies, IEEE Trans. Knowl. Data Eng. 23 (5) (2011) 683–698.
[19] L. Golab, H.J. Karloff, F. Korn, D. Srivastava, B. Yu, On generating near-optimal tableaux for conditional functional dependencies, PVLDB 1 (1) (2008) 376–390.
[20] J. Rammelaere, F. Geerts, Explaining repaired data with cfds, PVLDB 11 (11) (2018) 1387–1399.
[21] J. Rammelaere, F. Geerts, Revisiting conditional functional dependency discovery: Splitting the c from the fd, in: ECML PKDD, 2018.
[22] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J. Rudolph, M. Schönberg, J. Zwiener, F. Naumann, Functional dependency discovery: An experimental evaluation of seven algorithms, PVLDB 8 (10) (2015) 1082–1093.
[23] Y. Huhtala, J. Kärkkäinen, P. Porkka, H. Toivonen, TANE: an efficient algorithm for discovering functional and approximate dependencies, Comput. J. 42 (2) (1999) 100–111.
[24] S. Kruse, F. Naumann, Efficient discovery of approximate dependencies, PVLDB 11 (7) (2018) 759–772.
[25] T. Papenbrock, F. Naumann, A hybrid approach to functional dependency discovery, in: SIGMOD, 2016.

[26] Z. Wei, S. Link, Discovery and ranking of functional dependencies, in: ICDE, 2019, pp. 1526–1537.
[27] C.M. Wyss, C. Giannella, E.L. Robertson, Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances, in: DaWaK, 2001.
[28] P. Mandros, M. Boley, J. Vreeken, Discovering reliable approximate functional dependencies, in: SIGKDD, 2017.
[29] P. Mandros, M. Boley, J. Vreeken, Discovering reliable dependencies from data: Hardness and improved algorithms, in: ICDM, 2018.
[30] P. Mandros, D. Kaltenpoth, M. Boley, J. Vreeken, Discovering functional dependencies from mixed-type data, in: SIGKDD, 2020.
[31] F. Pennerath, P. Mandros, J. Vreeken, Discovering approximate functional dependencies using smoothed mutual information, in: SIGKDD, 2020.
[32] Y. Zhang, Z. Guo, T. Rekatsinas, A statistical perspective on discovering functional dependencies in noisy data, in: SIGMOD, 2020.
[33] S. Thirumuruganathan, L. Berti-Équille, M. Ouzzani, J. Quiané-Ruiz, N. Tang, Uguide: User-guided discovery of fd-detectable errors, in: SIGMOD, 2017.
[34] M. Mahdavi, Z. Abedjan, R.C. Fernandez, S. Madden, M. Ouzzani, M. Stonebraker, N. Tang, Raha: A configuration-free error detection system, in: SIGMOD, 2019, pp. 865–882.
[35] M. Mahdavi, Z. Abedjan, Baran: Effective error correction via a unified context representation and transfer learning, Proc. VLDB Endow. 13 (11) (2020) 1948–1961.
[36] X. Chu, I.F. Ilyas, P. Papotti, Discovering denial constraints, PVLDB 6 (13) (2013) 1498–1509.
[37] P.C. Arocena, B. Glavic, G. Mecca, R.J. Miller, P. Papotti, D. Santoro, Messing up with bart: error generation for evaluating data-cleaning algorithms, PVLDB 9 (2) (2015) 36–47.
[38] E.H.M. Pena, E.C. de Almeida, F. Naumann, Discovery of approximate (and exact) denial constraints, PVLDB 13 (3) (2019) 266–278.
[39] J. Szlichta, P. Godfrey, J. Gryz, Fundamentals of order dependencies, PVLDB 5 (11) (2012) 1220–1231.
[40] J. Szlichta, P. Godfrey, J. Gryz, C. Zuzarte, Expressiveness and complexity of order dependencies, PVLDB 6 (14) (2013) 1858–1869.
[41] H. Saxena, L. Golab, I.F. Ilyas, Distributed discovery of functional dependencies, in: ICDE, 2019, pp. 1590–1593.
[42] H. Saxena, L. Golab, I.F. Ilyas, Distributed implementations of dependency discovery algorithms, PVLDB 12 (11) (2019) 1624–1636.
[43] S. Schmidl, T. Papenbrock, Efficient distributed discovery of bidirectional order dependencies, VLDB J. 31 (1) (2022) 49–74.
[44] P. Schirmer, T. Papenbrock, S. Kruse, F. Naumann, D. Hempfing, T. Mayer, D. Neuschäfer-Rube, Dynfd: Functional dependency discovery in dynamic datasets, in: EDBT, 2019.
[45] R. Xiao, Y. Yuan, Z. Tan, S. Ma, W. Wang, Dynamic functional dependency discovery with dynamic hitting set enumeration, in: ICDE, 2022, pp. 286–298.