IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 6, 2019, accepted February 3, 2019, date of publication February 13, 2019, date of current version March 1, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2898707

One-Pass Inconsistency Detection Algorithms for Big Data

MEIFAN ZHANG, HONGZHI WANG, JIANZHONG LI, AND HONG GAO
Department of Computer Science and Technology, Harbin Institute of Technology, Harbin 150006, China

Corresponding author: Hongzhi Wang (wangzh@hit.edu.cn)

This work was supported by the NSFC under Grant U1509216, Grant U1866602, and Grant 61602129.

ABSTRACT Data in the real world is often dirty. Inconsistency is an important kind of dirty data; before repairing inconsistency, we need to detect them first. The time complexities of the current inconsistency detection algorithms are super-linear to the size of data and not suitable for the big data. For the inconsistency detection of big data, we develop an algorithm that detects inconsistency within the one-pass scan of the data according to both the functional dependency (FD) and the conditional functional dependency (CFD) in our previous work. In this paper, we propose inconsistency detection algorithms in terms of FD, CFD, and Denial Constraint (DC). DCs are more expressive than FDs and CFDs. Developing the algorithm to detect the violation of DCs increases the applicability of our inconsistency detection algorithms. We compare the performance of our algorithm with the performance of implementing SQL queries in MySQL and BigQuery. The experimental results indicate the high efficiency of our algorithms.

INDEX TERMS Inconsistency detection, big data, one-pass algorithm, data quality, denial constraint.

I. INTRODUCTION

Data quality problems in real world may cost not only billions of dollars in businesses, but also precious lives when they exist in medical data [1]. With the consideration of the serious consequences caused by data quality problems, techniques for detecting and fixing errors are in great demand. For big data, due to the volume feature, it has data quality problems in higher possibility.

Inconsistency is an important aspect of data quality problems. Inconsistency means that some tuples violate given rules. For effective inconsistency detection, some forms of rules are proposed such as functional dependency (FD) and conditional functional dependency (CFD) [2]. Denial constraints (DCs) are also regarded as inconsistency detection rules, they are more expressive than the traditional FDs and CFDs [3]–[6].

In [7] and [8], researchers proposed a SQL-based automatical detection method to identify the tuples violating CFDs. In [7], they get a tableaux merged by multiple CFDs, and translate that into a single pair of SQL queries. With this pair of SQL queries, they only need to scan the database twice. In the queries, they get a Macro by joining the tableaux with the whole dataset. However, join operation for big data still costs much. In [9], a method is proposed to detect and repair data structure inconsistencies automatically. However, they are not suitable for inconsistency detection on big data due to efficiency issues. As a result, they are difficult to scale to big data. We use an example to illustrate this point.

TABLE 1. An salary relation R.

	Dept	TP	Title	BP	AT	IL
t1	1	1500	1	1000	10	20
t2	1	2500	1	1000	10	20
t3	1	1500	2	1000	20	10
t4	1	1500	3	2000	20	10
t5	1	2500	2	2000	20	5
t6	1	1500	2	2000	30	10

Example 1: The schema in Table 1 is a relation of salary in a company. There are six attributes in this relation: Department (Dept), Total Pay (TP), Title Level (Title), Base Pay (BP), Attendance (AT), Ill and Leave (IL). We use the following two rules to detect the inconsistencies in this relation: one FD $\phi 1(R : Dept, Title \rightarrow BP, (\_, \_\|\_))$, one CFD $\phi 2(R : Dept, Title \rightarrow BP, (1, 2 \parallel 2000))$.

According to the SQL-based method, we need the following two queries to detect the inconsistencies. Q_V is a query for the FD $\phi 1$, and Q_C is for the CFD $\phi 2$.

Q_C : `select` t from R
`where` $t[Dept] = tp[Dept]$ AND $t[Title] = tp[Title]$ AND $t[BP] \neq tp[BP]$

The associate editor coordinating the review of this manuscript and approving it for publication was Zhao Zhang.

Q_V : select distinct $t[Dept], t[Title]$ from R
where $t[Dept] \asymp tp[Dept]$ AND $t[Title] \asymp tp[Title]$ AND $t[BP] \neq tp[BP]$
group by $t[Dept], t[Title]$
having count (distinct $t[Dept], t[Title]$) > 1

In the queries above, t is a tuple in Table 1 and tp is a single pattern tuple of a rule. The rule $t[A] \asymp tp[A]$ means that if $tp[A]$ is not "_", then $t[A] = tp[A]$; otherwise, $t[A]$ should be a constant in the domain $dom(A)$ of attribute A. The operators and the way of converting a DC into queries are introduced in reference [7], [10].

The first query returns the tuples violating $\phi 2$, which is shown in Table 2. And the second query returns the values of *Dept* and *Title* of inconsistent tuples. The tuples matching the records in Table 3 are those involved in the inconsistencies.

TABLE 2. The result of Q_C.

	Dept	TP	Title	BP	AT	IL
t3	1	1500	2	1000	20	10

TABLE 3. The result of Q_V.

Dept	Title
1	2

It is obviously that these two queries need two passes scanning of the data. And the results of FD detection only return the distinct values of the left hand side (LHS) of inconsistent tuples. We still need a list of queries for the inconsistent tuples. In this example, with the result of Q_V in Table 3, we still need one more query Q_R.

Q_R : select t from R
where $t[Dept] = 1$ AND $t[title] = 2$

This query returns the violations of $\phi 1$, and the result is $\{t_3, t_5, t_6\}$.

One query requires at least one pass of database scan, and the efficiency of a query is affected by the database size. This method may cost too much time for detecting inconsistencies in big data.

In order to make inconsistency detection more general, we also take DCs into consideration. Six operators $\{=, \neq, <, >, \leq, \geq\}$ are allowed in the DCs. These operators make DCs more expressive than FDs and CFDs. We use an example to indicate the violation of DCs.

Example 2: In the relation shown in Table 1, we have a rule like this: if $t1.AT > t2.AT$, then the value $t1.IL$ must be lower than or equal to $t2.IL$. This rule can be expressed as a DC:

$$\forall t1, \quad t2 \in R, \ \neg(t1[AT] > t2[AT] \wedge t1[IL] > t2[IL])$$

We can know from Table 1 that $t5$ conflicts with $t6$ since $t6[AT] > t5[AT]$ and $t6[IL] > t5[IL]$.

The DC in last example can be converted into one SQL query easily:

Q_{DC} : select $t1, t2$ from $R\ t1, R\ t2$
where $t1[AT] > t2[AT]$ AND $t1[IL] > t2[IL]$

However, this query is processed as a Cartesian product, meaning that the time complexity is $O(n^2)$. Though the denial constraints can be easily expressed in SQL queries, the low efficiency makes it not suitable for detecting the inconsistency of big data.

The inconsistency detection for big data motives us to design new algorithms that could accomplish the detection within one-pass of data scan. Considering reducing the times of accessing database, we group each attribute by its values. For the tuples with the same value on same attribute, we group their IDs together in a tuple set. Then, we attempt to get the violation with the grouped result. In the detection process with FDs and CFDs, we first get the tuples matching the LHS of a rule by calculating a list of intersections of the grouped tuple sets. At last, we determine the inconsistencies by checking that whether the tuple set matching the LHS is the subset of a tuple set matching the right hand side (RHS). In the process of detecting inconsistencies with DCs, we also make use of the grouped result. The violation of constant DCs can be captured easily similar to detecting the inconsistency of constant CFDs. Detecting with variable DCs is more complex. We proposed an efficient algorithm whose time complexity is $O(n \cdot logn)$. In any case, the detection process scans the data set only once.

We make following contributions in this paper.

Our first contribution is enriching our one-pass algorithm for detecting inconsistencies with FDs and CFDs with more details. We add more examples and prove the correctness of the detection algorithm.

Our second contribution is that we extend our algorithm and take a more general rule denial constraint into consideration. We proposed two algorithms for detecting inconsistency with constant DCs and variable DCs, respectively.

Our third contribution is the extensive experiment on both real and synthetic datasets. We compare our method with SQL-based methods. The experimental results demonstrate that the proposed methods are efficient and effective[1].

The remaining parts of this paper are organized as follows. In Section II, we survey related work for this paper. In Section III, we introduce the background of this paper including the definitions of inconsistencies and the rules used to detect inconsistencies. In Section IV, we propose the one-pass inconsistency detection method to detect the violation of FDs and CFDs. In Section V, we develop two algorithms to detect inconsistencies with DCs. In Section VI, we compare our detection methods with previous methods experimentally, and the experimental results on real datasets demonstrate

[1] In the experiments, we use our detection methods and the SQL-based method to detect the inconsistencies with same rules respectively. The detection results of our inconsistency detection method are same with the results of the SQL queries, and the results also accord with the reality.

the performance of our detection methods. In Section VII, we draw the conclusion and give a brief overview of our work in future.

II. RELATED WORK

As an important problem in data quality management, the inconsistency problem including both the detecting and repairing problem has been well studied [2], [7], [8], [11], [5], [12]–[16]. Some researchers use statistical model and propose some thresholds to identify the inconsistencies [17], [18]. Other researchers use constraints such as FDs and CFDs [2], [7], [8], [10]–[13], [19] to detect inconsistencies. In order to make the rules more general and more expressive, denial constraints (DCs) are also regarded as rules. DCs can express rules involving numerical values, with predicates such as ''greater than'' and ''less than''. DCs have been used for data cleaning as well as consistent query answering [3]–[6]. As the DCs are defined on predicates, they can be expressed in SQL queries easily. However the SQL queries usually have low efficiency. There is no efforts on detecting inconsistencies with high efficiency when regarding DCs as the data quality rules.

It is necessary to obtain rules before detecting inconsistencies with the rule-based detection methods. Some methods have been proposed to discover rules automatically [19], [20], which are mainly extensions to traditional FD. Reference [20] proposed a search algorithm with pruning strategy which can discover rules effectively. In [21], a method is proposed to automatically discover CFD. A DC discovering algorithm is proposed by Chu *et al.* [22] to automatically get the DCs from database. The rules used for detection may have conflicts. In order to solve this problem, a data cleaning framework is proposed to resolve conflicts in rules [23]. Bleifu [24] proposed a new algorithm called Hydra to automatically discover DCs. The work in [25] presents BFASTDC, a bitwise version of FASTDC that uses logical operations to form the auxiliary data structures from which DCs are mined.

Wang and Tang [26] proposed algorithms to check whether the fixing rules are consistent and devised algorithms for repairing data errors. CINA proposed in [27] analyses the constraints instability conditions to address the unstable context inconsistencies. In [28], they defined hazard patterns to identify and suppress the hazard from real inconsistencies. The hazard in that paper is similar to the STINs in CINA. Alpar and Winkelsträter [29] point out that both data quality breaches and correcting the violations by humans are expensive. Therefore, they use a metric to measure the consistency based on the rules and manually repair the least inconsistent transactions. A system for the discovery of approximate temporal functional dependencies is raised in [30]. It performs better than the traditional FDs when cleaning a temporal data set.

Recently researchers have been looking into automatic inconsistency detecting methods [9], which regards FDs as the main rules. Detecting methods based on CFDs are also proposed after the definition of CFD [7], [15]. Many previous methods only based on one kind of rules [2], [5], [9], [11], [31], few works combine different kind of rules together to detect inconsistencies. Li *et al.* [32] introduce a source selection method called SSID for inconsistency detection, however, multi-sources of data are required. Shaikh *et al.* [33] propose a novel method for detecting inconsistencies and incompleteness in access control policies with the help of data classification tools. Anand *et al.* [34] make efforts to detect the inconsistencies in business rules. However, the rules in these works [33], [34] are much different from the rules in our work.

The previous works on inconsistency detection mostly need redundant data inquiries. Many works aim at mining constraints and repairing violations [24]–[26], [35], however, few works make efforts in improving the efficiency of inconsistency detection. Queries are still used for inconsistency detection in some recent works [25], [35], [36]. There have been little efforts on detecting tuples involved in the inconsistencies with only one pass database scan. In the preliminary work [37], we proposed a one-pass algorithm to detect the violation of CFDs. In this paper, we extend the work and propose algorithms to detect the violation of DCs. We proved that the inconsistency detection with variable DCs can be converted into the problem of getting the ascending ordered pairs in an array. The time complexity can be reduced to $O(n \cdot logn)$.

III. PRELIMINARIES

In this section, we introduce some background of this paper including the definitions of inconsistencies and the rules used to detect inconsistencies. The rules in this section are defined in the syntax similar to those of FDs and CFDs in references [7], [10] and DCs in references [3], [22].

A. RULES

For a relational schema R, we use the rules containing normal form FDs (CFDs) and DCs in this paper.

Definition 1: Normal form: A FD or a CFD $\phi(R : X \rightarrow Y, T_\phi)$ is in normal form if (1)T_ϕ consists of a single pattern tuple tp. If $R : X \rightarrow Y$ is an FD, $tp = (\_, \_, \ldots, \_\|\_)$, which means tp consists of "_" only. (2)Y consists of a single attribute A.

After the definition of normal form, we write ϕ simply as $\phi(R : X \rightarrow A, tp)$. Each CFD (FD) not in normal form can be written as a set of CFDs (FDs) in normal form. For example, a CFD $\phi(R : Dept, Title, AT \rightarrow BP, TP, (1, 2, \_\|2000, \_))$ not in normal form can be written into two rules in normal form: (1) $\phi_1(R : Dept, Title \rightarrow BP, (1, 2\|2000))$, (2) $\phi_2(R : Dept, Title, AT \rightarrow TP, (1, 2, \_\|\_))$. With the help of rules in normal form, we can easily know the cells involved in the inconsistencies, since the RHS of the rules only contain one attribute. If a tuple t violates ϕ_1, it is clear that $t[Dept] = 1$, $t[Title] = 2$, $t[BP] \neq 2000$.

The normal form rules can be classified according to whether their RHS is a constant or not.

Definition 2: Constant RHS rules: The constant RHS rules contain the constant CFDs only. A CFD $\phi(R : X \rightarrow A, tp)$ is called a constant CFD if its pattern tuple tp consists of constants only. That is, $tp[A]$ is a constant and for each attribute $B \in X$, $tp[B]$ is a constant.

Definition 3: Variable RHS rules: The variable RHS rules contain both FDs and variable CFDs. A CFD $\phi(R : X \rightarrow A, tp)$ is called a variable CFD if tp[A] = "_". That is, the right hand side (RHS) of its pattern tuple is the unnamed variable "_". The RHS of FDs is also obviously the unnamed variable "_".

Definition 4: A denial constraint (DC) is in the form: $\forall t_\alpha, t_\beta, t_\gamma, ..., \neg(A_1 \wedge ... \wedge A_m)$. A_i are built-in atoms of the form $v_1 \theta v_2$ or $v_1 \theta c$, where v_1, v_2 is in the form of $t_x.A, x \in \{\alpha, \beta, \gamma, ...\}$, A is an attribute in R, c is a constant and $\theta \in \{=, \neq, <, >, \leq, \geq\}$.

We also classify denial constraints into two kinds: (1)constant DC, (2)variable DC.

Definition 5: Constant DC: A denial constraint where each A_i is built-in atoms of the form $v_1 \theta c$. v_1 is in the form of $t_x.A, x \in \{\alpha, \beta, \gamma, ...\}$, A is an attribute in R, c is a constant and $\theta \in \{=, \neq, <, >, \leq, \geq\}$.

Definition 6: Variable DC: A denial constraint where exist at least two A_i built-in the form $v_1 \theta v_2$. v_1, v_2 is in the form of $t_x.A, x \in \{\alpha, \beta, \gamma, ...\}$, A is an attribute in R, and $\theta \in \{=, \neq, <, >, \leq, \geq\}$.

FD is a special case of variable DC, where each A_i is built in the form $t_x.A = t_y.A, x, y \in \{\alpha, \beta, \gamma, ...\}$. DC is a more general integrity constraint in database practice.

B. INCONSISTENCY DETECTION

Inconsistency detection problem is to find the data violating given rules. An instance I of a schema R satisfies a rule ϕ, denoted by $I \models \phi$, (1) if $\phi(R : X \rightarrow A, tp)$ is a constant RHS rule, then for each tuple t in I, $t[X] = tp[X]$ implies $t[A] = tp[A]$, (2) if $\phi(R : X \rightarrow A, tp)$ is a variable RHS rule, then for each pair of tuples $t1$ and $t2$ in I, $t1[X] = t2[X]$ implies $t1[A] = t2[A]$, (3) if ϕ is a DC, the violation of a DC means that each predicates A_i in $\forall t_\alpha, t_\beta, t_\gamma ..., \neg(A_1 \wedge ... \wedge A_m)$ cannot be true at the same time.

The violations can be separated into two kinds according to the kind of rules.

Definition 7: Inconsistent tuples set: The inconsistent tuple set TS is the set of tuples violating the constant RHS rules and the constant DCs.

Definition 8: Inconsistent tuples sets group: A group of inconsistent tuples sets G is a group of tuple sets violating the variable RHS rules and variable DCs. Considering the violations of variable RHS rules, the tuples in the same group have the same LHS, tuples in different sets of the same group have different RHS values.

The inconsistent tuples detected by a constant RHS rule $\phi(R : X \rightarrow A, tp)$ can be grouped into one inconsistent tuple set TS. $\forall t \in TS, t[X] = tp[X]$ and $t[A] \neq tp[A]$.

Example 3: In Table 1, we detect inconsistent tuples with a constant RHS rule $\phi(R : Dept, Title \rightarrow BP, (1, 2 \| 2000))$, the inconsistent tuple set is $TS = \{t3\}$.

The inconsistent tuples detected by a variable RHS rule $\phi(R : X \rightarrow A, tp)$ must be grouped into groups of inconsistent tuple sets Gs. For each group G, we put the tuples with the same LHS and different RHS into different sets of the same group G.

Example 4: In Table 1, we detect inconsistent tuples with a variable RHS rule $\phi 1(R : Dept, Title \rightarrow BP, (\_, \_ \| \_))$, the inconsistent tuple sets groups G contains one group, $G[1] = \{\{t3\}, \{t5, t6\}\}$.

Example 5: We detect the inconsistency with these two Denial Constraints:

(1)one constant DC: $\forall t, \neg(t[AT] > 20 \wedge t[IL] \geq 10)$.

(2)one variable DC: $\forall t_1, t_2, \neg(t1[AT] > t2[AT] \wedge t1[IL] > t2[IL])$.

The violation of the first rule: $t6$, since $t6[AT] > 20$ and $t6[IL] \geq 10$.

The violation of the second rule: $t6$ conflicts with $t5$ since $t6[AT] > t5[AT]$ and $t6[IL] > t5[IL]$.

IV. DETECTING THE INCONSISTENCIES OF CFD

In this section, we describe our inconsistency detection algorithm. In Section IV-A, we introduce the framework of the algorithm, and then explain the four modules RANGE, GROUP, MATCH, and MERGE in the following subsections, respectively.

A. FRAMEWORK

In order to reduce the times of accessing database, we group each attribute by its values. And for the tuples with same value on same attribute, we group their IDs together in a tuple set and build a hash index for retrieving the tuple sets efficiently. Then we get the tuples matching the LHS by calculating a list of intersections of the grouped tuple sets. At last, we obtain the inconsistencies by checking whether the tuple set matching the LHS is the subset of a tuple set matching the RHS. In our detection method, we only require scanning the database once.

The pseudo code of our algorithm is shown in Algorithm 1. The algorithm has following four steps.

In the first step (line 3), we collect the set of the attributes and its values from the rules to reduce the data requiring accessing during detection. This step is introduced in detail in Section IV-B.

In the second step (line 4), we group the tuples for each attribute collected in the first step by its values. This step is introduced in Section IV-C.

In the third step (line 5-11), we detect the violation of the constant RHS rules. For one constant RHS rule $\phi(R : X \rightarrow A, tp)$, we get the tuple sets from the result of the second step with the attributes in $attr(\phi)$ and the values in tp. The tuples matching the LHS of ϕ are in $left\_set$, and for each X_i in X, $left\_set \cap = tupleset_{X_i = tp[X_i]}$. Likewise,

Algorithm 1 Detection

1: **Input:** a database D, a set of rules $\sum$.
2: **Output:** the inconsistencies $sets_\phi$ for each rule ϕ in $\sum$
3: $Aset = RANGE(\Sigma)$
4: $S = GROUP(D, Aset)$
5: **for** each constant CFD **do**
6: **for** attribute $X_i \in LHS(\phi)$ and its value $x_i = tp[X_i]$ **do**
7: $left\_set \cap = tuple\_set_{X_i=x_i}$
8: $right\_set \leftarrow tuple\_set_{RHS(\phi)=tp[A]}$
9: $sets_{\phi 0} \leftarrow left\_set - right\_set$
10: **for** each variable RHS rule **do**
11: $LS_\phi \leftarrow \{X_i \in LHS(\phi) | S[X_i]\}$
12: **for** each FD ϕ **do**
13: $i \leftarrow 0$
14: $left\_sets \leftarrow MATCH(\phi)$
15: **for** each $lset$ in $left\_sets$ and each $rset$ in the tuple sets of $S[RHS(\phi)]$ **do**
16: **if** $|lset \cap rset| < |lset|$ and $lset \cap rset \neq 0$ **then**
17: $sets_{\phi i} += lset \cap rset$
18: $i \leftarrow i + 1$
19: **for** each variable CFD ϕ **do**
20: $temp \leftarrow MERGE(LS_\phi[0], LS_\phi[1])$
21: **for** each $LS[i]$ and $i > 1$ **do**
22: $temp \leftarrow MERGE(temp, LS_\phi[i])$
23: $i \leftarrow i + 1$
24: $LS \leftarrow temp$
25: **return** V

$right\_set = tupleset\_A = tp[A]$. The inconsistencies can be calculated easily by $left\_set - right\_set$.

In the fourth step (line 12-32), we detect inconsistencies violating the variable RHS rules. In this step, we first consider the FDs. The MATCH procedure returns tuple sets in which the tuples agree on the LHS of ϕ. After getting $left\_sets$ from MATCH module and $right\_sets$ from $S[RHS(\phi)]$, we detect the violations with both $left\_sets$ and $right\_sets$. The tuples violating the variable RHS rules are not independent. We know that the tuples in the same $left\_sets_i$ and $right\_sets_j$ have the same attribute values in the LHS and RHS of ϕ, respectively. For each $left\_sets_i$, if it is the subset of $right\_sets_j$, there is no violation in $left\_sets_i$. Otherwise, there exist violations in $left\_sets_i$. In order to check whether $left\_sets_i$ is the subset of $right\_sets_j$ or not, we compare the size of $|left\_sets_i|$ and $|left\_sets_i \cap right\_sets_j|$. If $|left\_sets_i| > |left\_sets_i \cap right\_sets_j| \neq 0$, then the tuples in $left\_sets_i \cap right\_sets_j$ must be inconsistencies. After all the FDs are detected, we start checking the variable CFDs. For each variable CFD, if the embedded FD of the CFD is detected before, we only need to check the detected result with the constant in the CFD. Otherwise, we detect the inconsistencies just in the way that we detect violations of FDs.

Example 6: Consider the data in Table 1 and the rules in Example 1 again.

TABLE 4. Grouped records.

$tupleset_{Dept=1} = \{t1, t2, t3, t4, t5, t6\}$
$tupleset_{Title=1} = \{t1, t2\}$
$tupleset_{Title=2} = \{t3, t5, t6\}$
$tupleset_{Title=3} = \{t4\}$
$tupleset_{BP=1000} = \{t1, t2, t3\}$
$tupleset_{BP=2000} = \{t4, t5, t6\}$

First, we obtain the following $Aset$ with the algorithm RANGE which will be explained in detail in Section IV-B. $Aset = \{(Dept, \_), (Title, \_), (BP, \_)\}$.

Second, we get the tuple set calculated with the algorithm GROUP which will be explained in detail in Section IV-C. The grouped tuple sets are in Table 4.

The tuple set contains numerical tuple IDs, and the attribute value is an index of the grouped records, so the list of records is much smaller than the origin database. With the help of the list, we can easily detect inconsistencies with some intersection and difference operations. Then, we detect inconsistencies with the records in Table 4. The detection with CFD: $\phi_2(R : Dept, Title \rightarrow BP, (1, 2 \| 2000))$ is transformed into the following calculation.

$$\{tupleset_{Dept=1} \cap tupleset_{Title=2}\} \backslash tupleset_{BP=2000} = \{t3\}.$$

The detection with FD:$\phi_1(R : Dept, Title \rightarrow BP, (\_, \_ \| \_))$ is a little more complicated than CFD, since the RHS BP is not a constant value. We have to find the tuples matching the FD on the LHS, and no matching on the RHS. First, the tuples matching the LHS can be calculated with the following two intersections:

(1)intersection $tupleset_1$: $(tupleset_{Dept=1} \cap tupleset_{Title=1}) = \{t1, t2\}$

(2)intersection $tupleset_2$: $(tupleset_{Dept=1} \cap tupleset_{Title=2}) = \{t3, t5, t6\}$

We do not need to calculate the intersection($tupleset_{Dept=1} \cap tupleset_{Title=3}$), since $tupleset_{Title=3}$ only contains one tuple. As we know, one tuple is impossible to cause violation when detecting with variable CFDs.

If the intersection tuple sets contain more than one tuple, we use the results to check that whether the intersection is a subset of a RHS tuple set or not.

(1)$(tupleset_1 \cap tupleset_{BP=1000}) = \{t1, t2\} = tupleset_1$. That is, $tupleset_1$ is the subset of $tupleset_{D=d1}$. Thus, there is no violation in $tupleset_1$.

(2)$(tupleset_2 \cap tupleset_{BP=1000}) = \{t3\} \neq tupleset_2$, and $\{t3\} \neq \emptyset$. $tupleset_2$ is not the subset of any tuple set of attribute BP. Hence, $\{t3\}$ must cause violation.

(3)$(tupleset_2 \cap tupleset_{BP=2000}) = \{t5, t6\} \neq tupleset_2$, and $\{t5, t6\} \neq \emptyset$.

As the result, we obtain the inconsistencies of the rules:

The violation of $\phi 2 : \{t3\}$.

The violation of $\phi 1 : \{t3\}$ conflicts with $\{t5, t6\}$.

B. RANGE

This module aims to get the attributes and values involved in the rules. We get the set of the attributes and its values from the rules to reduce the data to be checked.

Algorithm 2 Range

```
1:  Input: a set of rules ∑.
2:  Output: attribute values involved in the rules.
3:  for each rule φ do
4:      for each attribute A ∈ attr(φ) do
5:          if A does not exist in Aset then
6:              Aset+ = A
7:              valueset_A+ = tp[A]
8:          else if valueset_A ≠ "_" then
9:              if tp[A] is "_" then
10:                 valueset_A ← {"_"}
11:             else if tp[A] ∉ valueset_A then
12:                 valueset_A+ = tp[A]
```

In the attribute set $S = \sum(attributeA, valuesetV)$, each attribute A is involved in the set $attr(\sum \phi)$. We check all the attributes and their values in the rules. If an attribute A is not in the attribute set $Aset$, we then store A and its value in $Aset$ and $valueset_A$ respectively (Line 5-7). Else if $valueset_A$ contains constant values, we need to check that whether the new value is a variable "_" or not (Line 8). If the answer is yes, we use a variable "_" to replace the values in $valueset_A$, otherwise, we store the distinct value in the $valueset_A$ (Line 9-12). In order to capture the distinct values, we use a hash table to store the values for each attribute. That is, the complexity of inserting a value is $O(1)$. The space complexity of this module is $O(|attr(\sum \phi)|)$ and the time complexity of this module is $O(S)$, where S is the number of the rules.

Example 7: In Table 1, we detect inconsistencies with following three rules:

$$\phi 1(R : Dept, Title \rightarrow BP, (\_, \_\|\_)),$$
$$\phi 2(R : Dept, Title \rightarrow BP, (1, 2\|2000)),$$
$$\phi 3(R : IL \rightarrow AT, (10\|20))$$

The attributes set $Aset$ of the above three rules contains five pairs of attribute and value set.

$$Aset = \{(Dept, \_), (Title, \_), (BP, \_), (AT, \{20\}), (IL, \{10\})\}.$$

C. GROUP

In the range module, we get the attribute set ($Aset$). For each attribute A, there is a value set ($Vset$) which consists of both the value of A and the tuple set ($Tset$), and all the tuples in the $Tset$ agree on the same value in attribute A. It is easy to get the tuples with certain value of certain attribute from the result.

The pseudo code of this step is shown in Algorithm 3. We store the attributes with constant values in $Aset$ and insert its value in the $Vsets$. If an attribute has no constant value, we only store the attribute in $Aset$ (line 3-10). We create a hash for each attribute involved in the rules. Then we scan each tuple in database. If the value does not exist in the hash indices, we store the value in $Vset$ and insert the tuple ID into its $Tset$ (line 11-22). Meanwhile, we insert the value into the bucket in the hash table of the attribute.

Algorithm 3 Group

```
1:  Input: a database D and the attribute set (Aset)
2:  Output: the data set S grouped by values and attributes
3:  for all attribute A and its value set V in Aset do
4:      store A in S
5:      if the value set of A is not "_" then
6:          for all value a in the value set of A do
7:              Vset_A+ = a
8:  for all tuple t ∈ D do
9:      for all A ∈ Aset do
10:         if the value set V of A is "_" or t[A] ∈ V then
11:             if t[A] is not in Vset_A then
12:                 Vset_A+ = t[A]
13:                 Tset_{A=t[A]}+ = t
14:             else if t is not in Tset_{A=t[A]} then
15:                 Tset_{A=t[A]}+ = t
```

The time complexity of this module is $O(|T| \cdot |A|)$, where $|T|$ is the number of tuples in the database, and $|A|$ is the number of attributes involved in rules. $O(1)$ is the cost of accessing one value in an attribute. We only scan the values of the attributes involved in the $Aset$ for just once. That is, we only require scanning a subset of the database. So the time complexity of this module is bounded by $O(|T|)$, where $|T|$ is the number of tuples in database.

This module requires the only one pass of the database scan in our detection algorithm. After this module, we get the intermediate result S. S consists of a list of $Asets$, each $Aset$ consists of an attribute and a $Vset$, and each $Vset$ consists of a list of pairs ($value$, $Tset$). $S = \sum Aset(attribute, Vset)$, and $Vset = \sum(value, Tset)$. For retrieving values from S efficiently, we build a hash index for attributes and values in S, respectively. The result of this module can return the tuple set containing the tuples with a certain attribute value without accessing the original database.

D. MATCH

As discussed in Section IV-A, the MATCH module returns tuple sets in which the tuples agree on the LHS of ϕ. In the MATCH module, we consider a strategy of processing the FDs. We store the intermediate results which will be used for the detection with other rules in order to avoid redundant calculation.

As shown in Algorithm 4, if LHS(ϕ) and LHS(ϕ') share the same subset, where ϕ and ϕ' are different rules, we store the merged results in a list named processed, since that will be used to detect violations of other rules (Line 5-12). Otherwise, we just merge the LHS of the rule (Line 13-20). The function MERGE used in this module will be explain in detail in Section IV-E.

LS_ϕ is a set of the attributes or conditions in the LHS of ϕ prepared to merge. We take an FD $\phi(R : A, B, C \rightarrow D, (\_, \_, \_\|\_))$ as an example. $LS_\phi = \{\{A\}, \{B\}, \{C\}\}$ at the

Algorithm 4 Match

Input: an FD ϕ
Output: the tuple set in which tuples agree on the LHS of ϕ
$i \leftarrow$ the size of processed attributes
while $|LS_\phi| \geq 2$ **do**
  if a subset A, B in LS_ϕ exists in $LS_{\phi'}$ and ϕ' is unprocessed **then**
    $processed[i] \leftarrow MERGE(A, B)$
    for all ϕ' unprocessed **do**
      if $\{A, B\}$ is the subset of $LS_{\phi'}$ **then**
        $LS_{\phi'} \leftarrow LS_{\phi'} - \{A, B\} + \{processed[i]\}$
    $i \leftarrow i + 1$
  else
    $temp \leftarrow MERGE(LS_\phi[0], LS_\phi[1])$
    for $1 < i < |LS_\phi|$ **do**
      $temp \leftarrow MERGE(temp, LS_\phi[i])$
      $i \leftarrow i + 1$
    $LS_\phi \leftarrow temp$
return LS_ϕ

beginning, and $|LS_\phi| = 3$. After merging attributes A and B, $LS_\phi = \{\{A, B\}, \{C\}\}$ and $|LS_\phi| = 2$

Example 8: Consider the data in Table 1, and the FD $\phi 1$ in example 1. We add another FD $\phi 4$:

$$\phi 1(R : Dept, Title \rightarrow BP, (\_, \_\|\_)).$$

$$\phi 4(R : Dept, Title, AT \rightarrow TP, (\_, \_, \_\|\_)).$$

First, we got the grouped tuple sets in the GROUP module. The data set is as follows.

$$\begin{aligned} S &= \{(Dept, Vset_{Dept}), (TP, Vset_{TP}), \\ &\quad (Title, Vset_{Title}), (BP, Vset_{BP}), (AT, Vset_{AT})\} \\ Vset_{Dept} &= \{(Dept = 1, \{t1, t2, t3, t4, t5, t6\})\} \\ Vset_{TP} &= \{(TP = 1500, \{t1, t3, t4, t6\}), \\ &\quad (TP = 2500, \{t2, t5\})\} \\ Vset_{Title} &= \{(Title = 1, \{t1, t2\}), \\ &\quad (Title = 2, \{t3, t5, t6\}), (Title = 3, \{t4\})\} \\ Vset_{BP} &= \{(BP = 1000, \{t1, t2, t3\}), \\ &\quad (BP = 2000, \{t4, t5, t6\})\} \\ Vset_{AT} &= \{(AT = 10, \{t1, t2\}), (AT = 20, \{t3, t4, t5\}), \\ &\quad (AT = 30, \{t6\})\} \end{aligned}$$

Then we detect tuple sets which contain tuples matching the FD on the LHS. We denote the tuple set ($Tset$) in which tuples agree on the same condition set ($Cset$) as $Tset_{Cset}$. The tuple sets matching $\phi 1$ on the left side are as follows.

$$Tset_{Dept=1,Title=1}:$$
$$(Tset_{Dept=1} \cap Tset_{Title=1}) = \{t1, t2\}$$
$$Tset_{Dept=1,Title=2}:$$
$$(Tset_{Dept=1} \cap Tset_{Title=2}) = \{t3, t5, t6\}$$

As we know, the attributes *Dept* and *Title* in the LHS of $\phi 1$ also exist in LHS of $\phi 4$, so we store the tuple sets matching the LHS of $\phi 1$ in the processed set in order to avoid redundant calculation. The results can be used in detecting the tuple sets matching $\phi 4$ in LHS. We get the intersections which contain more than one tuple as follows.

$$Tset_{Dept=1,Title=1,AT=10}:$$
$$(Tset_{Dept=1,Title=1} \cap Tset_{AT=10}) = \{t1, t2\}$$
$$Tset_{Dept=1,Title=2,AT=20}:$$
$$(Tset_{Dept=1,Title=2} \cap Tset_{AT=20}) = \{t3, t5\}$$

In this example, we do not calculate tuple set containing only one tuple likes $Tset_{Dept=1,Title=2} \cap Tset_{AT=30}$, since one tuple cannot violate a constant RHS rule alone. This strategy will be explained in detail in Section IV-E.

E. MERGE

This module is the most important part in the algorithm for detecting inconsistencies violating variable RHS rules. And this module dominates cost of inconsistency detection based on CFDs.

This block is a sub-module of the MATCH block. Merging two attributes A and B means to find the tuples agreeing on both A and B. The results contain a list of tuple sets. Each tuple set has a condition set. All the tuples in the same tuple set agree on same condition. For example, given a tuple set $\{t1, t2, t3\}$ and its condition set $\{A = 1, B = 2\}$, we mean $t1[A] = t2[A] = t3[A] = 1$, $t1[B] = t2[B] = t3[B] = 2$. To get the result merged by attributes A and B, we need to check each value a for A and each value b for B from the result of the GROUP module. $Tset_{A=a,B=b}$ is merged by $Tset_{A=a}$ and $Tset_{B=b}$. $Tset_{A=a,B=b} = Tset_{A=a} \cap Tset_{B=b}$.

The pseudo code of the merge algorithm is shown in Algorithm 5. We get two tuple sets from the two lists of tuple sets of the two attributes, separately. Then we check the size of these two tuple sets. If both of them and their intersection contain more than one tuple, we store the intersection tuple set in the list of tuple sets which is prepared as the output of this algorithm. Meanwhile, we store the conditions of the two tuple sets together as the condition set of the intersection

Algorithm 5 Merge

Input: two lists of tuple sets according to two different attributes sets M, N
Output: a list of tuple sets matching on all attributes in the input two lists
$i \leftarrow 0$
for all $m \in V\_set_M$ and $n \in V\_set_N$ **do**
  if $|tuple\_set_{M=m} \cap tuple\_set_{N=n}| > 1$ **then**
    $Cset_i \leftarrow \{M = m, N = n\}$
    $Tset_{Cset_i} \leftarrow tuple\_set_{M=m} \cap tuple\_set_{N=n}$
    $i \leftarrow i + 1$
return $Tset_{Cset_k}(k = 0, 1, .., i)$

(Line 6-7). The loop continues until all the tuple sets in one list make intersection with all the tuple sets in the other lists.

Example 9: We use the grouped result in Table 4, and we merge attributes *Dept* and *Title*:

$$(1)Tset_{Dept=1,Title=1} = \{t1, t2\},$$
$$(2)Tset_{Dept=1,Title=2} = \{t3, t5, t6\}.$$

We do not need to calculate the intersection ($Tset_{Dept=1} \cap Tset_{Title=3}$), since $Tset_{Title=3}$ contains only one tuple. As we know, one tuple is impossible to cause violation when detecting with variable CFD.

Then we analyze the cost of this block MERGE. As the tuples in $Tset_{A=a}$ and $Tset_{B=b}$ are ordered by the tuple ID, the cost of ($Tset_{A=a} \cap Tset_{B=b}$) is $O(M + N)$, where $M = |Tset_{A=a}|$ and $N = |Tset_{B=b}|$. We assume that there are p values $\{a_1, a_2, ..., a_p\}$ whose tuple set $|Tset_{A=a_i}| > 1$ in attribute A and q values $\{b_1, b_2, ..., b_q\}$ whose tuples set $|Tset_{B=b_i}| > 1$ in B. We only care about the tuple set containing more than one tuple, since one tuple causes no violation of a variable RHS rule.

$C(Merge(A, B))$ denotes the cost of *merge*(attribute A,attribute B)

$$C(Merge(A, B)) \tag{1}$$
$$= \sum_{1\le i\le p, 1\le j\le q} cost(Tset_{A=a_i} \cap Tset_{B=b_j}) \tag{2}$$
$$= \sum_{1\le i\le p} q \cdot |Tset_{A=a_i}| + \sum_{1\le j\le q} p \cdot |Tset_{B=b_j}| \tag{3}$$
$$= q \cdot \sum_{1\le i\le p} |Tset_{A=a_i}| + p \cdot \sum_{1\le j\le q} |Tset_{B=b_j}| \tag{4}$$
$$\le q \cdot tuplesnum + p \cdot tuplesnum = (p + q) \cdot tuplesnum \tag{5}$$

It shows that $C(Merge(A, B)) \le (p+q) \cdot tuplesnum$, where p is the number of distinct values in the tuple set containing more than one tuple in the grouped result of attribute A, and q is that in attribute B, *tuplesnum* is the number of tuples in the database. If both the value sets of A and B we got from the RANGE module are "_", $C(Merge(A, B)) = (p + q) \cdot tuplesnum$. The cost is affected by both the *tuplesnum* and the size of value sets of the attributes need to be merged. We prove the following proposition and propose a conception of redundancy to help us analyse the complexity of this module.

Theorem 1: The number of distinct values of one attribute with the $|Tset| > 1$ is no more than the $tuplesnum/2$.

Proof 1: We assume k as the number of distinct values of one attribute A with $|tupleset| > 1$. It means that there is a set S containing k tuplesets, and the size of each tupleset is at least 2. As we know, each tuple in the tupleset is different with the tuples in both the same tupleset and other tuplesets. The sums of all the $|tuplesets|$ of attribute A equals *tuplesnum*, then $\sum_{1\le i\le k} |S[i]| \le tuplesnum$. If $k > tuplesnum/2$, then $\sum_{1\le i\le k} |S[i]| > 2k > tuplesnum$ which conflicts with the result above. The assumption is not supposed, and the proposition is true.

Then we can give the upper bound of the p and q mentioned before. $p \le tuplenum/2$, and $q \le tuplenum/2$. $C(Merge(A, B)) \le (p + q) \cdot tuplenum \le (tuplenum)^2$. Thus the cost of this module $C(Merge(A, B))$ is bounded by both the tuplesnum and the number of distinct values in A, B. It also has an upper bound $O(|T|^2)$, the worst-case complexity, which is impossible in reality.

Definition 9: Redundancy: We define the redundancy (RDD) of an attribute with the rate $RDD(attributeA) = (1 - D_A/T)$, where D_A is the number of distinct values on attribute A, and T is the number of tuples in database. The redundancy of a database R is defined as $RDD(database) = (1 - \frac{1}{T} \sum_{A_i \in attr(R)} D_{A_i})$

According to the definition of redundancy, we can learn that database with high RDD must have a small number of distinct values. The RDD of some attributes in the real world may be very high. For example, in a relation of personal information, the RDD of an attribute "gender" must be very high, since there are only two distinct values "male" and "female". And in the real world, we only use FDs as constraints for the attributes with high RDD. There is no significance to use an FD for attributes such as "ID" whose values are all distinctive. In real world, the RDD of the dataset which can be restricted by FDs may be very high even close to 1.

With the help of redundancy, we show the complexity of this module. The time complexity of this module is $O(|D|\cdot|T|)$ where $|T|$ is the number of tuples in database, and $|D|$ is the number of distinct conditions preparing to be merged. After the definition of redundancy, we can use RDD to express $|D|$, $|D| = (|T| - RDD \cdot |T|)$. As in real world big data RDD is very likely to be high and even close to 1, meanwhile, $|D|$ will be much smaller than $|T|$, then the cost of this module can be performed in $O(T)$ time.

In the MERGE part, we discussed the complexity when there are two attributes involved in the LHS of a variable RHS rule. The cost of merging two attributes is $C(Merge(A, B)) \le (p+q) \cdot tuplesnum$, where p and q are the numbers of distinct values of attribute A and B respectively. For the variable RHS rule with more than two attributes on the left, we first merge the first two attributes and get the merged result. Then, we repeatedly merge the result with a new attribute until all the attributes are processed. Assume that there are three attributes A, B, C on the LHS, and the numbers of their distinct values are a, b, c. We first merge A and B, and the cost is $C(Merge(A, B)) \le (a + b) \cdot tuplesnum$. Then we merge the result and the attribute C. Suppose that the merged result of A and B contains m distinct value combinations, the cost in this step is $C(Merge((A, B), C)) \le (m + b) \cdot tuplesnum$. The upper bound of m is $a \cdot b$, but it is usually much smaller than that, since some value combinations may not cover more than one tuple. For example, if we merge the two attributes *Title* and *AT* in table 1, the result only contains two tuple sets: $Tupleset_{Title=1,AT=10} = \{t1, t2\}$ and $Tupleset_{Title=2,AT=20} = \{t3, t5\}$. The result does not contain

$Tupleset_{Title=2,AT=30} = \{t6\}$ and $Tupleset_{Title=3,AT=20} = \{t4\}$, since there is only one tuple in each of them. When the redundancy is high, the number of distinct values of each attribute involved in the rule is small, and the number of distinct value combinations may not be very large. That is, in each iteration, the cost is nearly linear.

In this module, we just use the grouped result S instead of the original database. We can get the tuple set with certain attribute value in constant time with the help of the hash index. The cost of one tuple set retrieve is $O(1)$. The cost of retrieving all tuple sets for the detection is bounded by $O(|\sum attr(\phi)|)$, where $|\sum attr(\phi)|$ is the number of attributes in the rules. This cost is much smaller than the cost of scanning the whole database. That is also a reason for the efficiency of our detection method.

F. CORRECTNESS

1) THE CORRECTNESS OF THE ALGORITHM FOR CONSTANT RHS RULES

Suppose we have a constant RHS rule $\phi(R : X \rightarrow A, tp)$, the tuples matching the LHS of this rule are in the set $Leftset = \{t|t[X] = tp[X], t \in R\}$, and the tuples matching the RHS of this rule are in the set $Rightset = \{t|t[A] = tp[A], t \in R\}$. Then the inconsistencies TS can be calculated by $(Leftset - Rightset)$.

Proof 2: In order to prove the algorithm for constant RHS rules, we need to prove that (1) $TS \subset (Leftset - Rightset)$: all the inconsistencies of this rule are in the set $(Leftset - Rightset)$, (2) $(Leftset - Rightset) \subset TS$: each tuple in the set $(Leftset - Rightset)$ is inconsistent.

(1)Assume that there exists one inconsistent tuple t not in the set $(Leftset - Rightset)$. Then, $t[X] = tp[X]$ and $t[A] \neq tp[A]$, that means $t \in Leftset \notin Rightset$. Therefore, $t \in (Leftset - Rightset)$. However, by the above assumption $t \notin Leftset - Rightset$. This contradiction leads us to conclude that the original assumption is incorrect. That is, $(Leftset - Rightset)$ covers all the inconsistencies of ϕ.

(2)Assume that there exists one tuple t in $(Leftset - Rightset)$ which is not the inconsistency. $t \in Leftset$ and $t \notin Rightset$ due to the assumption. That is, $t[X] = tp[X]$ and $t[A] \neq tp[A]$. Then, t is the inconsistent tuple according to the definition of inconsistency. This contradiction leads us to conclude that the original assumption is incorrect. That is, each tuple in the set $(Leftset - Rightset)$ is inconsistent.

2) THE CORRECTNESS OF THE ALGORITHM FOR VARIABLE RHS RULES

Suppose we have a variable RHS rule $\phi(R : X \rightarrow A, tp)$, $x_1, x_2, ..., x_p$ are all the distinct value combinations of the LHS, $a_1, a_2, ..., a_q$ are all the distinct values of the RHS. The tuples matching the LHS of this rule are in the set $Leftset_1, Leftset_2, ..., Leftset_p$, $Leftset_i = \{t|t[X] = x_i, t \in R\}$, and the tuples matching the RHS of this rule are in the sets $Rightset_1, Rightset_2, ..., Rightset_q$, $Rightset_j = \{t|t[A] = a_j, t \in R\}$. The inconsistencies are contained by a list of sets:$Iset_1, Iset_2, ..., Iset_p$, $Iset_i = \{Leftset_i \cap Rightset_j | Leftset_i \cap Rightset_j \neq 0, Leftset_i \cap Rightset_j \neq Leftset_i, 1 \leq j \leq q\}$. Tuples in each member of the $Iset_i$ conflict with other tuples in other members of the same $Iset_i$.

Proof 3: In order to prove the correctness of the algorithm for variable RHS rules, we need to prove that (1) each inconsistent tuple pair of this rule is contained in one of the *Isets*, (2) each tuple in the *Isets* conflicts with others.

(1)Assume that there exists a inconsistent tuple pair: $t1$,$t2$, and they are not contained by any $Iset_i$. We can know that $t1[X] = t2[X]$ and $t1[A] \neq t2[A]$ since the assumption. There exists one x_i and two different values a_m, a_n that $t1[X] = t2[X] = x_i$ and $t1[A] = a_m, t2[A] = a_n$. Then, $t1 \in (Leftset_i \cap Rightset_m)$ and $t2 \in (Leftset_i \cap Rightset_n)$. As the description above, $(Leftset_i \cap Rightset_m)$ and $(Leftset_i \cap Rightset_n)$ are different members of the same $Iset_i$. That is, we can we can find a $Iset_i$ whose two different members contain the tuple $t1$ and $t2$, respectively. The result contradicts the assumption.

(2)Assume that there exist two tuples $t1$, $t2$ where $t1$ and $t2$ are in the different members $(Leftset_i \cap Rightset_m)$, $(Leftset_i \cap Rightset_n)$ of the same $Iset_i$, and they are not inconsistent pair. It is clear from the assumption that $t1 \in (Leftset_i \cap Rightset_m)$ and $t1 \in (Leftset_i \cap Rightset_n)$. Then $t1 \in Leftset_i$, $t1 \in Rightset_m$, $t2 \in Leftset_i$, $t2 \in Rightset_n$. That is, $t1[X] = t2[X] = x_i$ and $t1[A] = a_m$, $t2[A] = a_n$. Since $(Leftset_i \cap Rightset_m)$ and $(Leftset_i \cap Rightset_n)$ are different, we can know that $m \neq n$ and $t1[A] \neq t2[A]$. Since $t1[X] = t2[X]$ and $t1[A] \neq t2[A]$, $t1$ and $t2$ conflict with each other. This contradiction leads us to conclude that the original assumption is incorrect.

V. DETECTING THE VIOLATION OF DC

We detect the violation of denial constraints with the result of the GROUP Module in the previous section. We propose two algorithms for constant DCs and variable DCs in Section V-A and Section V-B, respectively.

A. THE VIOLATION OF CONSTANT DC

The violation of constant DC can be detected easily with the result of GROUP Module. The process is similar to the detection with constant CFD. The pseudo code of our algorithm is shown in Algorithm 6.

We first get the result of the Group Module (Line 3). We can get the tuples matching each predicate simply with the grouped data. As all the predicates of one DC cannot be true at the same time due to the definition of DC, we can get the violation by capturing the tuples matching all the predicates at the same time. Taking one predicate into consideration, we get the tuples matching the predicate $t[A]\theta c$ by calculating the union of all the tuple sets whose value v of attribute A satisfies the predicate $v\theta c$. Then we get the tuples matching all the predicates by calculating the intersection of the tuples matching each predicate (Line 7). At last, each DC corresponds to one inconsistency set containing the tuples violating the DC.

Algorithm 6 The Inconsistency Detection Algorithm With Constant DC

1: **Input:** a relation R, a list of DCs $\sum$
2: **Output:** $\{V_1, V_2...V_m\}$, V_i is the set of tuples violating the ith DC in $\sum$
3: $G = Group()$ //We use $G(A, a)$ to indicate the tuples matching the condition $A = a$
4: **for** each DC ϕ_i in $\sum$ **do**
5: $V_i = \emptyset$
6: **for** each $A_i = \{t[A]\theta c\}$ in ϕ_i **do**
7: $Set\cap = \{\cup G[A, a]|a\theta c\}$
8: $V_i = Set$
9: **return** $\{V_1, V_2...V_m\}$

Example 10: Consider one constant DC ϕ: $\forall t, \neg(t[Title] \geq 2 \wedge t[BP] \geq 2000)$. We use this denial constraint to detect the inconsistency in Table 1.

We first get the tuples matching the predicates $t[Title] \geq 2$ and $t[BP] \geq 2000$ from the grouped table (Table 4), respectively.

(1)Tuples matching the predicate $t[Title] \geq 2$:

$$\begin{aligned} tupleset_{t[Title]\geq 2} &= tupleset_{Title=2} \cup tupleset_{Title=3} \\ &= \{t3, t4, t5, t6\} \end{aligned}$$

(2)Tuples matching the predicate $t[BP] \geq 2$:

$$tupleset_{t[BP]\geq 2} = tupleset_{BP=2000} = \{t4, t5, t6\}$$

The inconsistency of DC ϕ:

$$tupleset_{t[Title]\geq 2} \cap tupleset_{t[BP]\geq 2000} = \{t4, t5, t6\}.$$

B. THE VIOLATION OF VARIABLE DC

The inconsistency detection with variable DCs is more complex than that with constant DCs. In this section, we only consider the variable DCs in the form: $\forall t_\alpha, t_\beta, \neg(t_\alpha.A_1\theta_1\ t_\beta.A_1 \wedge ... \wedge t_\alpha.A_m\theta_m t_\beta.A_m)$, where $A_1...A_m$ is the attributes in R.

FD is a special case of DC when all the operators are '=' except one '$\neq$'. In this part, we first consider the $\theta_i \in \{<, >, \leq, \geq\}$. Then we extend the method to take operators $\{=, \neq\}$ into consideration. The extension can be processed easily by finding the tuple pairs matching the predicates whose operators are in $\{=, \neq\}$ from the result pair list consisting of tuples matching the predicates whose operators are in $\{<, >, \leq, \geq\}$.

For example, the inconsistent pairs of $\forall t1, t2\neg(t1[H] > t2[H] \wedge t1[T] > t2[T])$ in Table 5 is $(t1, t5)$, $(t2, t5)$, $(t6, t5)$, $(t4, t5)$. If we add a predicate in the DC and convert that into $\forall t1, t2\neg(t1[H] > t2[H] \wedge t1[T] > t2[T] \wedge t1[P] = t2[P])$, the new inconsistent pairs can be generated from the forward list by checking that whether the two elements in each pair are in the same tuple set in the grouped result of the attribute in the new predicates. Then we get the result $(t1, t5)$, since they are in the same tuple set $tupleset_{P=77} = \{t1, t5\}$.

TABLE 5. A elevation relation.

id	H	T	P
t1	2500	15	77
t2	2500	12.5	72
t3	1000	20	88
t4	1500	17.5	83
t5	3000	20	77
t6	2000	15	76

As we only take $\{<, >, \leq, \geq\}$ into consideration in this part, the denial constraints can be simplified to the form: $\neg(A_1\theta_1 \wedge ... \wedge A_m\theta_m)$. We find the violation of the rule by capturing the tuples matching $((A_1\theta_1) \wedge ... \wedge (A_m\theta_m))$.

Then a denial constraint can be simply converted into a SQL query. The denial constraint simplified as the $\neg((A >) \wedge (B >))$ meaning that if $t1[A] > t2[A]$ and $t1[B] > t2[B]$, then the $t1$ and $t2$ conflict with each other. We use a tuple pair $(t1, t2)$ to indicate the inconsistency. The violation of each variable DC is a list of inconsistent pairs.

Example 11: Consider the elevation relation in Table 5. There are three attributes involved in the table: Height(H), Temperature(T), Pressure(P). Assume that we have a denial constraint $\forall t1, t2\neg(t1[H] > t2[H] \wedge t1[T] > t2[T])$, which can be simplified as $\neg((H >) \wedge (T >))$. We attempt to get the inconsistencies by capturing the tuple pair$(t1, t2)$, where $(t1[H] > t2[H] \wedge t1[T] > t2[T])$. This denial constraint can be simply converted into the following SQL query.

Q_V : `select` $t1.id, t2.id$ from $R\ t1, R\ t2$
`where` $t1[H] > t2[H]$ AND $t1[T] > t2[T]$

However, the efficiency of this query is very low. This query is processed as a Cartesian product, meaning that the time complexity is $O(n^2)$. We attempt to capture a more efficient algorithm which is suitable for detecting the inconsistencies with DCs in big data. The time complexity of the new algorithm can be reduced to $O(n \cdot logn)$.

1) THE FRAMEWORK

In this part, we describe the framework of the algorithm to capture the inconsistencies with variable DCs. This algorithm starts from an extension of GROUP module, named GROUP'. The only difference between the GROUP' and GROUP is that we store the relative positions with the tuple IDs. Then we can get the violation from the relative position without checking the database. We proved a theorem to convert the problem of capturing the inconsistent tuple pair into getting the ascending ordered pair in a permutation.

We use the idea in [38] and [39] to get the ascending ordered pair. In [39], James proposed an $O(n \cdot logn)$ algorithm to solve the common subsequences problem. Khayyat *et al.* [38] proposed a fast inequality join method. They put columns to be joined in sorted arrays and use permutation arrays to encode positions of tuples in one sorted array. As the sorting can be processed in the GROUP module, we only need to adjust the GROUP module to capture the permutation array together with certain tuple sets.

TABLE 6. Group result of elevation relation.

$tupleset_{H=1000} = \{t3\}$
$tupleset_{H=1500} = \{t4\}$
$tupleset_{H=2000} = \{t6\}$
$tupleset_{H=2500} = \{t1, t2\}$
$tupleset_{H=3000} = \{t5\}$
$tupleset_{T=12.5} = \{t2\}$
$tupleset_{T=15} = \{t1, t6\}$
$tupleset_{T=17.5} = \{t4\}$
$tupleset_{T=20} = \{t3, t5\}$

TABLE 7. Group' result of elevation relation.

$tupleset_{T=12.5} = \{(t2, 4)\}$
$tupleset_{T=15} = \{(t1, 4), (t6, 3)\}$
$tupleset_{T=17.5} = \{(t4, 2)\}$
$tupleset_{T=20} = \{(t3, 1), (t5, 5)\}$

Algorithm 7 The Inconsistency Detection Algorithm With Variable DC

1: **Input:** a relation R, a variable DC ϕ
2: **Output:** a list V of tuple pair (ti, tj).
3: $G = GROUP'()$ // We add the relative position to each tuple stored in the grouped result.
4: **for** $i = 1$ to m **do** //m is the number of predicate A_i in ϕ
5: $V \cap = Relative\_Position(A_0, A_i, G)$
6: **return** V

We first describe the framework of algorithm with the pseudo code in Algorithm 7 in detail. In the third line of Algorithm 7, we extend the Group Module and store the relative positions with the tuple IDs. In order to get the relative positions, we first sort the tuples by the two attributes, respectively. Then we store the relative positions with the tuple IDs. We can get the tuples matching two predicates by the algorithm *RelativePosition*. Then we can capture the violation by calculating the intersection of the results of *RelativePosition*.

In order to explain the function of GROUP', we make the following example.

Example 12: Consider that we have the DC in last example, which can be simplified as $\neg((H\ >) \wedge (T\ >))$. We first group and sort (in ascending order) the tuples by two attributes respectively:

Group and sort by H:

$$1\{t3\}, 2\{t4\}, 3\{t6\}, 4\{t1, t2\}, 5\{t5\}.$$

Group and sort by T:

$$1\{t2\}, 2\{t1, t6\}, 3\{t4\}, 4\{t3, t5\}.$$

The relative position of T in H :

$$\{(t2, 4)\}, \{(t6, 3), (t1, 4)\}, \{(t4, 2)\}, \{(t3, 1), (t5, 5)\}.$$

Then we can get the tuples matching the two predicates by capturing the ascending ordered pair(making comparison with the second member of each item) in the relative position list $\{(t2, 4)\}, \{(t6, 3), (t1, 4)\}, \{(t4, 2)\}, \{(t3, 1), (t5, 5)\}$. The ascending ordered pairs are $((t2, 4), (t5, 5)), ((t6, 3), (t5, 5)), ((t1, 4), (t5, 5)), ((t4, 2), (t5, 5))$, and the corresponding inconsistent tuple pairs are $(t2, t5), (t6, t5), (t1, t5), (t4, t5)$. As we know, we can get the inversion pairs easily with a traditional function MergeSort. We proposed a function MSort developed from the MergeSort to capture the ascending ordered pairs. The function MSort is described in Algorithm 10. For instance, in the example above, we do not take the ascending ordered pair $((t6, 3), (t1, 4))$ and $((t3, 1), (t5, 5))$ into consideration since the operator of tuple T is "$>$" and the two members of each pair are in the same set. If the DC is changed as $\neg((H\ >) \wedge (T\ \geq))$, then the ascending ordered pair $((t6, 3), (t1, 4))$ and $((t3, 1), (t5, 5))$ should be taken into consideration, and the violation pairs should include $(t1, t6)$ and $(t1, t5)$. We first introduce two definitions before we analyse the different cases of a rule.

Definition 10: Ascending Ordered Pairs: Suppose that A and B are two members in a permutation and their positions in the permutation are $A.id$ and $B.id$ respectively. (A, B) is called an ascending ordered pair if $A.id < B.id$ and $A \leq B$.

Definition 11: Strictly Ascending Ordered Pairs: Suppose that A and B are two members in a permutation and their positions in the permutation are $A.id$ and $B.id$, respectively. (A, B) is called a strictly ascending ordered pair if $A.id < B.id$ and $A < B$.

Taking a DC containing two attributes as an example $\neg((H\theta_1) \wedge (T\theta_2))$, and the relative position of T in H is in a permutation $\{(t2, 4)\}, \{(t6, 3), (t1, 4)\}, \{(t4, 2)\}, \{(t3, 1), (t5, 5)\}$. We classify the rule into four cases according to the two operators.

(1) $\theta_1 =$ '$\geq$' and $\theta_2 =$ '$\geq$'. In this case, the inconsistencies are ascending ordered pairs in the relative position permutation: $((t2, 4), (t1, 4)), ((t2, 4), (t5, 5)), ((t6, 3), (t1, 4)), ((t6, 3), (t5, 5)), ((t1, 4), (t5, 5)), ((t4, 2), (t5, 5)), ((t3, 1), (t5, 5))$.

(2) $\theta_1 =$ '$>$' and $\theta_2 =$ '$\geq$'. In this case, the inconsistencies are the strictly ascending ordered pairs in the relative position permutation: $((t2, 4), (t5, 5)), ((t6, 3), (t1, 4)), ((t6, 3), (t5, 5)), ((t1, 4), (t5, 5)), ((t4, 2), (t5, 5)), ((t3, 1), (t5, 5))$. We can know that $((t2, 4), (t1, 4))$ is not the strictly ascending ordered pair. It is not the inconsistent pair since their positions in T are same.

(3) $\theta_1 =$ '$\geq$' and $\theta_2 =$ '$>$'. In this case, the inconsistencies are ascending ordered pairs in the relative position permutation and the two members of each pair are from different partitions: $((t2, 4), (t1, 4)), ((t2, 4), (t5, 5)), ((t6, 3), (t5, 5)), ((t1, 4), (t5, 5)), ((t4, 2), (t5, 5))$. We can know that $((t6, 3), (t1, 4))$ and $((t3, 1), (t5, 5))$ are not the inconsistent pairs since the two members of each pair are in the same partition.

(4) $\theta_1 =$ '$>$' and $\theta_2 =$ '$>$'. In this case, the inconsistencies are the strictly ascending ordered pairs in the relative position permutation and the two members of each pair are from different partitions: $((t2, 4), (t5, 5)), ((t6, 3), (t5, 5)), ((t1, 4), (t5, 5)), ((t4, 2), (t5, 5))$. As $((t2, 4), (t1, 4))$ is not the strictly ascending ordered pair, and $((t6, 3), (t1, 4))$,

$((t3, 1), (t5, 5))$ contain two members from the same partition, they are not inconsistent pairs.

In the four cases above, we only consider '$>$' and '$\geq$' since that we can change the relative position permutation to accommodate to different operators. For example, if $\theta_1 =$ '$>$' and $\theta_2 =$ '$<$', we can just change the relative position: $\{(t2, 2)\}, \{(t1, 2), (t6, 3)\}, \{(t4, 4)\}, \{(t5, 1), (t3, 5)\}$. Then, it can be regarded as the case (4) above.

We proved that the problem in these cases can be converted into capturing the ascending ordered pairs in the relative position permutation P.

Theorem 2: When detecting the violation of a DC $\neg((A, op_1) \wedge (B, op_2))$. There is an one-one correspondence between the candidate inconsistent tuple pair and the ascending ordered pair in the permutation P. If op_1 is '$>$', the correspondence is between the candidate inconsistent tuple pair and the strictly ascending ordered pair. If op_2 is '$>$', the correspondence is between the candidate inconsistent tuple pair and the ascending ordered pair whose two members are from different partitions in the permutation.

Proof 4: We just prove the condition in case (4) in previous discussion as an example. The proof of the other three cases can be easily captured in the similar way. When detecting the violation of a DC $\neg((A >) \wedge (B >))$. There is an one-one correspondence between the candidate inconsistent tuple pair and the strictly ascending ordered pair whose two members are from different partitions in the permutation P.

We proved the theorem in the following two steps:

(1) Assume that there exists one candidate inconsistent tuple pair $(t1, t2)$, A_i, A_j are the ids of $t1, t2$ respectively in the tuple list sorted by attribute A and B_i, B_j are the ids of $t1, t2$ respectively in the tuple list sorted by attribute B. Assume $(t1, t2)$ matches the two predicates $((A >), (B >))$, meaning that $t2[A] > t1[A], t2[B] > t1[B]$.

$$ListA : ...t1(A_i)...t2(A_j)...$$
$$ListB : ...t1(B_i)...t2(B_j)...$$
$$Permutation : ...A_i(B_i)...A_j(B_j)...$$

The relative position of $t1$ in the *ListA* is A_i and the relative position of $t2$ in the *ListA* is A_j. As $t2[A] > t1[A], t2[B] > t1[B]$, we can get that $A_i < A_j$ and $B_i < B_j$. Then (A_i, A_j) is a strictly ascending ordered pair. And since $B_i < B_j$, A_i and A_j must in different partitions of the permutation.

That is, the candidate inconsistent tuple pair $(t1, t2)$ corresponds to a strictly ascending ordered pair (A_i, A_j) whose two members are in different partitions.

(2) Assume that there exists a strictly ascending ordered pair (A_i, A_j) in the permutation, and A_i A_j are in different partitions of the permutation. Suppose A_i and A_j correspond to $t1$ and $t2$ respectively. The position of A_i in the permutation is B_i and the position of A_j in the permutation is B_j. As (A_i, A_j) is a strictly ascending ordered pair, then $A_i < A_j$. As A_i A_j are in different partitions of the permutation, then $B_i < B_j$. It is clear that $t2[A] > t1[A], t2[B] > t1[B]$ since $A_i < A_j$ and $B_i < B_j$. $(t1, t2)$ is a candidate inconsistent tuple pair.

That is, one strictly ascending ordered pair (A_i, A_j) whose two members are from different partitions corresponds to one inconsistent tuple pair $(t1, t2)$.

With the help of Theorem 2, the problem of capturing the candidate inconsistent tuple pair can be converted into getting the ascending ordered pair in the permutation. We will get ascending ordered pairs in Section V-B.2, and Section V-B.3.

2) RELATIVE POSITION

In the Algorithm 8, we get a list of tuple pairs. The two tuples in one pair match the given two predicates. The permutation P contains the relative positions (positions in the list sorted by the attribute of the first predicate) of the tuples sorted by the attribute of the second predicate. The permutation can be partitioned into several parts, we use *partition* to store the start position of each part. The permutation P and *partition* are set as empty sets at the beginning (Line 3-4). In Line 5-21, we use $G[B = bi].set$ to indicate the set of pairs in which each pair is in the form $(tuple\_id, position)$. *tuple_id* is the index of the tuple matching the condition $B = bi$ and *position* is the relative position of the tuple in the list sorted by the first predicate. In Line 5-12, θ_1 and θ_2 are inverse meaning that $((\theta_1 \in \{<, \leq\})$ and $(\theta_2 \in \{>, \geq\}))$ or $((\theta_1 \in \{>, \geq\})$ and $(\theta_2 \in \{<, \leq\}))$, then we need to get the inversion pair instead of the ascending ordered pair from the permutation. In order to avoid changing the algorithm of capturing the ascending ordered pairs, we simply adjust the relative positions in the permutation. We use the number of the values of first attribute

Algorithm 8 Relative Position

1: **Input:** the first predicate of a rule $A_0 = A\theta_1$, one predicate $A_i = B\theta_2$, Group result G
2: **Output:** a list V of tuple pair (ti, tj).
3: $permutationP = \emptyset$.
4: $partation = \emptyset$
5: **if** $((\theta_1 \in \{<, \leq\})and(\theta_2 \in \{>, \geq\}))or((\theta_1 \in \{>, \geq\})and(\theta_2 \in \{<, \leq\}))$ **then**
6: **for** each value b_i of attribute B **do**
7: **for** $j = 0$ to n **do** //$n$is the number of the $G[B = b_i].set$
8: int $pos = G[A].valuenum + 1 - G[B = bi].set[j].position$
9: add $(G[B = bi].set[j].tuple\_id, pos)$ into P.
10: add $P.size()$ into $partition$.
11: **else**
12: **for** each value b_i of attribute B **do**
13: **for** $j = 0$ to n **do** //$n$is the number of the $G[B = b_i].set$
14: int $pos = G[B = bi].set[j].position$
15: add $(G[B = bi].set[j].tuple\_id, pos)$ into P.
16: add $P.size()$ into $partition$.
17: Violation(P, $partition$, θ_1, θ_2)
18: **return** V.

to minus the relative position in order to get the new position (Line 8). Then we add the pair (*tuple_id*, *position*) into the permutation set P (Line 9). After checking the tuples sharing the same attribute value, the algorithm adds the position of the next part into the *partition* set. If the two operators are not inverse (Line 13-21), then there is no need to change the relative position. We use an example to explain the first condition (Line 5-12).

Example 13: In Example 12, we get the relative positions of T in H :

$$\{(t2, 4)\}, \{(t1, 4), (t6, 3)\}, \{(t4, 2)\}, \{(t3, 1), (t5, 5)\}.$$

Consider that we change the operator of the second predicate of DC in Example 12, and get a new rule simplified as $\neg((H\ >) \wedge (T\ <))$. Then, as the two operators are inverse, we need to adjust the relative position before storing that into the permutation P. Taking the $tupleset_{T=12.5} = \{(t2, 4)\}$ as an example, the relative position of $t2$ in the list sorted by the first attribute H is 4. We get the new position by the following calculation:

$$valuenum(H) + 1 - position = 2.$$

Then we repeat this step and add the pairs $(t2, 2)$, $(t1, 2)$, $(t6, 3)$, $(t4, 4)$, $(t3, 5)$, $(t5, 1)$ into the permutation P. At the same time, these pairs are partitioned into the following four parts: $\{(t2, 2)\}, \{(t1, 2), (t6, 3)\}, \{(t4, 4)\}, \{(t3, 5), (t5, 1)\}$. The second part starts from position 2, the third part starts from the position 4 and the last part starts from position 5. We store the first position of each part (start from the second part) into the partition set: $partition = \{2, 4, 5\}$.

3) VIOLATION

The Algorithm 9 is called at the end of the Algorithm 8. This algorithm can get the ascending ordered pairs of the permutation P. If the second operator $\theta_2 \in \{\leq, \geq\}$, then the algorithm can just call the *MSort* function (Line 2). The *MSort* function is developed from the traditional *MergeSort* Function, it is easy to get the ascending ordered pairs from the process of *MergeSort*. If the second operator $\theta_2 \in \{<, >\}$, which means that we do not care the ascending ordered pair whose two members involved in the same partition. For instance, in Example 13, the permutation is: $\{(t2, 2)\}, \{(t1, 2), (t6, 3)\}, \{(t4, 4)\}, \{(t5, 1), (t3, 5)\}$, the pairs $(t1, t6)$ and $(t5, t3)$ are not included in the candidate inconsistent pair list since they are in the same partition, though their positions are in ascending order. Then, the algorithm applies the *MSort* function on each partition (Line 6), and *Merge* all the partitions together (Line 13).

Algorithm 9 Violation

violation(P, *partition*, θ_1, θ_2)

1: **if** $\theta_2 \in \{\leq, \geq\}$ **then**
2: MSort(0, $P.size - 1$, P, $theta_1$,1)
3: **else**
4: $start \leftarrow 0$
5: **for** $i = 0$ to $partition.size$ **do**
6: MSort($start$, $partition[i] - 1$, P, θ_1, 2)
7: $start = partition[i]$
8: **for** $i = 0$ to $partition.size$ **do**
9: **if** $i + 1 = partition.size()$ **then**
10: break
11: Merge(0, $partition[i] - 1$, $partition[i + 1] - 1$, P, θ_1, 1)

Algorithm 10 MSort

MSort(*left*, *right*, P, θ_1,*kind*)

1: **if** *left* $<$ *right* **then**
2: $mid = (left + right)/2$
3: MSort(*left*, *mid*, P, *kind*)
4: MSort($mid + 1$, *right*, P, *kind*)
5: Merge(*left*, *right*, *mid*, P, $theta_1$, *kind*)

Example 14: In example 12, the ascending ordered pair $(3, 4)$ and $(1, 5)$ is not involved into the pairs in ascending order. Consider that we change the second operator of DC in Example 11.and get a new DC $\neg((H >) \wedge (T \geq))$.

The relative position of T in H:

$$\{(t2, 4)\}, \{(t6, 3), (t1, 4)\}, \{(t4, 2)\}, \{(t3, 1), (t5, 5)\}.$$

We can simply apply the function *MSort* this time since the second operator is " $\geq$" instead of " $>$", there is no need to take the partition into consideration.

The ascending ordered pairs are $((t2, 4), (t5, 5))$, $((t1, 4), (t5, 5))$, $((t6, 3), (t5, 5))$, $((t4, 2), (t5, 5))$, $((t6, 3), (t1, 4))$, $((t3, 1), (t5, 5))$ and the corresponding inconsistent tuple pairs are $(t2, t5)$, $(t1, t5)$, $(t6, t5)$, $(t4, t5)$, $(t6, t1)$, $(t3, t5)$.

The *MSort* and *Merge* function (Algorithm 10,11) are developed from the traditional *MergeSort* function. We just adjust the function *Merge*. If the first operator $\theta_1 \in \{<, >\}$, then the *op* in Line 9 (Function *Merge*) is " $\geq$", which means that the pair $(t1, t2)$ where $t1[A] = t2[A]$ would not be included in the candidate inconsistent pairs list. Else, the operator *op* is " $>$". The variable *kind* in the input indicates the situation in the function *Violation*. If $kind = 2$, meaning that the current pair is from the same partition, then the algorithm would not add the pair into the candidate violation list. If $kind = 1$, then the algorithm will add the pair into the inconsistent pairs list (Line 13-17 in Function *Merge*).

After introducing the two algorithms for detecting the violation of DCs, we analyze the complexity and correctness of the algorithms. As the complexity of GROUP module has been analyzed before, we just discuss the complexity after the GROUP module.

Complexity Analysis:

The complexity of the detection algorithm with the constant DCs:

Time Complexity: It is easy to detect the inconsistencies with constant DCs, similar to detect inconsistencies with

Algorithm 11 Function Merge

Merge($left$, mid, $right$, P, θ,$kind$)

```
temp = ∅
i = left, j = mid + 1
while i ≤ mid and j ≤ right do
    if θ ∈ {<, >} then
        op = " >="
    else
        op = " >"
    if P[i].position op P[j].position then
        add P[i] into temp
        i ← i + 1
    else
        if kind = 1 then
            for m = i to mid do
                add (P[m], P[j]) into V
        add P[j] into temp
        j ← j + 1
for p = i to mid do
    add P[p] into temp
for q = j to right do
    add P[q] into temp
i = 0
for k = left to right do
    P[k] ← temp[i]
    i ← i + 1
```

constant RHS rules. With the help of the grouped result, we can capture the tuples matching certain predicate in time $O(1)$. The rest of the job is to calculate a list of intersections. As the ids of tuples in each tuple set are sorted in order, the complexity of calculating one intersection is $O(n)$. Then the time complexity of this part is $O(n)$, where n is the number of records in the relation.

Space Complexity: The space complexity in this algorithm is the complexity of calculating the intersection of two sorted set. It is clearly that the complexity is $O(n)$.

The complexity of the detection algorithm with the variable DCs:

Time Complexity: We adjust the GROUP module and store the relative position together with certain tuple. The time complexity of GROUP' is $O(n \cdot logn)$, where n is the number of records in the relation. After getting the permutation containing the relative positions, we convert the problem of capturing the inconsistent tuple pairs into getting the ascending ordered pairs in the permutation. The method of getting the ascending ordered pairs in the permutation develops from *MergeSort* algorithm whose complexity is $O(n \cdot logn)$. In the process of merging two sorted arrays into a single sorted array which costs $O(n)$, the ascending ordered pairs are detected when exchanges appear. And the tree structure for *MergeSort* method has height at most $O(logn)$. Thus, the time complexity of *MergeSort* is $O(n \cdot logn)$. Since the ascending ordered pairs can be detected in the process of *MergeSort*, the complexity of this part is $O(n \cdot logn)$ as well.

Space Complexity: The space complexity for MergeSort is $O(n)$. We add a permutation containing the relative position, the space complexity is $O(n)$. Thus, the space complexity of this part is $O(n)$ as well.

The correctness of the algorithms:

The correctness of the algorithm for constant DCs:

Suppose we have a constant DC $\neg(A_1 \wedge ... \wedge A_m)$. Each A_i is built-in atoms of the form $t[A]\theta c$, A is an attribute in R, c is a constant and $\theta \in \{=, \neq, <, >, \leq, \geq\}$. Tuples matching each predicate A_i are collected in the tuple set S_i. Then, the inconsistencies can be calculated by $(S_1 \wedge ... \wedge S_m)$.

Proof 5: In order to prove the correctness of the algorithm for constant DCs, we need to prove that (1)all the inconsistencies of this rule are in the set $(S_1 \wedge ... \wedge S_m)$, (2)each tuple in the set $(S_1 \wedge ... \wedge S_m)$ is inconsistent.

(1) Assume that there is an inconsistent tuple t, and $t \notin (S_1 \wedge ... \wedge S_m)$. As t violates the DC, t must match all the predicates $A_1, A_2, ..., A_m$. That is, $t \in S_1, t \in S_2, \ldots, t \in S_m$. Then, we can get $t \in (S_1 \wedge ... \wedge S_m)$ which contradicts the assumption, the assumption is incorrect.

(2) Assume that there is a tuple $t \in (S_1 \wedge ... \wedge S_m)$, and t does not violate the rule. We can know that $t \in S_1$, $t \in S_2, \ldots, t \in S_m$ from the assumption. That is, t matches all the predicates $A_1, A_2, ..., A_m$. Then, t must violate the rule by the definition of DC. This contradiction leads us to conclude that the original assumption is incorrect.

The correctness of the algorithm for variable DCs:

We had proved that detecting the inconsistent pairs with a variable DC can be converted into getting the ascending ordered pair in a permutation (Theorem 2). As we know, the inversion pairs can be captured easily by a most common sorting method MergeSort. The process of capturing the ascending ordered pairs is similar to capture the inversions. That is, our method developed from the function MergeSort can capture the inconsistencies violating variable DCs.

VI. EXPERIMENTAL STUDY

In this section, we used two real-world datasets and two synthetic datasets to evaluate the performance of our inconsistency detection algorithm experimentally.

Dataset 1: This dataset is a synthetic dataset. It contains a relation of elevation which is shown in Table 5. We extend the attributes and involve two other attributes in the relation. There are five attributes in the relation: H (Height), T (Temperature), P (Air Pressure), O (Oxygen level), D (atmosphere density). There are 30M tuples in the dataset.

We use the rules such as

$$(1)t1, t2\neg(t1.T > t2.T \wedge t1.P > t2.P \wedge t1.H > t2.H),$$
$$(2)t1, t2\neg(t1.H > t2.H \wedge t1.O > t2.O).$$

Dataset 2: This dataset is generated with the distribution of an original dataset about Child Health. The original dataset contains 25000 tuples of human heights and weights

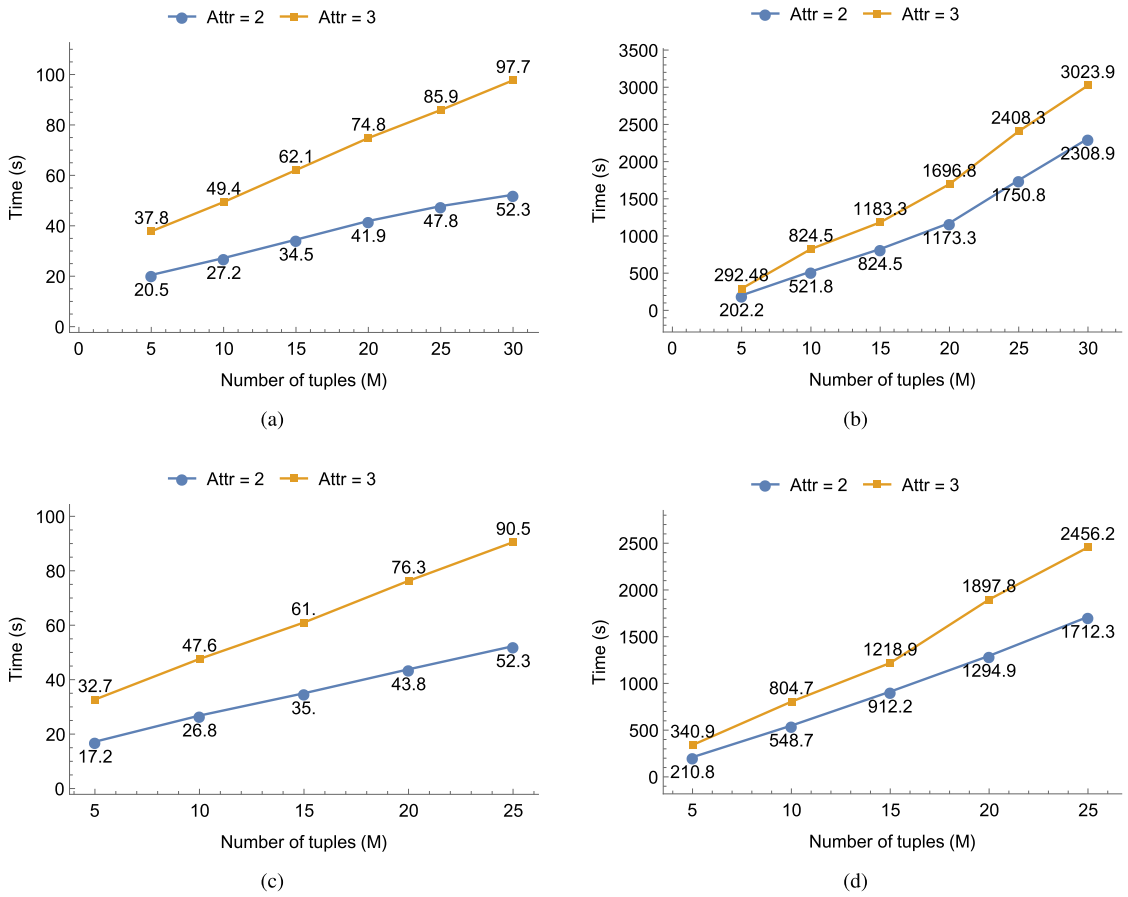

FIGURE 1. The impact of ATTR. (a) Detection with cDCs (Dataset1). (b) Detection with vDCs (Dataset1). (c) Detection with cDCs (Dataset2). (d) Detection with vDCs (Dataset2).

of 18 years old children. It can be found from this website[2]. We generate 25M tuples based on the distribution of the Child Health dataset. In addition, we add 2 attributes about body mass index(BMI) and FatIndex. The WHO definition is: a BMI greater than or equal to 25 is overweight a BMI greater than or equal to 30 is obesity. We set FatIndex according to the WHO definition. If BMI≥30, we set FatIndex as 3; else if BMI≥25, we set FatIndex as 2; otherwise, it is set as 1.

Dataset 3: This dataset is a synthetic dataset about salary. This dataset contains 30M tuples, and the attributes of this dataset is shown in Table 1.

Dataset 4: This dataset is U.S. Pollution Data [3] about pollution including NO_2, O_3, SO_2 and CO. It contains 1.75M number of tuples and 29 attributes. We use the attributes about the mean, the maximum and air quality index of each pollution to conduct our experiments.

We conducted all the experiments on a Windows 7 machine with a 3.10GHz Intel CPU and 4GB of Memory. Each experiment was run 5 times, and the average result is reported. In the experiments, we use our detection methods and the SQL-based methods to detect the inconsistencies with same rules, respectively. The DBMS we used to implement the SQL queries is MySQL, and the data is stored in the DBMS without index. We also compare the performance of our method with the performance of implementing SQL queries in BigQuery. The SQL queries are formed like the examples in Section 1. The detection results of our inconsistency detection method are same with the result of the SQL queries, and the results also accord with the reality.

The performance of the cRHS and vRHS inconsistency detection algorithm is shown in our previous work [37]. We will only discuss the performance of the DCs inconsistency detection algorithm in the following part.

CFD Inconsistency Detection

A. THE PERFORMANCE OF THE DC INCONSISTENCY DETECTION ALGORITHM

The Impact of noise on detection time In this part, we evaluate the performance of the inconsistency detection algorithm we proposed to detect the violation of DCs.

EXP 1: The impact of number of attributes and tuples in datasets: We use "ATTR" and "DataSize" to indicate the number of attributes and the number of tuples, respectively. The experimental result is shown in Figure 1. In this experiment we test the impact of number of tuples and number of

[2] http://wiki.stat.ucla.edu/socr/index.php/SOCR_Data_Dinov_020108_HeightsWeights

[3] https://www.kaggle.com/sogun3/uspollution

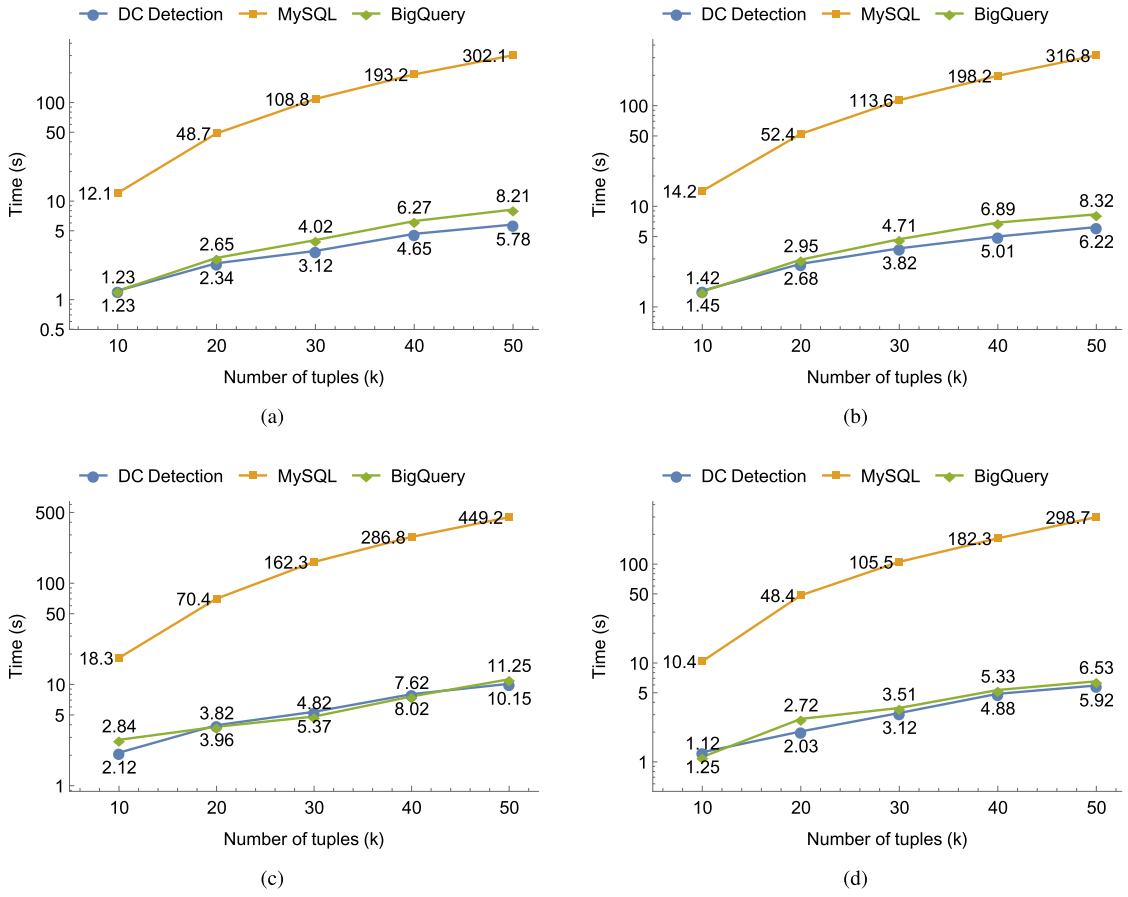

FIGURE 2. Performance Comparison. (a) Dataset 1. (b) Dataset 2. (c) Dataset 3. (d) Dataset 4.

attributes on the detection time. We conduct the DCs inconsistency detection algorithm on the Dataset 1 and Dataset 2. The experimental result on the other two datasets are similar to that shown in Figure 1. The number of tuples ranges from 5M to 30M, in 5M increments.

We use two lists of constant DCs to detect the violation in the datasets. Each list contains five DCs. In the first list, there are two attributes in each DC. In the second list, there are three attributes in each DC. Fig. 1(a) and Fig. 1(c) indicate that the detection time increases with the number of tuples. As in our method, the violation of constant DCs is captured by calculating the intersection of the tuples matching each predicates, more attributes means more intersections. As shown in Fig. 1(a) and Fig. 1(c), it is clear that the detection time also increases with the number of attributes.

We use two variable DCs to detect the violation. The first one contains two attributes, and the second one contains three attributes. The complexity of our detection method is $O(n \cdot logn)$. Fig. 1(b) and Fig. 1(d) indicate that the detection time increases with the number of tuples, and the line accords with the theoretical result. The number of attributes also has impact on the detection time when detecting with variable DCs. The reason is that in our detection method more attributes result in more intersections. That is, the detection time also increases with the number of attributes.

EXP 2: The efficiency of our detection method: In this experiment, we compare our DCs inconsistency detection algorithm with the SQL-based methods. We conduct the experiment on the four datasets, the experimental result is shown in Figure 2. As we know, a denial constraint can be easily converted into a SQL query. We compare the performance of our detection time with the SQL-based methods. We implement the SQL queries in MySQL and BigQuery. This experiment is processed on a small number of tuples due to the low efficiency of query processing in MySQL and the limit of files loaded from a local data source to BigQuery. The noise of each dataset is nearly 5%. The SQL query of a denial constraint is processed as a Cartesian product, meaning that the time complexity is $O(n^2)$. The complexity of our detection method is $O(n \cdot logn)$. It is also clear that our detection method is much more efficient than the performance of conducting queries in MySQL. We can also learn from this figure that the efficiency of our method is comparable to BigQuery.

EXP 3: The impact of noise: In this experiment, we test the impact of noise on detection time. Figure 3 shows the result of conducting this experiment on Dataset 1 and Dataset 2. The number of tuples is fixed as 5M. We use a variable DC containing three attributes to detect the inconsistent tuple pairs. The noise ranges from 1% to 5%, in 1% increments. We observe from Figure 3 that the noise of the database has

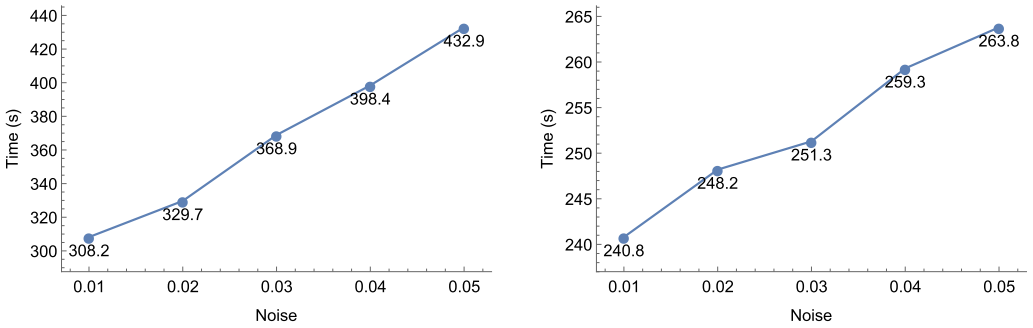

FIGURE 3. The Impact of noise on detection time. (a) The impact of noise (Dataset 1). (b) The impact of noise (Dataset 2).

impact on the detection time. The result is different from that when detecting with variable CFD. More dirty data means more inconsistent tuple pairs. As we adopt a variable DC containing three attributes, we need to make intersection of the tuple pairs. The complexity of intersection is $O(n)$, it causes the impact of noise on the detection time.

Summary of Experiments: In summary, the experiments have following results.

(1) Our detection method is much more efficient than conducting SQL queries in MySQL, and comparable to BigQuery. The complexity of our method is $O(n \cdot logn)$ when detecting the inconsistency of database with variable DCs.

(2) The detection time of our method is affected by both the number of tuples and the number of attributes.

(3) The noise of database has impact on the detection time.

VII. CONCLUSIONS

We study the inconsistency detection problem in this paper. We enrich our CFD inconsistency detection algorithm with more examples to show the details and the proof of the correctness. Detecting inconsistencies with high efficiency benefits the database management for improving data quality. We also proposed two algorithms to detect the inconsistencies with constant DCs and variable DCs, respectively. The experimental results indicate the high efficiency of our detection algorithms.

Our future study will focus on repairing the inconsistencies in big data. We aim to develop efficient methods to restore the consistency of big data.

REFERENCES

[1] W. W. Wayne, "Data quality and the bottom line: Achieving business success through a commitment to high quality data," Data warehouse Inst., Renton, WA, USA, Tech. Rep. 1, 2002. [Online]. Available: www.dw-institute.com

[2] S. Kolahi and L. V. S. Lakshmanan, "On approximating optimum repairs for functional dependency violations," in *Proc. 12th Int. Conf. Database Theory (ICDT)*, Saint Petersburg, Russia, Mar. 2009, pp. 53–62. doi: 10.1145/1514894.1514901.

[3] L. Bertossi, *Database Repairing and Consistent Query Answering* (Synthesis Lectures on Data Management). San Rafael, CA, USA: Morgan Claypool Publishers, 2011. doi: 10.2200/S00379ED1V01Y201108DTM020.

[4] L. Bertossi, L. Bravo, E. Franconi, and A. Lopatenko, "The complexity and approximation of fixing numerical attributes in databases under integrity constraints," *Inf. Syst.*, vol. 33, nos. 4–5, pp. 407–434, 2008. doi: 10.1016/j.is.2008.01.005.

[5] X. Chu, I. F. Ilyas, and P. Papotti, "Holistic data cleaning: Putting violations into context," in *Proc. 29th IEEE Int. Conf. Data Eng. (ICDE)*, Brisbane, QLD, Australia, Apr. 2013, pp. 458–469. doi: 10.1109/ICDE.2013.6544847.

[6] A. Lopatenko and L. Bravo, "Efficient approximation algorithms for repairing inconsistent databases," in *Proc. 23rd Int. Conf. Data Eng. (ICDE)*, Istanbul, Turkey, Apr. 2007, pp. 216–225. doi: 10.1109/ICDE.2007.367867.

[7] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Conditional functional dependencies for data cleaning," in *Proc. 23rd Int. Conf. Data Eng. (ICDE)*, Istanbul, Turkey, Apr. 2007, pp. 746–755. doi: 10.1109/ICDE.2007.367920.

[8] S. Ma, L. Duan, W. Fan, C. Hu, and W. Chen, "Extending conditional dependencies with built-in predicates," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 12, pp. 3274–3288, Dec. 2015. doi: 10.1109/TKDE.2015.2451632.

[9] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," in *Proc. ACM SIGPLAN Conf. Object-Oriented Program. Syst., Lang. Appl. (OOPSLA)*, Anaheim, CA, USA, Oct. 2003, pp. 78–95. doi: 10.1145/949305.949314.

[10] W. Fan and F. Geerts, *Foundations of Data Quality Management* (Synthesis Lectures on Data Management). San Rafael, CA, USA: Morgan Claypool Publishers, 2012. doi: 10.2200/S00439ED1V01Y201207DTM030.

[11] P. Bohannon, M. Flaster, W. Fan, and R. Rastogi, "A cost-based model and effective heuristic for repairing constraints by value modification," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, , USA, Jun. 2005, pp. 143–154. doi: 10.1145/1066157.1066175.

[12] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma, "Improving data quality: Consistency and accuracy," in *Proc. 33rd Int. Conf. Very Large Data Bases* Vienna, Austria: University of Vienna, Sep. 2007, pp. 315–326. [Online]. Available: http://www.vldb.org/conf/2007/papers/research/p315-cong.pdf

[13] W. Fan, F. Geerts, N. Tang, and W. Yu, "Inferring data currency and consistency for conflict resolution," in *Proc. 29th IEEE Int. Conf. Data Eng. (ICDE)*, Brisbane, QLD, Australia, Apr. 2013, pp. 470–481. doi: 10.1109/ICDE.2013.6544848.

[14] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas, "Guided data repair," *Proc. VLDB Endowment*, vol. 4, no. 5, pp. 279–289, 2011. [Online]. Available: http://portal.acm.org/citation.cfm?id=1952378&CFID=12591584&CFTOKEN=15173685

[15] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu, "Towards certain fixes with editing rules and master data," *Proc. VLDB Endowment*, vol. 21, no. 2, pp. 213–238, 2012. doi: 10.1007/s00778-011-0253-7.

[16] J. Wijsen, "Condensed representation of database repairs for consistent query answering," in *Database Theory—ICDT*, Siena, Italy, Jan. 2003, pp. 375–390. doi: 10.1007/3-540-36285-1_25.

[17] F. Korn, S. Muthukrishnan, and Y. Zhu, "Checks and balances: Monitoring data quality problems in network traffic databases," in *Proc. 29th Int. Conf. Very Large Data Bases (VLDB)*, Berlin, Germany, Sep. 2003, pp. 536–547. [Online]. Available: http://www.vldb.org/conf/2003/papers/S17P01.pdf

[18] H. Xiong, G. Pandey, M. Steinbach, and V. Kumar, "Enhancing data analysis with noise removal," *IEEE Trans. Knowl. Data Eng.*, vol. 18, no. 3, pp. 304–319, Mar. 2006. doi: 10.1109/TKDE.2006.46.

[19] F. Chiang and R. J. Miller, "Discovering data quality rules," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 1166–1177, 2008. [Online]. Available: http://www.vldb.org/pvldb/1/1453980.pdf

[20] L. Golab, H. J. 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. [Online]. Available: http://www.vldb.org/pvldb/1/1453900.pdf
[21] W. Fan, F. Geerts, L. V. S. Lakshmanan, and M. Xiong, "Discovering conditional functional dependencies," in *Proc. 25th Int. Conf. Data Eng. (ICDE)*, Shanghai, China, Mar./Apr. 2009, pp. 1231–1234. doi: 10.1109/ICDE.2009.208.
[22] X. Chu, I. F. Ilyas, and P. Papotti, "Discovering denial constraints," *Proc. VLDB Endowment*, vol. 6, no. 13, pp. 1498–1509, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p1498-papotti.pdf
[23] F. Geerts, G. Mecca, P. Papotti, and D. Santoro, "The LLUNATIC data-cleaning framework," *Proc. VLDB Endowment*, vol. 6, no. 9, pp. 625–636, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p625-mecca.pdf
[24] T. Bleifuß, S. Kruse, and F. Naumann, "Efficient denial constraint discovery with hydra," *Proc. VLDB Endowment*, vol. 11, no. 3, pp. 311–323, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol11/p311-bleifub.pdf
[25] E. H. M. Pena and E. C. de Almeida, "BFASTDC: A bitwise algorithm for mining denial constraints," in *Proc. 29th Int. Conf. Database Expert Syst. Appl. (DEXA)*, Sep. 2018, pp. 53–68. doi: 10.1007/978-3-319-98809-2_4.
[26] J. Wang and N. Tang, "Dependable data repairing with fixing rules," *J. Data Inf. Qual.*, vol. 8, nos. 3–4, pp. 16:1–16:34, 2017. [Online]. Available: http://doi.acm.org/10.1145/3041761
[27] C. Xu, W. Xi, S. Cheung, X. Ma, C. Cao, and J. Lu, "Cina: Suppressing the detection of unstable context inconsistency," *IEEE Trans. Softw. Eng.*, vol. 41, no. 9, pp. 842–865, Sep. 2015. doi: 10.1109/TSE.2015.2418760.
[28] W. Xi, C. Xu, W. Yang, X. Ma, P. Yu, and J. Lu, "Suppressing detection of inconsistency hazards with pattern learning," *Inf. Softw. Technol.*, vol. 74, pp. 219–229, Jun. 2016. doi: 10.1016/j.infsof.2015.08.003.
[29] P. Alpar and S. Winkelsträter, "Assessment of data quality in accounting data with association rules," *Expert Syst. Appl.*, vol. 41, no. 5, pp. 2259–2268, 2014. doi: 10.1016/j.eswa.2013.09.024.
[30] Z. Abedjan, C. G. Akcora, M. Ouzzani, P. Papotti, and M. Stonebraker, "Temporal rules discovery for Web data cleaning," *Proc. VLDB Endowment*, vol. 9, no. 4, pp. 336–347, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol9/p336-abedjan.pdf
[31] N. Talukder, M. Ouzzani, A. K. Elmagarmid, and M. Yakout, "Detecting inconsistencies in private data with secure function evaluation," Dept. Comput. Sci., Purdue Univ., West Lafayette, IN, USA, Tech. Rep. 11-006, 2011.
[32] L. Li, X. Feng, H. Shao, and J. Li, "Source selection for inconsistency detection," in *Proc. 23rd Int. Conf. Database Syst. Adv. Appl. (DASFAA)*, Gold Coast, QLD, Australia, May 2018, pp. 370–385. doi: 10.1007/978-3-319-91458-9_22.
[33] R. A. Shaikh, K. Adi, and L. Logrippo, "A data classification method for inconsistency and incompleteness detection in access control policy sets," *Int. J. Inf. Secur.*, vol. 16, no. 1, pp. 91–113, 2017. doi: 10.1007/s10207-016-0317-1.
[34] K. Anand, P. K. Chittimalli, and R. Naik, "An automated detection of inconsistencies in SBVR-based business rules using many-sorted logic," in *Proc. 20th Int. Symp. Practical Aspects Declarative Lang. (PADL)*, Los Angeles, CA, USA, Jan. 2018, pp. 80–96. doi: 10.1007/978-3-319-73305-0_6.
[35] M. H. Farid, A. Roatis, I. F. Ilyas, H. F. Hoffmann, and X. Chu, "CLAMS: Bringing quality to data lakes," in *Proc. Int. Conf. Manage. Data SIGMOD*, San Francisco, CA, USA, Jun./Jul. 2016, pp. 2089–2092. doi: 10.1145/2882903.2899391.
[36] X. Oriol and E. Teniente, "Incremental checking of OCL constraints with aggregates through SQL," in *Proc. 34th Int. Conf Conceptual Modeling*, Stockholm, Sweden, Oct. 2015, pp. 199–213. doi: 10.1007/978-3-319-25264-3_15.
[37] M. Zhang, H. Wang, J. Li, and H. Gao, "One-pass inconsistency detection algorithms for big data," in *Proc. 21st Int. Conf. Database Syst. Adv. Appl. (DASFAA)*, Dallas, TX, USA, Apr. 2016, pp. 82–98. doi: 10.1007/978-3-319-32025-0_6.
[38] Z. Khayyat *et al.*, "Lightning fast and space efficient inequality joins," *Proc. VLDB Endowment*, vol. 8, no. 13, pp. 2074–2085, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p2074-khayyat.pdf
[39] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Commun. ACM*, vol. 20, no. 5, pp. 350–353, 1977.

MEIFAN ZHANG was born in Harbin, Heilongjiang, China, in 1992. She received the bachelor's degree in computer science from the Harbin Institute of Technology, in 2014, where she is currently pursuing the Ph.D. degree. Her research interests include big data analytics, data quality, and machine learning.

HONGZHI WANG was born in 1978. He received the Ph.D. degree in computer science from the Harbin Institute of Technology, in 2008.

From 2008 to 2010, he was an Assistant Professor with the Harbin Institute of Technology. From 2010 to 2015, he was an Associate Professor. Since 2015, he has been a Professor with the Department of Computer Science and Technology, Harbin Institute of Technology. His research interests include big data management, data quality, graph data management, and web data management.

Prof. Wang was a recipient of the Microsoft Fellowship, the Chinese Excellent Database Engineer, and the IBM PHD Fellowship.

JIANZHONG LI was born in 1950. He received the B.S. degree from Heilongjiang University, in 1975.

He was with the University of California at Berkeley, as a Visiting Scholar, in 1985. From 1986 to 1987 and from 1992 to 1993, he was a Staff Scientist with the Information Research Group, Lawrence Berkeley National Laboratory, Berkeley, CA, USA. He was a Visiting Professor with the University of Minnesota at Minneapolis, Minnesota, MN, USA, from 1991 to 1992, and from 1998 to 1999. Since 1998, he has been a Professor with the Department of Computer Science and Technology, Harbin Institute of Technology. His current research interests include database management systems, data warehousing and data mining, sensor network, and data intensive supercomputing. He has authored three books, including the *Parallel Database Systems*, and the *Principle of Database Systems and Digital Library*.

Prof. Li was a recipient of awards and honors, including the Chairman of the ACM SIGMOD China and the Director of the China Computer Federation.

HONG GAO received the Ph.D. degree from the Harbin Institute of Technology.

She is currently a Professor and a Doctoral Supervisor with the Harbin Institute of Technology. She has published over 100 papers in her career. She has long involved in the research work of massive data computation and quality management, wireless sensor networks, and graphic data management and computation. She was a recipient of awards and honors, including the Assistant Director of the China Computer Federation Technical Committee on Databases, a member of the China Computer Federation Technical Committee on Sensor Network, and the Deputy Director of the Massive Data Computing Lab.