Efficient Discovery of Relaxed Functional Dependencies

Mengran Li
School of Computer Science, Fudan University, China
mrli22@m.fudan.edu.cn

Zijing Tan
School of Computer Science, Fudan University, China
Shanghai Key Laboratory of Data Science
zjtan@fudan.edu.cn

Honghui Yang
School of Computer Science, Fudan University, China
22212010046@m.fudan.edu.cn

Shuai Ma
SKLSDE Lab, Beihang University, China
mashuai@buaa.edu.cn

ABSTRACT

This paper studies the discovery of relaxed functional dependencies (RFDs). We consider RFDs that relax restrictions in both value equality and constraint satisfaction: treating values as equal if their distance is less than a given similarity threshold, and considering RFDs with violations below a given error threshold as valid. As a highly non-trivial extension of the row-based approach to functional dependency (FD) discovery, we present the first algorithm capable of discovering all valid and minimal RFDs. We extend the structure called "*difference-set*" for *predicates* that are combinations of attributes and similarity thresholds. We present an efficient method for difference-set construction, incorporating optimizations for both time and space complexity. When inferring RFDs from difference-sets, we enumerate RFDs based on the subsumption relationship of their right-hand-side predicates to share computations. An extensive experimental evaluation verifies that the proposed discovery algorithm is faster than baseline methods up to orders of magnitude and effective in finding hidden FDs from dirty data.

PVLDB Reference Format:
Mengran Li, Zijing Tan, Honghui Yang, and Shuai Ma. Efficient Discovery of Relaxed Functional Dependencies. PVLDB, 18(7): 2044-2056, 2025.
doi:10.14778/3734839.3734843

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/YukinoMR/FastRFD.

1 INTRODUCTION

As an important part of data profiling [1, 2], dependency discovery methods for identifying hidden dependencies from data have received consistent and extensive attention. Functional dependency (FD) is one of the most important types of dependency. Formally, an FD $X \rightarrow A$ states that whenever two tuples in an instance r share the same values for the set of attributes X, they also agree in their values in attribute A. In practice, real-world data are often dirty [16, 21], which hinders the validity of FDs. Relaxations of FDs, known as relaxed FDs (RFDs) [9, 52], have been proposed to deal with such situations. In this paper, we study the problem of RFD discovery, and consider RFDs that relax restrictions in value equality and constraint satisfaction simultaneously. The formal definition of such RFDs will be provided in Section 3. In what follows, we first given an illustrative example.

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:10.14778/3734839.3734843

Example 1: Relational instance r in Table 1 is about hospitalized patients, where attribute Acuity refers to the level of care required by patients, and attribute Los denotes the length of stay in a hospital. There are some data issues in the attribute *Department*: the values in t_3 and t_4 should both be "Internal Medicine", and the value in t_9 should be "Eye Clinic". There is also an error in the attribute *RoomNumber* of t_6, where the correct value should be "205". *RoomNumber* no longer determines *Department* due to these issues, affecting the validity of the FD *RoomNumber* $\rightarrow$ *Department*.

We propose a RFD: $RoomNumber_{(0)} \xrightarrow{0.1} Department_{(1)}$, which relaxes the previous FD by allowing for similarity instead of strict equality and permitting fewer violations. This RFD states that for two tuples with the same room number, the difference between their values in *Department* should be no more than 1. The absolute value and string edit distance are used to calculate the distance between numeric and string values respectively, and similarity threshold 0 is used to represent equality, while threshold 1 is the distance between two very similar values in *Department*. This RFD is allowed to be violated by partial data, with an error threshold set at 0.1. The criterion g_1 [25] is employed to measure the degree of violation, which is defined as the proportion of violating tuple pairs (ignoring reflexive pairs). Among the 90 tuple pairs, 8 pairs violate the RFD with relaxation only in value equality. Since the g_1 value falls below the threshold 0.1, this RFD is considered valid on the instance r.

Both relaxations are necessary. RFDs with relaxations in value equality are suitable to handle minor spelling errors, different abbreviation forms, or small errors in numerical values. However, errors resulting from more complex causes may also exist, as seen in tuples t_4 and t_6. If relaxation in constraint satisfaction is not allowed, then a high similarity threshold on attribute *Department* is required to tolerate errors, potentially causing many different data values to be considered the same. Conversely, ignoring the relaxation of value equality can lead to an overestimation of the RFD violation, causing it to be overlooked during the discovery. □

The example shows that data errors may distort inherent data constraints to some extent. With relaxations in both value equality and constraint satisfaction, RFDs are capable of effectively capturing hidden data constraints present in dirty data. Although desirable, manually designing RFDs is more challenging than FDs, which are already known to be difficult [5, 38, 39, 57]. This motivates the

Table 1: Instance r with hospitalized patients.

	Department (DEP)	Name	Gender	Age	Symptoms (SYM)	RoomNumber (RN)	Acuity	Los
t_1	Internal Medicine	Lisa	Female	69	Heart Arrhythmia	101	3	96
t_2	Internal Medicine	Tom	Male	53	Heart Arrhythmia	102	2	82
t_3	Internal Madicine (Medicine)	Jack	Female	32	Heart Hurt	101	1	29
t_4	Orthopaedics (Internal Medicine)	Cloud	Male	45	Heart Arrhythmia	102	2	72
t_5	Pulmonology	Tina	Female	46	Shortness of breath	205	2	23
t_6	Pulmonology	Aerith	Female	46	Shortness of breath	118 (205)	2	26
t_7	Eye Clinic	Root	Male	71	Cataract	118	2	28
t_8	Eye Clinic	Frank	Male	23	Trachoma	118	1	20
t_9	Ey (Eye) Clinic	Halley	Female	80	Cataract	118	2	35
t_{10}	Eye Clinic	Schneider	Male	36	Trachoma	117	1	26

research on discovery methods for RFDs. Discovering RFDs is necessarily much more difficult than FDs due to a significantly larger search space resulting from combinations of attributes and similarity thresholds, with multiple thresholds possible per attribute, as well as additional computations for measuring value similarity and quantifying violations. To our best knowledge, most methods for RFD discovery [8, 14, 15, 20, 26] only consider RFDs relaxing in one aspect. The only work [10] that aims to discover RFDs relaxing in both aspects can only approximate the discovery result without a guarantee of completeness (details are discussed in Section 2). This work aims to provide an effective and efficient solution for RFD discovery, addressing the limitations of previous research.

Contributions & Organization.

(1) RFD discovery framework. For RFDs allowing for value similarity and fewer violations quantified with measure g_1 (Sections 3), we present a discovery framework (Section 4), which can be regarded as a highly non-trivial extension of the row-based approach to FD discovery [18, 37, 58]. A structure, called as *difference-set*, is built to encode similarity between attribute values within tuple pairs, and minimal and valid RFDs are inferred from this structure.

(2) Discovering RFDs. (a) We give an efficient method to construct difference-set (Section 5). Difference-set is kept in a condensed representation to reduce space complexity, and computations are organized column-wise, combined with caching and clustering techniques to reduce time complexity. (b) We present the first method capable of discovering the complete set of minimal and valid RFDs (Section 6). We enumerate RFDs based on the *subsumption* relationship of their right-hand-side (RHS) predicates and introduce several novel pruning rules. (c) Our method is adapted to find top-k RFDs based on a utility function involving multiple factors.

(3) Experimental study (Section 7). We verify the following. Our method (a) can be orders of magnitude faster than row-based and column-based baseline methods; (b) significantly outperforms [10] that also aims for RFDs with relaxations in both aspects; and (c) can effectively identify FDs hidden in dirty data.

2 RELATED WORK

Discovery techniques have been studied for different dependencies, such as order dependencies (ODs) [11, 23, 24, 28, 31, 53–55], denial constraints (DCs) [12, 33, 40, 41, 43, 59] and differential dependencies (DDs) [27, 50]. ODs and DCs subsume FDs, but not RFDs with relaxation in value equality. DDs can specify constraints on differences with operators "$\leq$" and "$>$". DDs generalize RFDs with relaxation in value equality but not in constraint satisfaction.

Discovery methods for FDs and its variants have been well researched. Most methods aim for the *complete* set of minimal and valid FDs, while some works [4, 22, 32] only approximate the result, *i.e.,* trading correctness and (or) completeness for efficiency. Another line of research [34–36, 42, 61] adopts information-theoretic or probabilistic interpretations, and only aims for *top* FDs. The method for discovering *embedded* FDs from data with missing values is presented in [56]. There are also studies on the discovery of conditional FDs [17, 19, 44, 45], for identifying conditions to generate partial data and discovering FDs that hold within them. From a different perspective, most methods consider the *static* setting, where FDs are found in a dataset r. In contrast, some methods [6, 7, 48, 60] consider the *dynamic* setting, aiming to find FDs from $r + \triangle r$ in response to a set $\triangle r$ of updates, based on the known FD set on r.

The goal of this study is to find the complete set of minimal and valid RFDs under the static setting. In the rest of this section, we investigate works that are close to ours.

Discovery methods for FDs. Discovery methods for FDs have been well studied and can be roughly divided into three categories. Column-based approaches [3, 20] traverse the space of FDs according to a lattice structure, validate FDs and prune the search space by leveraging found valid FDs. Row-based approaches [18, 37, 58] build a data structure by comparing attribute values of tuple pairs, and then infer valid FDs from it. Hybrid approaches [5, 39, 57] adopt the row-based strategy on sample data to identify candidate FDs, and then refine them for the final result by applying the column-based strategy. A previous evaluation [38] indicates that column-based methods often exhibit good scalability *w.r.t.* the number of tuples, while row-based methods typically scale well with the number of attributes. Hybrid methods integrate the strengths of both approaches, thus offering better performance in most cases.

As previously noted, the RFDs we consider differ significantly from FDs and introduce many challenges to the discovery problem.

Discovery methods for RFDs. Different definitions of RFDs and related discovery methods have been proposed. [15, 20, 26] consider RFDs with relaxation in satisfaction (also known as approximate FDs). The error measure g_1 [25] adopted in [26] is computed as the proportion of violating tuple pairs, while the measure g_3 [25] considered in [15, 20] represents the proportion of tuples that need to be removed to make a violated FD hold. Regarding the error threshold setting, [20, 26] apply the same threshold across all RFDs, while [15] allows for different thresholds for different RFDs, under the assumption that the upper bound on the proportion of errors is known for each attribute. Different from these works, [8] considers RFDs relaxing from value equality to similarity, allowing multiple optional similarity thresholds for each attribute. The methods in [8], [15, 20], and [26] can be broadly categorized as row-based methods, column-based methods, and hybrid methods, respectively.

To our best knowledge, [10] presents the only method to discover RFDs relaxing in both aspects. It first builds a distance matrix where each entry records the distance between a tuple pair on a specific attribute, and then employs a column-based strategy that enumerate RFDs and validates them using the matrix. This work differs in the following. (1) We use the same similarity threshold settings as in [8], while [10] uses the same single threshold for all attributes. Since different thresholds can denote varying degrees of similarity, our setting significantly enhances the expressiveness of RFDs. (2) We use criterion g_1 to measure the satisfaction of RFDs, as opposed to criterion g_3 used by [10]. A RFD considered valid under g_1 may not be valid under g_3, and vice versa. The computation of g_3 becomes intractable when the restriction in value equality is simultaneously relaxed [10], preventing [10] from ensuring the completeness of discovery result. The inability to guarantee completeness can significantly impact the effectiveness, as will be experimentally studied. In contrast, computing g_1 value for a RFD relaxing in both aspects can be done in polynomial time, allowing us to efficiently offer the complete result.

3 PRELIMINARIES

In this section, we provide the definition of RFD. Let R denote a relational schema: an attribute set $\{A^1, \ldots, A^{|R|}\}$ where $|R|$ is the size of R. Let r denote an instance of R and $|r|$ its size. Let t and s denote tuples in r, and t_A denote the value of attribute A in tuple t.

Distance and threshold. Distance functions measure the similarity of values. The attribute domain can suggest an appropriate function, such as the absolute difference for numerical values and the edit distance for strings. More sophisticated functions can also be used. For example, the distance between two synonyms in a specific domain can be defined as very small. We use Φ_A to denote the function used for attribute A and do not depend on specific functions. $\Phi_A(t, s)$ returns the distance between t_A and s_A.

Two values are considered similar if their distance is no larger than a *similarity threshold*. Each attribute requires its own threshold setting due to variations in data types and distributions [8, 47]. Appropriate thresholds for an attribute can be set by users or be automatically identified, *e.g.,* the distance between two similar and frequently occurring strings in a dataset. Such techniques have been well studied [8, 47, 50, 51] and are not the focus of this research. We consider multiple thresholds possible for each attribute and do not depend on specific techniques to determine them. In the following, we assume (1) a set of similarity thresholds for each $A \in R$, denoted by Thr_A, is determined in a pre-processing step and an input of our discovery method; and (2) $0 \in Thr_A$ for every A; $\Phi_A(t, s) = 0$ iff t and s have the same value in A. Similar to [8, 10], we use absolute values as thresholds, which facilitates comparison. Normalization can also convert absolute values to establish thresholds.

Table 2: The set $\mathcal{P}$ of all predicates

p_1: $DEP_{(1)}$	p_2: $DEP_{(0)}$	p_3: $Name_{(0)}$	p_4: $Gender_{(0)}$
p_5: $Age_{(20)}$	p_6: $Age_{(10)}$	p_7: $Age_{(0)}$	p_8: $SYM_{(3)}$
p_9: $SYM_{(0)}$	p_{10}: $RN_{(1)}$	p_{11}: $RN_{(0)}$	p_{12}: $Acuity_{(0)}$
p_{13}: $Los_{(10)}$	p_{14}: $Los_{(0)}$		

Predicates. The combination of Φ_A and a threshold θ from Thr_A constitutes a *predicate* on A, denoted by A_θ when Φ_A is clear from the context. A tuple pair (t, s) satisfies A_θ, written as $(t, s) \asymp A_\theta$, if $\Phi_A(t, s) \leq \theta$. We say A_{θ^i} subsumes A_{θ^j}, written as $A_{\theta^i} \succ A_{\theta^j}$, if $\theta^i > \theta^j$. It is easy to see that $(t, s) \asymp A_{\theta^i}$, if $(t, s) \asymp A_{\theta^j}$ and $A_{\theta^i} \succ A_{\theta^j}$. We write $A_{\theta^i} \succeq A_{\theta^j}$ if $A_{\theta^i} \succ A_{\theta^j}$ or $\theta^i = \theta^j$.

We extend the subsumption relation to predicate sets. For two distinct sets X, Y of predicates, X subsumes Y, written as $X \succ Y$, if for each predicate $p \in X$, there exists a predicate $p' \in Y$ such that $p \succeq p'$. It can be seen that (t, s) satisfies all the predicates of X, if (t, s) satisfies all the predicates of Y; the subsumption relationship between X and Y expresses the inclusion relationship of the sets of tuple pairs that satisfy X and Y. We write $X \succeq Y$ if $X \succ Y$ or $X = Y$.

Example 2: Recall Table 1. In the following examples, we abbreviate attributes *Department*, *Symptoms*, and *RoomNumber* as *DEP*, *SYM*, and *RN*, respectively. Let $U = \{DEP_{(0)}, Age_{(20)}, RN_{(1)}\}$ and $U' = \{DEP_{(1)}, RN_{(1)}\}$. We have $U' \succ U$, since $DEP_{(1)} \succ DEP_{(0)}$. □

Relaxed functional dependencies (RFDs). Our RFD definition is adapted from [8, 10]. With a given instance r of R, A RFD λ is of the form $A^i_{\theta^i}, \ldots, A^j_{\theta^j} \xrightarrow{\Psi,\, \epsilon} A^k_{\theta^k}$, where $A^i_{\theta^i}, \ldots, A^j_{\theta^j}, A^k_{\theta^k}$ are predicates on attributes $A^i, \ldots, A^j, A^k$, respectively, and $A^i, \ldots, A^j, A^k$ are distinct. Ψ is a function that quantifies the satisfaction of λ, and ϵ is a given error threshold. A tuple pair (t, s) satisfies λ, written as $(t, s) \models \lambda$, iff $(t, s) \asymp A^k_{\theta^k}$ if $(t, s) \asymp A^i_{\theta^i} \wedge \ldots \wedge (t, s) \asymp A^j_{\theta^j}$. Otherwise, (t, s) violates λ, written as $(t, s) \not\models \lambda$. Ψ is defined based on g_1 [25], a common measure to quantify dependency violations [12, 23, 26, 33, 40, 59]. For r and λ, $\Psi(\lambda, r)$ is the ratio of the number of tuple pairs violating λ to the total number of tuple pairs in r^2 (ignoring reflexive tuple pairs).

$$\Psi(\lambda, r) = \frac{|\{(t,s) \mid (t,s) \in r^2 \wedge t \neq s \wedge (t,s) \not\models \lambda\}|}{|r|^2 - |r|}$$

Example 3: Consider $\lambda = RN_{(0)} \longrightarrow DEP_{(1)}$. Although we relax the equality constraint on attribute *DEP*, (t_6, t_7) still violates λ, because the difference between the value of t_6 and that of t_7 in *DEP* is larger than 1. So are (t_6, t_8), (t_6, t_9) and (t_2, t_4). We have $\Psi(\lambda, r) = 0.089$. □

Valid and minimal RFD. For r and $\lambda = A^i_{\theta^i}, \ldots, A^j_{\theta^j} \xrightarrow{\Psi,\, \epsilon} A^k_{\theta^k}$,

(1) λ is *valid* on r, iff $\Psi(\lambda, r) \leq \epsilon$.

(2) λ is *minimal* on r, iff there does not exist a distinct RFD λ' such that (a) λ' is valid on r; (b) λ' has $A^k_{\theta^{k'}}$ as its RHS predicate and $\theta^{k'} \leq \theta^k$; and (c) for each left-hand-side (LHS) predicate $A^m_{\theta^{m'}}$ of λ', λ has $A^m_{\theta^m}$ on the LHS and $\theta^m \leq \theta^{m'}$.

Example 4: (Example 3 continued.) $\lambda = RN_{(0)} \xrightarrow{0.1} DEP_{(1)}$ is valid since $0.089 < 0.1$. Consequently, $RN_{(0)}, SYM_{(0)} \xrightarrow{0.1} DEP_{(1)}$ is not minimal, because its LHS predicate set contains that of λ and it has the same RHS as λ. Suppose all the available predicates are given in

Table 2. Since neither $RN_{(1)} \xrightarrow{0.1} DEP_{(1)}$ nor $RN_{(0)} \xrightarrow{0.1} DEP_{(0)}$ is verified to be valid, λ is minimal. □

Remarks. The definition of a minimal RFD involves both the containment of predicate sets and the subsumption relationship between predicates, while the definition of a minimal FD only involves the containment of attribute sets; thus, minimal FDs are a special case of minimal RFDs. The definition adheres to the semantics of logical implication: if a RFD λ is not minimal due to the existence of λ', then the validity of λ' implies that of λ on any instance r.

RFD discovery. Given an instance r of R, a set Thr_A of similarity thresholds for each attribute $A \in R$, and an error threshold ϵ, the problem of RFD discovery is to find all minimal valid RFDs on r.

4 FRAMEWORK FOR RFD DISCOVERY

In this section, we provide our RFD discovery algorithm framework, which can be seen as a highly non-trivial extension of the row-based approach to FD discovery [18, 37, 58]. Let $\mathcal{P}$ denote the set of predicates in all attributes of R and $|\mathcal{P}|$ its size. For a RFD λ, let LHS_λ and RHS_λ denote its LHS and RHS predicate sets, respectively. Table 3 summarizes the frequently used notations in this paper.

We extend the concept of "*difference-set*" to RFDs based on predicates, while the original definition for FDs is based on attributes [58].

Difference-set. With the set $\mathcal{P}$ of predicates, the *difference-set* $DS(t, s)$ for two distinct tuples t and s is the set of predicates violated by (t, s), *i.e.,* $DS(t, s) = \{A_\theta \mid A_\theta \in \mathcal{P} \wedge \Phi_A(t, s) > \theta\}$.

Example 5: In Table 2, we give $\mathcal{P}$ for the instance r in Table 1. It can be verified that $DS(t_1, t_2) = \{Name_{(0)}, Gender_{(0)}, Age_{(10)}, Age_{(0)}, RN_{(0)}, Acuity_{(0)}, Los_{(10)}, Los_{(0)}\}$. □

Whether a given RFD λ is satisfied by a pair (t, s) can be determined based on $DS(t, s)$, which further facilitates the computation of the g_1 value of λ. The formal results are stated as follows.

Proposition 1: (1) $\forall t, s \in r\ (t \neq s), (t, s) \not\models \lambda$, iff (a) $RHS_\lambda \in DS(t, s)$; and (b) $LHS_\lambda \cap DS(t, s) = \emptyset$.

(2) $\Psi(\lambda, r) = \frac{|\{(t,s)\ \mid\ t,s \in r\ (t \neq s)\ \wedge\ RHS_\lambda \in DS(t,s)\ \wedge\ LHS_\lambda\ \cap\ DS(t,s)\ =\ \emptyset\}|}{|r|^2 - |r|}$

Proof: (1) $(t, s) \not\models \lambda$, iff (t, s) satisfies all the LHS predicates, but does not satisfy the RHS predicate . (2) This directly follows from (1). □

Example 6: Recall Table 1. For any RFD λ with $DEP_{(1)}$ on the RHS, a tuple pair whose difference-set does not contain $DEP_{(1)}$, *e.g.,* (t_1, t_2), never violates λ. As a counterexample, $DS(t_1, t_4)$ contains $DEP_{(1)}$. To make λ satisfied by (t_1, t_4), λ needs at least one predicate from $DS(t_1, t_4)$ on its LHS. For instance, (t_1, t_4) does not violate $\lambda = RN_{(0)} \longrightarrow DEP_{(1)}$ as $RN_{(0)} \in DS(t_1, t_4)$. □

A naive row-based discovery method. The connection between difference-sets and valid RFDs motivates a naive discovery method as follows: (1) Enumerate every tuple pair to generate its difference-set. (2) Enumerate all RFDs by choosing one RHS predicate and several LHS predicates, and for each RFD, remove difference-sets that do not contain the RHS predicate or contain at least one of the LHS predicates. A RFD is valid if the proportion of the remaining difference-sets to all difference-sets is below the given threshold ϵ. (3) Remove non-minimal RFDs. However, this naive method is highly expensive. It takes both time and space complexity of $O(|r|^2)$ to construct difference-sets. Candidate RFD enumeration is in $O(\sum_{A \in R} 2^{(|\mathcal{P}| - |Thr_A|)} |Thr_A|)$. Validating each RFD directly on difference-sets takes $O(|r|^2)$ time. Finally, performing pairwise minimality checks on the set Ω of all valid RFDs takes $O(|\Omega|^2)$ time.

Table 3: Notations

Symbol	Description	Section
$\Phi_A(t, s)$	the distance between t_A and s_A	Section 3
Thr_A	the set of similarity thresholds used for A	Section 3
A_θ	a predicate on A using threshold θ	Section 3
$X \succ Y, X \succeq Y$	subsumption relation between predicate sets	Section 3
$\mathcal{P}$	the predicate set for R	Section 4
LHS_λ, RHS_λ	the LHS and RHS predicate sets of λ	Section 4
$DS(t, s)$	the difference-set of a tuple pair (t, s): the set of predicates violated by (t, s)	Section 4
$DS(r)$	the set of all the distinct difference-sets	Section 5
$DS(t_i, \cdot)$	the set of difference-sets of (t_i, t_j) for a given t_i and every t_j $(i < j)$	Section 5
$DS_A(t_i, \cdot)$	the part of $DS(t_i, \cdot)$ regarding A	Section 5
$DS_{A_{\theta^i}}(r)$	difference-sets within $DS(r)$ that contain A_{θ^i}	Section 6
$DS_A(r)$	$\cup_{\theta^i \in Thr_A} DS_{A_{\theta^i}}(r)$	Section 6

Overview of our optimizations. (1) A well-designed difference-set construction method can have far better space and time complexity (Section 5). There is no need to store the difference-sets of all tuple pairs, but only those distinct difference-sets and the number of tuple pairs that generate each difference-set. As many tuple pairs indeed generate the same difference-set, the actual storage required is often small. By constructing difference-sets column-wise and combining techniques such as clustering and caching, the number of distance calculations between attribute values can also be significantly reduced. (2) The efficiency of the enumeration and minimality check process can be greatly improved, by considering the subsumption relationship of the RHS predicates of RFDs, employing difference-sets to refine RFDs instead of validating RFDs with difference-sets, and introducing effective pruning rules (Section 6).

5 DIFFERENCE-SET CONSTRUCTION

In this section, we give a method for difference-set construction. In the following, we assume that each tuple is uniquely identified, ranging from 1 to $|r|$, and use $t_1 \ldots, t_{|r|}$ to denote tuples.

Algorithm. We present our algorithm, referred to as DiffBuilder (Algorithm 1), for difference-set construction on r. It outputs the set $DS(r)$ of distinct difference-sets where each difference-set has a count representing the number of tuple pairs that lead to it. From a high-level workflow, DiffBuilder enumerates every t_i and computes the difference-sets produced by pair (t_i, t_j) for every t_j where $i < j \leq |r|$; the set of these difference-sets is denoted by $DS(t_i, \cdot)$ (lines 4-16). Merging $DS(t_i, \cdot)$ for all $t_i \in r$ results in $DS(r)$ (line 17), as (t_i, t_j) and (t_j, t_i) share the same difference-set. The computation of $DS(t_i, \cdot)$ is conducted column-wise: each time a *partial* difference-set for an attribute $A \in R$ is built (lines 5-15), denoted by $DS_A(t_i, \cdot)$, which carries values only in A. $DS(t_i, \cdot)$ is obtained by combining together $DS_A(t_i, \cdot)$ for every $A \in R$ (line 16).

In what follows, we introduce the supporting techniques of DiffBuilder in detail and show illustrative examples in Figure 1.

Representation of difference-set. The difference-set of a tuple pair is defined as the set of violated predicates, but a more efficient storage

Algorithm 1: DiffBuilder

```
Input: instance $r$ of schema $R$
Output: $DS(r)$: the set of distinct difference-sets for tuple pairs
        from $r$, where each difference-set has a count representing
        the number of tuple pairs that produce the difference-set
1  $cache \leftarrow \emptyset$
2  $DS(r) \leftarrow \emptyset$
3  foreach tuple $t_i \in r$ do
4      $DS(t_i, \cdot) \leftarrow \emptyset$
5      foreach $A \in R$ do
6          if $cache_A.contains(t_i[A])$ then
7              $tempResult \leftarrow cache_A\ (t_i[A])$
8              $DS_A(t_i, \cdot) \leftarrow$ remove tuple identifier $t_k$ from
                 $tempResult$ for all $k \leq i$
9              if $lastTuple(clus_A(t_i)) = t_i$ then
10                 $cache_A.remove\ (t_i[A])$
11             else $cache_A.update(t_i[A], DS_A(t_i, \cdot))$
12         else
13             $DS_A(t_i, \cdot) \leftarrow$ call Algorithm AttBuilder with $t_i$ and $A$
14             if $t_i \neq lastTuple(clus_A(t_i))$ then
15                 $cache_A.add(t_i[A], DS_A(t_i, \cdot))$
16         $DS(t_i, \cdot) \leftarrow$ combine $DS_A(t_i, \cdot)$ into $DS(t_i, \cdot)$
17     $DS(r) \leftarrow$ merge $DS(t_i, \cdot)$ into $DS(r)$
```

format is used in implementation. Since each predicate on A is identified by its similarity threshold, a unique number is assigned to each predicate (threshold): the larger the threshold value, the smaller the assigned number. For example, number 1 corresponds to the largest threshold, while number $|Thr_A|$ corresponds to the smallest threshold of 0. Within each (partial) difference-set, we only store the largest assigned number among all the *satisfied* predicates, and store 0 if all the predicates are *violated*.

For example, for the two predicates $DEP_{(0)}$ and $DEP_{(1)}$ on DEP, their numbers are "2" and "1", respectively. Consequently, "2", "1" and "0" are stored to represent the difference-sets $\{\}$, $\{DEP_{(0)}\}$ and $\{DEP_{(0)}, DEP_{(1)}\}$, respectively, as shown in Figure 1.

Compressing difference-set. In $DS_A(t_i, \cdot)$, the two partial difference-sets produced by (t_i, t_j) and (t_i, t_k) are redundant if t_j and t_k have the same value in A. We adopt a compression scheme similar in spirt to [41][1]. The idea is to keep only distinct difference-sets and save a list of tuple identifiers (TIDs) for each difference-set *ds*, where every tuple in the list, when combined with t_i, leads to *ds*. Our method to compute $DS_A(t_i, \cdot)$ generates difference-sets in this representation, as will be detailed in Algorithm AttBuilder.

The compressed representation of $DS_{DEP}(t_1, \cdot)$ is in Figure 1; each difference-set is denoted by a threshold number and a TID list.

Combining partial difference-sets. Suppose $DS_A(t_i, \cdot) = \{<m, U_m>, \ldots, <n, U_n>\}$, where U_m is the TID list associated with threshold θ^m. Similarly, suppose $DS_B(t_i, \cdot) = \{<m', U_{m'}>, \ldots, <n', U_{n'}>\}$ for another attribute B. Now consider the combination of them. For each value k from $DS_A(t_i, \cdot)$ and each value l' from $DS_B(t_i, \cdot)$, a new distinct partial difference-set (k, l') for AB is produced, and the associated TID list is the intersection of U_k and $U_{l'}$.

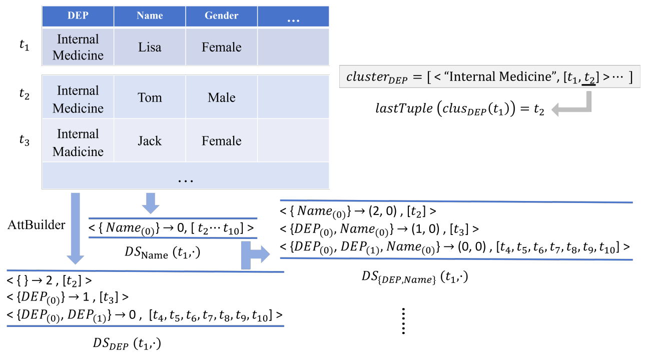

DEP
Name
Gender
…
t_1
Internal Medicine
Lisa
Female
t_2
Internal Medicine
Tom
Male
t_3
Internal Madicine
Jack
Female
…
$cluster_{DEP} = [<$ "Internal Medicine", $[t_1, t_2] > \cdots]$
$lastTuple\ (clus_{DEP}(t_1)) = t_2$
AttBuilder
$< \{Name_{(0)}\} \rightarrow 0, [t_2 \cdots t_{10}] >$
$DS_{Name}\ (t_1, \cdot)$
$< \{\} \rightarrow 2\ , [t_2] >$
$< \{DEP_{(0)}\} \rightarrow 1\ , [t_3] >$
$< \{DEP_{(0)}, DEP_{(1)}\} \rightarrow 0\ ,\ [t_4, t_5, t_6, t_7, t_8, t_9, t_{10}] >$
$DS_{DEP}\ (t_1, \cdot)$
$< \{Name_{(0)}\} \rightarrow (2, 0)\ , [t_2] >$
$< \{DEP_{(0)}, Name_{(0)}\} \rightarrow (1, 0)\ , [t_3] >$
$< \{DEP_{(0)}, DEP_{(1)}, Name_{(0)}\} \rightarrow (0, 0)\ , [t_4, t_5, t_6, t_7, t_8, t_9, t_{10}] >$
$DS_{\{DEP,Name\}}\ (t_1, \cdot)$

Figure 1: DiffBuilder for tuple t_1

We show $DS_{\{DEP,Name\}}(t_1, \cdot)$ in Figure 1, which is obtained based on $DS_{DEP}(t_1, \cdot)$ and $DS_{Name}(t_1, \cdot)$.

After processing all the attributes, we get $DS(t_i, \cdot)$. For each difference-set in $DS(t_i, \cdot)$, the count of tuples in its TID list indicates the number of tuple pairs that lead to the difference-set. We discard the list and keep only the count for each difference-set.

Merging difference-sets. If a difference-set in $DS(r)$ appears in, for example, $DS(t_i, \cdot), \ldots, DS(t_k, \cdot)$, then the count associated with that difference-set is the sum of the counts from $DS(t_i, \cdot), \ldots, DS(t_k, \cdot)$.

Clusters. We use *clusters* to organize tuples (TIDs). Each cluster for an attribute A is a pair $\langle key, l\rangle$, where l is the list of tuples that share the same value *key* in A, and TIDs in l are stored in ascending order. The clusters for a numerical attribute are further sorted based on their *key* values in ascending order. For example, the clusters for *Acuity* are organized in a list [$<1, [t_3, t_8, t_{10}]>$, $<2, [t_2, t_4, t_5, t_6, t_7, t_9]>$, $<3, [t_1]>$]. Two operations are defined on clusters: $clus_A(t)$ returns the cluster in A that contains t, and $lastTuple(c_k)$ returns the last tuple in a cluster c_k. Both operations take $O(1)$. We present examples in Figure 1.

Caching. We adopt a caching technique to improve efficiency. A cache is created for each attribute, denoted as $cache_A$ for the cache corresponding to attribute A. Each element in $cache_A$ using $t_i[A]$ as its key takes $DS_A(t_i, \cdot)$ as its value (lines 11 and 15). If the key $t_i[A]$ is found in the cache (line 6), then $t_j[A] = t_i[A]$ for some $j < i$. If $j < i$, then the tuples that t_i needs to compare are a subset of the tuples that t_j needs to compare. Consequently, we can easily obtain $DS_A(t_i, \cdot)$ by simply removing t_k from the cached result for all $k \leq i$ (lines 7-8). The cached result is discarded if it will no longer be used, which happens when t_i is the last tuple in its cluster (lines 9-10). If the key $t_i[A]$ does not exist in the cache, then Algorithm AttBuilder is called to compute $DS_A(t_i, \cdot)$ (line 13). When memory is low, we prefer to discard cached results for numerical attributes or results with fewer future uses, *i.e.*, results whose key values belong to clusters containing fewer tuples after t_i (not shown).

Algorithm. We now present our algorithm, called AttBuilder (Algorithm 2), for computing $DS_A(t_i, \cdot)$. AttBuilder treats categorical and numerical attributes in different ways. For a categorical attribute A, it deals with all tuples after t_i one by one. For each t_j ($j > i$), it identifies the cluster containing t_j and all tuples after t_j (lines 6-7), as all of them produce the same difference-set with t_i. A set *checkedSet* is used to save those tuples (line 8), to avoid processing

[1]Our idea is inspired by the method to build *evidence sets* in DC discovery, but RFDs consider differences between values, while DCs [12, 13] concern orders of values. This distinction leads to significant differences in data structures and computation methods.

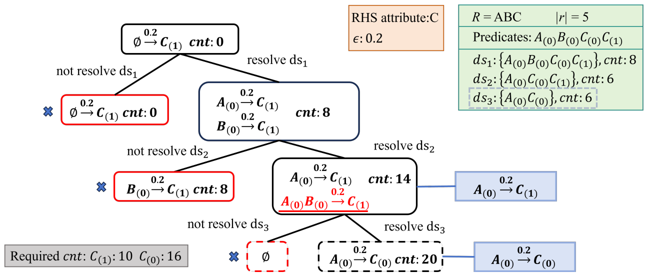
Algorithm 2: AttBuilder

```
   Input: tuple $t_i$ and attribute $A$
   Output: the partial difference-sets $DS_A(t_i, \cdot)$ for $t_i$ and $A$
1  $DS_A(t_i, \cdot) \leftarrow \emptyset$
2  if $A$ is a categorical attribute then
3      $checkedSet \leftarrow \emptyset$
4      foreach $t_j \in \{t_{i+1}, \ldots, t_{|r|}\}$ do
5          if $t_j \notin checkedSet$ then
6              $cls \leftarrow clus_A(t_j)$                // $t_j$'s cluster
7              $tidList \leftarrow \{t_k \mid t_k \in cls \wedge k \geq j\}$
8              $checkedSet \leftarrow checkedSet \cup \{t_k \mid t_k \in cls \wedge k \geq j\}$
9              $dist \leftarrow \Phi_A(t_i, t_j)$          // distance calculation
10             $m \leftarrow$ the largest satisfied threshold number according
                 to $dist$, or 0 if $dist$ is larger than the max threshold
11             add <$m$, $tidList$> into $DS_A(t_i, \cdot)$ if no <$m$, $L$> exists in
                 $DS_A(t_i, \cdot)$, otherwise merge $tidList$ into $L$
12 else                                         // $A$ is a numerical attribute
13     $clsList \leftarrow$ all clusters $cls$ such that the distance between $cls.key$
         and $t_i[A]$ is less than the maximum threshold
14     $othertidList \leftarrow [t_{i+1}, \ldots, t_{|r|}]$
15     foreach $cls \in clsList$ do
16         $t_j \leftarrow lastTuple(cls)$
17         if $j > i$ then
18             $dist \leftarrow \Phi_A(t_i, t_j)$          // distance calculation
19             $m \leftarrow$ the largest satisfied threshold number
20             $tidList \leftarrow \{t_k \mid t_k \in cls \wedge k > i\}$
21             $othertidList \leftarrow othertidList \setminus \{t_k \mid t_k \in cls \wedge k > i\}$
22             add <$m$, $tidList$> into $DS_A(t_i, \cdot)$ if no <$m$, $L$> exists in
                 $DS_A(t_i, \cdot)$, otherwise merge $tidList$ into $L$
23     add <0, $othertidList$> into $DS_A(t_i, \cdot)$
```

them again (line 5). After the required distance calculation, the threshold number is determined and put into $DS_A(t_i, \cdot)$ together with the TID list (lines 9-11). For a numerical attribute, AttBuilder first identifies all the clusters containing tuples that, when combined with t_i, can contribute tuple pairs that satisfy some predicates (line 13). As clusters for a numerical attribute are ordered, those clusters can be found by identifying a start position and an end position in the cluster list using a binary search (not shown), thus avoiding checking all the clusters. AttBuilder then processes the tuples in those clusters that come after t_i (lines 15-22). AttBuilder also identifies all the tuples that, when combined with t_i, *do not* satisfy any predicates during the same process (lines 14 and 21).

Example 7: Consider $DS_{Acuity}(t_3, \cdot)$. Since only tuples in the same cluster as t_3 can satisfy $Acuity_{(0)}$, it suffices to visit the cluster $<1, [t_3, t_8, t_{10}]>$. In $DS_{Acuity}(t_3, \cdot)$, $<1, [t_8, t_{10}]>$ represents tuples that, when combined with t_3, satisfy $Acuity_{(0)}$, while $<0, [t_4, t_5, t_6, t_7, t_9]>$ denotes tuples that do not satisfy any predicates with t_3. In DiffBuilder, $DS_{Acuity}(t_3, \cdot)$ is cached with "1" as the key. $DS_{Acuity}(t_8, \cdot)$ and $DS_{Acuity}(t_{10}, \cdot)$ are then obtained leveraging the cache. For instance, $DS_{Acuity}(t_8, \cdot) = \{<1, [t_{10}]>, <0, [t_9]>\}$; it retains only the tuples after t_8 from the cached result. □

Space and time complexity. (1) It takes $O(|DS(t_i, \cdot)|(|r| + |R|))$ space to compute $DS(t_i, \cdot)$. Threshold numbers for all the attributes and a TID list are saved within each difference-set. We use $|r|$ bits to store each TID list, where a bit set to 1 indicates that the corresponding tuple is in the list. $DS(t_i, \cdot)$ finally takes $O(|DS(t_i, \cdot)||R|)$



RHS attribute:C
ϵ: 0.2
R = ABC $|r|$ = 5
Predicates: $A_{(0)}B_{(0)}C_{(0)}C_{(1)}$
ds_1: $\{A_{(0)}B_{(0)}C_{(0)}C_{(1)}\}$, cnt: 8
ds_2: $\{A_{(0)}C_{(0)}C_{(1)}\}$, cnt: 6
ds_3: $\{A_{(0)}C_{(0)}\}$, cnt: 6
$\emptyset \xrightarrow{0.2} C_{(1)}$ cnt: 0
not resolve ds$_1$
resolve ds$_1$
$\emptyset \xrightarrow{0.2} C_{(1)}$ cnt: 0
$A_{(0)} \xrightarrow{0.2} C_{(1)}$ $B_{(0)} \xrightarrow{0.2} C_{(1)}$ cnt: 8
not resolve ds$_2$
resolve ds$_2$
$B_{(0)} \xrightarrow{0.2} C_{(1)}$ cnt: 8
$A_{(0)} \xrightarrow{0.2} C_{(1)}$ cnt: 14
$A_{(0)}B_{(0)} \xrightarrow{0.2} C_{(1)}$
$A_{(0)} \xrightarrow{0.2} C_{(1)}$
not resolve ds$_3$
resolve ds$_3$
Required cnt: $C_{(1)}$: 10 $C_{(0)}$: 16
$\emptyset$
$A_{(0)} \xrightarrow{0.2} C_{(0)}$ cnt: 20
$A_{(0)} \xrightarrow{0.2} C_{(0)}$

Figure 2: RFD generation

space after all TID lists are discarded. $DS(r)$ takes $O(|DS(r)||R|)$ space; usually, $|DS(r)| \ll |r|^2$, as experimentally verified (shown in Table 4). All cached results take $O(|r| \sum_{A \in R} |clus_A|(|Thr_A| + 1))$ space, where $|clus_A|$ denotes the number of clusters in A. Useless cached results are discarded promptly. DiffBuilder in total takes $O(\max_{t_i \in r} |DS(t_i, \cdot)|(|r|+|R|) + \sum_{A \in R} |clus_A|(|Thr_A|+1)|r| + |DS(r)||R|)$ space, since each $DS(t_i, \cdot)$ is computed in sequence.

(2) It takes $O(|r||R| + \sum_{A \in R} |clus_A| log(|clus_A|))$ time to build and sort clusters for all attributes. Assuming that all cache operations take $O(1)$ time, it requires $O(\sum_{A \in R} |clus_A|^2)$ time to calculate distances and $O(|r| \sum_{A \in R} |clus_A|(|Thr_A| + 1) + |r| \sum_{t_i \in r} |DS(t_i, \cdot)|)$ time to process TID lists, with clustering and caching techniques. The time complexity of DiffBuilder is the sum of the three parts. All the bitwise operations on TID lists, such as intersection and union, are very efficiently implemented with roaring bitmaps [29].

6 RFD DISCOVERY WITH DIFFERENCE-SET

In this section, we first provide an algorithm to discover all valid and minimal RFDs leveraging $DS(r)$ built by DiffBuilder, and then adapt it to discover only top-k RFDs based on a utility function.

Before detailing our algorithm, we explain several key technical aspects. For ease of understanding, we show each difference-set as a predicate set, regardless of our underlying implementation. We denote by $DS_{A_{\theta i}}(r)$ the subset of $DS(r)$ consisting of all the difference-sets that contain a given predicate $A_{\theta i}$, *i.e.*, $DS_{A_{\theta i}}(r) = \{ds \mid ds \in DS(r) \wedge A_{\theta i} \in ds\}$.

RFD refinement with difference-sets. In generating each RFD, our method begins by selecting a RHS predicate and then incrementally adds LHS predicates until the RFD becomes valid or is deemed impossible. We utilize difference-sets in $DS(r)$ to guide the process of adding LHS predicates, based on Proposition 1. As illustrated before in Example 6, for a RFD λ with $A_{\theta i}$ on its RHS, and a tuple pair tp with its difference-set ds: (a) The pair tp may potentially violate λ only when ds contains $A_{\theta i}$, *i.e.*, $ds \in DS_{A_{\theta i}}(r)$. (b) If ds contains $A_{\theta i}$ and λ contains no predicates from ds on its LHS, then tp violates λ. This violation can be resolved by adding to the LHS of λ any predicate from ds that is not on attribute A.

Example 8: Figure 2 illustrates our approach. We consider a relation containing 3 attributes and 5 tuples, and show all the predicates and difference-sets along with the number of tuple pairs that produce each difference-set (denoted by cnt). Given that the error threshold ϵ is 0.2, a valid RFD needs to be satisfied by a minimum of 16 out of 20 tuple pairs. In this example, we consider RFDs with $C_{(1)}$ on the RHS. Since $DS_{C_{(1)}}(r)$ only includes ds_1 and ds_2, any RFD is always satisfied by the 6 tuple pairs producing ds_3. Therefore, every valid RFD needs to be satisfied by 10 more tuple pairs.

We start with an empty LHS predicate set. (1) First consider ds_1. $\emptyset \xrightarrow{0.2} C_{(1)}$ is violated by the tuple pairs producing ds_1 because its LHS has no predicates from ds_1. Since a valid RFD can be violated by some tuple pairs, we have the option to resolve ds_1 or not, leading to two branches. In the branch where ds_1 is unresolved, the predicates in ds_1 cannot be used because using them will resolve ds_1. The branch terminates immediately due to the lack of available predicates. In the other branch, the 2 predicates within ds_1 that are not on attribute C are added individually to the LHS to generate 2 RFDs, each of which is satisfied by an additional 8 tuple pairs. (2) Next consider ds_2. $A_{(0)} \xrightarrow{0.2} C_{(1)}$ contains a LHS predicate from ds_2, and is confirmed as valid because the number of satisfying tuple pairs meets the requirement. In contrast, adding $A_{(0)}$ to the LHS of $B_{(0)} \xrightarrow{0.2} C_{(1)}$ results in a non-minimal RFD. □

Processing order of RHS predicates. We adopt a strategy to unify the processing of RFDs that use different RHS predicates on the same attribute, by addressing them in *descending order* of their RHS threshold values. This is because a smaller RHS threshold represents a stricter constraint, which is thus harder to satisfy.

Proposition 2: If $\theta^i > \theta^j$, then
(1) $\forall ds \in DS(r), A_{\theta^j} \in ds$ if $A_{\theta^i} \in ds$;
(2) $DS_{A_{\theta^i}}(r) \subseteq DS_{A_{\theta^j}}(r)$;

Example 9: (Example 8 continued.) We consider RFDs with $C_{(0)}$ on the RHS. The processing does not start from scratch as $DS_{C_{(1)}}(r) \subset DS_{C_{(0)}}(r)$; only branches that yield valid RFDs in Example 8 are promising ones. Since $ds_3 \in DS_{C_{(0)}}(r)$, every valid RFD with RHS predicate $C_{(0)}$ should be satisfied by 6 more tuple pairs compared to that with $C_{(1)}$. We begin with $A_{(0)} \xrightarrow{0.2} C_{(0)}$, which is already satisfied by the tuple pairs producing ds_3. Notably, the validity of $A_{(0)} \xrightarrow{0.2} C_{(0)}$ implies that $A_{(0)} \xrightarrow{0.2} C_{(1)}$ is no longer minimal. □

Algorithm. We present our algorithm, called RFDD (Algorithm 3), for finding the complete set of valid and minimal RFDs. We denote by $DS_A(r)$ the subset of $DS(r)$ consisting of all difference-sets that contain at least one predicate on attribute A, and denote the count associated with each difference-set ds by *ds.cnt*. For each attribute $A \in R$, RFDD first sorts difference-sets within $DS_A(r)$ and divides them into *stages* (lines 2-4). The rationale is that if $DS_A(r)[m]$ is the first difference-set that does not contain a predicate A_{θ^i}, then all the difference-sets before $DS_A(r)[m]$ belong to $DS_{A_{\theta^i}}(r)$. RFDD next builds an array *cntAry* with $|Thr_A|$ elements (lines 5-6). For a RFD λ with RHS predicate A_{θ^i}, difference-sets in $DS(r) \setminus DS_{A_{\theta^i}}(r)$ correspond to tuple pairs that always satisfy λ. Thus, *cntAry*$[i]$ represents the number of remaining tuple pairs that must satisfy λ to make λ valid. Finally, RFDD calls Procedure DSEnum to find all valid and minimal RFDs with RHS predicates on A (line 7). The meanings of every parameter will be explained shortly.

Algorithm. DSEnum (Algorithm 4) is presented to find all valid and minimal RFDs with RHS predicates on A, by utilizing difference-sets to generate RFDs and eliminating non-minimal or invalid RFDs through various effective rules. The enumeration of all difference-sets from $DS_A(r)$ is organized in a tree structure where each node

Algorithm 3: RFD Discovery (RFDD)

```
  Input: the set $DS(r)$ of distinct difference-sets, the set $\mathcal{P}$ of
         predicates, the set $Thr_A$ of similarity thresholds for each
         $A \in R$, and the error threshold $\epsilon$
  Output: the complete set $\Sigma$ of minimal and valid RFDs
1 foreach $A \in R$ do
2     sort difference-sets in $DS_A(r)$ in descending order, based on the
        number of predicates on $A$ contained in each difference-set
3     foreach $ds$ in $DS_A(r)$ do
4         $ds.stage \leftarrow$ the minimum $i$ such that $A_{\theta^i} \in ds$
5     for $i \leftarrow 1$ to $|Thr_A|$ do
6         $cntAry[i] \leftarrow (|r|^2\text{-}|r|)\times(1\text{-}\epsilon) - \sum_{ds \in DS(r) \setminus DS_{A_{\theta^i}}(r)} ds.cnt$
7     DSEnum($A$, 1, $cntAry$, $\mathcal{P} \setminus \{A_{\theta^k} \mid \theta^k \in Thr_A\}$, $\emptyset$)
```

corresponds to a (recursive) call of DSEnum. The parent-child relationship of nodes follows the order of difference-sets within $DS_A(r)$, similar to the demonstration shown in Figure 2. Each time DSEnum gets called, it employs the i-th difference-set to refine LHS predicate sets in Ω. The predicates available for use are restricted to the set *predSet*. Counts stored in *cntAry* are adjusted after RFD refinement and utilized to check the validity of RFDs. When first called in RFDD, DSEnum takes the first difference-set within $DS_A(r)$, an empty set Ω, and all predicates except those on A as *predSet*.

DSEnum first checks the current processing state. If the current results already form valid RFDs as indicated by *cntAry*, then they are added to the result set Σ (line 3). The operations at a node are immediately terminated if difference-sets have been used up, the predicate on A with the minimum threshold has been addressed, or no LHS predicate sets exist (line 4). If the current difference-set is in a different *stage* than the previous one, then DSEnum begins processing RFDs using a new RHS predicate with a smaller threshold; pruning is applied if no valid RFDs exist for the previous RHS predicate (lines 5-6). The LHS predicate sets that do not intersect with the current difference-set are collected (line 8).

Under the node for the i-th difference-set ds, the tree forks into two branches. (a) In the branch where ds is unresolved, the set of available predicates is adjusted because the predicates appearing in ds cannot be used afterwards (line 10). For those LHS predicate sets that have no more predicates available for refinement, DSEnum collects those already forming valid RFDs by invoking Procedure Check (lines 12-14). After that, DSEnum is recursively called (line 15). (b) In the branch where ds is resolved, counts stored in *cntAry* are decreased (lines 18-19). DSEnum enumerates all the available predicates, each time uses one of them to refine LHS predicate sets, and discards non-minimal ones immediately (lines 20-24). Predicate sets that cannot be further refined are handled similarly to those at the preceding branch (lines 25-26), while all the others are collected (line 27). Finally, DSEnum is recursively called (line 28).

Procedure Collect. Collect is invoked to add valid RFDs to Σ. It finds the smallest threshold value θ^k for the RHS predicate that still forms valid RFDs (lines 31-32), and combines minimality check to discard non-minimal RFDs (lines 34 and 36).

Procedure Check. Given a predicate set ω, Check identifies all the remaining difference-sets that intersect with ω, subtracts the counts of such difference-sets and calls Collect if valid RFDs are found.

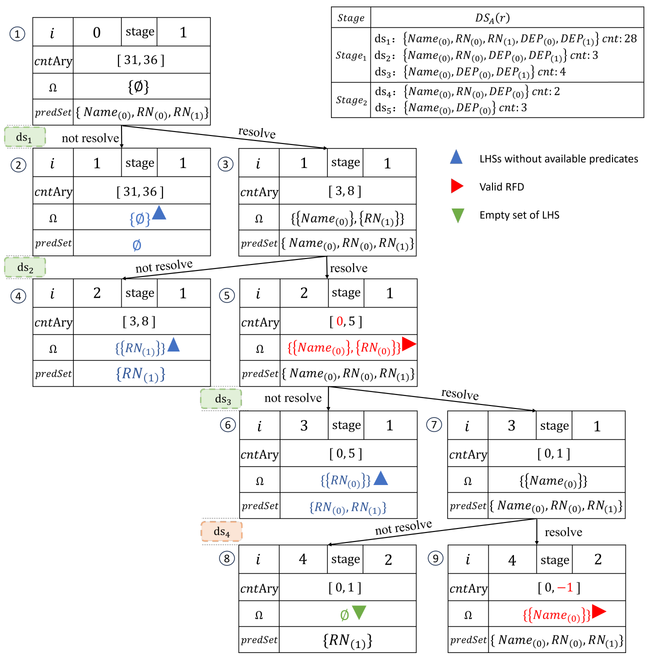
Algorithm 4: Difference-set Enumeration (DSEnum)

```
1  Procedure DSEnum($A$, $i$, $cntAry$, $predSet$, $\Omega$)
2      if $i = 1$ then $stage \leftarrow 1$ else $stage \leftarrow DS_A(r)[i\text{-}1].stage$
3      if $cntAry[stage] \leq 0$ then Collect($A$, $\Omega$, $stage$, $cntAry$)
4      if $i > |DS_A(r)| \vee cntAry[Thr_A|] \leq 0 \vee \Omega = \emptyset$ then return
5      $ds \leftarrow DS_A(r)[i]$; $curStage \leftarrow ds.stage$
6      if $curStage \neq stage \wedge cntAry[stage] > 0$ then return
7      $stage \leftarrow curStage$
8      $\Omega^- \leftarrow \{\omega \in \Omega \mid \omega \cap ds = \emptyset\}$
9      $\Omega \leftarrow \Omega \setminus \Omega^-$
       /* The branch where $ds$ is unresolved */
10     $predSet \leftarrow predSet \setminus ds$
11     if $\Omega^- \neq \emptyset$ then
12         foreach $\omega \in \Omega^-$ such that $predSet \setminus \omega = \emptyset$ do
13             $\Omega^- \leftarrow \Omega^- \setminus \omega$
14             Check ($i$+1, $\omega$, $stage$, $cntAry$)
15         DSEnum($A$, $i+1$, $cntAry$, $predSet$, $\Omega^-$)
16     recover the changes done in lines 10 and 13
       /* the branch where $ds$ is resolved */
17     if $predSet \cap ds \neq \emptyset$ then
18         for $k \leftarrow stage$ to $|Thr_A|$ do
19             $cntAry[k] \leftarrow cntAry[k]$ - $ds.cnt$
20         foreach $\omega \in \Omega^-$ do
21             foreach $p \in (ds \setminus \omega) \cap predSet$ do
22                 $\omega' \leftarrow \omega \cup \{p\}$
23                 if $\exists \eta \in \Omega$ where $\eta \succeq \omega'$ then continue
24                 remove all $\eta$ from $\Omega$ where $\omega' \succeq \eta$
25                 if $predSet \setminus \omega' = \emptyset$ then
26                     Check ($i$+1, $\omega'$, $stage$, $cntAry$)
27                 else $\Omega \leftarrow \Omega \cup \{\omega'\}$
28         DSEnum($A$, $i+1$, $cntAry$, $predSet$, $\Omega$)
29
30 Procedure Collect($A$, $candLHSs$, $stage$, $cntAry$)
31     for $k \leftarrow stage$ to $|Thr_A|$ do
32         if $cntAry[k] > 0$ then break
33     foreach $X \in candLHSs$ do
34         if $\nexists \eta \rightarrow A_{\theta j} \in \Sigma$ where $\eta \succeq X$ and $j \geq k$ then
35             $\Sigma \leftarrow \Sigma \cup \{X \rightarrow A_{\theta k}\}$
36             remove all $X \rightarrow A_{\theta j}$ from $\Sigma$ where $j < k$
37
38 Procedure Check($k$, $\omega$, $stage$, $cntAry$)
39     for $ds \leftarrow DS_A(r)[k]$ to $DS_A(r)[|Thr_A|]$ do
40         if $\omega \cap ds \neq \emptyset$ then
41             for $k \leftarrow ds.stage$ to $|Thr_A|$ do
42                 $cntAry[k] \leftarrow cntAry[k]$ - $ds.cnt$
43     if $cntAry[stage] \leq 0$ then Collect($A$, $\{\omega\}$, $stage$, $cntAry$)
```

Example 10: Consider the instance r in Table 1 and the predicate set in Table 2. We only take into account attributes DEP, $Name$ and RN in this example. We consider RFDs with RHS predicate on DEP and set $\epsilon = 0.1$. Figure 3 presents execution details of DSEnum.

(1) At node ①, $\Omega = \{\emptyset\}$, $predSet = \{Name_{(0)}, RN_{(0)}, RN_{(1)}\}$ and $cntAry = [31, 36]$. First consider node ② where ds_1 is not resolved. Since $predSet \setminus ds_1 = \emptyset$, no branches are forked under this node. Then consider node ③ where ds_1 is resolved. Each predicate from $predSet \cap ds_1$ is used as a LHS predicate set, except for $RN_{(0)}$.

(2) Two valid $LHSs$ $\{Name_{(0)}\}$ and $\{RN_{(0)}\}$ appear in node ⑤ for



Stage
$DS_A(r)$
$Stage_1$
ds$_1$: $\{Name_{(0)}, RN_{(0)}, RN_{(1)}, DEP_{(0)}, DEP_{(1)}\}$ cnt: 28
ds$_2$: $\{Name_{(0)}, RN_{(0)}, DEP_{(0)}, DEP_{(1)}\}$ cnt: 3
ds$_3$: $\{Name_{(0)}, DEP_{(0)}, DEP_{(1)}\}$ cnt: 4
$Stage_2$
ds$_4$: $\{Name_{(0)}, RN_{(0)}, DEP_{(0)}\}$ cnt: 2
ds$_5$: $\{Name_{(0)}, DEP_{(0)}\}$ cnt: 3
▲ LHSs without available predicates
▶ Valid RFD
▼ Empty set of LHS
① i 0 stage 1 $cntAry$ [31, 36] Ω $\{\emptyset\}$ $predSet$ $\{Name_{(0)}, RN_{(0)}, RN_{(1)}\}$
ds$_1$
not resolve
resolve
② i 1 stage 1 $cntAry$ [31, 36] Ω $\{\emptyset\}$ ▲ $predSet$ $\emptyset$
③ i 1 stage 1 $cntAry$ [3, 8] Ω $\{\{Name_{(0)}\}, \{RN_{(1)}\}\}$ $predSet$ $\{Name_{(0)}, RN_{(0)}, RN_{(1)}\}$
ds$_2$
not resolve
resolve
④ i 2 stage 1 $cntAry$ [3, 8] Ω $\{\{RN_{(1)}\}\}$ ▲ $predSet$ $\{RN_{(1)}\}$
⑤ i 2 stage 1 $cntAry$ [0, 5] Ω $\{\{Name_{(0)}\}, \{RN_{(0)}\}\}$ ▶ $predSet$ $\{Name_{(0)}, RN_{(0)}, RN_{(1)}\}$
ds$_3$
not resolve
resolve
⑥ i 3 stage 1 $cntAry$ [0, 5] Ω $\{\{RN_{(0)}\}\}$ ▲ $predSet$ $\{RN_{(0)}, RN_{(1)}\}$
⑦ i 3 stage 1 $cntAry$ [0, 1] Ω $\{\{Name_{(0)}\}\}$ $predSet$ $\{Name_{(0)}, RN_{(0)}, RN_{(1)}\}$
ds$_4$
not resolve
resolve
⑧ i 4 stage 2 $cntAry$ [0, 1] Ω $\emptyset$ ▼ $predSet$ $\{RN_{(1)}\}$
⑨ i 4 stage 2 $cntAry$ [0, −1] Ω $\{\{Name_{(0)}\}\}$ ▶ $predSet$ $\{Name_{(0)}, RN_{(0)}, RN_{(1)}\}$

Figure 3: Running example of DSEnum

the RHS predicate $DEP_{(1)}$ (stage 1), but the search continues to test $DEP_{(0)}$. This is finally verified at node ⑨, where $Name_{(0)} \xrightarrow{0.1} DEP_{(1)}$ is removed from Σ while $Name_{(0)} \xrightarrow{0.1} DEP_{(0)}$ is added to Σ in Procedure Collect. There is no need to further process ds_5 under node ⑨, as valid RFDs for the final stage (stage 2) have been found.

(3) At node ⑥, no predicates are available to refine the LHS predicate set $\{RN_{(0)}\}$. After $RN_{(0)} \xrightarrow{0.1} DEP_{(0)}$ is verified to be invalid in Check, branches under ⑥ are pruned.

Finally, $Name_{(0)} \xrightarrow{0.1} DEP_{(0)}$ and $RN_{(0)} \xrightarrow{0.1} DEP_{(1)}$ are discovered as valid and minimal RFDs. □

Proposition 3: Algorithm RFDD finds the complete set of valid and minimal RFDs.

Proof: (1) *Validity.* The validity of RFDs is checked at each node. All RFDs at a node use the same RHS predicate and resolve identical difference-sets. Each position in $cntAry$ corresponds to a specific RHS predicate, indicating the number of remaining tuple pairs to be satisfied by valid RFDs. A value of zero or less in any position indicates that the LHS predicate sets at that node, along with the associated RHS predicate, form valid RFDs.

(2) *Minimality.* The minimality of every valid RFD is checked when added to Σ (in Collect). This process involves first checking for any RFDs in Σ that hinder the minimality of the new RFD, and then removing all RFDs that are no longer minimal due to the new one. Only the equivalent LHS predicate set is considered in the second step. This is because the visit to a node that does not resolve a difference-set always occurs before the node that resolves the difference-set, ensuring that every LHS predicate set generated at a later node cannot be a proper subset of those at an earlier node.

(3) *Completeness.* RFDD enumerates every attribute and generates RFDs with RHS predicates on that attribute using DSEnum. For each

difference-set *ds*, DSEnum creates two branches: one that does not resolve *ds* and one that does. The resolving branch utilizes each available predicate to create new LHS predicate sets. The above process exhaustively generates all valid RFDs. The pruning rules only eliminate invalid or non-minimal RFDs. If a branch cannot yield valid RFDs with a RHS threshold, it cannot do so with a smaller one either (line 6). DSEnum checks if any LHS predicate set already forms a valid RFD (in Check) before discarding it (lines 14 and 26). Minimality checks occur within a node (lines 23-24) or when adding valid RFDs to Σ (in Collect), removing only non-minimal RFDs. □

Complexity. The complexity of RFDD is the sum of the complexities of DSEnum applied to each attribute. We measure the complexity of the enumeration algorithm DSEnum in the size of the search space. In the search tree for attribute A, the number of nodes is $O(2^{|DS_A(r)|})$ and the total number of distinct LHS predicate sets at all nodes is $O(2^{|\mathcal{P}_{\overline{A}}|})$, where $|\mathcal{P}_{\overline{A}}|$ denotes the number of predicates appearing in $DS_A(r)$ that are associated with attributes other than A. RFD generation only uses predicates within $DS_A(r)$, and in the two branches under a node, their LHS predicate sets are disjoint; in the branch where a difference-set is unresolved, predicate sets that intersect with that difference-set and its predicates are excluded from future processing. Operations on *cntAry* take $O(|DS_A(r)||Thr_A|)$ time in total. The minimality check takes $O(|\Omega|^2)$ within a node and $O(|\Sigma||\Omega|)$ when Procedure Collect is invoked at a node, where $|\Omega|$ denotes the number of LHS predicate sets at that node.

Remark. Enumerating valid RFDs inherently exhibits worst-case exponential complexity. The actual running complexity primarily depends on the number of RFDs generated for validation during the search and the overhead of removing non-minimal RFDs. In our approach, each refinement of a LHS predicate set is restricted to the predicates within a specific difference-set, thereby ensuring that the difference-set is resolved. This reduces the generation of invalid RFDs and allows us to keep track of the counts of tuple pairs that need to be satisfied instead of RFD validations; the complexity of RFDD is independent of $|r|$. We also eliminate branches that cannot generate valid RFDs as early as possible and perform minimality checks early at each node to substantially decrease the minimality check cost when adding valid RFDs to the result set. In contrast, the baseline method generates RFDs by enumerating combinations of predicates and validating them with difference-sets. This results in a significantly larger RFD space of $O(2^{|\mathcal{P}|-|Thr_A|})$ for each attribute A, consequently numerous invalid RFDs, and redundant, costly validation operations. Performing pairwise minimality checks among all valid RFDs can also be expensive.

Top-k discovery. We adapt RFDD to discover only top-k RFDs. This version, denoted by RFDD$_{\text{top}}$, aids users in selecting several *meaningful* RFDs, suitable for certain scenarios. We first present a *utility* function that combines some common metrics [8, 47, 50, 51]. The utility of a RFD λ is computed as follows:

$$utility(\lambda) = \frac{frac(RHS_\lambda) \cdot support(LHS_\lambda)}{|LHS_\lambda|}$$

(1) RHS threshold number: without loss of generality, assume that $RHS_\lambda = A_{\theta k}$. Recall that on A the minimum threshold of 0 corresponds to number $|Thr_A|$, while the maximum threshold corresponds to number 1. $frac(RHS_\lambda) = k/|Thr_A|$, where a large value indicates a strong constraint on the RHS and is desirable.

Table 4: Datasets and execution statistics (TL: 100^+ hours for dataset EQ, 10^+ hours for others)

Dataset Properties				Results		Running Time (seconds)		
Dataset	$\|r\|$	$\|R\|$	$\|\mathcal{P}\|$	$\|DS(r)\|$	$\|\Sigma\|$	RowRFD	ColRFD	FastRFD
Bridges	108	12+1	47	2,563	7,383	116	TL	**11.5**
Echo	132	13+0	47	5,389	14,249	743	TL	**148**
Iris	150	1+4	22	1,278	355	0.604	71.2	**0.538**
Foodstamp	150	0+5	14	99	32	0.086	5.5	**0.061**
Glass	214	0+11	47	10,358	25,084	3,579	TL	**651**
Balance	625	1+4	10	30	13	**0.125**	18.8	0.18
Restaurant	864	5+1	25	307	28	1.43	81.7	**0.387**
Car	1,728	7+0	18	1,466	381	2.02	6,112	**0.512**
Wine	4,898	0+9	26	39,859	1,435	774	TL	**676**
Abalone	4,177	1+8	37	23,545	790	1,154	TL	**1,010**
Emissions	8,088	0+5	16	251	37	40.5	6,518	**10.8**
Pcm	9,342	10+2	49	8,787	345	96.6	TL	**35.9**
Vocab	21k	1+4	6	24	16	72.1	4,643	**9.1**
Chess	28k	4+3	16	429	210	150	TL	**40.9**
Claim	112k	8+3	29	26,596	586	7,858	TL	**1,841**
Stock	122k	2+5	33	21,917	170	8,297	TL	**1,069**
Flight	150k	8+5	62	33,465	2,388	9,437	TL	**2,698**
Struct	169k	1+5	25	1,098	30	6,135	TL	**231**
EQ	1,000k	0+12	72	7,992	96	209,752	TL	**52,761**

(2) *Support*: support(LHS_λ) is the proportion of tuple pairs that satisfy all the LHS predicates, and measures how frequently λ can be applied. A high support indicates that more tuples are constrained by λ, implying better utility in downstream tasks.

(3) LHS cardinality: $|LHS_\lambda|$ is the number of LHS predicates. A small $|LHS_\lambda|$ implies succinctness, avoiding potential overfitting RFDs.

Implementation. Different from the other two metrics, the *support* is computationally expensive. Based on $DS(r)$, an inverted index is built to enhance efficiency. For each predicate, a bit array with $|DS(r)|$ elements is used, where a "1" at a position indicates that the corresponding difference-set in $DS(r)$ does not contain the predicate. With these bit arrays, the difference-sets that do not contain multiple predicates simultaneously can be efficiently obtained.

7 EXPERIMENTAL STUDY

In this section, we experimentally evaluate our approach.

Datasets. The tested datasets are shown in Table 4, and they have been extensively used in prior work [8, 15, 27, 41]. $|R|$ is given in the form of # textual attributes + # numerical attributes. We set similarity thresholds on each dataset in the same way as [8].

Algorithms. Our RFD discovery method, called FastRFD, is implemented in Java. It first constructs $DS(r)$ with DiffBuilder (Algorithm 1), and then finds RFDs with RFDD (Algorithm 3) based on $DS(r)$. FastRFD is compared with the following algorithms (all implemented in Java) in terms of efficiency and (or) effectiveness.

(1) Two baseline RFD discovery methods are developed based on column-wise and row-wise strategies respectively. (a) ColRFD is a non-trivial extension of the method Tane [20]. It enumerates candidate RFDs by traversing a lattice built upon the subsumption relationship of predicate sets, and then validates them via tuple pair comparisons. The results of distance calculations between strings are cached to avoid repeated calculations. (b) RowRFD is a non-trivial extension of the method FastFD [58]. It first builds

the difference-sets of all tuple pairs in a column-by-column fashion enhanced with clustering, and obtains $DS(r)$ by removing duplicate difference-sets. It then employs a simplified version of RFDD, referred to as RFDD$^-$, which deals with RFDs that use different RHS predicates on the same attribute independently.

(2) Dimϵ [10][2], the only known algorithm to find RFDs relaxing in both aspects. As stated in Section 2, the results of FastRFD and Dimϵ are not comparable, because they use different criteria to measure violations and Dimϵ cannot guarantee the completeness.

(3) Domino [8], PYRO [26], DAFDiscover [15] and FastDD [27], the SOTA discovery methods for RFDs with relaxation in value equality, RFDs with relaxation in constraint satisfaction, and DDs[3].

Running environment. We use a machine with an Intel Xeon E-2224 3.4G CPU, 64GB of memory and CentOS, and report the average of three runs. We set the error threshold ϵ to 0.01 by default.

Experimental results. We next report our findings.

Exp-1: FastRFD **against baseline methods.** Table 4 presents the runtime of algorithms. The time limit for the largest dataset EQ is 100 hours, and 10 hours for others; results exceeding these limits are marked as TL. We also show $|DS(r)|$ and $|\Sigma|$. Recall that $DS(r)$ is the intermediate result that connects the two parts of FastRFD (and RowRFD). FastRFD can well process datasets with different input and output scales and data distribution characteristics, and can be orders of magnitude faster than the compared algorithms.

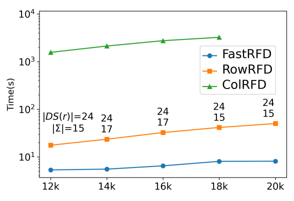
(1) ColRFD fails to process most datasets within the time limit, and on the datasets it can complete, it is on average more than two orders of magnitude slower than FastRFD. The results justify our choice of the row-based strategy: (a) RFD discovery is constrained by the inherently exponential complexity in $|\mathcal{P}|$. Row-based methods can perform much better compared to column-based ones when facing a large search space. (b) The repeated distance calculations in RFD validation are very costly even when distances between strings are cached in ColRFD; building difference-sets column-wise in FastRFD and RowRFD effectively avoids these redundancies.

(2) RowRFD is already a highly non-trivial row-based discovery method, but FastRFD still beats it by on average 5.2X and up to 26.6X. The advantage of FastRFD over RowRFD comes from the memory and operational benefits of keeping difference-sets in the condensed representation, and the benefit brought by the unified processing of RFDs using RHS predicates on the same attribute.

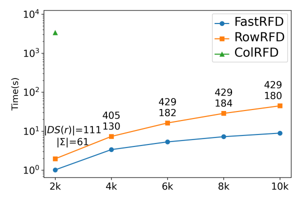
Exp-2: Scalability. We first compare the scalability of different algorithms, and then study our scalability in depth.

Scalability comparison. We compare the scalability by varying parameters and limit the running time to 1 hour.

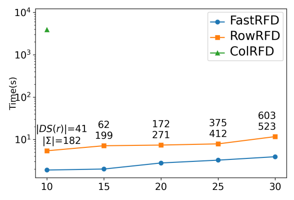
(1) Using Vocab and Chess, we study the impact of $|r|$ in Figure 4a and Figure 4b, respectively. ColRFD does not scale well on Vocab, exceeding the time limit when $|r|$ reaches 20k. FastRFD scales well: as $|r|$ increases from 12k to 18k, the time of FastRFD increases from 5.4s to 8.1s and the advantage of FastRFD over RowRFD (resp. ColRFD) expands from 3.3X to 5.2X (resp. 290X to 401X). On Chess,



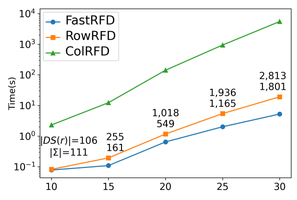
(a) Vocab: varying $|r|$



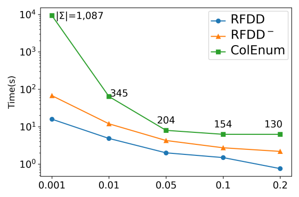
(b) Chess: varying $|r|$



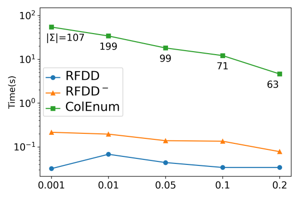
(c) Flight: varying $|\mathcal{P}|$



(d) Glass: varying $|\mathcal{P}|$



(e) Pcm: varying ϵ



(f) Bridges: varying ϵ

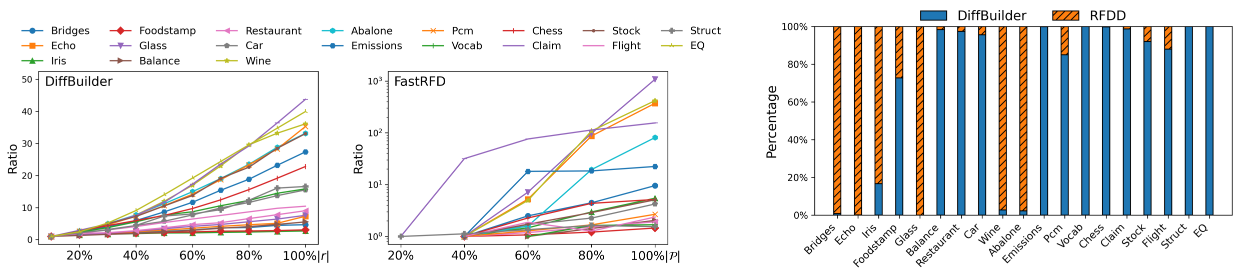
Figure 4: Scalability comparison

ColRFD can be completed within the time limit only when $|r| =$ 2K. As $|r|$ increases from 2k to 10k, the time taken by FastRFD increases from 1s to 8.9s, as opposed to 1.9s to 44s by RowRFD. The results verify the scalability advantage of FastRFD.

(2) We vary $|\mathcal{P}|$ by varying $|R|$ or considering more thresholds on certain attributes. When one predicate set is varied to another larger one, the inclusion relationship between them is ensured. In Figures 4c and 4d, we use Flight ($|r| = 5$k) and Glass to show two representative scenarios where either DiffBuilder or RFDD dominates FastRFD. On Flight, computing difference-sets takes a significant portion of the runtime of both FastRFD and RowRFD. The two methods show similar scalability since they both perform difference-set construction column-wise, but FastRFD is still on average 2.9X and up to 3.6X faster. With the increase of $|\mathcal{P}|$ on Glass, the time for inferring RFDs from difference-sets increases rapidly while that for difference-set construction only slightly increases due to the small instance size. The proportion of RFDD finally accounts for over 95% of the time of FastRFD (not shown). Overall, the time of FastRFD increases by 67X, compared to 2,387X for ColRFD and 233X for RowRFD. The advantage of FastRFD over ColRFD verifies the scalability advantage of row-based algorithms over column-based ones, while that of FastRFD over RowRFD mainly comes from the superiority of RFDD over RFDD$^-$.

(3) We finally study the impact of ϵ. Since ϵ only affects the runtime of RFDD, we conduct experiments to compare methods for obtaining RFDs based on $DS(r)$: (a) RFDD. (b) RFDD$^-$, the second part of RowRFD. (c) ColEnum, which enumerates RFDs in the same way as

[2]The code is at https://dastlab.github.io/dime/ (last accessed 2025/4/6).
[3]The code is at https://dast-unisa.github.io/Domino-SW/, https://github.com/HPI-Information-Systems/pyro, https://github.com/agithuber2023/DAFDiscover and https://github.com/TristonK/FastDD, respectively (last accessed 2025/4/6).



(a) varying $|r|$ or $|\mathcal{P}|$

(b) time decomposition

Figure 5: Scalability of our methods

ColRFD but validates them via $DS(r)$ based on Proposition 1. We select two datasets where the impact of ϵ on $|\Sigma|$ varies.

The results for Pcm are shown in Figure 4e. RFDD is on average 124X faster than ColEnum and 2.7X faster than RFDD$^-$. The former comparison verifies that using difference-sets to refine RFDs can be far more efficient than enumerating RFDs and then validating them based on difference-sets. As analyzed in Section 6, the former approach significantly reduces the generation of invalid RFDs, thereby greatly decreasing the search space, which is crucial for enumeration algorithms with exponential complexity. Additionally, the latter comparison shows that organizing RFDs based on the subsumption of their RHS predicates can further enhance performance. $|\Sigma|$ decreases as ϵ increases on Pcm. With the increase of ϵ, a previously valid RFD may be replaced by a more *generalized* RFD, *i.e.,* a RFD with a smaller threshold in its RHS predicate, fewer LHS predicates or larger threshold(s) in its LHS predicate(s). When multiple valid RFDs share the same generalization, the total number of discovered RFDs will decrease.

The results for Bridges shown in Figure 4f exhibit a trend that slightly differs. $|\Sigma|$ is 107 for ϵ = 0.001 but increases to 199 for ϵ = 0.01; the number of newly established valid RFDs exceeds that of RFDs that are removed due to not being minimal. The time of ColEnum (and RFDD$^-$) decreases because valid RFDs (even with a larger quantity) are discovered early in the traversal as ϵ increases. RFDD slightly degrades as ϵ changes from 0.001 to 0.01. We observe that a pruning rule in RFDD (line 6 of DSEnum) takes effect for ϵ = 0.001, making RFDD highly efficient, but it no longer works for ϵ = 0.01 when the restriction in RFD satisfaction is further relaxed.

More scalability results. Using all the datasets, we test the scalability of our methods in Figure 5. In Figure 5a, we calculate the ratio of running time as the proportions of $|r|$ and $|\mathcal{P}|$ change, using the first configuration of each dataset as the baseline. We display the time of DiffBuilder only when varying $|r|$, since $|r|$ does not directly affect RFDD (the input for RFDD is $DS(r)$, but not r). The scalability of DiffBuilder is much better than $O(|r|^2)$: when $|r|$ increases tenfold, the maximum growth ratio of the time is 43 and the median is 15.8. The differences in scalability across different datasets are primarily related to $|DS(r)|$. According to the complexity analysis provided in Section 5, if $|DS(r)|$ (and $|DS(t_i, \cdot)|$) increases significantly with $|r|$, it will reduce the benefits brought by difference-set compression, leading to higher computational complexity.

When adjusting $|\mathcal{P}|$, each attribute must have at least one predicate, so some datasets start experiments at 40% of $|\mathcal{P}|$. The time of FastRFD reflects the cumulative effects of DiffBuilder and RFDD. Since RFDD is much more sensitive to $|\mathcal{P}|$ than DiffBuilder, the time of FastRFD significantly increases in datasets where the time for RFDD constitutes the vast majority of FastRFD, possibly exhibiting exponential growth. Consequently, we observe significant differences in the scalability of FastRFD with respect to $|\mathcal{P}|$ across different datasets. In Figure 5b, we present the proportions of DiffBuilder and RFDD within FastRFD, highlighting a clear relationship with the scalability with respect to $|\mathcal{P}|$.

Exp-3: Handling dirty data. We compare different methods in their abilities of identifying FDs from dirty data. We first conduct FD discovery on r to identify the set Σ of minimal and valid FDs as the ground truth, and then inject errors into r to generate a dirty dataset r'. We next run each algorithm on r' and measure the *recall* (R), *precision* (P) and *F-measure* (F) of the result set relative to Σ.

We inject errors by selecting 5% tuples, and for each tuple, modifying all the values in the RHS attributes of FDs from Σ. Each value has a 50% chance of being modified to a new value at a distance of at most 2 from it, and a 50% chance of being replaced with another value from the active domain. We set similarity thresholds 0, 1, and 2 on each attribute for FastRFD and Domino. We test FastRFD and PYRO with ϵ = 0.01 and 0.001, and report the setting where both algorithms perform better. For DAFDiscover, the upper bound on the proportion of errors is set for each attribute based on our noise injection method. We set similarity threshold to 2 and ϵ to 0.1 for Dimϵ, as Dimϵ only supports one similarity threshold for all attributes and uses a different way to quantify violations. We also run exact FD discovery on r', since injecting errors does not imply that all the original valid FDs become invalid.

We report the results on three datasets in Table 5. Discovery methods for some kinds of RFDs usually provide better *recall* on dirty data compared to the exact discovery method. However, Dimϵ performs particularly poorly for Foodstamp. Only a small number of valid results exist in this dataset, and Dimϵ misses all of them because it does not guarantee completeness. Domino does not tolerate constraint violations, while PYRO and DAFDiscover do not support relaxation in value equality. Thus, when faced with dirty datasets that contain multiple types of errors, they often add too many attributes on the LHS (similar to overfitting) in the discovery process, which can notably reduce their precision in some cases.

Table 5: Comparison of effectiveness

Dataset		Method					
		FastRFD ($\epsilon = 0.001$)	Dimϵ [10] ($\epsilon = 0.1$)	Domino [8]	PYRO [26] ($\epsilon = 0.001$)	DAFDiscover [15]	ExactFD
Struct	R	**1.0**	0.62	0.62	**1.0**	**1.0**	0.56
	P	**0.76**	0.67	0.21	0.69	0.69	0.47
	F	**0.86**	0.64	0.31	0.81	0.81	0.51
Stock	R	0.96	0.42	0.90	**1.0**	**1.0**	0.82
	P	0.76	**0.84**	0.23	0.21	0.26	0.75
	F	**0.85**	0.57	0.37	0.35	0.41	0.78
Foodstamp	R	0.86	0	0.29	0.57	**1.0**	0.29
	P	**0.86**	0	0.14	0.40	0.50	0.14
	F	**0.86**	0	0.19	0.47	0.66	0.19

Table 6: Ranking RFDs

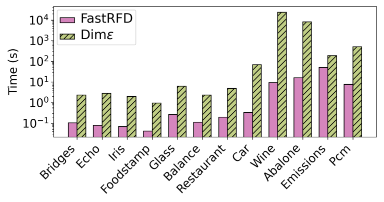
Dataset	Precision	Example
Foodstamp	0.8	λ_1: $income_{(10)} \xrightarrow{0.01} supplincome_{(0)}$
		λ_2: $participation_{(0)}, income_{(10)} \xrightarrow{0.01} tenancy_{(0)}$
Bridges	0.65	λ_3: $REL_{(0)}, type_{(2)} \xrightarrow{0.01} clear_{(0)}$
		λ_4: $erected_{(20)}, span_{(2)}, REL_{(0)} \xrightarrow{0.01} material_{(0)}$
Wine	0.8	λ_5: $FA_{(0.4)}, RS_{(4)}, TSD_{(4)} \xrightarrow{0.01} quality_{(0)}$
		λ_6: $FA_{(0.4)}, TSD_{(4)} \xrightarrow{0.01} sulphates_{(0.1)}$

FastRFD performs the best across all criteria in Struct, and achieves the best overall performance in Stock and Foodstamp, while other methods usually have significant limitations in either *precision* or *recall*. The results verify that compared to the other algorithms, FastRFD is more effective in identifying FDs from dirty data.

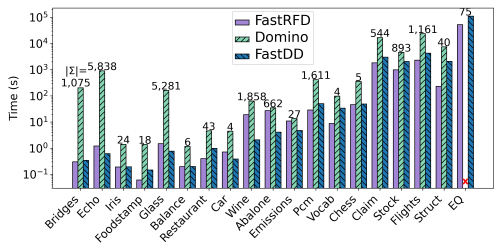
Exp-4: Top-k discovery. We verify top-k discovery by employing RFDD$_{\text{top}}$ in FastRFD. Using Foodstamp, Bridges and Wine, we identify top-20 RFDs on each dataset and manually judge the meaningfulness of them. In Table 6, we report the *precision*, which is the proportion of RFDs marked as meaningful. Based on our utility function, relatively satisfactory results can be obtained. The attribute descriptions and detailed discovery results are provided [30].

We showcase some results in Table 6. (1) λ_1 states that families with very similar incomes should receive the same treatment regarding whether they have supplemental incomes. The difference of 10 in the attribute *income* is small, as the values in *income* range from 0 to 2,995, while *supplincome* contains a boolean value. The corresponding exact FD and the version only relaxing the restriction in value equality hold with exceptions and are invalid. Although the version only relaxing the restriction in FD satisfaction is found valid, λ_1 is more general and applicable in practice because it does not require values in *income* to be strictly equal. (2) λ_4 states that bridges built during the same period, if they have similar spans and are either all elevated or all non-elevated, are constructed from the same material. The corresponding exact FD only applies to bridges with exactly the same year of construction and span, with much lower support. (3) λ_5 indicates that slight variations in the concentrations of fixed acidity, residual sugar, and total sulfur dioxide barely affect the quality. Both relaxations are necessary in this RFD.

The discovered RFDs can be utilized for enhancing data quality. For instance, data violating RFDs are clear targets in data cleaning, and attributes involving relaxed value equality may indicate the presence of data precision problems.



(a) FastRFD against Dimϵ



(b) FastRFD against Domino and FastDD

Figure 6: Runtime comparisons

Exp-5: FastRFD **against** Dimϵ**.** We modify FastRFD to use the same similarity threshold for all attributes as Dimϵ, and set ϵ to 0.1 for both algorithms. Dimϵ only supports datasets with $|r| \leq$ 35K and cannot process some datasets within a 10-hour limit. As reported in Figure 6a, FastRFD is on average 295X faster than Dimϵ, with a median speedup of 27 times. The results here and those from Exp-3 verify that FastRFD far outperforms Dimϵ in terms of both effectiveness and efficiency.

Exp-6: FastRFD **against** Domino **and** FastDD**.** By disabling the relaxation in constraint satisfaction with $\epsilon = 0$, FastRFD is adapted to find RFDs with relaxation only in value equality, just like Domino. FastDD [27] is also used to generate the same output, by considering only the "$\leq$" operator. The results in Figure 6b show that FastRFD significantly beats Domino and is on average 92.9X faster (Domino fails on EQ due to time limit). FastRFD is on average 1.77X faster than FastDD, outperforming it on 13 out of 19 datasets, with its advantage particularly evident in some of the more time-consuming datasets. FastRFD can also serve as an efficient solution to the discovery of RFDs relaxing restrictions only in value equality.

8 CONCLUSION

We consider RFDs relaxing restrictions in value equality and constraint satisfaction simultaneously, and have presented the first algorithm for discovering all valid and minimal RFDs. We have developed novel methods to build difference-sets and find RFDs based on difference-sets, and verified the effectiveness and efficiency of our approach through an extensive experimental evaluation.

We intend to extend our approach to distributed environments [46, 49], for addressing the scalability limitation of a single machine.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China 62172102, U24B20143 and U22B2021. For any correspondence, please refer to Zijing Tan and Shuai Ma.

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2017. Data Profiling: A Tutorial. In *SIGMOD*. 1747–1751.

[2] Ziawasch Abedjan, Lukasz Golab, Felix Naumann, and Thorsten Papenbrock. 2018. *Data Profiling*. Morgan & Claypool Publishers, San Rafael.

[3] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. 2014. DFD: Efficient Functional Dependency Discovery. In *CIKM*. 949–958.

[4] Tobias Bleifuß, Susanne Bülow, Johannes Frohnhofen, Julian Risch, Georg Wiese, Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. 2016. Approximate Discovery of Functional Dependencies for Large Datasets. In *CIKM*. 1803–1812.

[5] Tobias Bleifuß, Thorsten Papenbrock, Thomas Bläsius, Martin Schirneck, and Felix Naumann. 2024. Discovering Functional Dependencies through Hitting Set Enumeration. *Proc. ACM Manag. Data* 2, 1 (2024), 43:1–43:24.

[6] Bernardo Breve, Loredana Caruccio, Stefano Cirillo, Vincenzo Deufemia, and Giuseppe Polese. 2023. IndiBits: Incremental Discovery of Relaxed Functional Dependencies using Bitwise Similarity. In *ICDE*. 1393–1405.

[7] Loredana Caruccio and Stefano Cirillo. 2020. Incremental Discovery of Imprecise Functional Dependencies. *ACM J. Data Inf. Qual.* 12, 4 (2020), 19:1–19:25.

[8] Loredana Caruccio, Vincenzo Deufemia, Felix Naumann, and Giuseppe Polese. 2021. Discovering Relaxed Functional Dependencies Based on Multi-Attribute Dominance. *IEEE Trans. Knowl. Data Eng.* 33, 9 (2021), 3212–3228.

[9] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2016. Relaxed Functional Dependencies - A Survey of Approaches. *IEEE Trans. Knowl. Data Eng.* 28, 1 (2016), 147–165.

[10] Loredana Caruccio, Vincenzo Deufemia, and Giuseppe Polese. 2020. Mining relaxed functional dependencies from data. *Data Min. Knowl. Discov.* 34, 2 (2020), 443–477.

[11] Jixuan Chen, Yifeng Jin, Yihan Li, Zijing Tan, Weidong Yang, and Shuai Ma. 2023. Effective and Efficient Lexicographical Order Dependency Discovery. *IEEE Trans. Knowl. Data Eng.* 35, 9 (2023), 9700–9714.

[12] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Discovering Denial Constraints. *PVLDB* 6, 13 (2013), 1498–1509.

[13] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *ICDE*. 458–469.

[14] Xiaoou Ding, Yida Liu, Hongzhi Wang, Chen Wang, Yichen Song, Donghua Yang, and Jianmin Wang. 2024. Efficient Relaxed Functional Dependency Discovery with Minimal Set Cover. In *ICDE*. 3519–3531.

[15] Xiaoou Ding, Yixing Lu, Hongzhi Wang, Chen Wang, Yida Liu, and Jianmin Wang. 2024. DAFDiscover: Robust Mining Algorithm for Dynamic Approximate Functional Dependencies on Dirty Data. *Proc. VLDB Endow.* 17, 11 (2024), 3484 – 3496.

[16] Wenfei Fan and Floris Geerts. 2012. *Foundations of Data Quality Management*. Morgan & Claypool Publishers, San Rafael.

[17] Wenfei Fan, Floris Geerts, Jianzhong Li, and Ming Xiong. 2011. Discovering Conditional Functional Dependencies. *IEEE Trans. Knowl. Data Eng.* 23, 5 (2011), 683–698.

[18] Peter A. Flach and Iztok Savnik. 1999. Database Dependency Discovery: A Machine Learning Approach. *AI Commun.* 12, 3 (1999), 139–160.

[19] Lukasz Golab, Howard J. Karloff, Flip Korn, Divesh Srivastava, and Bei Yu. 2008. On generating near-optimal tableaux for conditional functional dependencies. *PVLDB* 1, 1 (2008), 376–390.

[20] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Comput. J.* 42, 2 (1999), 100–111.

[21] Ihab F. Ilyas and Xu Chu. 2019. *Data Cleaning*. ACM, New York City.

[22] Ihab F. Ilyas, Volker Markl, Peter J. Haas, Paul Brown, and Ashraf Aboulnaga. 2004. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *SIGMOD*. 647–658.

[23] Yifeng Jin, Zijing Tan, Jixuan Chen, and Shuai Ma. 2023. Discovery of Approximate Lexicographical Order Dependencies. *IEEE Trans. Knowl. Data Eng.* 35, 4 (2023), 3684–3698.

[24] Yifeng Jin, Lin Zhu, and Zijing Tan. 2020. Efficient Bidirectional Order Dependency Discovery. In *ICDE*. 61–72.

[25] Jyrki Kivinen and Heikki Mannila. 1992. Approximate Dependency Inference from Relations. In *ICDT*. 86–98.

[26] Sebastian Kruse and Felix Naumann. 2018. Efficient Discovery of Approximate Dependencies. *PVLDB* 11, 7 (2018), 759–772.

[27] Shulei Kuang, Honghui Yang, Zijing Tan, and Shuai Ma. 2024. Efficient Differential Dependency Discovery. *Proc. VLDB Endow.* 17, 7 (2024), 1552–1564.

[28] Philipp Langer and Felix Naumann. 2016. Efficient order dependency detection. *VLDB J.* 25, 2 (2016), 223–241.

[29] Daniel Lemire, Gregory Ssi Yan Kai, and Owen Kaser. 2016. Consistently faster and smaller compressed bitmaps with Roaring. *Softw. Pract. Exp.* 46, 11 (2016), 1547–1569.

[30] Mengran Li, Zijing Tan, Honghui Yang, and Shuai Ma. 2024. https://github.com/YukinoMR/FastRFD (last accessed 2025/4/10).

[31] Pei Li, Jaroslaw Szlichta, Michael H. Böhlen, and Divesh Srivastava. 2022. ABC of order dependencies. *VLDB J.* 31, 5 (2022), 825–849.

[32] Qiongqiong Lin, Yunfan Gu, Jingyan Sai, Jinfei Liu, Kui Ren, Li Xiong, Tianzhen Wang, Yanbei Pang, Sheng Wang, and Feifei Li. 2023. EulerFD: An Efficient Double-Cycle Approximation of Functional Dependencies. In *ICDE*. 2878–2891.

[33] Ester Livshits, Alireza Heidari, Ihab F. Ilyas, and Benny Kimelfeld. 2020. Approximate Denial Constraints. *PVLDB* 13, 10 (2020), 1682–1695.

[34] Panagiotis Mandros, Mario Boley, and Jilles Vreeken. 2017. Discovering Reliable Approximate Functional Dependencies. In *SIGKDD*. 355–363.

[35] Panagiotis Mandros, Mario Boley, and Jilles Vreeken. 2018. Discovering Reliable Dependencies from Data: Hardness and Improved Algorithms. In *ICDM*. 317–326.

[36] Panagiotis Mandros, David Kaltenpoth, Mario Boley, and Jilles Vreeken. 2020. Discovering Functional Dependencies from Mixed-Type Data. In *SIGKDD*. 1404–1414.

[37] Heikki Mannila and Kari-Jouko Räihä. 1994. Algorithms for Inferring Functional Dependencies from Relations. *Data Knowl. Eng.* 12, 1 (1994), 83–99.

[38] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *PVLDB* 8, 10 (2015), 1082–1093.

[39] Thorsten Papenbrock and Felix Naumann. 2016. A Hybrid Approach to Functional Dependency Discovery. In *SIGMOD*. 821–833.

[40] Eduardo H. M. Pena, Eduardo Cunha de Almeida, and Felix Naumann. 2019. Discovery of Approximate (and Exact) Denial Constraints. *PVLDB* 13, 3 (2019), 266–278.

[41] Eduardo H. M. Pena, Fábio Porto, and Felix Naumann. 2022. Fast Algorithms for Denial Constraint Discovery. *PVLDB* 16, 4 (2022), 684–696.

[42] Frédéric Pennerath, Panagiotis Mandros, and Jilles Vreeken. 2020. Discovering Approximate Functional Dependencies using Smoothed Mutual Information. In *SIGKDD*. 1254–1264.

[43] Chaoqin Qian, Menglu Li, Zijing Tan, Ai Ran, and Shuai Ma. 2023. Incremental discovery of denial constraints. *VLDB J.* 32, 6 (2023), 1289–1313.

[44] Joeri Rammelaere and Floris Geerts. 2018. Explaining Repaired Data with CFDs. *PVLDB* 11, 11 (2018), 1387–1399.

[45] Joeri Rammelaere and Floris Geerts. 2018. Revisiting Conditional Functional Dependency Discovery: Splitting the "C" from the "FD". In *ECML PKDD 2018*. 552–568.

[46] Hemant Saxena, Lukasz Golab, and Ihab F. Ilyas. 2019. Distributed Implementations of Dependency Discovery Algorithms. *Proc. VLDB Endow.* 12, 11 (2019), 1624–1636.

[47] Philipp Schirmer, Thorsten Papenbrock, Ioannis K. Koumarelas, and Felix Naumann. 2020. Efficient Discovery of Matching Dependencies. *ACM Trans. Database Syst.* 45, 3 (2020), 13:1–13:33.

[48] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. 2019. DynFD: Functional Dependency Discovery in Dynamic Datasets. In *EDBT*. 253–264.

[49] Sebastian Schmidl and Thorsten Papenbrock. 2022. Efficient distributed discovery of bidirectional order dependencies. *VLDB J.* 31, 1 (2022), 49–74.

[50] Shaoxu Song and Lei Chen. 2011. Differential dependencies: Reasoning and discovery. *ACM Trans. Database Syst.* 36, 3 (2011), 16:1–16:41.

[51] Shaoxu Song, Lei Chen, and Hong Cheng. 2014. Efficient Determination of Distance Thresholds for Differential Dependencies. *IEEE Trans. Knowl. Data Eng.* 26, 9 (2014), 2179–2192.

[52] Shaoxu Song, Fei Gao, Ruihong Huang, and Chaokun Wang. 2022. Data Dependencies Extended for Variety and Veracity: A Family Tree. *IEEE Trans. Knowl. Data Eng.* 34, 10 (2022), 4717–4736.

[53] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. 2017. Effective and Complete Discovery of Order Dependencies via Set-based Axiomatization. *PVLDB* 10, 7 (2017), 721–732.

[54] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. 2018. Effective and complete discovery of bidirectional order dependencies via set-based axioms. *VLDB J.* 27, 4 (2018), 573–591.

[55] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. 2013. Expressiveness and Complexity of Order Dependencies. *PVLDB* 6, 14 (2013), 1858–1869.

[56] Ziheng Wei, Sven Hartmann, and Sebastian Link. 2021. Algorithms for the discovery of embedded functional dependencies. *VLDB J.* 30, 6 (2021), 1069–1093.

[57] Ziheng Wei and Sebastian Link. 2019. Discovery and Ranking of Functional Dependencies. In *ICDE*. 1526–1537.

[58] Catharine M. Wyss, Chris Giannella, and Edward L. Robertson. 2001. FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances. In *DaWaK*. 101–110.

[59] Renjie Xiao, Zijing Tan, Haojin Wang, and Shuai Ma. 2022. Fast approximate denial constraint discovery. *Proc. VLDB Endow.* 16, 2 (2022), 269–281.

[60] Renjie Xiao, Yong'an Yuan, Zijing Tan, Shuai Ma, and Wei Wang. 2022. Dynamic Functional Dependency Discovery with Dynamic Hitting Set Enumeration. In *ICDE*. 286–298.

[61] Yunjia Zhang, Zhihan Guo, and Theodoros Rekatsinas. 2020. A Statistical Perspective on Discovering Functional Dependencies in Noisy Data. In *SIGMOD*. 861–876.