Big Data Mining and Analytics

Volume 3 | Issue 1 Article 5

2020

Mining Conditional Functional Dependency Rules on Big Data

Mingda Li
the Department of Computer Science, University of California, Los Angles, CA 90095, USA.

Hongzhi Wang
the Department of Computer Science and Technology, Harbin Institute of Technology, Harbin 150000, China.

Jianzhong Li
the Department of Computer Science and Technology, Harbin Institute of Technology, Harbin 150000, China.

Follow this and additional works at: https://tsinghuauniversitypress.researchcommons.org/big-data-mining-and-analytics

 Part of the Computer Engineering Commons, Computer Sciences Commons, and the Data Science Commons

Recommended Citation

Mingda Li, Hongzhi Wang, Jianzhong Li. Mining Conditional Functional Dependency Rules on Big Data. Big Data Mining and Anyalytics 2020, 03(01): 68-84.

This Research Article is brought to you for free and open access by Tsinghua University Press: Journals Publishing. It has been accepted for inclusion in Big Data Mining and Analytics by an authorized editor of Tsinghua University Press: Journals Publishing.

BIG DATA MINING AND ANALYTICS
ISSN 2096-0654 06/06 pp 68–84
Volume 3, Number 1, March 2020
DOI: 10.26599/BDMA.2019.9020019

Mining Conditional Functional Dependency Rules on Big Data

Mingda Li, Hongzhi Wang*, and Jianzhong Li

Abstract: Current Conditional Functional Dependency (CFD) discovery algorithms always need a well-prepared training dataset. This condition makes them difficult to apply on large and low-quality datasets. To handle the volume issue of big data, we develop the sampling algorithms to obtain a small representative training set. We design the fault-tolerant rule discovery and conflict-resolution algorithms to address the low-quality issue of big data. We also propose parameter selection strategy to ensure the effectiveness of CFD discovery algorithms. Experimental results demonstrate that our method can discover effective CFD rules on billion-tuple data within a reasonable period.

Key words: data mining; conditional functional dependency; big data; data quality

1 Introduction

With the accumulation of data at present, databases have become increasingly large. At the same time, due to the difficulty in manual maintenance and variations of data sources, big data involves a high possibility of quality problems which make them difficult to use. Therefore, cleaning techniques are crucial for effective use of big data.

Conditional Functional Dependency (CFD) discovery algorithms[1] are powerful tools in data cleaning, because they can find the hidden relation among items. Such relation can help us find dirty tuples which can be modified accordingly. The Functional Dependencies (FDs) can be considered as special forms of CFDs. High-quality rules are the core of effective data cleaning systems with CFDs.

High-quality CFD discovery on big data brings two challenges. First, the volume of big data requires a highly efficient and scalable discovery algorithm with complexity that is linear or sublinear to the data size. Second, big data may involve further data-quality problems. Thus, a clean training set can hardly be prepared for CFD discovery and a fault-tolerant CFD discovery approach is necessary.

Owing to the importance of such an approach, some CFD discovery algorithms have been proposed. However, none of them could address the aforementioned challenges. Most existing methods such as the method in Ref. [2] discover high-quality rules with data mining algorithms on a small but clean dataset efficiently. However, these approaches are unsuitable for big data cleaning due to the lack of representative dataset. Other methods for discovering CFDs on dirty datasets such as the method in Ref. [3] need many passes over the dataset to find approximate CFDs, but this method may not be effective on a big dataset that cannot be loaded into the main memory.

Therefore, developing a scalable method is necessary to mine high-quality rules from big data with size larger than the main memory. To achieve this goal, we design a scalable and systemic algorithm. We sample from the big data first to obtain an effective training set within one pass of scan. Then, we discover CFDs based on sampling results. The reason for sampling before discovery is that, without sampling, we have to scan the big dataset many times to mine patterns and calculate support for finding CFDs. This task is time-consuming, especially when the

• Mingda Li is with the Department of Computer Science, University of California, Los Angles, CA 90095, USA. E-mail: limingda@hit.edu.cn.
• Hongzhi Wang and Jianzhong Li are with the Department of Computer Science and Technology, Harbin Institute of Technology, Harbin 150000, China. E-mail: wangzh@hit.edu.cn; lijzh@hit.edu.cn.
* To whom correspondence should be addressed.
Manuscript received: 2019-09-28; accepted: 2019-10-09

@ The author(s) 2020. The articles published in this open access journal are distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/).

dataset is larger than the memory. Another purpose of sampling is to filter dirty items and keep clean ones. The following example illustrates the need for sampling.

Example 1 Table 1 is changed from the example in Ref. [4]. It is about a customer with the basic information such as Country Code (CC), Area Code (AC), Phone Number (PN), name (NM), and address (street [STR], city [CT], and ZIP code [ZIP]).

From Table 1, we can find the traditional FD sets f_1 and f_2:

$$f_1 : [\mathrm{CC}, \mathrm{AC}] \to \mathrm{CT},$$
$$f_2 : [\mathrm{CC}, \mathrm{AC}, \mathrm{PN}] \to \mathrm{STR},$$

and the CFD sets $\beta_1 - \beta_5$:

$\beta_1 : ([\mathrm{CC}, \mathrm{ZIP}] \to \mathrm{STR}, (40, \_\_ \parallel \_\_))$,

$\beta_2 : ([\mathrm{CC}, \mathrm{AC}] \to \mathrm{CT}, (01, 112 \parallel \mathrm{NYC}))$,

$\beta_3 : ([\mathrm{CC}, \mathrm{AC}] \to \mathrm{CT}, (01, 108 \parallel \mathrm{MH}))$,

$\beta_4 : ([\mathrm{CC}, \mathrm{AC}] \to \mathrm{CT}, (40, 1069 \parallel \mathrm{EDI}))$,

$\beta_5 : ([\mathrm{CC}, \mathrm{NM}] \to \mathrm{STR}, (4731, \mathrm{Steve} \parallel \mathrm{Low\ St.}))$.

However, if the dataset is about the customers in America, then the dirty items t_9 and t_{10} with CT-SYD and LON, which are not in the US, will have few similar items. With the sampling method to find representative samples, we need to ignore them. Moreover, we neglect t_{11}, because we cannot find items that have more than two attributes similar to those of t_{11}, which are not sufficient to find a CFD that shows the hidden relation among most of the items. Therefore, the following rule sets φ_1 and φ_2 could be discovered from the dataset without t_9, t_{10}, and t_{11}:

$$\varphi_1 : ([\mathrm{CC}, \mathrm{ZIP}] \to \mathrm{STR}, (40, \_\_ \parallel \_\_)),$$
$$\varphi_2 : ([\mathrm{AC}] \to \mathrm{CT}, (\_\_ \parallel \_\_)).$$

If we clean the dataset based on the rule sets $\beta_1 - \beta_5$, then t_9 and t_{10} will conform to β_5, and be treated as clean items. However, with the new rule set, these items will become dirty. Meanwhile, as the attributes of the two new rules are less, we do not have to compare them many times. Therefore, the data cleaning with only two rules φ_1 and φ_2 is more efficient than that with five rules $\beta_1 - \beta_5$.

Table 1 Example 1.

Item	CC	AC	PN	NM	STR	CT	ZIP
t_1	01	108	11080176	Ian	Three Ave.	MH	2221
t_2	01	108	11080176	Jack	Tree Ave.	MH	2221
t_3	01	112	11120101	Joe	High St.	NYC	02ED1
t_4	01	108	11120101	Jim	Elm Str.	MH	2221
t_5	40	1069	41690101	Ben	High St.	EDI	02ED1
t_6	40	1069	41690177	Ian	High St.	EDI	02ED1
t_7	40	108	41690177	Ian	Port PI	MH	02WB2
t_8	01	1069	11120101	Sean	3rd Str.	UN	2233
t_9	4731	108	233323	Steve	Low St.	SYD	XXXX
t_{10}	4731	XXXX	3456123267	Steve	Low St.	LON	2112E
t_{11}	8E11	979797	678345	Laola	4th St.	MH	322233

From Example 1, we can find that the selection of training set is important. Meanwhile, for big data, using a small set of items is the only possible approach to rule discovery. Thus, selecting a representative training set from big data is crucial. For the big dataset with size larger than the memory, we attempt to accomplish sampling in one-pass as the sampling method for estimating the confidence of CFDs in Ref. [5].

In summary, the developed rule discovery method that is suitable for big data with size larger than the memory requires the following features, which the existing methods do not have:

(1) A small but representative training set should be selected in one-pass scanning of the data.

(2) The method to discover rules from items should tolerate the wrong records in the training set.

(3) Owing to the tradeoff between effectiveness and efficiency, a mechanism that tunes the parameter according to the need of applications should be provided.

Therefore, we propose a method for discovering a high-quality CFD set. Such an approach could tolerate data-quality problems and meet user requirements for a dataset with size larger than the memory. The contributions of this study are summarized as follows:

(1) We design Representative and Random Sampling for CFDs (BRRSC): a sampling method to obtain a suitable training set from CFD discovery in a single scanning of data. According to the theoretical analysis and experiments, BRRSC is a sub-linear algorithm that is suitable for big data.

(2) We propose Dynamical Fault-tolerant CFD discovery (DFCFD) algorithm that can tolerate error data to discover CFDs by our proposed method. DFCFD can be changed according to different data sizes and parameters of the dirty dataset to obtain the best CFD set.

(3) To resolve conflicts among the discovered CFD set, we propose a graph-based algorithm with each CFD as a node and the conflict relationship between two CFDs as an edge. In this algorithm, the conflict-free CFD set is computed as the maximal weight independent set on the graph.

(4) To meet the various requirements for CFD discovery, we design an adaptive parameter computation strategy for CFD discovery. We define four dimensions

of user requirements. Users are allowed to decide the most important aim in the discovery and set limits for the other three. Thereafter, we propose a multi-objective programming to solve this parameter determination problem.

(5) We verify experimentally the performance and scalability of our algorithm. We compare the time for discovering CFDs and the quality of the CFDs with previous methods for different data sizes and parameters. To test the optimality of the parameter selection method, we compare the effectiveness of different choices of parameters using the controlling variable method. We use real-world big data to show the effectiveness of our method.

We introduce the preliminary definitions and the framework of our solution in Sections 2 and 3, respectively. The sampling method is proposed in Section 4. In Section 5, we develop error-tolerant CFD discovery algorithms and conflict-resolving algorithms. An adaptive parameter selection algorithm is proposed in Section 6. In Section 7, we perform extensive experiments to verify the efficiency and effectiveness of proposed algorithms. Finally, we draw the conclusion in Section 9.

2 Priliminary

In this section, we first review some definitions of CFDs and then define the problem.

2.1 Background

A CFD is a pair $(X \to A, t_P)$, where X is a set of attributes in the items, A is a single attribute decided by X, and t_P is a pattern tuple with attributes in X and A. For an attribute C in $X \cup A$, $t_P[C]$ is either a constant or an undetermined variable denoted as "__". We define X and A as Left Hand Side (LHS) and Right Hand Side (RHS) for a CFD, respectively. A pattern tuple "$\|$" is used to separate X and A attributes.

We call a CFD as constant CFD if t_P consists of constants only, i.e., $t_P[A]$ as a constant and $t_P[B]$ as a constant for all $B \in X$. It is called a variable CFD if $t_P[A]$ is "__", and the value of $t_P[B]$ depends on that of $t_P[A]$. The general CFDs include both of the variable and constant CFDs.

Among the CFD sets $\beta_1 - \beta_5$ in Example 1, β_1 is a variable CFD, while $\beta_2 - \beta_5$ are constant CFDs. In the CFD sets φ_1 and φ_2, φ_1 and φ_2 are both variable CFDs.

When we find CFDs, we should avoid trivial and redundant CFDs to increase efficiency. To achieve this goal, we define the minimal CFDs. A minimal CFD must be a nontrivial and left-reduced CFD first.

A CFD $(X \to A, t_P)$ is trivial when $A \in X$. If a CFD is trivial, it is always correct when the attribute in X is equal to the same attribute in A. It is always wrong when the equality relationship is not met. Therefore, we only study the nontrivial CFDs in this paper. We call the constant CFD $(X \to A, (t_P \parallel a))$ a left-reduced CFD if no set of attributes Z is included in X to make a new CFD $(Z \to A, (t_P \parallel a))$. Similarly, we call a variable CFD left-reduced if for any $Z \subset X$, $(Z \to A, (t_P \parallel a))$ cannot be proved suitable, and no $t_{P'}[X]$ is more general than $t_P[X]$ to make the $(X \to A, (t_{P'} \parallel a))$ correct. To determine the confidence level of a CFD, we say that a tuple supports a CFD when it satisfies the condition in φ.

2.2 Problem definition

Given a dataset that may be quite large, our goal is to find a high-quality CFD set that contains constant and variable CFDs. As the major part of big data is clean, we regard a CFD set as high-quality when most tuples in the big data support it. Meanwhile, a high-quality CFD set should control its CFD number. Thus, we need to discover a CFD set that contains a minimal number of CFDs with most tuples supporting it. Measuring the quality is difficult when considering the number of CFDs and supporters. Therefore, in the experiments, we used a standard CFD set discovered on a clean dataset. Then, we modified the dataset to make it dirty and utilized our method to discover our set of CFDs on it. We evaluate our set of CFDs by comparing them with the standard CFD set.

3 Framework

The framework of the proposed method is shown in Fig. 1. In the working process described by Fig. 1, to obtain a high-quality CFD set from big data, we firstly obtain samples through the algorithm proposed in Section 4.2 in one-pass scanning. Then, an error-tolerant CFD discovery algorithm in Section 5.1 is developed to find CFDs from the samples. Thereafter, we establish a

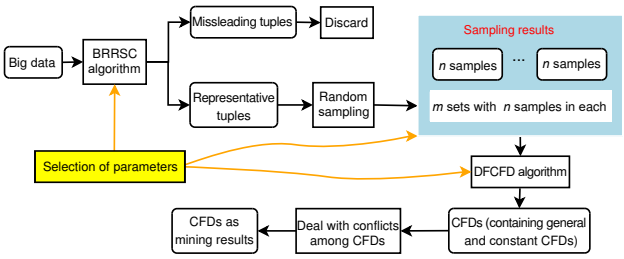


Fig. 1 Framework of the whole process.

weighted undirected graph including CFDs as nodes (Section 5.2.1) and add an edge between two CFDs to represent a conflict (Section 5.2.2). To address the conflicts, we adapt the algorithm in Ref. [3] to find a maximal weighted independent set (Section 5.2.3). Meanwhile, to satisfy various requests from users, we propose a novel method to select the most suitable parameters for CFD discovery (Section 6). In summary, the proposed system framework is separated into four parts: a sampling algorithm, an error-tolerant dynamical CFD discovery algorithm, a method that deals with conflicts among CFDs, and the selection of parameters.

4 Representative and Random Sampling for CFDs (RRSC)

We use sampling method to select a small but representative dataset for CFD discovery. Although reservoir sampling[6] can ensure the equal possibility for each tuple to be sampled with unknown size of the entire data, the representativeness of the sample cannot be ensured. Thus, inspired by the reservoir sampling, we propose a novel sampling algorithm that calculates the number of the same attributes of samples to decide whether a tuple is suitable. To ensure that our samples represent all types of suitable tuples, we select multiple sets of samples from a big dataset. Then, we find CFDs on each sample set. We then finally synthesize the entire CFD set by modeling all discovered CFDs as a weighted graph, and find the subset with the largest weights.

We suppose that the number of the groups and samples in each group are n and m, respectively. In Section 4.1, we first propose a multiple-pass scan algorithm through which we identify n groups of popular items iteratively. This algorithm is divided into two phrases: the first extraction and the second to n-th extractions where m denotes the number of items in each group, because the second to n-th extractions represent a process of iteration different from the first extraction. During the second to n-th extractions where n is the group number, we need to compare the samples with both current and original sampling results. However, scanning a dataset multiple times for big data is infeasible. In Section 4.2, we explain how to perform the iteration in once scan.

4.1 Multiple-pass scan algorithm

We start from the criteria for sample selection and then describe the algorithm in Section 4.1.1. The sample is divided into two parts to ensure effectiveness. The first group of m items is obtained primarily as the base, and the second to n-th groups are sampled iteratively until all types of popular items are sampled. We will discuss these two algorithms in Sections 4.1.2 and 4.1.3, respectively.

4.1.1 Tuple section criteria

First of all, we should avoid special and unpopular samples, which are misleading tuples such as t_9, t_{10}, and t_{11} in Example 1. A misleading tuple is a tuple with the following features:

(1) If a tuple has at least one incomplete attribute, such as t_9 and t_{10}, we treat it as a misleading tuple.

(2) If we compare the attributes of a tuple t with popular tuples and find that the number of the same attributes is smaller than a threshold ϵ, t is treated as a misleading tuple and is defined according to the method in Section 6.

Second, avoiding similar items is necessary to prevent over-fitting. To achieve this goal, we adopt the second to n-th iteration. In the i-th sample where $2 \leqslant i \leqslant n$, we compare it with the samples obtained from the first to $(i-1)$-th sample. If the number of the same attributes between the current item and early results is larger than a threshold, then this item is considered too similar for sampling results and given up.

4.1.2 Representative and Random Sampling for CFDs for the First group (FRRSC)

The first group is generated by the framework similar as reservoir sampling, which is suitable for sampling on the size-unknown data within once scan. The difference is that the replacement of sample considers the criteria in Section 4.1.1.

We first include the front m tuples in the sample S. For each of the following tuples t, we decide whether it is the misleading tuple. If t is incomplete, we do not add it to S directly. Otherwise, we use $1/q$ as the selection probability t, where q is the number of tuples in S with sharing more than ϵ attributes with t such that extremely unpopular tuples are selected in low probability.

The pseudo code of the algorithm is shown in Algorithm 1.

We firstly initialized S as the first m complete tuples (lines 1–7). For each tuple N_i, if it is complete and it shares more than ϵ attributes than some tuples in S (in lines 10–12), it replaces some tuples in S randomly line 14).

Example 2 We attempt to sample 7 popular items from the dataset shown in Example 1. We first pick $t_1 - t_7$ to S. Then, we compare t_8 with the samples in S. If we set ϵ as 2, then we can find that $\text{cmp}(S_i, t) \geqslant \epsilon$,

Algorithm 1 FRRSC

Input: N: the dataset, m: sample sizeB'
Output: The sample S

1: $k \leftarrow 0$
2: $i \leftarrow 0$
3: **while** (**do** $k < m$ and $i < |N|$)
4: **if** N_i is complete **then**
5: $S_k \leftarrow N_i$
6: $k \leftarrow k + 1$
7: $i \leftarrow i + 1$
8: **while** $i < |N|$ **do**
9: **if** $N[t]$ is complete **then**
10: **for** $j = 0$ to $m - 1$ **do**
11: **if** $\text{cmp}(N_i, S_j) \geqslant \epsilon$ **then**
12: $q \leftarrow q + 1$; $k \leftarrow$rand$[1, q]$;
13: **if** $k \leqslant m$ **then**
14: $S_j \leftarrow N_i$
15: break
16: $i \leftarrow i + 1$
17: return S

Note: $\text{cmp}(T[1, i], t)$ shows the number of the same attributes shared by two tuples.

because t_8[CC] $= t_3$[CC] and t_8[PN] $= t_3$[PN]. Thus, we generate a random number from 1 to 8. If we generate 2, then S_2=t_8 rather than t_2.

Thereafter, for t_9, we find that the item is incomplete and discard it. Thus, we check t_{10} without changing q. For t_{10}, we can find that it is also a misleading tuple, because it is incomplete.

We check t_{11} and find that it is complete. However, when we compare it with samples pointed by S, we find that no sample can have more than two similar attributes, which shows that it has the second feature of misleading tuples. Thus, it is also a misleading tuple. Having no further tuples to select, we obtain the samples: t_1, t_8, t_3, t_4, t_5, t_6, and t_7.

Theorem 1 shows the effectiveness of the proposed algorithm.

Theorem 1 The FRRSC can keep the probability of sampling for all popular tuples the same and avoid obtaining misleading tuples.

4.1.3 Representative and Random Sampling for CFDs for the left groups (TRRSC): Extraction of the second to n-th groups of items

By calculating the number of the same attributes of the tuple with samples in $T[1], T[2], \ldots, T[i-1]$, we ensure that the samples are popular (a sample exists with no less than b' same attributes) but different from the samples obtained in previous iterations (no sample has more than b same attributes) and establish a new sample set for it.

The pseudo code of the algorithm is shown in Algorithm 2. Such a function is invoked for $n - 1$ times to generate $T[2]$ to $T[n]$, and we know that some

Algorithm 2 TRRSC

Input: N is big dataset. m samples in each group. b and b' set by us due to the data type and demand of user as standards of similarity. The samples $T[1]$ to $T[i-1]$ and each set is with m indexes from $T[a, 1]$ to $T[a, m](1 \leqslant a \leqslant i - 1)$.
Output: The group of indexes from $T[1]$ to $T[n]$.

1: $p = i \times m$;
2: number $i \times m + 1$ to N tuples from 1 to $N - i \times m$;
3: $t = 1$; $q = 1$;
4: **while** there exists $(t + 1)^{\text{th}}$ item in N **do**
5: $t = t + 1$;
6: **if** $N[t]$ is complete **then**
7: **for** $i = 1$ to $\min(q, m)$ **do**
8: **if** $\text{cmp}(T[1, i], t) \geqslant b$ **then**
9: **for** $j = 1$ to $i - 1$ **do**
10: **for** $k = 1$ to m **do**
11: **if** $(\text{cmp}(T[j, k], t) \geqslant b')$ **and** $(\text{cmp}(T[j, k], t) \leqslant b$ (**for** all $1 \leqslant j \leqslant i - 1$, $1 \leqslant k \leqslant m$)) **then**
12: $q = q + 1$; $k =$rand$[1, q]$;
13: **if** $k \leqslant m$ **then** $T[i, k]$ point to $N[t]$;
14: **if** $q \geqslant m$ **then** we start the next iteration;
15: **else**
16: $n = i - 1$
17: output $T[1]$ to $T[n]$ as sampling result.

Note: $\text{cmp}(T[1, i], t)$ shows the number of the same attributes when $i = 1$; rand$[1, q]$ is 1 when $q = 1$ which guarantees the 1^{th} tuple be added as what we want.

new tuples have not been found. Then, we perform the $(i + 1)$-th iteration to find the new type of $T[n]$.

When we select the i-th ($i \leqslant n$) group of samples, we set $i \times m + 1$ as the starting number firstly (lines 1 and 2) .The reason is that as we select at least m items each time, no popular items are found from 1 to $i \times m$ in the i-th sampling.

We then obtain samples from $i \times m + 1$ to N. We re-number the $i \times m + 1$ to N tuples as items from 1 to $N - i \times m$. We also set two variables t and q to show the number of tuples to deal with and the number of new types of found popular tuples, respectively (line 3). To generate $T[i, 1]$, we check tuples from the first one to validate whether they can meet our new criteria.

The new criteria is to compare the t-th tuple with the samples in $T[1]$ to $T[i-1]$ (line 9). If a sample has more than b' same attributes with t, and no sample shares more than b attributes with t, we set $k = 1$ and add this tuple as the first one (line 11). We check it to prevent samples from being too similar to make the CFDs strict. b is a high limit ensuring that the chosen tuple is not similar to the selected samples, and b' is a lower bound to prevent selected samples from being too special to make CFD useless.

Then, we continue to add new tuples. Instead, as for the comparison with samples in $T[1]$ to $T[i - 1]$, each attribute in the t-th tuple is compared with each sample

in $T[i]$ (line 7). If at least a sample in $T[i]$ shares more than b attributes with t-th (line 8), we compare it with samples in $T[1]$ to $T[i-1]$.

Therefore, we increase q by 1 and generate a k in $[1, q]$ (line 12). We compare k with m to decide whether to replace the sample in $T[i]$ in the same manner as FRRSC (line 13). If no sample in $T[i]$ has at least b attributes that are the same as t, some attributes of t are blank, or no sample exists in $T[1]$ to $T[i-1]$. Following our new criteria, we treat it as a new tuple (line 14).

Finally, when no new tuple is left, if we find the number of popular tuples similar to samples in $T[i]$ represented by $q > m$, we know that some new tuples have not been found. Then, we perform the $(i+1)$-th iteration to find the new kind of tuple (line 18). When we find $q \leqslant m$, we know that almost all kinds of popular tuples have been found (lines 20 and 21).

We use an example to demonstrate the process of the algorithm.

Example 3 If we have found a sampling set of $T[1] = t_1, t_2, t_3, t_4$ and want to find the second sampling set, we start from the 5-th element of t. We compare t_5 with samples in $T[1]$.

If we set $b' = 2$ and $b = 3$, we find that $t_5[\text{STR}] = t_3[\text{STR}]$ and $t_5[\text{ZIP}] = t_3[\text{ZIP}]$.

As no samples in $T[1]$ share 3 attributes with t_5, we add t_5 to $T[2]$ as the first sample.

We find that t_6 shares more than 3 attributes with t_5. Then, we compare t_6 with the samples in $T[1]$ and find that t_3 shares 2 attributes with t_6, but no sample shares 3 attributes with it. Thus, we add t_6 to $T[2]$. Then, we can find 3 attributes in t_7 the same as those in t_6. Meanwhile, t_1 in $T[1]$ has two attributes the same as t_7, and no item in $T[1]$ shares 3 attributes with t_7. Then we add t_7 to $T[2]$. Since we find that no item in $T[2]$ has 3 attributes the same as t_8, we give it up and turn to t_9. Then, we find t_9 and t_{10} are incomplete, and t_{11} is special.

Therefore, $T[2] = t_5, t_6, t_7$, which is extremely small. Therefore, we quit $T[2]$ and return $T[1]$ as the sampling result. Effectiveness analysis Theorem 2 shows the effectiveness of proposed algorithm.

Theorem 2 For popular items similar to the sampling set $T[i]$, we ensure that their probability is sampled the same in the i-th sampling and avoid sampling misleading items in TRRSC.

Time complexity analysis. To the process of second to n-th times of sampling, we know that for the i-th sample, we need to compare each item with items in $T[1], T[2], \ldots, T[i-1]$. Therefore, we need to compare for $(i-1) \times m$ times. The total times are $(i-1) \times m \times N \times r$ for the i-th extraction. Therefore, the total times of comparing are as follows: For the Input/Output (I/O) process, the datas are scanned for once. Thus, the time complexity is $O(n)$.

4.2 One-pass sampling algorithm

Algorithm overview. To handle a big dataset, we design an algorithm to compass all iterations in once scan. Initially, we make m indexes in $T[1]$ pointing to the first m tuples and establish an array q. Each element $q[i]$ is the number of the tuples similar to $T[i]$.

Then, we compare each new scanned tuple with samples. If a sample in $T[i]$ has more than b same attributes with it, we add $q[i]$ by 1 and add it into $T[i]$ if $q[i] < m$. When $q[i] \geqslant m$, we generate a random number k in $(1, q[i])$. If k is no larger than m, we add it as the k-th sample in $T[i]$. Otherwise, we abandon it.

If no sample has more than b same attributes with the tuple, we check whether a sample has no less than b' same attributes with it, because it may be special. If such a sample exists, then we know that it is not special and put it into $T[i+1]$. Otherwise, it is abandoned.

When sampling from real big data, we observed that the possibility of the popular tuple being sampled is extremely small. If we firstly generate a random number k and compare attributes only when k is no larger than $n \times m$, then we will reduce the comparison times. As the cost, some tuples will be lost when counting items are similar to $T[a]$. The reason is that even though a new tuple is similar to $T[a]$, we do not know whether it is similar without comparing it with tuples in sample sets when $k > m$. This condition leads to the wrong deletion of $T[a]$, because the amount of its similar tuples is smaller than m. For big data, $T[a]$ always has more than m similar tuples. Therefore, after all reservoirs are full ($\min(q[a]) \geqslant m$ $(0 < a \leqslant n)$), we generate a random number before comparing new tuples with other samples.

Algorithm description. The pseudo code is shown in Algorithm 3. We first set the i-th entry in $T[1]$ as a pointer to the i-th item, initialize a variable t and an array $q[n]$ (lines 2–4). $q[i]$ is the number of tuples similar to those samples in $T[i]$. t is increased by 1 and when there exists a reservoir that is not full ($\min(q[a]) < m$ $(0 < a \leqslant n)$), we compare each attribute in $N[t]$ with samples in $T[1], \ldots, T[i]$ (lines 9 and 10).

If at least one sample in $T[a]$ has more than b attributes with the same amount as the attributes of $N[t]$

Algorithm 3 BRRSC

Input: Dataset N, m samples in each group. b and b' are set by us due to data type and user demand as standards of similarity.
Output: The groups of indexes $T[1]$ to $T[n]$.

```
1: for w = 1 to m do
2:     T[1, w] point to N[w];
3: t = m; label= 0;
4: q[1] = m;
5: while there is N[t + 1] do
6:     t = t + 1;
7:     if min_{0<a≤n}(q[a]) < m then
8:         if N[t] is complete then
9:             for w = 1 to i do
10:                for j = 1 to min(m, q[w]) do
11:                    if cmp(T[w, j], t) ⩾ b then
12:                        if q[b] > m then
13:                            q[b] = q[b] + 1; k =rand[1, q[b]];
14:                            if k ⩽ m then
15:                                T[b, k] point to N[t]
16:                        else
17:                            q[b] = q[b] + 1; T[b, q[b]] point
   to N[t]
18:                    else if cmp(T[b, j], t) ⩾ b′ then
19:                        label=1
20:            if label==1 then
21:                T[i, 1] point to N[t]
22:    else if min_{0<a≤n}(q[a]) ⩾ m then
23:        k =rand[1, t]
24:        if k ⩽ m × n then
25:            if N[t] is complete then
26:                for w = 1 to i do
27:                    for j = 1 to min(m, q[w]) do
28:                        if cmp(T[w, j], t) ⩾ b then
29:                            q[b] = q[b] + 1;
30:                            T[b, k%m] point to N[t]
31:                        else if cmp(T[b, j], t) ⩾ b′ then
32:                            label=1;
33:                if label==1 then
34:                    T[i, 1] point to N[t];
35:        output T[1], T[2], . . . , T[n];
```

(line 11), then we increase $q[a]$ by 1 and generate a random integer k in $[1, q[a]]$ when $q[a] \geqslant m$ (lines 12 and 13). When $k \leqslant m$, we replace sample $T[a, k]$ with $N[t]$ (lines 14 and 15). When $k > m$, we find a new tuple. When $q[a] < m$, we add t as $T[a, q[a] + 1]$ directly (line 17).

When we compare $N[t]$ with samples in $T[1]$, $T[2], \ldots, T[i]$, we also check whether an item has more than b' attributes similar to $N[t]$ and set the label as 1 to show that such an item exists. If no item has more than b same attributes with $N[t]$, then we check whether the label is 1. If the label is 1, showing that a sample has more than b' same attributes with t, we build a new group $T[i + 1]$ and denote it as $T[i + 1, 1]$ (lines 21 and 22).

When all reservoirs are full ($\min(q[a]) \geqslant m$ ($0 < a \leqslant n$)), we generate a random integer k in $[1, t]$, and compare each attribute in $N[t]$ with samples in $T[1], T[2], \ldots, T[i]$ (lines 29 and 30) only when $k \leqslant n \times m$ (line 27). We use $n \times m$ rather than m as the high limit for n sample sets. Then, if at least one sample in $T[a]$ has more than b attributes with the same amount as the attributes of $N[t]$, we increase $q[a]$ by 1 and replace the sample $T[a, k\%m]$ with $N[t]$ (line 33). During comparison, we also let label equal to 1 to show that such an item has more than b' attributes the same as $N[t]$. We build a new group $T[i + 1]$ and denote it as $T[i + 1, 1]$ (line 37). Therefore, we synthesize the two phases in FRRSC and TRRSC in once scan. Finally, the results are $T[1], T[2], \ldots,$ and $T[n]$ (line 41).

Example 4 We compare t_6 with $T[1]$ and find that no sample in $T[1]$ has more than 3 attributes the same with it. However, when comparing it with $T[2]$, we find that it has 5 attributes the same with $T[2, 1]$, which is t_5 actually. Then, as $q[2] = 1 < 3$, we insert t_6 directly to $T[2, 2]$.

When we come to t_7, it is compared with $T[1]$, and the result is the same as t_5 and t_6. However, when we compare it with $T[2]$, we find that it has 3 attributes the same as t_6. Meanwhile, as $q[2] = 2 < 3$, we add t_7 to $T[2, 3]$ directly.

For t_8, we find that no sample in $T[1]$ and $T[2]$ has more than 3 attributes the same with them. However, it has 2 attributes the same with t_3 tuple. Therefore, we add it to $T[3, 1]$.

For t_9 and t_S, we can find that both of these two items are incomplete, and we abandon them directly. Thereafter, we find that no item in $T[1]$, $T[2]$, and $T[3]$ has more than 2 attributes the same as t_{11}'s attributes. Finally, by checking $T[1]$, $T[2]$ and $T[3]$, we find that S, $q[1] = 4 \geqslant 3$, $q[2] = 3 \geqslant 3$, and $q[3] = 1 < 3$.

Thus, we abandon $T[3]$, and leave $T[1] = \{t_2, t_3, t_4\}$ and $T[2] = \{t_5, t_6, t_7\}$ as sampling results. $n = 2$ is the number of groups.

Effectiveness analysis. Theorem 3 shows the effectiveness of the proposed algorithm.

Theorem 3 For the popular complete tuples in a big dataset which is all similar to the same $T[i]$ sampling set, the probability of extraction remains the same in BRRSC. The misleading tuples cannot be sampled in BRRSC.

Time complexity analysis. Different from the second to n-th extraction, a comparison is not needed for all the sampled items to ensure that the item is new. We can add it to its similar $T[i]$ directly. When $\min(q[a]) < m$, as the average times is $n/2$ compared with the sampling items, the time complexity $f_1(|N|) = r \times m \times (n/2) \times |N_f| = O(c)$.

N_f is a small part of N, which can make each sample set $T[i]$ have more than m items. When $\min(q[a]) \geqslant m$, we firstly generate k in $[1, t]$ before we compare the attributes of the item. The probability that we compare attributes is $p_1 = (m \times n)/t$. Therefore, for N_b which shows a large part of N except N_f, the time complexity $f_2(|N|)$ is

$$f_2(N) = \Big(\frac{1}{|N_f|} + \frac{1}{|N_f|+1} + \frac{1}{|N_f|+2} + \frac{1}{|N_f|+3} + \cdots + \frac{1}{|N|}\Big) \times m \times n \times r \times (n/2) = O(\ln(|N|)).$$

The complexity of time $f(|N|)$ for the entire process is

$$f(|N|) = f_1(|N|) + f_2(|N|) = (\ln(|N|) - a + |N_f|) \times m \times n \times r \times (n/2) = O(\ln(|N|).$$

This shows that the complexity of sampling is $O(\ln(|N|))$, which is sub-linear to the dataset.

5 CFD Discovery for Big Data (BDC)

After sampling, we need to find rules on n small datasets. For the discovery, we still have following problems to solve:

(1) Although we use the RRSC, some special or dirty samples may remain. The CFD discovery algorithm should be fault-tolerant.

(2) As variable types of big data exist, we have to make our method fit different conditions. We also have to ensure that the CFD set is complete. Therefore, our method should discover both constant and variable CFDs, and be able to tolerate faults. Such an algorithm is introduced in Section 5.1.

(3) Owing to errors in the training set, we can find conflicts in CFDs produced by an algorithm. To resolve the conflicts, we establish a graph-based method to find correct CFDs by finding disconnected subsets with the largest weights in Section 5.2.

5.1 DFCFD algorithm

DFCFD is designed to find CFDs from the results of sampling. We improve three CFD discovery algorithms TANE for CFDs (CTANE), FastCFD, and CFDMiner[7] to Big data TANE for CFDs (BCTANE), Big data Fast CFD (BFCFD), and Big data CFD Minder (BCFDM) by accepting some CFDs with limited confidence to tolerate fault. We find that different algorithms have preference for specific big data. Therefore, we ensemble different algorithms. During ensemble, we utilize the same process of different methods.

The entire work of the DFCFD algorithm is shown in Fig. 2. We have two choices of algorithm combinations, which are introduced in the following.

BCTANE. To improve CTANE, we use a threshold e to decide whether to accept a CFD. For each CFD, we set a variable $u' = |T|$ ($T \subseteq r$, and a CFD is absolutely correct for items in T), where $|T|$ denotes the number of the samples in T which is a set of samples, and r is a sample set for CFD discovery. Then we obtain a new variable $u = u'/|T'|$ ($T' \subseteq r$, which conforms to the left side (premise) of CFD). We improve CTANE by adding the following two steps:

(1) When we cut a limb, we change the rule if $u_{\text{CFD}} \leqslant e$, and then we cut the limb.

(2) When we calculate the supporters for a CFD, we think that items with the same LHS can support CFD when RHS is empty or wrong (which means that similarity$> e$).

BFCFD. To develop FastCFD, we change its procedure FindMin to adapt to datasets with special or dirty ones. When FindMin determines whether a constant t_a makes constant CFD $(X \to A, (t_P \parallel t_a))$ valid, we check whether there is no $X' \subseteq X$ in size $|X| - 1$ making CFD $(X' \to A, (t_P[X'] \parallel t_a))$ valid in FastCFD. However, for big data, many samples may be incomplete or contain errors. Thus we make the BFCFD allow some different items to make CFD $(X' \to A, (t_P[X'] \parallel t_a))$ valid, when following constraint is satisfied: $u' = |T|(T \subseteq r; \text{CFD}(X' - > A,$

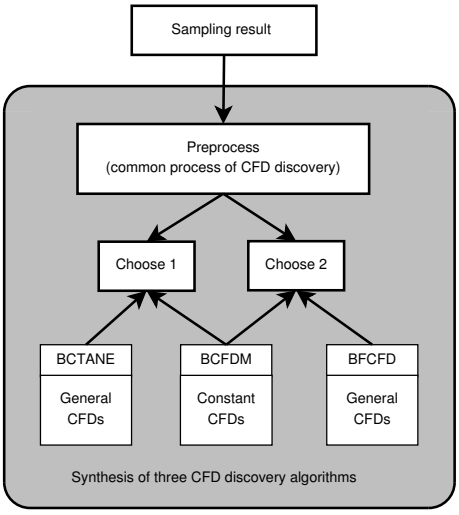


Fig. 2 Overview of DFCFD algorithm.

$(t_P[X']||t_a))$ is right for items in T); $u = u'/|T'|(T' \subseteq r$ and it conforms to $t_P[X'])$.

For the constant CFDs, when $u > e$, we say that CFD $(X' \to A, (t_P[X'] \parallel t_a))$ is valid and acceptable.

Then, in FindMin, to find variable CFDs from big data, we use a threshold of error e to tolerate the wrong samples. We revise the constraints as follows:

(1) If the number of $X' \subseteq X$ in size $|X| - 1$ making $Y \cup (X \setminus X')$ cover $D_A^m(r_{t_P[X']})$ is less than $e \times I$.

(2) If number of $Y' \subseteq Y$ of size $|Y| - 1$ making Y' cover $D_A^m(r_{t_P[X]})$ is less than $e \times i'$.

If (1) and (2) are both satisfied, then the variable CFD is accepted.

BCFDM. We change the CFDMiner in a manner similar to the aforementioned two improvements. In the third step of CFDMiner, we check the free item set (Y, s_p) in list L with the following constraints (the number of attributes in Y is shown by i).

For each subset $Y' \not\subset Y$ such that $(Y', s_p[Y']) \not\subset L$, we replace RHS$(Y, s_p)$ with RHS$(Y', s_p[Y'])$. However, the RHS$(Y', s_p[Y'])$ cannot lead to a left-reduced constant CFD.

For big data, we can ignore these wrong tuples and the constraint is modified as follows:

If the number of the subsets Y' satisfying $Y' \not\subset Y$ and RHS$(Y', s_p[Y'])$ is smaller than e, then a left-reduced constant CFD is less than $e \times i$. When we compare the items to the wrong item, if the similarity of similar items and the wrong one is larger than e, then we gather the similar ones with the wrong one, find there is no left-reduced constant CFD. If the above condition is met, then we can accept (Y, s_p).

Integration of three algorithms. To synthesize these three algorithms, we should merge the same or similar processes of these three methods to accelerate the entire process by preprocessing. According to Ref. [1] and similar to our improved algorithms, all of these three original algorithms need to know the supporters of different attribute sets. Therefore, we firstly generate the number of supporters for different attributes and place them in a hash table. Then, using the hash table, we can reduce repeat calculations in the process of finding CFDs by three algorithms.

To select the algorithms, we need to consider their different preferences. As we have not changed much about the three algorithms, the function of the improved algorithms is similar to that of the original ones. Then, according to Ref. [1], we can find that CTANE cannot run to completion when arity is above 17, and it can be sensitive to support threshold and outperform FastCFD when the dataset is large with small arity. However, FastCFD can outperform CTANE when arity is larger than 17 and can do well for small datasets with few attributes. Furthermore, CFDMiner can always outperform the other two by three orders of magnitude making us ignore its efficiency. Therefore, we select BCTANE and BCFDM when arity is smaller than 17 and items are more than a million. When arity is larger than 17, we utilize BFCFD and BCFDM together.

5.2 Dealing with conflicts between CFDs

With dirty data in the training set, the discovered CFDs may involve conflicts. As we premise that the large part of the dataset is clean, we attempt to find a maximum compatible rule subset. Thus, we model the CFD set as a weighted undirected graph including CFDs as nodes. We add a line between two nodes when a conflict occurs between two CFDs. The weight of each node represents number of its supporters. Then, the problem of finding a maximum compatible rule subset is converted to finding a maximal weight independent set of nodes from the graph. To solve this problem, we develop linking rules and the Maximal Weight Independent Discovery (MWID) algorithm. In this section, we first introduce how to obtain the weight of each node (Section 5.2.1), and then we represent the conflicts between CFDs by linking rules (Section 5.2.2). Finally, we use the MWID algorithm to find a maximal weight independent set (Section 5.2.3).

5.2.1 Calculating the weight of each node

We use the number of supporters of a CFD as weight of each node in WCFD. The WCFD is a weighted undirected graph for CFDs. For constant CFDs, such a number could be computed by Structured Query Language (SQL), but the process is more difficult for variable CFDs.

Thus, we propose a new method to calculate the supporters of variable CFDs. We first build a rank of the number $(r_1, r_2), (r_2, r_3), (r_3, r_4), \ldots$ for the samples with n samples in them. We should note that the ranker has a large distance in the back. For the half of n, we think the supporters are large enough to ignore the difference between them. Thus, we can set the last rank as $(n/2, n)$.

With the rank, we can set the threshold k instead of e in finding CFDs by FastCFD or CTANE as $r_1, r_2, r_3, \ldots$. If a CFD exists in the CFD set for $k = r_i$ and does not exist in the CFD set for $k = r_{i+1}$, then we can set the amount of supporters for the CFD as int$[(r_i + r_{i+1})/2]$. However, if the CFD reaches the

final rank, then we use 80% of n as its supporters.

5.2.2 Discovery of the conflict between two CFDs

When we decide whether conflict exists between two CFDs, we design a deciding-linking rule. Through such rule, we can decide whether to set a line between two CFD nodes to show conflict between them. We discuss linking rules in two cases with two CFDs and multiple CFDs.

For two CFDs, C_1: $(X_1 \to A_1, (t_P[X_1] \parallel t_1))$; C_2: $(X_2 \to A_2, (t_P[X_2]||t_2))$. We firstly decide whether conflict exists between C_1 and C_2. We can divide the problems into three situations according to the relationship between X_1 and X_2. Without generality, we suppose $|X_1| \leqslant |X_2|$.

T1. $X_1 \subset X_2$. Only if A_1 is the same as A_2, can conflict occur.

T1-1. If C_1 and C_2 are both constant CFD, then only when $t_p[X_1C_1] = t_p[X_1C_2]$ but the $t_p[A_1C_1] \neq t_p[A_2C_2]$, is there a conflict between them. Here $t_p[X_1C_1]$ and $t_p[X_1C_2]$ mean the range of the attribute set X_1 in C_1 and C_2 which are the same for other attributes, e.g., C_1: $(F, G \to A, (1, 2 \parallel 1))$ and C_2: $(F, G, H \to A, (1, 2, 3 \parallel 3))$.

T1-2. If C_1 and C_2 are both variable CFDs, then when "__" is for different attribute, there can be conflict. There must be at least one attribute r_i in X_1 that is a variable attribute with "__" for its range and a constant data for r_i in X_2 to create a conflict, e.g., C_1: $(F, G \to A, (\_\_, 2 \parallel \_\_))$ and C_2: $(F, G, H \to A, (1, 2, \_\_ \parallel \_\_))$. We know that for C_1, when F is 1, A is a constant. However, from C_2, we know that when $F = 1$ and H is changed, A is changed with H.

T1-3. If C_1 is a variable and C_2 is a constant, conflict cannot exist between two CFDs, because when $X1 \subset X$ and C_1 is variable, the C_2 can be a kind of situation of it.

T1-4. If C_1 is a constant and C_2 is a variable, when $t_p[X_1C_1] = t_p[X_1C_2]$, but in X_2 a variable attribute exists that is not in X_1. Thus, when $A_1 = A_2$, A_2 is more general than A_1. Then, a conflict occurs, e.g., rules C_1: $(F, G \to A, (1, 2 \parallel 2))$ and C_2: $(F, G, H \to A, (1, 2, \_\_ \parallel \_\_))$. We know that when $F = 1$ and $G = 2$, A in C_1 should be a constant. However, it is a variable with different H. Then C_1 and C_2 are in conflict.

T2. $X_1 = X_2$. Only if A_1 is the same as A_2, can there be conflict.

T2-1. If C_1 and C_2 are both constant CFD, then only $t_p[X_1C_1] = t_p[X_2C_2]$ but $t_p[A_1C_1] \neq t_p[A_2C_2]$ can imply a conflict, e.g., C_1: $(F, G \to A, (1, 2 \parallel 1))$ and C_2: $(F, G \to A, (1, 2 \parallel 3))$.

T2-2. If C_1 and C_2 are both variable CFDs, then when "__" is for different attributes, there can be conflict, e.g., C_1: $(F, G \to A, (\_\_, 2 \parallel \_\_))$ and C_2: $(F, G \to A, (1, \_\_ \parallel \_\_))$. For C_1, when F is 1, A is a constant. However, from C_2, when $F = 1$ and G is different, the A can change.

T2-3. If C_1 is variable and C_2 is constant, it cannot generate conflict for C_2 that can be treated as a special situation for C_1.

T3. $X_1 \subset X_2$. In this case, conflict occurs only when A_1 is the same as A_2. If $X_1 \cap X_2 = \varnothing$, comparing these CFDs is unnecessary. Thus, $X_1 \cap X_2 = \varnothing$ should be satisfied to find a conflict. We suppose that $X_1 \cap X_2 = E$, where E is the attribute set shared by X_1 and X_2.

T3-1. If the C_1 and C_2 are both constant CFDs, conflict cannot exist between the two CFDs. As they cannot include the situation of the other, conflict cannot occur.

T3-2. If C_1 and C_2 are both variable CFDs, then when "__" is the range for all the attributes in one CFD and in another CFD, both the attributes inside and outside E have fixed ranges, e.g., C_1: $(F, G, H \to A, (\_\_, \_\_, \_\_ \parallel \_\_))$ and C_2: $(F, L, Q \to A, (1, 2, \_\_ \parallel \_\_))$. For C_1 when $F = 1, G = 2$, and $H = 8$, A is a constant. From C_2, we can know that when $F = 1, G = 2$, and $H = 8$ but $Q \neq H$. Thus, A in C_2 is different.

T3-3. If one CFD is a variable and another CFD is a constant, conflict cannot exist between them because constant CFD can be seen as a special case for the other CFD when $X_1 \not\subset X_2$.

When we find conflict among more than two CFDs, we can integrate the conditions of generating conflict into a rule M1. The only condition generating conflict is that for a variable CFD, no less than two constant CFDs show that it is wrong. We suppose that three CFDs exist, which contain a variable CFD and two constant CFDs.

$$C_1 : (X_1 \to A_1, (t_P[X_1] \parallel t_1)),$$
$$C_2 : (X_2 \to A_2, (t_P[X_2] \parallel t_2)),$$
$$C_3 : (X_3 \to A_3, (t_P[X_3] \parallel t_3)).$$

M1. If there is conflict among them, A_1, A_2, and A_3 must be the same attribute. At least one attribute is shared by X_1, X_2, and X_3. We denote such attribute set by U. Meanwhile, in one CFD, the range of U is "__" which means variable, and A in this CFD is also a variable. However, in other CFDs, U and A are both constants. Then we suppose that C_1 is a variable while C_2 and C_3 are both constant. We find that we

can synthesize different conditions: $\{X_1 \not\subset X_2, X_1 \subset X_3 X_1 = X_2\}$ and all other conditions in one rule. Let $E = X_1 \cap X_2 \cap X_3$, then if one attribute in E is"__" for C_1 and it is the same constant data for C_2 and C_3. To the other attributes in E, the range of them is the same for three CFDs. Then, if A in C_2 is different from C_3, a conflict occurs.

According to Rule M1, in the case that $X_1 \not\subset X_2 \not\subset X_3$, consider three rules C_1: $(F, G, H \to A, (\_\_, 1, 2 \parallel \_\_))$, C_2: $(F, G, Q \to A, (1, 1, 4 \parallel 1))$, and C_3: $(F, G, W \to A, (1, 1, 4 \parallel 2))$. We can discover from C_1 that when $G = 1$ and $H = 2$, F can decide A. However, in C_2 and C_3, we discover that when $G = 1$ and $H = 2$, F cannot decide A. In the case that $X_1 \subset X_2 = X_3$, consider rules C_1: $(F, G \to A, (\_\_, 2 \parallel \_\_))$, C_2: $(F, G, Q \to A, (1, 1, 4 \parallel 1))$, and C_3: $(F, G, W \to A, (1, 1, 4 \parallel 2))$. We can discover from the preceding discussions that F cannot decide A when $G = 2$ by itself. A conflict occurs between them.

For all the different relationships among X_1, X_2, and X_3, we can see that the Rule M1 can work for all the conditions. If we want to see the conflict among more than two CFDs, we can only determine the conflict when one CFD is variable and the others are constant. However, the constant CFDs of the others cannot show the variable CFD. Therefore, no matter what kind of relationship among X_1, X_2, and X_3, we can always check the conflict by M1.

For the conflict between two CFDs C_1 and C_2, we can just build a line between them as in Fig. 3. However, for more than two CFD nodes, we need to put constant CFDs together as a new node and leave the variable CFD alone. The weight of a combined node N_{sum} in Fig. 4 is $K_{\text{sum}} = K_1 + K_2 + K_3 + K_4 + \cdots + K_n$. Other CFDs having conflict with $C_1, C_2, \ldots, C_n$ also have conflict with the N_{sum} in Fig. 4.

5.2.3 MWID

As the premise for our method of finding CFDs from big dirty data is that the large part of the dataset is clean, we attempt to find a maximum compatible rule subset. Then, with the maximum subset, we can cover the largest number of tuples in the big dataset. As the maximal independent discovery problem, an NP-hard problem[8] is a special case with this problem with the weight of each vertex as 1. The Maximal Weight Independent Set (MWIS) discovery problem is also an NP-hard problem.

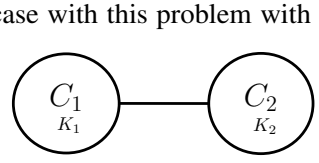


Fig. 3 Build a line between C_1 and C_2 when there is conflict.

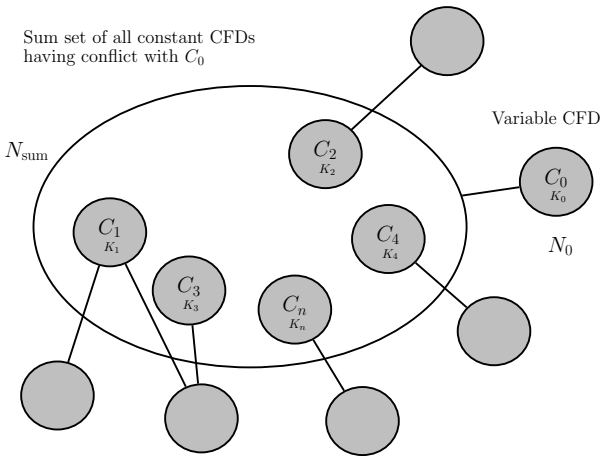


Fig. 4 Put constant CFDs together and leave the variable CFD alone.

To find the MWIS from an undirected graph, we design an algorithm MWID by improving algorithm FastMIS in Ref. [3]. FastMIS introduces a randomized algorithm to find Maximal Independent Set (MIS). It computes an MIS in a distributed manner. However, the computed MIS contains the largest number of nodes and does not consider the weight. Therefore, we modify some steps in the FastMIS to generate the MWIS. In FastMIS, MIS is obtained in three steps. The first two steps are as follows:

(1) Each node v chooses a random value $r(v) \in [0, 1]$ and sends it to its neighbors.

(2) If $r(v) < r(w)$ for all neighbors $w \in N(v)$, then node v enters MIS and informs its neighbors.

The two steps ensure that if a node v joins the MIS, then v's neighbors do not join MIS at the same time. Through this method, the node with the globally smallest value will always join the MIS to find the maximal independent set, which has been proved in Ref. [3]. When considering the weight, we need to ensure that the nodes with larger weight have more possibility to join the MIS. Thus, we cannot let the range to select a random value keeping the same. The modified algorithm is as follows:

(1) Compare the weight w_v of each node v with each weight w_n of its neighbors. If $w_v > w_n$, then we generate a random value $r(v) \in [0, 0.5)$ and give it to this neighbor. If $w_v < w_n$, then we generate a random value $r(v) \in (0.5, 1]$ for its neighbor. When $w_v = w_n$, we set $r(v) = 0.5$ and send it to this neighbor.

(2) If $r(v) \times w_v > r(w) \times w_n$ for all neighbors $w \in N(v)$, node v enters MIS and informs its neighbors.

In this manner, we can make the node with larger weight and fewer neighbors be added more easily to obtain MWIS.

Algorithm description. The pseudo code is shown in Algorithm 4. The algorithm runs multiple rounds, each of which corresponds to a phase. We introduce a single phase with pseudo code. The input is an adjacent matrix A of the WCFD graph. Then we set the variable scale as the number of nodes in the graph. With the scale, we obtain an array M[scale] to record found MWIS (line 1). In a single phase, for each node v, we compare the weight of v with the weight of each of its neighbors. If the weight of v is larger, then we generate a random number k from $[0, 0.5)$ and give it to w (lines 5 and 6). If the weight of v is equal to w, then we give 0.5 to w (lines 7 and 8). If the weight of v is smaller, we generate k from $(0.5, 1]$ and assign it to w (lines 9 and 10).

After we generate random numbers, for each node v, we set a label as 1 (line 12). We compare the random number of the neighbor of v with $r(v)$. If the $r(w) \times w_n$ is no smaller than $r(v) \times w_v$, then we let the label be 0 (line 15). After we finish the comparison, if label is still 1, we add v to M[scale], and move v and all edges adjacent to v (lines 17 and 18). Then, we start another phase when a node is found in G (lines 19 and 20).

Time complexity analysis. As the modified algorithm only adds the process of comparing weight, we can use the constraints provided in Ref. [3] to help analyze the time complexity. The probability in a single phase that at least a quarter of all edges are removed is at least $1/3$. Then with less than $1/3$ for the probability, many (potentially all) edges are removed. The probability that less than $1/4$ of edges are removed is more than $2/3$. Therefore, the removed edges are approximately $1/3 \times 1 + 2/3 \times 1/4 = 1/2$.

As at least $1/3$ of phases are "good" and can remove at least a quarter of edges, we need $\log 4/3(m)$ good phases, where the m is the number of the edges in G. The last two edges will certainly be removed in the next phase. We consider the extra time of comparing for each node, and we obtain the $(3 \log 4/3(m) + 1) \times c \in O(\log n)$ as time complexity, where c is a number no larger than the number of nodes in G.

Algorithm 4 MWID

Input: A graph G. The **scale** for the number of points in the graph.
Output: The maximal weight independent set M[scale].

```
1:  M[scale] = {};
2:  for v = 1 to scale do
3:      for each neighbor w of v do
4:          switch cmp(v, w)
5:              case 1: k =rand[0, 0.5];
6:                      r(w) = k;
7:              case 2: k = 0.5;
8:                      r(w) = k;
9:              case 3: k =rand[0.5, 1];
10:                     r(w) = k;
11: for v = 1 to scale do
12:     label= 1;
13:     for each neighbor w of v do
14:         if r(v) × w_v ⩽ r(w) × w_n then
15:             label= 0;
16:     if label= 1 then
17:         add v to M[scale];
18:         remove v and all edges adjacent to v from G;
19: if there is node in the G then
20:     go to 2;
```

Note: cmp(v, w) aims to compare the weight of v with that of w. If the weight of v is larger, it returns 1. If the weight of v equals that of w, it returns 2. For the condition in which v is smaller, we obtain 3.

6 Parameter Selection

In CFD discovery algorithms, the following parameters should be known:

(1) High limit of the number of groups extracted from dataset (n).

(2) Number of the items in each group (m).

(3) Least number of the same attributes to decide whether a tuple is similar to others (b).

(4) Highest number of same attributes that a special item has with popular items (b').

(5) Threshold we set when finding CFDs (e).

In this section, we discuss the parameter selection methods based on user requirements. The requirements include four dimensions that have tradeoffs. The four dimensions of CFD discovery methods are as follows:

(1) Time of finding CFDs (CW). We always want less time for CFDs discovery. The time of our algorithm is the sum time of sampling and discovering the CFDs from samples.

(2) Quality of CFDs (QC). We want to improve the quality of CFDs to make it fit the CFDs found on the clean dataset. This dimension is described by the percentage of CFDs from the clean dataset covered by those found in the dirty set.

(3) Time of cleaning data with our CFDs (CC). Another target of CFDs discovery is to clean data efficiently. We measure the time by cleaning data with the CFD set.

(4) Quality of cleaning (denoted by QD). Meanwhile, we need to ensure that our CFDs clean the data effectively. We use the percentage of dirty items in the dataset found by the CFD set to measure.

For these parameters, a user could select a dimension as the one with the highest priority. We denote such dimension as OD. For others, the tolerate range is set. As an example, a possible demand description is as follows:

(1) We want the discovery time to be as small as possible.

(2) The lowest quality of CFDs we allow is 96%.

(3) The longest allowed time of using CFDs to clean our dataset is 3 hours.

(4) The lowest quality of the cleaning result is 95%.

We designed experimental methods to obtain these parameters according to these requirements. The data for this experiment is generated by the TPC-H. We generate a small tuple set with the same amount of attributes as those tuples in big data to be cleaned, which can make our data similar to big data and ensure that the functions found from our data can work effectively with the big data.

In each experiment, we vary one parameter p_1 with the others unchanged, use our method to find CFDs and clean the dataset by the discovered CFDs. Then we measure the amount for four aims. With several rounds of experiments, we draw a curve about the four goals and p_1. By fitting such a curve, we can determine four functions between CW, QC, CC, QD, and the parameter p_1.

With the same process, we obtain the functions between CW, QC, CC, QD, and other parameters. We denote the relation function between p_i and CW as $f_{\mathrm{CW}}(p_i)$ which is similar to QC, CC, and QD.

Finally, we integrate all the functions to obtain equations:

$$\mathrm{CW}=\sum_{i=1}^{r} f_{\mathrm{CW}}(p_i)/r,\ \mathrm{QC}=\sum_{i=1}^{r} f_{\mathrm{QC}}(p_i)/r,$$

$$\mathrm{CC}=\sum_{i=1}^{r} f_{\mathrm{CC}}(p_i)/r,\ \mathrm{QD}=\sum_{i=1}^{r} f_{\mathrm{QD}}(p_i)/r.$$

Then, we can formalize the problem as an optimization problem with description of one of CW, QC, CC, and QD as optimization goal, and other equations as well as the input range requirements as the constraint. By applying simplex algorithm, we obtain the optimized solution for this problem to determine the parameters. For example, in our experiment, the problem is solved as follows.

We can obtain $\mathrm{QC}=\sum_{i=1}^{r} f_{\mathrm{QC}}(p_i)/r, \mathrm{CC}=\sum_{i=1}^{r} f_{\mathrm{CC}}(p_i)/r,\ \mathrm{QD}=\sum_{i=1}^{r} f_{\mathrm{QD}}(p_i)/r,\ n\times m<N/\ 50\,000,$ $50\%<e<1, n\geqslant 2,\ 2<b'<b<15, \mathrm{QC}\geqslant 0.95, \mathrm{CC}\leqslant 130, \mathrm{QD}\geqslant 0.95,\ \mathrm{CW}\leqslant 130.$

Using the simplex algorithm, we obtain parameter set $\{n=11, m=4000, e=0.9, b'=4, b=9\}$ which is proven as the best choice in Section 7.2.2.

7 Experiment

To verify the efficiency and effectiveness of the proposed algorithms, we perform extensive experiments in this section.

7.1 Experimental settings

The experiments were conducted on both synthetic datasets and real-life data. We firstly use synthetic data generated by TPC-H, which is a decision support benchmark and can generate data in any size to evaluate the performance and scalability of our algorithm and optimality of the method of choosing parameters. We also used real dataset names from the UCI machine learning repository (http://archive.ics.uci.edu/ml/), dblp (http://dblp.uni-trier.de/) namely SUSY dataset, and article dataset as shown in Table 2 to check the effect of the method on real data .

All algorithms are implemented in Java. The program has been tested on a PC with Intel Core i7 4770 (3.4 GHz) and 8 GB of memory running Ubuntu operating system. Each experiment was repeated three times and the average is reported.

We use the following parameters to evaluate the proposed algorithms:

(1) The time of finding CFDs from the dataset.

(2) The quality of discovered CFDs is measured by the percentage of the CFD sets discovered by our approach on dirty data and those obtained from the clean data.

(3) The time of cleaning data with discovered CFDs.

(4) The quality of data cleaned by discovered CFDs is measured by the percentage of data cleaned according to the CFD sets discovered by our approach on dirty data and those obtained from the clean data.

Also, to test the optimality of the method, we choose parameters in Section 6 and compare the effect of different choices of parameters using the controlling variable method.

7.2 Experiment result

7.2.1 Performance and scalability experiments

We show the performance and scalability of our

Table 2 Parameters of real dataset.

Dataset	Arity	Size (number of tuples)
SUSY dataset	18	5 000 000
Article dataset	23	220 000

algorithm through different data sizes and arities. We use CFDs discovered on the clean data by combining algorithms (the original CFD discovery algorithms) as baseline. Then, we modify 8% of the generated data to make them dirty and usable for testing.

(1) Efficiency experiments

(a) Impact of tuple number

We varied the tuple number from 1×10^5 to 1.2 billion with 16 attributes for each tuple. The maximal data size is 2.1×10^{11}. The discovery time is shown in Fig. 5a, where "combined" refers to the original algorithms for CFD discovery and "improved" refers to the algorithm proposed in this paper and DDQS as the experiment for Ref. [9]. The horizontal axis is in logarithm scale. The reason we do not use x directly is that other two algorithms can only work for small data and we want to use a very big size to show the data size that our algorithm can deal with. From Fig. 5, we obtain the following observations:

When DBSIZE (the number of tuples in the database) is smaller than 7×10^4, the response time of our method is higher than that of the combined and DDQS algorithm. This condition shows that due to the time of sampling and conflict resolution, our method performed poorly with small data.

When DBSIZE$> 7\times10^4$, the original algorithms and DDQS find CFDs more slowly. The reason is that when DBSIZE is large enough, it costs more time to find CFDs than sampling and combining different sets.

The increasing speed of other two lines is higher than ours, which shows that our algorithm is effective for big data.

(b) Impact of attribute number

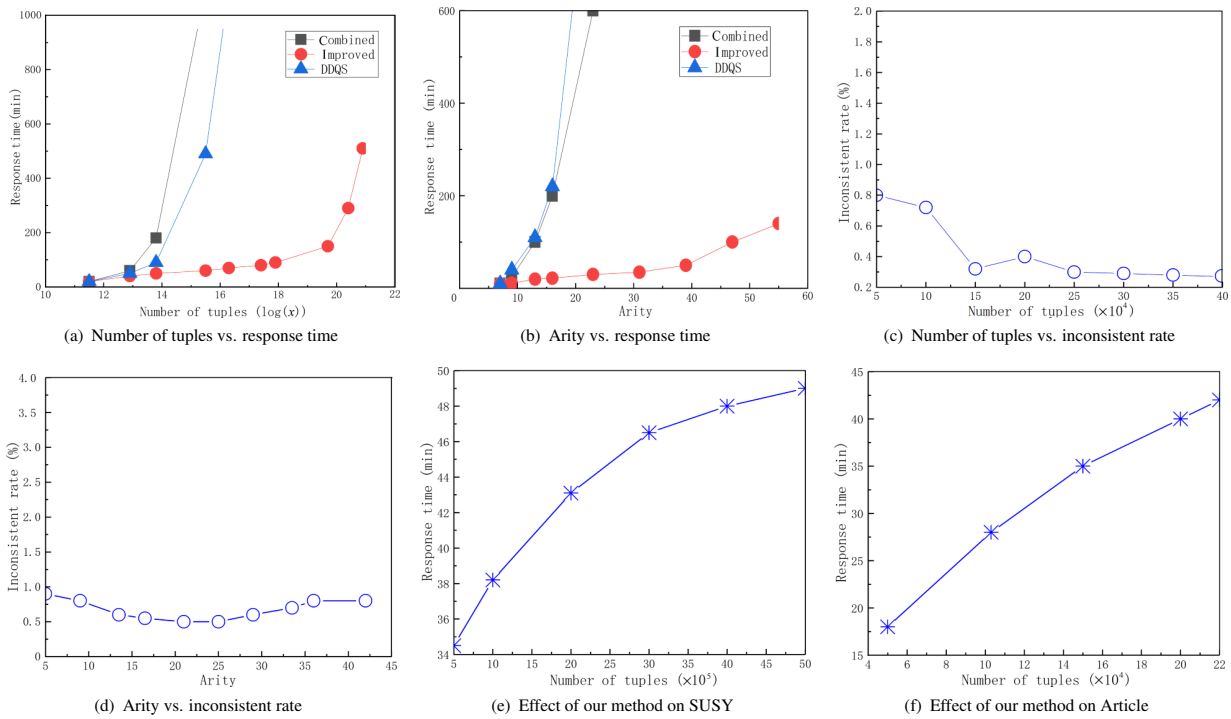
We vary the attribute number from 7 to 55, and fix the tuple number 1×10^6. From Fig. 5b, we can find that the two lines are both index functions which show the index form of r in the objective function of Section 6. However, the line for our method is more gentle. When arity is smaller than 23, the combined algorithms are faster because no sample is needed. For the arity of more than 25 attributes, our method outperforms the combined algorithms. Our method can save over 20% of the time when arity is 55. Compared with other lines in the graph, our method performs better for the data with a large number of attributes.

(2) Precision experiments

We add the CFDs found by the methods in the gather of original algorithms (discover CFDs using the original algorithms) to obtain a standard set of CFDs. By computing the percentage of the standard set of CFDs that are not covered by our CFDs, we evaluate the precision of our algorithm.

(a) Impact of tuple number

We varied the tuple number from 5×10^4 to 4×10^5 tuples with 16 attributes for each item. The line shows the percentage of the CFDs found by the methods in the combined algorithm. The y-axis represents the percentage of CFDs which are generated by the



(a) Number of tuples vs. response time
(b) Arity vs. response time
(c) Number of tuples vs. inconsistent rate
(d) Arity vs. inconsistent rate
(e) Effect of our method on SUSY
(f) Effect of our method on Article

Fig. 5 Experiment result.

combined algorithm and covered by CFDs from our algorithm, and is called inconsistent rate.

From Fig. 5c, we have the following observations:

The inconsistency rate is less than 1%, which shows that our method can always find CFDs that are similar to those found by original methods.

When tuples are large, the inconsistency rate is smaller than that for a small number of tuples. This result proves that our method is more scalable for big data.

When DBSIZE is larger than 3×10^4, we can use CFDs that we find as a standard set of CFDs due to precision $> 96.5\%$.

(b) Impact of attribute number

We varied the arity from 5 to 42 by fixing the tuples as 1×10^5. From Fig. 5d, we can see that the CFDs found by our method are highly similar to those standard CFDs no matter what the arity is. When arity is 25, the effect is the best of all. However, when the arity is extremely large or extremely small, the result of our CFDs worsens. For extremely large arity, wrong items may be concentrated when we change items by ourselves. We will obtain some similar wrong samples, which make our CFDs appear to be wrong. This is caused by people and to real dataset, this will not happen. For extremely small arity, we can find that CFDs are few, thereby making the base small.

7.2.2 Optimality of parameters

To show the effectiveness of our parameter selection method, we change one parameter, and leave the others unchanged and compare the four dimensions. We use TPC-H to generate the dataset and use the default constraint for parameters as $\{n = 11, m = 4000, e = 0.9, b' = 4, b = 9\}$ and only change a parameter in these parameters. In each set of experiments, we compare the results with parameters $\{n = 11, m = 4000, e = 0.9, b' = 4, b = 9\}$ obtained by our method and those with different values of the optimization goal. In each table, the column with grey background is the result with optimal parameters.

The results with n as the optimization goal is shown in Table 3. From the results, we find that only when n is 11 can the four objectives be satisfied and the CW is as short as impossible. When it is large, the time for finding CFDs increases and when it is extremely small, the CFDs we find are not so accurate. From Table 4, we can see that the optimization result for m is 4000 items in each group of sampling. When it is extremely large, we can find that the time of finding CFDs and the cleaning dataset is extremely large. When it is extremely small, QC becomes worse and the result of cleaning is poor.

We know from Table 5 that when e is 0.9, we can obtain the best results. When e is extremely small, we find many wrong CFDs and spend a large amount of time cleaning the data. When e is extremely large, we can be too strict to tolerate wrong tuples and leave CFDs.

The results with b as the optimization goal are shown in Table 6. We can see that when it is too small or too large, the CFDs we find will be not so accurate.

We can find from Table 7 that we should choose 4 as the amount of b'. Although when b' is 5, we can also accept the result, while the CW less than 4 makes us abandon it.

Above all, we can see that our selection of $n = 11$, $m = 4000, e = 0.9, b' = 4$, and $b = 9$ can work best to make CW as small as possible and satisfy the low limits for other aims.

Table 3 Optimization of n.

n	CW (min)	QC	CC (min)	QD
5	75	0.972	102	0.83
8	83	0.990	115	0.94
11	92	0.994	123	0.96
14	123	0.994	132	0.97
17	157	0.993	139	0.98

Table 4 Optimization of m.

m	CW (min)	QC	CC (min)	QD
2000	99	0.985	95	0.92
4000	102	0.994	123	0.96
6000	121	0.995	132	0.97
8000	155	0.995	141	0.97
10 000	190	0.997	151	0.98

Table 5 Optimization of e.

e	CW (min)	QC	CC (min)	QD
0.60	120	0.951	150	0.86
0.70	107	0.979	138	0.89
0.80	99	0.987	129	0.93
0.90	92	0.994	123	0.96
0.97	86	0.985	120	0.94

Table 6 Optimization of b.

b	CW (min)	QC	CC(min)	QD
7	81	0.990	102	0.94
9	92	0.994	123	0.96
10	103	0.993	128	0.95
12	124	0.991	131	0.94

Table 7 Optimization of b'.

b'	CW (min)	QC	CC (min)	QD
2	95	0.972	147	0.96
4	92	0.994	123	0.96
5	101	0.990	117	0.95
6	104	0.983	109	0.93

7.2.3 Test on real data

We use real datasets from the UCI machine learning repository and DBLP, namely, the SUSY and article datasets, article data is to test the effectiveness of our method on real data.

Figure 5e shows the time of discovering CFDs from the SUSY dataset. We use different parts of the dataset to test the scalability. We observe that when we increase the tuple number, time increases around linear with the data size. This result shows that our method can deal with real big data in a linear effect. The largest size of the data is 6.2×10^9. For the dataset from DBLP in Fig. 5f, we vary DBSIZE from 5×10^4 to 2×10^5. The largest size of the data is 2.1×10^9. We also find that the time of finding CFDs increases linearly with the data size, thereby showing the linear cleaning effect of our method on the real big data. Thus, the experimental results on the real data verify the analysis results.

8 Related Work

In this section, we present a brief survey of the related work.

(1) Concept of dependencies. A set of data quality rules are often created to improve data consistency. Once the inconsistent items exist in the database, some rules are violated. Thus, errors are discovered and revised accordingly. In general, the integrity constraints should be used as a data-quality detection rule to improve data consistency[1,5,7,9,10]. The theory of conditional dependencies, including CFDs[11] and Conditional Inclusion Dependencies (CINDs)[12], develops the traditional FDs and inclusion dependencies to capture the common mistakes in realistic data. For the conditional FDs, Refs. [11, 12] study the problems including the consistency, logical implication, and axiomatic for dependency language. Based on Refs. [11, 12], a variety of extensions for conditional dependencies have been proposed in Refs. [1, 13–15] to develop the capacity of illustrating conditional dependencies without the growth of the computational complexity.

(2) Rule mining. To use dependencies as data quality rules, the first problem is how to obtain these dependencies. References [2, 16] design the automatic discovery algorithms for finding CFDs. However, the algorithms in them both need to work on a clean and representative dataset. In Ref. [3], CFDs can be discovered from a dirty dataset. However, the process can be hardly finished for the dataset with size larger than the memory. Meanwhile, the complexity of the algorithm in Ref. [3] is large for big data.

(3) Algorithms used in rules mining for big data. Many methods have been proposed to find rules on big data. In Ref. [4], an on-demand algorithm is proposed to generate an optimal tableau for given CFDs. In Ref. [6], various sampling and sketching techniques are used to estimate the confidence of a CFD with a small number of passes (one or two) over input using a small space.

(4) Rule analysis and optimization. As the data quality rule set may contain conflicts, we need to find out consistent constraint rules (i.e., a maximum consistent subset) as data quality rules. The computational complexity of this problem is extremely high. For CFDs, finding the maximum consistent subset of rules is proven to be NP-complete[17]. When we consider both the CFDs and CINDs, this problem is undecidable. Thus, approximate algorithms to calculate a maximum consistent subset for CFDs have been proposed in Ref. [11].

(5) Error detection. Error detection means capturing data errors by the consistent subset of the data-quality rules. This method finds the tuples in violation of the data-quality rules. In Refs. [11, 14], for centralized storing relational databases, the approaches are designed to detect the tuples in violation of CFDs and CINDs automatically based on SQL query processing.

9 Conclusion

For big data, rule discovery in data cleaning brings new challenges. To solve this problem, we proposed a novel CFD discovery method for big data. For the volume feature of big data, we designed a sampling algorithm to obtain typical samples by scanning data only once. Then, on the sample set, we adapted existing CFD discovery algorithms to tolerate the fault. By integrating these modified methods, we discovered a preliminary CFD set. To increase the quality in the discovered rule set, we designed a graph-based rule selection algorithm. Considering that a user may have different requirements for CFD discovery, we proposed a strategy to select parameters according to the requirements of users. The experimental results demonstrated that the proposed algorithm is suitable for big data and outperforms existing algorithms. Future work includes extending the proposed algorithm to parallel platforms and modifying the proposed algorithm to discover other rules.

Acknowledgment

This paper was partially supported by the National Key R&D Program of China (No. 2018YFB1004700), the

National Natural Science Foundation of China (Nos. U1509216, U1866602, and 61602129), and Microsoft Research Asia.

References

[1] W. F. Fan, F. Geerts, X. B. Jia, and A. Kementsietsidis, Conditional functional dependencies for capturing data inconsistencies, *ACM Trans. Database Syst.*, vol. 33, no. 2, p. 6, 2008.

[2] W. G. Chen, W. F. Fan, and S. Ma, Analyses and validation of conditional dependencies with built-in predicates, in *Proc. 20th Int. Conf. Database and Expert Systems Applications*, Linz, Austria, 2009, pp. 576–591.

[3] B. Bollobás, *Modern Graph Theory*. New York, NY, USA: Springer-Verlag, 1998.

[4] L. Golab, H. Karloff, F. Korn, D. Srivastava, and B. Yu, On generating near-optimal tableaux for conditional functional dependencies, *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 376–390, 2008.

[5] G. Cormode, L. Golab, K. Flip, A. McGregor, D. Srivastava, and X. Zhang, Estimating the confidence of conditional functional dependencies, in *Proc. 2009 ACM SIGMOD Int. Conf. Management of Data*, Providence, RI, USA, 2009, pp. 469–482.

[6] J. S. Vitter, Random sampling with a reservoir, *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, 1985.

[7] W. F. Fan, F. Geerts, J. Z. Li, and M. Xiong, Discovering conditional functional dependencies, *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 5, pp. 683–698, 2011.

[8] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[9] F. Chiang and R. J. Miller, Discovering data quality rules, *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 1166–1177, 2008.

[10] C. Batini and M. Scannapieco, *Data Quality: Concepts, Methodologies and Techniques*. Berlin, Germany: Springer, 2006.

[11] J. Chomicki, Consistent query answering: Five easy pieces, in *Proc. 11th Int. Conf. Database Theory*, Barcelona, Spain, 2007, pp. 1–17.

[12] L. Bertossi, Consistent query answering in databases, *ACM SIGMOD Record*, vol. 35, no. 2, pp. 68–76, 2006.

[13] W. F. Fan, Dependencies revisited for improving data quality, in *Proc. 27th ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems*, Vancouver, Canada, 2008, pp. 159–170.

[14] E. Rahm and H. H. Do, Data cleaning: Problems and current approaches, *IEEE Data Eng. Bull.*, vol. 23, no. 4, pp. 3–13, 2010.

[15] L. Bravo, W. F. Fan, and S. Ma, Extending dependencies with conditions, in *Proc. 33rd Int. Conf. Very Large Data Bases*, Vienna, Austria, 2007, pp. 243–254.

[16] L. Bravo, W. F. Fan, F. Geerts, and S. Ma, Increasing the expressivity of conditional functional dependencies without extra complexity, in *Proc. 24th Int. Conf. Data Engineering*, Cancun, Mexico, 2008, pp. 516–525.

[17] A. K. Kalavagattu, Mining approximate functional dependencies as condensed representations of association rules, Master dissertation, Arizona State University, Phoenix, AZ, USA, 2008.

Mingda Li is a fifth year PhD student in UCLA. His research interest lies in learning semantic information by embedding different objects (words, IPs, and utterances, etc.) with deep neural networks and making the learning process more efficient via boosted negative sampling or a more scalable design of the learning platform, etc.

Hongzhi Wang received the PhD degree from Harbin Institute of Technology in 2008. He is a professor and PhD supervisor of Harbin Institute of Technology, the secretary general of ACM SIGMOD China, CCF outstanding member, a member of CCF databases and big data committee. His research fields include big data management and analysis, database, graph management, and data quality. He was "starring track" visiting professor at MSRA and postdoctoral fellow at University of California, Irvine. He has been PI for more than 10 national or international projects including NSFC key project, NSFC projects and National Technical support project, and co-PI for more than 10 national projects include 973 project, 863 project, and NSFC key projects. He also serves as a member of ACM Data Science Task Force. He has won first natural science prize of Heilongjiang Province, MOE technological first award, Microsoft Fellowship, IBM PhD Fellowship, and Chinese excellent database engineer. His publications include over 200 papers including VLDB, SIGMOD, SIGIR papers, 4 books, and 3 book chapters. His PhD thesis was elected to be outstanding PhD dissertation of CCF and Harbin Institute of Technology. He serves as the reviewer of more than 20 international journal including *IEEE TKDE* and PC member of over 30 internal conference. His papers were cited more than 1000 times.

Jianzhong Li is professor and doctoral supervisor at Harbin Institute of Technology. He is a senior member of CCF. His research interests include database, parallel computing, and wireless sensor networks, etc.