Efficient Differential Dependency Discovery

Shulei Kuang
School of Computer Science, Fudan University, China
21210240017@m.fudan.edu.cn

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

Zijing Tan*
School of Computer Science, Fudan University, China
zjtan@fudan.edu.cn

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

ABSTRACT

Differential dependencies (DDs) are proposed to specify constraints on the *differences* between values, where the semantics of *difference* can be "similar", "dissimilar" and beyond. DDs subsume functional dependencies (FDs), and find valuable applications in tasks such as violation detection, duplicate identification, and quantitative data cleaning, among others. In this paper we present an efficient DD discovery method for finding hidden DDs from data. We encode differences between values in a novel structure called the "diff-set", and present a set of techniques for constructing the diff-set, discovering valid DDs with set cover enumeration of the diff-set, and eliminating non-minimal DDs. Our extensive experimental evaluation verifies that our method outperforms the existing DD discovery method up to orders of magnitude. Furthermore, our method is adapted to discover an important subclass of DDs, known as *relaxed* FDs (RFDs), and is also up to orders of magnitude faster than the state-of-the-art RFD discovery method.

PVLDB Reference Format:
Shulei Kuang, Honghui Yang, Zijing Tan, and Shuai Ma. Efficient Differential Dependency Discovery. PVLDB, 17(7): 1552 - 1564, 2024.
doi:10.14778/3654621.3654624

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

1 INTRODUCTION

Data profiling techniques [1, 2] aim to find hidden meta-data from datasets, and are actively studied in the literature due to their practical demands. Data dependencies are one of the most important types of meta-data, and hence, methods for discovering dependencies have drawn much attention in recent years.

In this paper, we tackle the problem of discovering differential dependencies (DDs). DDs [44] are proposed to specify constraints on the *differences* between values, a departure from dependencies that only concern the equality of values, *e.g.,* functional dependencies (FDs). DDs subsume not only FDs but also some variants of FDs that relax the equality to "similarity", *e.g.,* relaxed functional dependencies (RFDs) [5] and metric functional dependencies (MFDs) [25].

The formal definition of DDs will be reviewed in Section 3. Below we give an example to illustrate the form and usefulness of DDs.

Example 1: Relation instance r_1 in Table 1 shows house information. Each tuple carries the address, type, numbers of bedrooms and bathrooms, and area of a house. Due to an input error, there is a typo in the attribute Type of tuple t_4 (the typo is shown in red after the right arrow). We showcase a few DDs holding on r_1.

(1) φ_1 = [Address ($\leq$ 0)] $\rightarrow$ [Type ($\leq$ 1)]. This DD states that for two houses with the same address, the difference between their values in Type should be no more than 1. Without loss of generality, we assume that the *edit distance* is used to measure the difference between strings. This DD applies to tuples with the same value in Address and *very similar* values in Type, which is necessary to deal with the typo in tuple t_4. Note the FD Address $\rightarrow$ Type, *i.e.,* the DD [Address ($\leq$ 0)] $\rightarrow$ [Type ($\leq$ 0)], does not hold.

(2) φ_2 = [Type ($\leq$ 1)] $\wedge$ [Bedroom ($\leq$ 1)] $\rightarrow$ [Area ($\leq$ 25)]. This DD states that for two houses of the same type (tolerating the typo), the difference between theirs values in Area should be no more than 25, if the difference between their values in Bedroom is no more than 1. We herein use *absolute difference values* for numerical attributes Bedroom and Area. Note *thresholds* are inferred from the instance, *e.g.,* "25" is the difference between values of t_5 and t_6 in Area.

(3) φ_3 = [Type ($\leq$ 1)] $\wedge$ [Bathroom ($>$ 1)] $\rightarrow$ [Bedroom ($>$ 2)]. This DD states a constraint on two houses of the same type: the difference between their values in Bedroom should be larger than 2, if the difference between their values in Bathroom is larger than 1. Besides the semantics of "similar" expressed with operator "$\leq$", this DD exhibits the semantics of "dissimilar" with "$>$".

DDs can be used in various data management tasks. By allowing small variances in values, DDs can serve all use cases that RFDs and MFDs can serve. For example, φ_1 reveals a hidden *determinant* relationship between Address and Type, which cannot be captured by FDs. Moreover, as t_3 and t_4 share identical values in all attributes except for Type, a *duplicate detection* method [26] can utilize this DD to determine that t_3 and t_4 refer to the same entity and merge them. DDs can state complex constraints on the difference between values, and hence also lend themselves well to *quantitative* data cleaning tasks [38]. Constraints concerning orders of attributes, *e.g.,* denial constraints (DCs) [9], can state that if a house t has more bedrooms than another house t', then the area of t should be larger than t'. However, DCs cannot specify constraints on the difference between areas of the two houses, making it difficult to find a value suitable for cleaning an erroneous cell in area. Incorporating DDs

*Zijing Tan is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 7 ISSN 2150-8097.
doi:10.14778/3654621.3654624

Table 1: House Information (relation instance r_1)

	Address	House Type (Type)	Bedroom	Bathroom	Area (m^2)
t_1	Apt. 1603, No 16, 225 Handan Road	Apartment	1	1	65
t_2	Apt. 901, No 11, 225 Handan Road	Apartment	2	1	80
t_3	Apt. 502, No 1, 225 Handan Road	Apartment	4	2	155
t_4	Apt. 502, No 1, 225 Handan Road	Apartment→Aparment	4	2	155
t_5	Unit 3, 1850 Songhu Road	Townhouse	4	3	275
t_6	Unit 12, 833 Guohong Road	Townhouse	3	2	250
t_7	Unit 156, 899 Jiangwan Road	Detached House	5	3	350
t_8	Unit 222, 1555 Zhongqing Road	Detached House	8	5	630

into data cleaning processes with DCs [9, 10, 16, 17, 40] can improve the accuracy, since considering constraints specified by DDs helps provide proper attribute values. □

Manually designing DDs is necessarily tedious and error-prone, even for experts. In fact, it is often impractical due to the large number of relations and the presence of attribute names that lack semantic meaning in database systems. With this comes the need for DD discovery algorithms that can automatically find DDs from data. However, DD discovery involves a much larger search space than FD discovery, as it considers combinations of attributes and thresholds, with multiple thresholds possible for each attribute. Additionally, computing the difference between values is required in DD discovery, rather than simply checking their equivalence in FD discovery. These challenges make discovering DDs much more complex than discovering FDs. Our experimental findings indicate that existing DD discovery methods do not scale well on real-life datasets, highlighting the need for a more efficient discovery method.

Contributions & Organizations. In this paper, we present a new and efficient DD discovery method.

(1) We introduce our DD discovery algorithm, which is based on an innovative framework (Section 4). Our approach introduces the concept of "diff-set" and reformulates the DD discovery problem for a given instance r as *set cover enumeration* of the diff-set of r, along with *minimality* check operations for DDs. We lay out the theoretical foundation of our approach.

(2) We provide efficient techniques to construct the diff-set (Section 5). We give a condensed representation of the diff-set to encode results of *differential functions*, and propose to compute the diff-set in a column by column manner enhanced with auxiliary structures.

(3) We present a novel method that finds valid DDs with set cover enumeration of the diff-set, and combines minimality checks to identify minimal DDs (Section 6).

(4) We conduct an extensive experimental evaluation to verify our approach (Section 7). Our DD discovery method significantly outperforms existing ones [44] up to orders of magnitude. We also adapt our method to the discovery of an important subclass of DDs, known as RFDs. Our adapted version is far more efficient than the state-of-the-art method for discovering RFDs [5].

2 RELATED WORK

Dependency discovery methods have been extensively studied in the literature for many kinds of dependencies. See, *e.g.*, some recent works [3, 14, 22–24, 27, 31, 35–37, 39, 41–43, 51, 53–56, 58, 60, 61]. In this section, we investigate works close to ours.

Foundation of DDs. The definition of DDs is proposed in [44], together with related theoretical issues, such as the implication problem and a sound and complete inference system for DDs. They are used to define *minimal* and *valid* DDs, as the target of DD discovery in [44] and this work.

The relationship between DDs and other dependencies. DDs are different from constraints concerning *orders* of attribute values, *e.g.,* DCs [9] and order dependencies (ODs) [18, 19, 47–50]. DCs and ODs are related to the order of values, *e.g.,* $t.A > s.A$, but not their difference, *e.g.,* $|t.A - s.A|$. DDs also differ from sequential dependencies (SDs) [20]. An SD in the form of $X \rightarrow_g Y$ states that when tuples are sorted on X, the distance between the values in Y of two successive tuples should be within a given threshold g. Matching dependencies (MDs) [12, 13] also concern the *similarity* (difference) of values, but are proposed for record matching across possibly different relations: if some attributes of tuples *match* then the tuples should have the same values in some other attributes, where the *match* is defined in terms of similarity operators.

FDs concern *equality* of values, which is a special case of *difference*. There are many variants of FDs studied in the literature; please refer to [6, 46] for surveys on the topic. In particular, relaxed functional dependencies (RFDs) [5] generalize the equality of values to the similarity of them. A RFD $A_{\phi_1} \rightarrow B_{\phi_2}$ states that if tuples have similar values in A *w.r.t.* a *similarity function* ϕ_1, then their values in B should also be similar *w.r.t.* a similarity function ϕ_2. RFDs generalize metric functional dependencies (MFDs) [25], since MFDs only permit variations in RHS attribute values. DDs use *differential functions* for expressing the semantics of *similarity*, *dissimilarity* and beyond, and hence subsume FDs, MFDs and RFDs.

Discovery of DDs. The first algorithm to discover DDs is presented in [44], which is built upon a *column-based* framework originally proposed for discovering FDs [21]. It traverses the search space of candidate DDs according to a lattice, and presents rules to prune invalid or non-minimal DDs. The DD discovery method given in [28] assumes a user-defined threshold is used as the upper-bound of distance intervals of the left-hand-side (LHS) differential functions. Another method [29] relates the discovery of DDs to association rules, and adopts a measure of *interestingness* to prune the search space. [28, 29] do not aim for the complete set of minimal valid DDs, and only find a subset of the DDs discovered by [44].

Our work discovers the same complete set of minimal valid DDs as [44], and hence differs from [28, 29]. Compared to [44], our method differs in the following. (1) Our approach can be regarded as a highly non-trivial generalization of *row-based* approaches to FD discovery [32, 33, 59]. Discovering DDs requires to consider multiple differential functions on the same attribute, resulting in a much larger search space than discovering FDs. Row-based approaches usually outperform column-based ones in terms of the scalability with the size of the search space [34]. These observations inspire the design of our method. (2) We present a set of novel techniques underlying our approach, including an encoding scheme, and efficient methods for diff-set construction and for recasting DD discovery as set cover enumeration of the diff-set plus minimality checks.

As noted earlier, RFDs are a subclass of DDs. To our best knowledge, the state-of-the-art RFD discovery method is given in [5]. It first compares all tuple pairs to compute results of similarity functions, and then exploits the idea of *dominance* to infer RFDs. Our DD discovery method can be easily modified to discover only RFDs.

3 PRELIMINARIES

In this section, we review notations of DDs [44]. We use R to denote a relational schema (an attribute set), r to denote an instance of R, t, s to denote tuples in r, and t_A to denote the value of t in $A \in R$.

Distance measure. For $A \in R$, $dom(A)$ denotes the domain of A. A *distance measure* d_A can be defined on A: $d_A(u, v)$ is a value that measures the *difference* between u and v. The measure d_A should have four properties: (a) non-negativity, (b) identity, (c) symmetry and (d) triangle inequality. There are many distance measures studied in the literature [8, 11], *e.g.,* the absolute difference for numerical values and the edit distance for string values.

Differential function [44]. A *(singleton) differential function* $\phi[A]$ specifies a *constraint* on the difference between attribute values in A, based on d_A. Specifically, $\phi[A]$ is in the form of $[A\ (op\ \theta)]$, where the operator $op \in \{\leq, >\}$ and θ is a threshold. We say a tuple pair (t, s) satisfies $\phi[A] = [A\ (op\ \theta)]$ if $d_A(t_A, s_A)\ op\ \theta$, written as $(t, s) \asymp \phi[A]$. Otherwise, we write $(t, s) \not\asymp \phi[A]$. Since d_A satisfies symmetry, $(s, t) \asymp \phi[A]$ iff $(t, s) \asymp \phi[A]$.

A differential function $\phi[X]$ defined on a set of attributes $X \subseteq R$ is the conjunction of constraints on the difference between values in $A_i \in X$; that is, $\phi[X] = \bigwedge_{A_i \in X} \phi_i[A_i]$. A tuple pair (t, s) satisfies $\phi[X] = \bigwedge_{A_i \in X} \phi_i[A_i]$, written as $(t, s) \asymp \phi[X]$, if $(t, s) \asymp \phi_i[A_i]$ for every $A_i \in X$. We write $(t, s) \not\asymp \phi[X]$, if $(t, s) \not\asymp \phi_i[A_i]$ for any $A_i \in X$.

Example 2: Consider r_1 in Table 1. For ϕ[Type] = [Type ($\leq$ 1)] where $d_{Type}(u, v)$ is the edit distance between u and v, $(t_3, t_4) \asymp \phi$[Type] but $(t_3, t_5) \not\asymp \phi$[Type]. For ϕ[Type, Bedroom] = [Type ($\leq$ 1)] $\wedge$ [Bedroom (> 2)] where $d_{Bedroom}(u, v)$ is the absolute difference value between u and v, we have $(t_7, t_8) \asymp \phi$[Type, Bedroom]. □

Differential dependency [44]. A differential dependency (DD) is in the form of $\phi_L[X] \rightarrow \phi_R[A]$, where $X \subseteq R$, $A \in R \setminus X$, and $\phi_L[X]$ and $\phi_R[A]$ are differential functions on X and A, respectively. A tuple pair (t, s) satisfies $\phi_L[X] \rightarrow \phi_R[A]$, iff $(t, s) \asymp \phi_R[A]$ if $(t, s) \asymp \phi_L[X]$. For an instance r of R, we say $\phi_L[X] \rightarrow \phi_R[A]$ holds (is valid) on r, written as $r \models \phi_L[X] \rightarrow \phi_R[A]$, iff every pair (t, s) in r^2 satisfies $\phi_L[X] \rightarrow \phi_R[A]$. A DD states that for any two tuples, if the difference between their values in X satisfies the constraint specified by $\phi_L[X]$, then the difference between their values in A should also satisfy the constraint specified by $\phi_R[A]$.

Discovery methods typically aim for only *minimal* dependencies. The minimality of DDs is based on the *subsumption* of differential functions [44]. Specifically, a differential function $\phi[X]$ is said to *subsume* another function $\phi'[Y]$, written as $\phi[X] \succeq \phi'[Y]$, if $\forall (t, s)$, we have $(t, s) \asymp \phi[X]$ if $(t, s) \asymp \phi'[Y]$. For example, [Type ($\leq$ 2)] subsumes (a) [Type ($\leq$ 2)] $\wedge$ [Bedroom (> 1)], (b) [Type ($\leq$ 1)] and (c) [Type ($\leq$ 0)] $\wedge$ [Bedroom (> 3)]. Note the subsumption concerns not only *set containment*, but also operators and thresholds. We write $\phi[X] \succ \phi'[Y]$, if $\phi[X] \succeq \phi'[Y]$ and $\phi[X] \neq \phi'[Y]$.

Minimal DD. On a given instance r, a DD $\gamma = \phi_L[X] \rightarrow \phi_R[A_i]$ is *minimal* if there does not exist a distinct DD $\gamma' = \phi'_L[Y] \rightarrow \phi'_R[A_i]$ holding on r, such that $\phi'_L[Y] \succeq \phi_L[X]$ and $\phi_R[A_i] \succeq \phi'_R[A_i]$.

Intuitively, γ' imposes a "weaker" constraint on the LHS and a "stronger" constraint on the RHS than γ. It is easy to prove that γ holds on r if γ' holds on r. Since the validity of γ' always guarantees that of γ, only γ' is output by a DD discovery method.

Determining differential functions. To construct the search space of DD discovery, differential functions must be determined. The domain of an attribute usually suggests a distance measure on the attribute, and proper thresholds are the key to meaningful differential functions. Criteria to determine similarity thresholds have been studied not only in the context of DDs [44, 45] but also in those of RFDs and MDs [5, 42]. Below we briefly review them.

(1) *Thresholds from data* [5, 42]. Rather than ask users to provide thresholds, thresholds can be inferred from the given instance. Different thresholds can be used on the same attribute. For example, [Bedroom($\leq$ 1)] and [Bedroom($\leq$ 3)] can denote different degrees of similarity, while [Bedroom($\leq$ 0)] denotes equality.

(2) *Support* [5, 42, 44, 45]. The support of a differential function is the proportion of tuple pairs satisfying the function. When the function is used as the LHS of a DD, the support measures the proportion of tuple pairs the DD applies to. A threshold is usually preferable if it leads to functions with high support.

(3) *Dependent quality* [45]. An improper threshold can incur a meaningless differential function. For example, if $d_A(u, v)$ is always smaller than 10 for any u, v in $dom(A)$, then using $[A\ (\leq 10)]$ as the RHS of a DD is not interesting. This is because the DD always holds no matter what LHS function is used, but it is not clear whether the RHS indeed *depends on* the LHS.

With a given instance r, we assume a set Ψ of *singleton* differential functions, *i.e.,* $\phi[A]$, $\forall A \in R$, is determined in a pre-processing step and taken as an input of our DD discovery. Our method does not depend on any specific techniques to determine differential functions, and can support arbitrary similarity measures and configurable thresholds.

DD discovery. With a set Ψ of singleton differential functions, the problem of DD discovery is to find the complete set of minimal valid DDs on a given instance r.

4 FRAMEWORK OF OUR APPROACH

In this section, we present and justify the framework of our DD discovery approach.

Overview. Consider our discovery framework shown in Figure 1. As noted in Section 3, a set Ψ of singleton differential functions is determined on a sample of the given instance r in a pre-processing step. Taking Ψ and r as inputs, our discovery method finds the

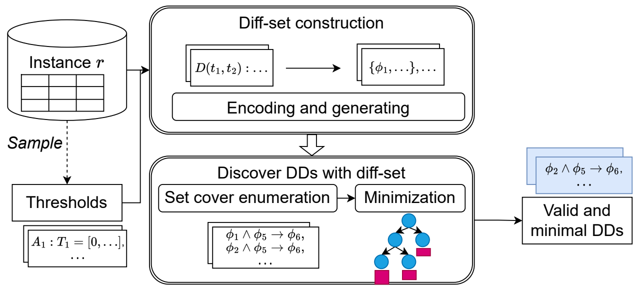


Figure 1: Overview of our DD discovery method

complete set of minimal valid DDs, consisting of two phases. In the first phase, a data structure called the "diff-set" is built, for encoding results of differential functions with respect to tuple pairs from r^2. Based on a novel encoding scheme, we develop efficient techniques for generating the diff-set, as detailed in Section 5. In the second phase, we recast discovering valid DDs as *enumerating set covers* of the diff-set and take additional operations to eliminate non-minimal DDs, as detailed in Section 6.

In the rest of this section, we give the theoretical foundation of our two-phase approach, starting with the definition of diff-set.

Diff-set *w.r.t.* differential functions. With a given set Ψ of singleton differential functions $\phi[A], \forall A \in R$, (1) the diff-set of a tuple pair (t, s) from r^2 is $D(t, s) = \{ \phi[A] \in \Psi \mid (t, s) \not\asymp \phi[A] \}$, *i.e.,* the set of differential functions that (t, s) does *not* satisfy. Note $D(t, s) = D(s, t)$ since $d_A(t_A, s_A) = d_A(s_A, t_A)$. (2) The diff-set of an instance r is $D_r = \{D(t, s) \mid t, s \in r, D(t, s) \neq \emptyset\}$, *i.e.,* the set of non-empty and distinct diff-sets of tuple pairs from r^2.

Example 3: In Table 2, we give an example set Ψ of differential functions for the instance r_1 in Example 1. It can be verified that (a) $D(t_1, t_7) = \{ \phi_1, \phi_4, \phi_5, \phi_6, \phi_8, \phi_9, \phi_{11}, \phi_{12}, \phi_{14}, \phi_{15}, \phi_{16} \}$; and (b) $D(t_3, t_8) = D(t_1, t_7)$. □

Note D_r is a set of sets, and each element of D_r consists of differential functions from Ψ. Also note that the size $|D_r|$ of D_r is usually much smaller than $|r|^2$ ($|r|$ is the number of tuples in r), because different tuple pairs can produce the same diff-set.

Valid DD and diff-set. For a function $\phi[A] \in \Psi$, we use $D_r(\phi[A])$ to denote the set $\{ U \mid U \in D_r \wedge \phi[A] \in U\}$, *i.e.,* the subset of D_r with only diff-sets that contain $\phi[A]$. Each U in $D_r(\phi[A])$ is a diff-set produced by a tuple pair (or several tuple pairs with the same diff-set) that does *not* satisfy $\phi[A]$. A key observation is that if $\phi[A]$ is used as the RHS of a DD, then at least another function in U must be used in the LHS to make the tuple pair(s) satisfy the DD.

Example 4: (Example 3 continued.) Recall $\phi_{16} \in D(t_1, t_7)$; the pair (t_1, t_7) does *not* satisfy ϕ_{16}. Consider the DD φ_2 = [Type ($\leq$ 1)] $\wedge$ [Bedroom ($\leq$ 1)] $\rightarrow$ [Area ($\leq$ 25)], *i.e.,* $\phi_5 \wedge \phi_9 \rightarrow \phi_{16}$, which has ϕ_{16} on the RHS. It is satisfied by (t_1, t_7), since ϕ_5 (and also ϕ_9) belongs to $D(t_1, t_7)$ and is used on the LHS of φ_2. As a counter example, (t_1, t_7) does *not* satisfy $\phi_{10} \rightarrow \phi_{16}$; $\phi_{10} \notin D(t_1, t_7)$. □

Formally, we have the following result that establishes the connection between valid DDs with $\phi[A]$ on the RHS and $D_r(\phi[A])$.

Proposition 1: Suppose $\phi_L[X] = \bigwedge_{A_i \in X} \phi_i[A_i]$. $\phi_L[X] \rightarrow \phi[A]$ ($A \notin X$) is a valid DD on r, iff for each $U \in D_r(\phi[A])$, there exists some $\phi_i[A_i]$ in $\phi_L[X]$ such that $\phi_i[A_i] \in U$.

Table 2: Example Differential Functions

ϕ_1: *Address*(≤ 0)	ϕ_2: *Address*(> 0)	ϕ_3: *Address*(> 4)
ϕ_4: *Type* (≤ 0)	ϕ_5: *Type*(≤ 1)	ϕ_6: *Type*(≤ 9)
ϕ_7: *Type*(> 9)	ϕ_8: *Bedroom*(≤ 0)	ϕ_9: *Bedroom*(≤ 1)
ϕ_{10}: *Bedroom*(> 2)	ϕ_{11}: *Bathroom*(≤ 0)	ϕ_{12}: *Bathroom*(≤ 1)
ϕ_{13}: *Bathroom*(> 1)	ϕ_{14}: *Bathroom*(> 3)	ϕ_{15}: *Area*(≤ 0)
ϕ_{16}: *Area*(≤ 25)	ϕ_{17}: *Area*(> 90)	ϕ_{18}: *Area*(> 210)

Proof: By definition, we prove every tuple pair from r^2 satisfies $\phi_L[X] \rightarrow \phi[A]$, iff for each $U \in D_r(\phi[A])$, there exists some $\phi_i[A_i]$ in $\phi_L[X]$ such that $\phi_i[A_i] \in U$.

(1) We prove every tuple pair satisfies $\phi_L[X] \rightarrow \phi[A]$, if each U in $D_r(\phi[A])$ contains some $\phi_i[A_i]$. Every pair whose diff-set does not belong to $D_r(\phi[A])$ obviously satisfies $\phi_L[X] \rightarrow \phi[A]$. For a pair (t, s) whose diff-set belongs to $D_r(\phi[A])$, there must exist some $\phi_i[A_i] \in D(t, s)$ according to the assumption. We know (t, s) satisfies $\phi_L[X] \rightarrow \phi[A]$, because $(t, s) \not\asymp \phi_L[X]$ if $(t, s) \not\asymp \phi_i[A_i]$.

(2) We prove each U in $D_r(\phi[A])$ must contain some $\phi_i[A_i]$, if every tuple pair satisfies $\phi_L[X] \rightarrow \phi[A]$. $D_r(\phi[A])$ is empty if all pairs satisfy $\phi[A]$. Otherwise, for U in $D_r(\phi[A])$, without loss of generality, let $U = D(t, s)$. We know (t, s) satisfies $\phi_L[X] \rightarrow \phi[A]$ according to the assumption. The pair (t, s) cannot satisfy $\phi_L[X]$ since $(t, s) \not\asymp \phi[A]$. Therefore, (t, s) must dissatisfy some $\phi_i[A_i]$ in $\phi_L[X]$, which implies that $\phi_i[A_i] \in D(t, s) = U$. □

DD discovery with set cover enumeration. If we take $\phi_L[X]$ as a subset and $D_r(\phi[A])$ as a subset family, both defined on Ψ, then Proposition 1 tells us that if $\phi[A]$ is used as the RHS of a valid DD, then the LHS of the DD, *i.e.,* $\phi_L[X]$, intersects with every element of $D_r(\phi[A])$. Such $\phi_L[X]$ is referred to as a *set cover*, *a.k.a.* hitting set, of $D_r(\phi[A])$ in the literature. Discovering all valid DDs with $\phi[A]$ on the RHS is related to finding all set covers of $D_r(\phi[A])$, *i.e.,* the problem of set cover enumeration [15, 30].

Please note that $\phi_L[X]$ is a set cover of $D_r(\phi[A])$, but the reverse is not always true. This is because there may be multiple differential functions on the same attribute, while a DD can use at most one differential function for each attribute by definition. A special treatment is needed when finding valid DDs with set cover enumeration. A more intricate issue concerns the minimality. A set cover is minimal if no subset of it is also a set cover. We may aim for minimal DDs directly from minimal set covers. However, a minimal cover does not always imply a minimal DD. This is because the minimality of set covers is built upon set containment, while the minimality of DDs concerns the *subsumption* of differential functions.

Example 5: It can be verified that {[Type ($\leq$ 0)], [Bedroom ($\leq$ 0)]} is a minimal set cover of D_r([Area($\leq$ 25)]). However, [Type ($\leq$ 0)] $\wedge$ [Bedroom ($\leq$ 0)] $\rightarrow$ [Area ($\leq$ 25)] is not a minimal DD. This is because {[Type ($\leq$ 1)], [Bedroom ($\leq$ 1)]} is also a minimal set cover and [Type ($\leq$ 0)] $\wedge$ [Bedroom ($\leq$ 0)] $\rightarrow$ [Area ($\leq$ 25)] is *not* minimal if [Type ($\leq$ 1)] $\wedge$ [Bedroom ($\leq$ 1)] $\rightarrow$ [Area ($\leq$ 25)] is valid. □

Hence, additional minimality checks are needed for identifying minimal DDs from DDs discovered with set cover enumeration.

Remarks. Our approach can be regarded as a highly non-trivial extension of *row-based* techniques for FD discovery [32, 33, 59, 61]. We highlight the differences as follows. (1) FDs only concern the equality of values, making FD discovery a special case of DD discovery; Ψ contains only functions of the form $\phi[A] = [A(\leq 0)]$. The consideration of multiple differential functions on one attribute and the computation of functions beyond equality significantly complicate the construction of the diff-set, as detailed in Section 5. (2) There is a one-to-one relationship between a minimal valid FD and a minimal set cover [59, 61]. In contrast, finding minimal valid DDs requires special treatment in set cover enumeration and additional minimality check operations, as noted earlier. We will present novel techniques to address the issues in Section 6.

5 DIFF-SET CONSTRUCTION

In this section we present techniques for diff-set construction. We provide a novel scheme to encode every diff-set of tuple pair with a condensed representation, present a method to build the diff-set of r in a column-by-column fashion, and partition data for dealing with large datasets and building diff-set with parallelism.

Encoding of diff-set. Recall the diff-set $D(t,s)$ is the set of differential functions that (t,s) does not satisfy. During the stage of diff-set construction, we adopt a novel encoding scheme to save $D(t,s)$ as an *integer*. This condensed representation reduces the memory usage, which in turn improves the efficiency of diff-set construction. In the sequel we assume $R = \{A_1, A_2, \ldots, A_{|R|}\}$, where $|R|$ is the number of attributes of R.

With the given set Ψ of singleton differential functions, thresholds used in functions on an attribute A_i are known (suppose we use 0 in $[A_i\ (\leq 0)]$ for every A_i, to express the semantics of "equality" on A_i). We sort these thresholds in ascending order, and save them in a list denoted by T_i. We use $|T_i|$ to denote the number of elements in T_i, and $T_i[k]$ to denote the k-th element of T_i ($0 \leq k \leq |T_i| - 1$). The thresholds are employed to generate $|T_i| + 1$ *intervals*, *i.e.,* $[0, 0]$, $(T_i[0]{=}0, T_i[1]], \ldots, (T_i[|T_i| - 1], \infty)$. For each interval, we assign an *interval sequence number* (ISN) to it, which is in the range of $[0, |T_i|]$. Every distance value belongs to exactly one interval. We use $\#_{A_i}(dist)$ to denote the ISN on attribute A_i for a distance value *dist*, which is formally defined in Equation 1.

$$\#_{A_i}(dist) = \begin{cases} 0 & dist = 0 \\ k & T_i[k-1] < dist \leq T_i[k] \\ |T_i| & dist > T_i[|T_i| - 1] \end{cases} \quad (1)$$

For a tuple pair (t,s), $\#_{A_i}(d_{A_i}(t_{A_i}, s_{A_i}))$ determines whether each differential function on A_i is satisfied by the pair or not. Taken together, the set of ISNs for attributes of R determines $D(t,s)$.

Proposition 2: Two tuple pairs (t,s) and (t',s') have the same ISN for every $A_i \in R$, iff $D(t,s) = D(t',s')$.

We further encode all ISNs of (t,s) into an *integer*, as a condensed representation of $D(t,s)$. The computation of the encoding is given in the following Equations. We also use $D(t,s)$ to denote the *code* of $D(t,s)$, when it is clear from the context. To simplify the presentation, we denote by a_i the ISN on A_i, *i.e.,* $a_i = \#_{A_i}(d_{A_i}(t_{A_i}, s_{A_i}))$.

$$S_i = \prod_{k=1}^{i}(|T_k| + 1) \qquad (1 \leq i \leq |R| - 1) \quad (2)$$

$$D(t,s) = a_1 + a_2 \times S_1 + \ldots + a_{|R|} \times S_{|R|-1} \quad (3)$$

Except for a_1, each a_i is associated with a weight S_{i-1} computed with Equation (2), and the weighted sum of all a_i is used as the code of $D(t,s)$ (Equation 3). The rationale is that a_i can be computed from the code reversely, as shown below. In the equation, *mod* and *div* denote remainder and integer division, respectively.

$$a_i = \begin{cases} D(t,s)\ mod\ S_1 & i = 1 \\ (D(t,s)\ mod\ S_i)\ div\ S_{i-1} & 1 < i < |R| \\ D(t,s)\ div\ S_{|R|-1} & i = |R| \end{cases} \quad (4)$$

Example 6: For the set Ψ of differential functions shown in Table 2, let $R = \{A_1$ = Address, A_2 = Type, A_3 = Bedroom, A_4 = Bathroom, A_5 = Area$\}$. We have $T_1 = [0,4]$, $T_2 = [0,1,9]$, $T_3 = [0,1,2]$, $T_4 = [0,1,3]$, and $T_5 = [0,25,90,210]$. As an example, T_5 is used to generate 5 intervals, *i.e.,* [0,0], (0,25], (25, 90], (90, 210], (210, ∞). According to Equation (2), $S_1 = 3$, $S_2 = 3 \times 4 = 12$, $S_3 = 3 \times 4 \times 4 = 48$, and $S_4 = 3 \times 4 \times 4 \times 4 = 192$.

Now consider a pair (t_1, t_7). Let $dist_i = d_{A_i}(t_1[A_i], t_7[A_i])$ and $a_i = \#_{A_i}(dist_i)$. We have $a_5 = 4$, since $dist_5 = 350 - 65 = 285$ and $dist_5 \in (210, \infty)$. Similarly, $a_1 = 2$, $a_2 = 3$, $a_3 = 3$ and $a_4 = 2$. According to Equation (3), $D(t_1, t_7) = 2 + 3 \times 3 + 3 \times 12 + 2 \times 48 + 4 \times 192 = 911$. We can recall a_i ($i \in [1,5]$) from the code of $D(t_1, t_7)$ with Equation (4). Specifically, $a_5 = 911\ div\ 192 = 4$, $a_4 = (911\ mod\ 192)\ div\ 48 = 2$, $a_3 = (911\ mod\ 48)\ div\ 12 = 3$, $a_2 = (911\ mod\ 12)\ div\ 3 = 3$, and $a_1 = 911\ mod\ 3 = 2$. □

Remarks. We highlight benefits of our encoding scheme. (i) Saving ISNs is usually more memory-efficient than saving distance values, and encoding all ISNs into one integer further reduces memory footprint. (ii) The diff-set D_r of r consists of distinct diff-sets of tuple pair. Based on Proposition 2, duplicate diff-sets can be efficiently identified by checking the equivalence of their codes (integers). (iii) Since every S_i in Equation (2) can be pre-computed, the computation of Equation (3) is very efficient. Besides, according to Equation (3) the code of $D(t,s)$ can be *incrementally* computed, each time for an a_i. This enables us to compute diff-sets of tuple pair in a column-by-column fashion, as illustrated below.

Computing diff-set column by column. It is more efficient to build the diff-set column by column, which puts computations concerning the same attribute together. Additionally, auxiliary structures can be created to speed up the computations. To make our solution as general as possible, we do not leverage indexing techniques designed for specific metrics [8]. Instead, we employ two simple yet effective optimizations that apply to most attribute types.

(1) For an attribute $A_i \in R$, our first optimization is the *clustering* method that puts all tuples with the same value in A_i in the same cluster. Tuples in the same cluster have *no* difference between their values in A_i, and all tuple pairs across the same two clusters have the same difference. Computing distance measures for cluster pairs is usually much more efficient than tuple pairs, because the number of clusters is typically much smaller than the number of tuples and the cost of clustering is linear in the number of tuples.

(2) Our second optimization applies to ordered attributes, *e.g.,* numerical attributes, time and date. For distance measures on these attributes, a common property is that if t is before t' and t' is before t'' after sorting by an ordered attribute A_i, then the distance

between values of t and t' in A_i is no greater than that of t and t'' in A_i. We can exploit this property to reduce computations.

Auxiliary structures. Before giving details of our algorithm, we present auxiliary structures used in it. We use *position list index* (Pli) [21, 27, 35] to save clusters. We denote the Pli on attribute A_i by π_{A_i}, which is a set of *clusters*. Each cluster is a pair $\langle k, l \rangle$, where k is a value in $dom(A_i)$ and l is the set of tuples with the same value k in A_i. Only tuple identifiers (*ids*) are saved in l to reduce memory footprint. For ordered attributes, we further sort clusters in π_{A_i} by k in descending order, resulting in a list of clusters.

Example 7: For the instance r_1 in Table 1, $\pi_{\text{Bedroom}} = [\langle 8, \{t_8\}\rangle, \langle 5, \{t_7\}\rangle, \langle 4, \{t_3, t_4, t_5\}\rangle, \langle 3, \{t_6\}\rangle, \langle 2, \{t_2\}\rangle, \langle 1, \{t_1\}\rangle]$ is a list of clusters, while π_{Type} is a set of clusters. □

Algorithm. BuildDiff (Algorithm 1) takes as input the instance r, and outputs the encoding of diff-set D_r. Storing $D(t_j, t_k)$ for $j < k$ suffices since $D(t_j, t_k) = D(t_k, t_j)$. Initially we set all $D(t_j, t_k) = 0$ (line 1). $D(t_j, t_k) = 0$ iff t_j, t_k have the same values in all attributes (the interval sequence number (ISN) on every attribute is 0). Hence, $D(t_j, t_k)$ needs to be updated for an attribute A_i, if t_j, t_k have different values in A_i. Attributes of R are processed one by one, and a Pli structure is built for each of them (lines 3 and 9).

We first consider *non-ordered* attributes, *e.g.,* textual attributes. For an attribute A_i and two clusters c_m, c_n in π_{A_i}, the ISN for $d_{A_i}(c_m.k, c_n.k)$ is first computed (line 6), and Procedure Update is then called to update diff-sets of all tuple pairs across c_m and c_n (line 7, lines 23-26). Our encoding scheme naturally supports *incremental* updates. With the ISN *seqNumber*, $D(t_j, t_k)$ is updated by adding the product of *seqNumber* and S_{i-1} to it (lines 22 and 26).

We then consider ordered attributes, *e.g.,* numerical attributes, time and date. For each cluster c_m and each threshold $T_i[j]$, we find cluster c_{end} that is the first cluster after c_{start} such that $d_{A_i}(c_{end}.k, c_m.k) > T_i[j]$ (lines 10-13). All tuple pairs across c_m and a cluster between c_{start} and c_{end} satisfy the same set of differential functions on A_i (lines 14-15), so do tuple pairs across c_m and a cluster that is either the final c_{end} *w.r.t.* c_m or after the final c_{end} (lines 17-18). Note the required ISNs on A_i are directly obtained with positions in T_i (lines 15 and 18). Since clusters in π_{A_i} are sorted, there are additional optimizations. The technique of binary search is employed to find c_{end} (line 13), and after processing $T_i[j]$, the treatment for $T_i[j+1]$ starts from the cluster where the previous search stops (line 16).

Example 8: (Example 6 continued.) For attribute A_2 (Type), $\pi_{A_2} = \{$ c_1: ⟨Apartment, $\{t_1, t_2, t_3\}$⟩, c_2: ⟨Aparment, $\{t_4\}$⟩, c_3: ⟨Townhouse, $\{t_5, t_6\}$⟩, c_4: ⟨ Detached House, $\{t_7, t_8\}$⟩ $\}$. We have $d_{A_2}(c_1.k, c_2.k) = 1$, and hence $\#_{A_2}(d_{A_2}(c_1.k, c_2.k)) = 1$. The Procedure Update is called to update $D(t_1, t_4)$, $D(t_2, t_4)$ and $D(t_3, t_4)$ accordingly.

For attribute A_4 (Bathroom), $\pi_{A_4} = [c_1\text{: }\langle 5, \{t_8\}\rangle, c_2\text{: }\langle 3, \{t_5, t_7\}\rangle, c_3\text{: }\langle 2, \{t_3, t_4, t_6\}\rangle, c_4\text{: }\langle 1, \{t_1, t_2\}\rangle]$. No updates are caused by c_1 and $T_4[0] = 0$, or by c_1 and $T_4[1] = 1$. For c_1 and $T_4[2] = 3$, cluster c_4 is found since $d_{A_4}(c_4.k, c_1.k) > 3$. All tuple pairs across c_1 and c_2 or across c_1 and c_3 are processed by calling Update with *seqNumber* = 2. There are no more thresholds on A_4. Hence, all tuple pairs across c_1 and c_4 are processed by calling Update with *seqNumber* = 3. □

Complexity. Operations in BuildDiff mainly consist of three parts. (1) Building clusters with hashing takes $O(|r|)$, and sorting clusters for an ordered attribute A_i additionally takes $O(|\pi_{A_i}|\log(|\pi_{A_i}|))$,

---

Algorithm 1: Build the diff-set D_r of r (BuildDiff)

Input: the relational instance r
Output: the encoding of diff-set D_r of r

1 $D_r \leftarrow$ an array of $|r|(|r|-1)/2$ elements where all elements are 0
2 **foreach** *non-ordered attribute* $A_i \in R$ **do**
3   build π_{A_i} for A_i
4   **foreach** *cluster* $c_m \in \pi_{A_i}$ **do**
5   **foreach** *cluster* $c_n \in \pi_{A_i} \setminus c_m$ **do**
6   *seqNumber* $\leftarrow \#_{A_i}(d_{A_i}(c_m.k, c_n.k))$
7   Update(*seqNumber*, A_i, c_m, c_n, D_r)
8 **foreach** *ordered attribute* $A_i \in R$ **do**
9   build π_{A_i} for A_i
10   **foreach** *cluster* $c_m \in \pi_{A_i}$ **do**
11       $c_{start} \leftarrow c_m$
12   **foreach** *threshold* $T_i[j] \in T_i$ **do**
13          $c_{end} \leftarrow$ the first cluster after c_{start} such that $d_{A_i}(c_{end}.k, c_m.k) > T_i[j]$
14   **foreach** *cluster* c_n *between* c_{start} *and* c_{end} **do**
            // excluding c_{start} and c_{end}
15   Update(j, A_i, c_m, c_n, D_r)
16          $c_{start} \leftarrow c_{end}$
17   **foreach** *cluster* c_n *such that* $c_n = c_{end}$ *or* c_n *is after* c_{end} **do**
18   Update($|T_i|$, A_i, c_m, c_n, D_r)
19 $D_r \leftarrow$ distinct diff-sets of tuple pair in D_r
20
21 **Procedure** Update(*seqNumber*, A_i, c_1, c_2, D_r)
22    $\triangle diff \leftarrow$ *seqNumber* $\times S_{i-1}$
23   **foreach** *tuple* $t_j \in c_1.l$ **do**
24   **foreach** *tuple* $t_k \in c_2.l$ **do**
25   **if** A_i *is a non-ordered attribute* **and** $j < k$ **then**
26             $D(t_j, t_k) \leftarrow D(t_j, t_k) + \triangle diff$
27   **if** A_i *is an ordered attribute* **then**
28   **if** $j < k$ **then** $D(t_j, t_k) \leftarrow D(t_j, t_k) + \triangle diff$
29   **else** $D(t_k, t_j) \leftarrow D(t_k, t_j) + \triangle diff$

---

where $|\pi_{A_i}|$ is the number of clusters in π_{A_i}. (2) Computing distance measures for cluster pairs (instead of tuple pairs) takes $|\pi_{A_i}|^2$ for an unordered attribute A_i. In the worst case, it also takes $|\pi_{A_i}|^2$ if A_i is an ordered attribute, but in practice some comparisons across clusters can be avoided with binary search (line 13). (3) The code of $D(t_j, t_k)$ is updated for an attribute A_i iff t_j, t_k have different values in A_i. Each update requires one *addition* operation, while the *multiplication* operation is shared by all tuple pairs across the same two clusters (line 22). We experimentally find the total number of updates for an attribute is much smaller than $|r|^2$ in most cases and every update incurs a very small cost.

Dealing with large datasets. BuildDiff employs an array whose size is quadratic in $|r|$ to save intermediate results. In practice we may fail to afford the array in memory if $|r|$ is relatively large. To address the limitation, we adopt a partition technique similar in spirit to [60]. We partition r into blocks $r_1, \ldots, r_k$, and each time run BuildDiff with one block or two blocks (instead of the whole set of tuples of r); each time we deal with tuple pairs either from ${r_m}^2$ ($m \in [1, k]$) or from $r_m \times r_n$ ($m \neq n$). For the case of two blocks, BuildDiff is slightly modified to build a Pli structure on each block and process clusters from different blocks. Finally, *partial* diff-sets from different runs of BuildDiff are merged to form D_r, by removing duplicate diff-sets of tuple pair.

Parallelism. Partitioning r not only enables the support for large datasets, but also parallelism. Besides the baseline method that *serializes* computations on different blocks and block pairs, we develop a parallel version that utilizes multi-threaded parallelism, a feature readily supported by modern multi-core CPUs. The parallel version employs multiple threads to compute partial diff-sets in parallel, and uses *concurrent queues* to resolve potential read-write and write-write conflicts when merging partial results.

Generating D_r. As a complementary step, we restore every integer code in D_r to its normal form, *i.e.,* a set of differential functions, after the processing of BuildDiff. This is done by decoding the integer into ISNs (Equation 4) and finding unsatisfied differential functions according to ISNs. The total cost is linear in $|D_r|$ but irrelevant of $|r|$; it is usually trivial compared with the cost of BuildDiff.

6 DISCOVERING DDS WITH SET COVER ENUMERATION

In this section, we present a novel method to discover DDs, which combines set cover enumeration techniques with specialized minimality check operations for DDs.

Algorithm. GenDD (Algorithm 2) takes as inputs the diff-set D_r and the set Ψ of differential functions, and outputs the complete set of minimal valid DDs on r. To simplify the presentation, a LHS differential function $\phi[X]$ is considered as a subset of Ψ, so is every diff-set of tuple pair U in $D_r(\phi[A_i])$ for a RHS function $\phi[A_i]$.

Each time GenDD takes a differential function from Ψ, and finds DDs with the function as the RHS. Functions in Ψ are sorted in a *partial* order such that $\phi'[A_i]$ is before $\phi[A_i]$ if $\phi[A_i] \succ \phi'[A_i]$ (line 2). The rationale is that the minimality of a DD with $\phi'[A_i]$ on the RHS is always irrelevant of any DD with $\phi[A_i]$ on the RHS. For example, the minimality of $\phi_L[X] \rightarrow$ [Type (≤ 1)] is irrelevant of $\phi'_L[X'] \rightarrow$ [Type (≤ 2)] no matter what $\phi_L[X]$ and $\phi'_L[X']$ are. As will be seen shortly, sorting Ψ in the order helps improve the efficiency of minimality check.

With a function, say $\phi[A_i]$, GenDD finds the set Γ of LHS functions for valid DDs with $\phi[A_i]$ as the RHS, by calling Function Cover with the set of *available* functions and the set of diff-sets of tuple pair containing $\phi[A_i]$ (line 5); note all other functions on A_i cannot be used on the LHS (line 4). GenDD then performs minimality check on Γ by calling Function Minimize; Minimize also considers the set Σ of DDs that have already been discovered (line 6). Finally, newly discovered minimal valid DDs are added into Σ (lines 7-8).

Function Cover performs set cover enumeration of $D_r(\phi[A_i])$. It first generates a candidate LHS function for each element in Ψ' (line 11), and then employs every diff-set of tuple pair from $D_r(\phi[A_i])$ to refine candidates until every candidate intersects with every diff-set, *i.e.,* every candidate forms a set cover of $D_r(\phi[A_i])$. Specifically, if a candidate, say γ, does not intersect with a diff-set of tuple pair, say U, then a new candidate $\gamma \cup \{\phi'[A_j]\}$ is generated for each $\phi'[A_j] \in U \backslash \{\phi[A_i]\}$, if it is minimal in terms of set containment and γ does not already contain a function on A_j (lines 12-20). In this way, Cover enumerates *all* possible ways to refine γ *w.r.t.* U.

Algorithm 2: DD discovery based on D_r (GenDD)

Input: the diff-set D_r of r, and the set Ψ of singleton differential functions $\phi[A_i]$, $\forall A_i \in R$
Output: the set Σ of minimal and valid DDs on r

1 $\Sigma \leftarrow \emptyset$
2 sort Ψ based on a partial order, such that $\forall \phi'[A_i], \phi[A_i] \in \Psi$, $\phi'[A_i]$ is before $\phi[A_i]$ if $\phi[A_i] \succ \phi'[A_i]$
3 **foreach** $\phi[A_i] \in \Psi$ **do**
4     $\Psi' \leftarrow \{\phi'[A_j] \in \Psi \mid i \neq j\}$
5     $\Gamma \leftarrow$ Cover(Ψ', $D_r(\phi[A_i])$)
6     $\Gamma \leftarrow$ Minimize(Σ, Γ, $\phi[A_i]$)
7   **foreach** $\phi_L[X] \in \Gamma$ **do**
8         $\Sigma \leftarrow \Sigma \cup \{\phi_L[X] \rightarrow \phi[A_i]\}$
9
10 **Function** Cover(Ψ', $D_r(\phi[A_i])$)
11     $\Gamma \leftarrow \{\{\phi\} \mid \phi \in \Psi'\}$
12   **foreach** $U \in D_r(\phi[A_i])$ **do**
13         $\Gamma^- \leftarrow \{\gamma \in \Gamma \mid \gamma \cap U = \emptyset\}$ // no intersection with U
14         $\Gamma \leftarrow \Gamma \backslash \Gamma^-$
15   **foreach** $\gamma \in \Gamma^-$ **do**
16   **foreach** $\phi'[A_j] \in U \backslash \{\phi[A_i]\}$ **do**
17   **if** $\exists \phi''[A_j] \in \gamma$ **then**
18   **continue** // already a function on A_j
19   **if** $\nexists \gamma' \in \Gamma$ *such that* $\gamma' \subseteq (\gamma \cup \{\phi'[A_j]\})$ **then**
20                     $\Gamma \leftarrow \Gamma \cup \{\gamma \cup \{\phi'[A_j]\}\}$
21   **return** Γ
22
23 **Function** Minimize(Σ, Γ, $\phi[A_i]$)
24     $\Gamma_{full} \leftarrow \{\phi_L[X] \mid \phi_L[X] \rightarrow \phi'[A_i] \in \Sigma \wedge \phi[A_i] \succ \phi'[A_i]\}$
25   sort Γ based on a partial order, such that $\forall \phi_L[X], \phi'_L[X'] \in \Gamma$, $\phi'_L[X']$ is before $\phi_L[X]$ if $\phi'_L[X'] \succ \phi_L[X]$
26     $\Gamma_{new} \leftarrow \emptyset$
27   **foreach** $\phi_L[X] \in \Gamma$ **do**
28   **if** $\nexists \phi'_L[X'] \in \Gamma_{full}$ *such that* $\phi'_L[X'] \succeq \phi_L[X]$ **then**
29             $\Gamma_{full} \leftarrow \Gamma_{full} \cup \phi_L[X]$
30             $\Gamma_{new} \leftarrow \Gamma_{new} \cup \phi_L[X]$
31   **return** Γ_{new}

Every LHS function returned by Cover is a minimal set cover, but is *not* necessarily minimal in terms of subsumption of differential functions. Moreover, minimality of DDs also concerns DDs with different RHS functions. Hence, the calling of Minimize is necessary. Minimize first identifies all LHS functions that may affect the minimality of DDs with $\phi[A_i]$ on the RHS (line 24). It then sorts Γ based on a partial order (line 25), similar in spirit to the sorting in line 2. Taken together, the sort operations enable us to perform minimality check with a linear scan of the new LHS functions and existing ones (lines 27-28). A LHS function $\phi_L[X]$ passing the minimality check is employed to check the minimality of functions after it (line 29), and collected in the output (line 30).

Example 9: We illustrate the running of Function Cover in Figure 2. There are four differential functions $\{\phi_1, \phi_2, \phi_3, \phi_4\}$ and the set $D_r(\phi_1)$, as shown in the figure. Suppose Cover is called for DDs with ϕ_1 on the RHS. Initially, we have the set Γ of candidate LHS functions. The LHS of a valid DD should intersect with every element of $D_r(\phi_1)$. To achieve this goal, Cover employs diff-sets from $D_r(\phi_1)$ to refine candidates from Γ. Suppose Cover processes diff-sets in the order of $\phi_1\phi_3$, $\phi_1\phi_2\phi_3$ and $\phi_1\phi_2\phi_4$. (1) For $U = \phi_1\phi_3$, the set Γ^- contains

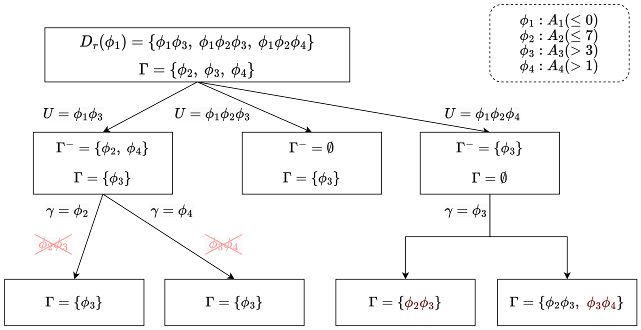


Figure 2: Example 9 for Function Cover

diff-sets that do not intersect with U. Cover removes Γ^- from Γ, and refines candidates in Γ^- by including more differential functions. The only candidate $\phi_2\phi_3$ generated from $\gamma = \phi_2$ is not minimal *w.r.t.* Γ. Similarly for $\gamma = \phi_4$. (2) Every candidate in Γ already intersects with $U = \phi_1\phi_2\phi_3$. (3) For $U = \phi_1\phi_2\phi_4$, two new candidates $\phi_2\phi_3$ and $\phi_3\phi_4$ are generated from ϕ_3, and are added into Γ since they pass the minimality check. □

Further optimizations. We present a novel structure to maintain LHS functions of discovered DDs, which helps effectively skip irrelevant ones when checking the minimality of a DD (used in line 28 of GenDD). To simplify the presentation, and without loss of generality, we illustrate our technique with an example.

Example 10: The example is shown in Figure 3. For the set Γ of LHS functions of newly discovered DDs with the same RHS, we aim to check their minimality in terms of DDs discovered before. According to the subsumption of RHS differential functions, the set Σ_{full} is identified (line 24 of GenDD). Σ_{full} is organized as a prefix tree, where the parent-child relationship is established by following the order of attributes, *i.e.,* $A_1, A_2, \ldots, A_{|R|}$. Each node denotes a combination of an attribute and one of the two operators, and every LHS function in Σ_{full} is saved in a leaf node by following the path.

LHS functions in Γ are sorted by considering their subsumption relationships (line 25), and processed in the order. For example, $\phi_2\phi_5$ must be processed before $\phi_1\phi_5$. (1) To check the minimality of ϕ_3: $[A_1\ (> 1)]$, the node labeled with "$A_1, >$" is visited. Since ϕ_3 is already saved in the node, ϕ_3 fails in the minimality check. (2) To check the minimality of ϕ_4: $[A_2\ (\leq 0)]$, the node labeled with "$A_2, \leq$" should be visited. Since this node does not exist yet, ϕ_4 passes the check. In addition, the node is inserted into the tree, for checking remaining functions in Γ. (3) To check the minimality of $\phi_2\phi_5$: $[A_1\ (\leq 1)] \wedge [A_2\ (\leq 1)]$, both the node with "$A_1, \leq$" and the node with "$A_2, \leq$" (the new node) are visited. The visit to the node with "$A_1, \leq$" terminates at its child leaf node. Since $\phi_2\phi_5$ passes the minimality check, it is inserted into the leaf node. (4) $\phi_1\phi_5$ is processed similarly as $\phi_2\phi_5$, but it is not minimal because $\phi_2\phi_5$ already exists in the tree. □

Proposition 3: Algorithm GenDD finds the complete set of minimal and valid DDs.

Proof: *Validity.* Every DD generated by GenDD has at most one differential function for each attribute (lines 4 and 17-18), complying with the definition of DDs. Function Cover is an approach to set

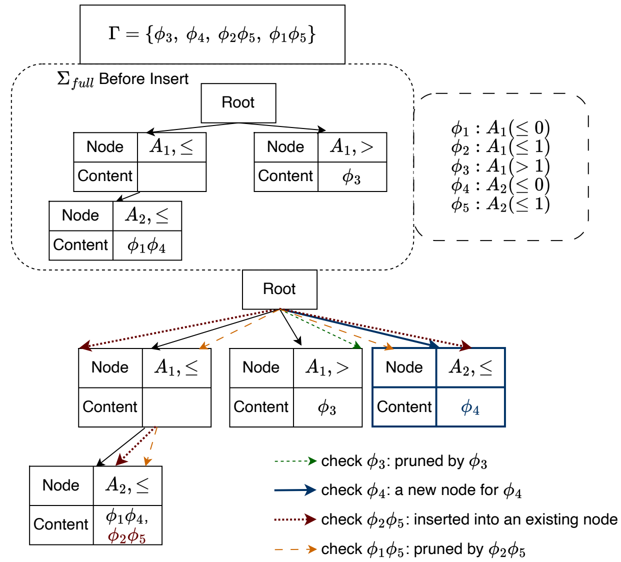


Figure 3: Example 10 for Minimality Check

cover enumeration of $D_r(\phi[A_i])$. For every $\phi_L[X]$ generated by Cover, the validity of $\phi_L[X] \rightarrow \phi[A_i]$ follows from Proposition 1.

Minimality. The function Cover only returns minimal set covers. Considering elements already in Γ suffices to determine the minimality of a new element in terms of set containment (line 19), since the size (the number of differential functions) of every element in Γ monotonically increases (line 20). Based on the output of Cover, Minimize considers the subsumption of differential functions to further eliminate non-minimal DDs. The sort operation on Ψ (line 2) and that on Γ (line 25) ensure that the minimality check can be performed in a single pass (line 28).

Completeness. The completeness is guaranteed, since (a) Cover returns all minimal set covers of $D_r(\phi[A_i])$, and (b) Minimize removes only non-minimal DDs. □

Complexity. The worst-case complexity of GenDD is exponential in the size $|\Psi|$ of Ψ; the size $|\Sigma|$ of Σ may grow exponentially with $|\Psi|$. The worst-case complexity of the minimality check is $|\Sigma|^2$ but much smaller in practice; usually only a very small proportion of Σ is visited for checking the minimality of a DD with our optimization. Note the complexity of GenDD is by nature irrelevant of $|r|$.

7 EXPERIMENTAL EVALUATIONS

In this section, we conduct an experimental evaluation to verify the effectiveness and efficiency of our DD discovery approach, and to analyze our methods and optimizations in detail.

7.1 Experimental settings

Datasets. We used a host of datasets [5, 36, 60] in our experimental evaluation. Their properties are given in Table 3, with the number $|r|$ of tuples and the number $|R|$ of attributes (textual attributes + numerical attributes). We also give the number $|\Psi|$ of differential functions considered on each dataset.

Table 3: Datasets and Execution Statistics for DD Discovery Algorithms (TL denotes more than 24 hours, and ML denotes running out of Java heap space of 100GB)

Dataset Properties				Results		Running Time (seconds)			
Dataset	$\|r\|$	$\|R\|$	$\|\Psi\|$	$\|D_r\|$	$\|\Sigma\|$	BF	TD-PO	IE-Hybrid	FastDD
Iris	150	1+4	19	443	102	0.428	0.293	0.299	**0.168**
Balance	625	1+4	10	132	6	0.184	0.188	0.184	**0.183**
Restaurant	864	5+1	26	4,473	423	13.85	4.36	3.33	**1.8**
Car	1,728	7+0	21	4,641	50	14.54	2.42	1.9	**0.594**
Cora	1,879	17+0	61	110,155	1,881,718	ML	ML	ML	**1,457**
Abalone	4,177	1+8	31	18,523	14,964	60,159	3,448	1,477	**4.7**
Pcm	9,342	10+2	42	191,931	72,252	TL	TL	TL	**109**
Tax	12k	9+6	52	2,253,295	1,295,130	TL	TL	ML	**836**
Vocab	21k	1+4	20	500	29	81.06	79.12	74.2	**27.3**
Adult	32k	9+6	43	5,528,919	1,011,677	TL	TL	TL	**1,458**
Claim	112k	8+3	43	1,063,798	119,939	TL	TL	TL	**7,278**
Atom	147k	6+7	53	42,025	5,139	ML	ML	ML	**1,248**
Flight	150k	8+5	49	85,068	25,384	TL	TL	TL	**2,932**
Struct	169k	1+5	29	1,177	162	4,750	4,711	4,361	**2,466**

Algorithms. All the algorithms are implemented in Java.

(1) Our DD discovery method FastDD is compared to existing ones [44]. We implemented three different versions presented in [44]. BF is a brute-force approach that validates all candidate DDs. TD-PO leverages subsumption orders to prune the search space in a top-down fashion, while IE-Hybrid can switch between top-down and bottom-up pruning modes. All the algorithms adopt the same settings: (a) the edit distance (resp. absolute difference) for textual (resp. numerical) attributes; and (b) the same set Ψ of differential functions. Thresholds on a dataset are derived from differences between attribute values of 200 sampled tuples (or all the tuples if $|r| < 200$), and an upper (resp. lower) bound is specified for "$\leq$" (resp. "$>$") to avoid meaningless results. On each attribute, 2 or 3 functions are used for each operator, and the support of every function is larger than a predefined minimal one.

(2) FastDD and IE-Hybrid are adapted to discover a subclass of DDs, namely RFDs. The adaptations, referred to as FastDD* and IE-Hybrid*, are compared with the state-of-the-art RFD discovery method Domino [5][1]. Domino uses the same distance measures as our method, but considers only the operator "$\leq$" and has built-in criteria to determine thresholds. FastDD* and IE-Hybrid* are modified to consider the same operator and thresholds as Domino, for the same output.

(3) We adapt FastDD* to compare with another RFD discovery method Dimϵ [7][2]. Dimϵ can find *approximate* RFDs holding on data with some exceptions, according to a predefined error rate ϵ. We set $\epsilon = 0$ in Dimϵ so as to find (exact) RFDs. Dimϵ allows only one user-defined threshold on each attribute. FastDD* is modified to use the same setting, ensuring the same output as Dimϵ.

As stated in Section 5, FastDD (FastDD*) partitions a large dataset into blocks to facilitate the diff-set construction. In our implementation, each block contains 10k tuples. Unless otherwise stated, FastDD (FastDD*) does not exploit parallelism.

Running environment. All the experiments are run on a machine with an Intel Xeon Bronze 3204 1.90G CPU (6 physical cores), 128GB of memory and CentOS Linux. The average of 3 runs is reported as the experimental results.

[1]The implementation of Domino is obtained from https://dast-unisa.github.io/Domino-SW/ (last accessed 2024/3/12).

[2]The implementation of Dimϵ is obtained from https://dastlab.github.io/dime/ (last accessed 2024/3/12).

7.2 Experimental results

Exp-1: DD discovery methods. We report the running time of all the methods in Table 3. The result is denoted by TL (resp. ML) if a method fails to terminate within 24 hours (resp. runs out of the heap space of 100 GB). We also show the size $|D_r|$ of the diff-set of r and the number $|\Sigma|$ of discovered DDs. These two factors usually have large impacts on the efficiency of DD discovery.

We see the following. (1) FastDD consistently beats all the methods from [44] on all the tested datasets, up to orders of magnitude faster. FastDD can efficiently handle datasets that vary significantly in $|r|$ and $|R|$ ($|\Psi|$), and performs well even if $|D_r|$ and $|\Sigma|$ are very large. Note $|r|$ and $|\Psi|$ affects the efficiency of the diff-set construction, $|D_r|$ and $|\Psi|$ determines the complexity of discovering DDs with diff-set, and $|\Sigma|$ is the size of the output of DD discovery.

(2) Although IE-Hybrid usually performs the best among the three methods from [44], it still fails to process some datasets within the time limit. IE-Hybrid follows the column-based strategy, which enumerates candidate DDs and prunes the search space based on DD validation results. Its efficiency is mainly controlled by the pruning power, which heavily depends on data distributions. Recall $|D_r|$ is the number of distinct diff-sets of tuple pair, and a large $|D_r|$ usually implies complex data distributions. The performance of IE-Hybrid usually degrades dramatically for a relatively large $|D_r|$. In contrast, the row-based strategy adopted by FastDD separates diff-set construction from DD discovery with diff-set, and only the complexity of the latter concerns $|D_r|$. The results show that FastDD can better deal with various data distributions.

(3) FastDD outperforms other methods in terms of memory usage. We experimentally find that FastDD suffices to deal with all tested datasets using less than 10 GB of heap space.

Exp-2: RFD discovery methods.

(1) We compare FastDD*, IE-Hybrid* and Domino in Table 4. Note the results cannot be compared to those in Table 3. FastDD* and IE-Hybrid* discover RFDs by using the same differential functions as Domino (Ψ in Table 4 differs from that in Table 3); on average 2 to 4 differential functions with the operator "$\leq$" are used on each attribute. Due to the inherent difficulties of enumeration algorithms, changes to Ψ can greatly alter the search space and discovery result, leading to a dramatic impact on efficiency. For example on Cora, $|\Sigma|$ varies dramatically from Table 3 to Table 4, so does the time.

We see the following. (a) FastDD* significantly outperforms the state-of-the-art RFD discovery method Domino. Compared to Domino, FastDD* is at least 5.4 and up to 4,969 times faster; the median is 22.1 times. IE-Hybrid* usually beats Domino on small datasets, but Domino can handle all the tested datasets. (b) FastDD* and Domino are more memory-efficient than IE-Hybrid*. They can process all tested datasets using less than 10 GB of memory.

(2) Using the same setting as Dimϵ, the comparison results of FastDD* and Dimϵ are given in Figure 4, for datasets that Dimϵ can process within time and memory limits. FastDD* is at least 3.4 and up to 2,988 times faster than Dimϵ; the median is 78 times.

Table 4: Execution Statistics for RFD Discovery Algorithms

Dataset Properties		Results		Running Time (seconds)		
Dataset	$\|\Psi\|$	$\|D_r\|$	$\|\Sigma\|$	IE-Hybrid*	Domino	FastDD*
Iris	22	1,278	24	0.311	8.4	**0.181**
Balance	10	30	21	0.192	2.4	**0.172**
Restaurant	25	1,561	43	2.2	37.1	**1.9**
Car	18	1,466	14	0.619	15.1	**0.597**
Cora	70	1,561	43	ML	18,799	**5.7**
Abalone	37	23,545	669	332	92.2	**4.1**
Pcm	49	8,787	1,630	TL	1,707	**88.9**
Tax	61	217,016	48,908	ML	765,333	**154**
Vocab	6	24	4	94.1	192	**24.7**
Adult	50	546,525	986	TL	44,093	**149**
Claim	29	26,596	123	TL	36,767	**6,759**
Atom	62	51,368	610	ML	30,551	**1,179**
Flight	61	33,465	1,216	TL	50,645	**2,796**
Struct	25	1,098	44	6,577	14,772	**2,502**

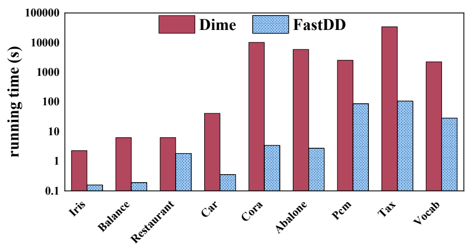
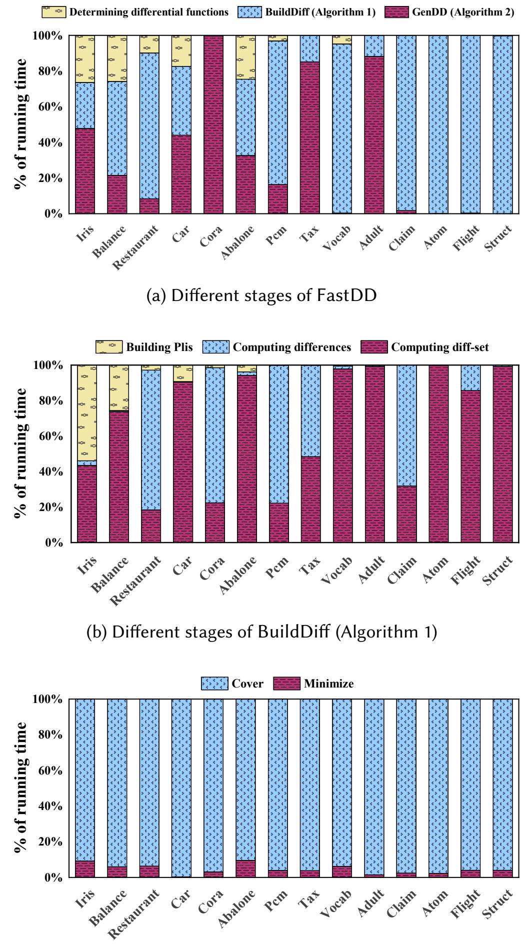


Figure 4: Comparison of Dime and FastDD*



(a) Different stages of FastDD

(b) Different stages of BuildDiff (Algorithm 1)

(c) Different stages of GenDD (Algorithm 2)

Figure 5: Time decomposition

Exp-3: Time decomposition. We study FastDD in detail by decomposing its running time, using the same setting as Exp-1.

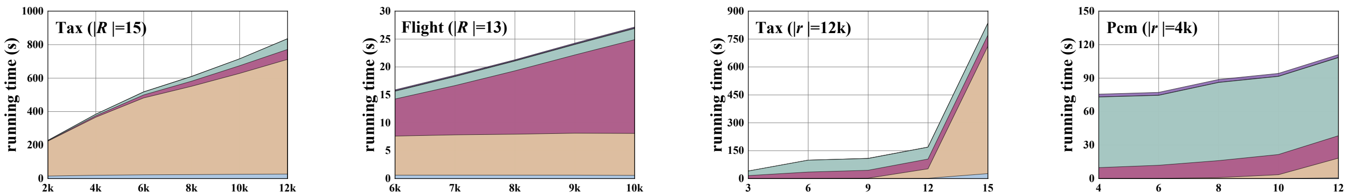
(1) We show the time of different stages of FastDD in Figure 5a, consisting of the time for (a) determining differential functions, (b) computing the diff-set with Algorithm BuildDiff (Section 5) and (c) discovering DDs with Algorithm GenDD (Section 6). The time for determining differential functions is always very short and negligible on most datasets; it is notable only when the total time is very short. This is because thresholds are determined with sampling in our implementation. BuildDiff usually takes a large proportion, and may even take almost all of the time on datasets with a large $|r|$, as expected. In contrast, GenDD governs the overall time on Cora, Tax and Adult. As shown in Table 3, a very large number of DDs are discovered on these datasets, and this inherent difficulty necessarily leads to more time for GenDD.

(2) By following the complexity analysis (Section 5), we decompose the time of BuildDiff into that for (a) building Plis, (b) computing distance measures, and (c) computing the diff-set. The results are shown in Figure 5b. Building Plis usually takes a small proportion due to its low computational complexity, while the ratio of the time for computing distance measures to the time for computing the diff-set differs considerably on datasets. The time for computing distances on an attribute mainly depends on the number of distinct values, especially long strings, since distance measures are computed for cluster pairs rather than tuple pairs in BuildDiff and it is very expensive to compute the edit distance of long strings.

(3) In Figure 5c, we decompose the time of GenDD into that for (a) function Cover and (b) function Minimize (Section 6). We find the efficiency of Cover is always the dominating factor. Our minimality check technique is verified to be very efficient even when the number of discovered DDs is huge.

Exp-4: Scalability of FastDD. We study the scalability of FastDD by varying $|r|$ or $|R|$. The results are reported in Figure 6.

(1) We first study the impact of $|r|$ with datasets Tax and Flight. FastDD scales well with $|r|$. The time increases from 229 seconds to 836 seconds as $|r|$ increases from 2k to 12k on Tax, and from 16 seconds to 28 seconds as $|r|$ increases from 6k to 10k on Flight. The effects of $|r|$ on different parts of FastDD indeed significantly differ. Specifically, (a) differential functions are determined on a random sample of r, with a time irrelevant of $|r|$. (b) The time for building Plis almost grows linearly with $|r|$. (c) The time used to compute differential functions does not depend on $|r|$, but depends on the number of distinct values. As $|r|$ increases, the time of this step increases significantly on Tax since many new values are introduced by new tuples, while it only slightly increases on Flight. Note the addition of new values can be partly seen from $|D_r|$; $|D_r|$ increases by more than 4 times on Tax. (d) The time for computing the diff-set increases, but the trend is better than the quadratic growth with $|r|$.



|r|	|Dr|	|Σ|
2k	506,594	889,649
4k	992,962	1,033,378
6k	1,393,001	1,143,123
8k	1,714,704	1,187,434
10k	1,991,463	1,236,411
12k	2,253,295	1,295,130

|r|	|Dr|	|Σ|
6k	35,217	46,926
7k	37,628	46,569
8k	39,930	45,943
9k	41,971	45,727
10k	43,907	44,726

|R|	|Dr|	|Σ|
3	62	6
6	3,815	473
9	35,135	7,278
12	493,071	133,743
15	2,253,295	1,295,130

|R|	|Dr|	|Σ|
4	340	39
6	1,118	289
8	9,004	2,870
10	54,761	11,490
12	191,931	72,252

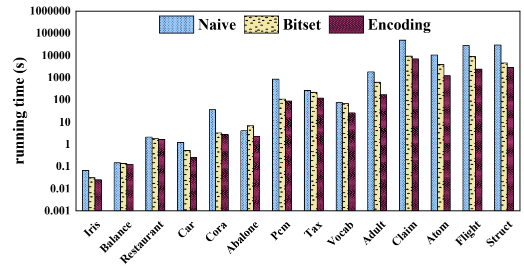
Figure 6: Scalability of FastDD **with** $|r|$ **or** $|R|$

This is because the diff-set of a tuple pair is updated for an attribute iff the two tuples have different values in the attribute, and some computations are shared by tuple pairs across the same two clusters. (e) The running time of GenDD depends on $|D_r|$ but not $|r|$. This explains why the time of GenDD dramatically increases on Tax, but only slightly increases on Flight.

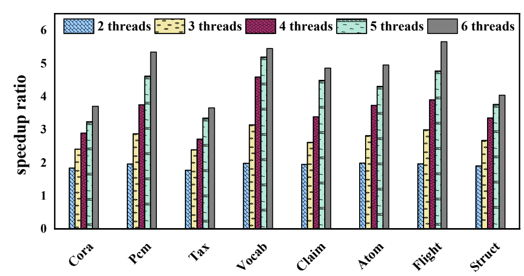
(2) We then study the impact of $|R|$ with datasets Tax and Pcm. When $|R|$ is varied, we vary $|\Psi|$ by deleting (resp. adding) functions on discarded (resp. new) attributes. We see the following. (a) Since operations for different attributes are entirely independent of each other, BuildDiff consistently takes more time as $|R|$ increases. However, the extent to which the increase in $|R|$ impacts the performance depends on whether new attributes introduce a substantial number of distinct values, particularly long string values. The increase in the time of BuildDiff is not very evident on Pcm, while the time almost triples as $|R|$ varies from 3 to 15 on Tax. Note the proportion of time used by BuildDiff decreases significantly on Tax, as the time of GenDD sharply increases. (b) $|D_r|$ always increases as $|R|$ increases, so does $|\Psi|$. Taken together, the time of GenDD is usually very sensitive to $|R|$, revealing inherent challenges associated with DD discovery. Since both $|D_r|$ and $|\Sigma|$ significantly increase on Tax and Pcm, the time of GenDD greatly goes up on the two datasets. The difference is that BuildDiff still governs the overall time on Pcm, while GenDD takes precedence on Tax as $|R|$ grows larger.

Within the two main parts of FastDD, we conclude that BuildDiff is more sensitive to $|r|$, while GenDD is more sensitive to $|R|$ and does not directly depend on $|r|$. Hence, the row-based strategy adopted by FastDD effectively separates the impact of $|r|$ from that of $|R|$, making FastDD a robust solution even for datasets that vary significantly in $|r|$, $|R|$ and underlying internal data distributions.

Exp-5: Comparison of methods to build diff-set. We compare BuildDiff against another two methods for diff-set construction. (a) Naive, which is a baseline method that compares all tuple pairs to determine the satisfaction of differential functions. (b) Bitset, which differs from BuildDiff only in its encoding scheme. Recall there are $|T_i| + 1$ intervals on attribute A_i (Section 5). Bitset uses $|T_i| + 1$ bits for each tuple pair to save the result on A_i; for a pair (t, s), all bits are initially set to be "0", and efficient bit operation is employed to set a bit to "1" if $d_{A_i}(t_{A_i}, s_{A_i})$ belongs to the corresponding interval.



(a) Comparison of different methods for diff-set construction



(b) Speed-up ratio of BuildDiff$^+$

Figure 7: Experimental results of Exp-5 and Exp-6

For $R = \{A_1, \ldots, A_{|R|}\}$, total $\sum_{i=1}^{|R|}(|T_i| + 1)$ bits are used for each tuple pair, and exactly $|R|$ bits are finally set to be "1".

The comparison results are shown in Figure 7a. BuildDiff consistently beats the other methods. Specifically, BuildDiff is on average 6.3 and up to 13 times faster than Naive; this comparison demonstrates the comprehensive strength of our solution. The advantage of BuildDiff becomes more evident on datasets with large $|r|$, as expected. The comparison of BuildDiff and Bitset in particular verifies the effectiveness of our encoding scheme. We see BuildDiff is on average 2 and up to 3.8 times faster than Bitset.

Exp-6: Speed-up ratio with parallelism. We also implement a parallel version of BuildDiff exploiting multi-threaded parallelism, called BuildDiff$^+$. The ratio of the time of BuildDiff$^+$ running with 1 thread to that of BuildDiff$^+$ with K threads is reported as the speed-up ratio of K threads. We vary the number of threads from 1 to 6,

Table 5: Ranking DDs

Dataset	Top-5 Precision	Top-10 Precision	Top-20 Precision
Abalone	0.8	0.8	0.85
Adult	1	0.8	0.8
Restaurant	0.6	0.7	0.55

which is readily supported by our machine. We test BuildDiff$^+$ on datasets with relatively large $|r|$. The results shown in Figure 7b tell us that BuildDiff$^+$ can well leverage the available threads; the speed-up ratio consistently increases as the number of threads increases. Specifically, the speed-up ratio with 2 threads is on average 1.91 and up to 1.98 on the tested datasets, and the ratio with 6 threads is on average 4.71 and up to 5.66.

Exp-7: Ranking DDs. We show ranking measures can help identify meaningful DDs from the discovery result. For DDs in the form of $\phi_L[X] \rightarrow \phi_R[A]$, we rank them first by the *support* of $\phi_L[X]$, *i.e.,* the proportion of tuple pairs satisfying $\phi_L[X]$, and then by the *succinctness* of $\phi_L[X]$, *i.e.,* the number $|X|$ of differential functions.

We perform DD discovery on Abalone, Adult and Restaurant, identify top-k DDs from the result based on the ranking, and manually label their meaningfulness. We define *precision* as the number of (labeled) meaningful DDs divided by k. The results are reported in Table 5 for k = 5, 10, 20. Relatively high precision values can be obtained, indicating the ability to efficiently identify meaningful DDs from the entire result using simple ranking methods.

DDs on Restaurant can effectively accommodate variant spellings and abbreviations used in values. $[\text{name}(\leq 0)] \wedge [\text{addr}(\leq 13)] \rightarrow [\text{phone}(\leq 8)]$ is an example of a discovered DD. Although this DD is easily understood, manually designing it is challenging, particularly in specifying the appropriate thresholds. We contend that requesting users to identify meaningful DDs from the top DDs discovered on a dataset is often more practical than asking them to provide DDs, especially when considering thresholds.

DDs discovered from Abalone are complex, involving differences in physical measurements and differences in the ages of abalones. Similarly, DDs on Adult are also intricate, explaining the reasons for different salary classes, For space limitation, we provide top-20 DDs and semantic descriptions of attributes online[3] for reference.

Exp-8: DDs for duplicate identification. Using dataset Restaurant as a testbed, we demonstrate the utility of DDs in duplicate identification [42, 44]. Due to values with variant spellings and abbreviations, different tuples in this dataset may refer to the same restaurant. The dataset is labeled, with tuples pertaining to the same restaurant sharing identical values in their "class" attribute. We perform DD discovery on Restaurant after removing this attribute.

We use DDs to classify tuples as either referring to the same restaurant or not. Tuples that satisfy all the LHS functions of a DD are considered to denote the same restaurant. The classification result is then verified based on the known labels in the "class" attribute. By utilizing DDs labeled as meaningful in the top-5 (or top-10) discovered DDs, the precision and recall of the classification task are 0.8 and 0.69 (or 0.75 and 0.85). Adding more DDs can enhance recall but may have a negative impact on precision, as expected.

[3]https://github.com/TristonK/FastDD-Exp/tree/main/Exp-7

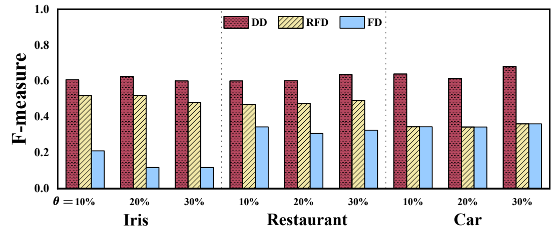


Figure 8: Comparison of DDs, RFDs and FDs

Overall, our preliminary experiment shows that DDs demonstrate good performance in identifying duplicates.

Exp-9: DDs for detecting inconsistencies. We verify the capabilities of DDs in detecting and resolving conflicts, compared with RFDs and FDs. We conduct dependency discovery on a dataset, and then introduce noise to it by randomly selecting θ% of the tuples and modifying a randomly chosen attribute for each selected tuple. We change the value of the selected attribute to a different value within the active domain. Using dependencies discovered on the original dataset, we first find all tuple pairs that violate at least one dependency, *i.e.,* performing violation detection on the dataset with added noise. Following the minimal change principle [4, 9], we then heuristically determine a minimum set V of tuples that guarantees each violating tuple pair has at least one tuple belonging to V, *i.e.,* V is a minimum cover of the hypergraph comprised of all conflicting tuple pairs. Note all data conflicts can be resolved by modifying only the tuples in V. We define *precision* p as the proportion of tuples in V that indeed contain noise, *recall* r as the proportion of all tuples containing noise that belong to V, and *f-measure* = $(2 \times p \times r)$ / $(p + r)$. The results for different settings considering DDs, RFDs and FDs are reported in Figure 8, as θ varies. We see employing DDs always leads to the best *f-measure*, mainly because DDs can better capture data conflicts compared to FDs and RFDs, resulting in significantly higher recall values, while its precision values remain stable. The results confirm the advantage of DDs compared to FDs and RFDs.

8 CONCLUSION

We have presented an efficient solution to DD discovery based on a new framework, by introducing the concept of diff-set and recasting DD discovery as set cover enumeration of the diff-set plus minimality checks. We have presented a novel scheme to encode the diff-set, and efficient methods for building the diff-set and discovering DDs from the diff-set. Our experimental evaluation has verified the efficiency and effectiveness of our approach.

Discovering DDs and using DDs in data management tasks can be integrated to form an *end-to-end* solution, in an iterative process with user interactions to further improve precision and recall. We intend to develop such systems, similar in spirit to [52, 57].

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China 62172102, 61925203 and U22B2021. We are really grateful to anonymous reviewers for their valuable comments and suggestions.

REFERENCES

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

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

[3] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. 2017. Efficient Denial Constraint Discovery with Hydra. *PVLDB* 11, 3 (2017), 311–323.

[4] Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. 2005. A Cost-Based Model and Effective Heuristic for Repairing Constraints by Value Modification. In *SIGMOD*. 143–154.

[5] 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.

[6] 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.

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

[8] Lu Chen, Yunjun Gao, Xuan Song, Zheng Li, Yifan Zhu, Xiaoye Miao, and Christian S. Jensen. 2023. Indexing Metric Spaces for Exact Similarity Search. *ACM Comput. Surv.* 55, 6 (2023), 128:1–128:39.

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

[10] Michele Dallachiesa, Amr Ebaid, Ahmed Eldawy, Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, and Nan Tang. 2013. NADEEF: a commodity data cleaning system. In *SIGMOD*. 541–552.

[11] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate Record Detection: A Survey. *IEEE Trans. Knowl. Data Eng.* 19, 1 (2007), 1–16.

[12] Wenfei Fan. 2008. Dependencies revisited for improving data quality. In *PODS*. 159–170.

[13] Wenfei Fan, Hong Gao, Xibei Jia, Jianzhong Li, and Shuai Ma. 2011. Dynamic constraints for record matching. *VLDB J.* 20, 4 (2011), 495–520.

[14] Wenfei Fan, Ziyan Han, Yaoshu Wang, and Min Xie. 2023. Discovering Top-k Rules using Subjective and Objective Criteria. *Proc. ACM Manag. Data* 1, 1 (2023), 70:1–70:29.

[15] Andrew Gainer-Dewar and Paola Vera-Licona. 2017. The Minimal Hitting Set Generation Problem: Algorithms and Computation. *SIAM J. Discret. Math.* 31, 1 (2017), 63–100.

[16] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2020. Cleaning data with Llunatic. *VLDB J.* 29, 4 (2020), 867–892.

[17] Stella Giannakopoulou, Manos Karpathiotakis, and Anastasia Ailamaki. 2020. Cleaning Denial Constraint Violations through Relaxation. In *SIGMOD*. 805–815.

[18] Seymour Ginsburg and Richard Hull. 1983. Order Dependency in the Relational Model. *Theor. Comput. Sci.* 26 (1983), 149–195.

[19] Seymour Ginsburg and Richard Hull. 1986. Sort sets in the relational model. *J. ACM* 33, 3 (1986), 465–488.

[20] Lukasz Golab, Howard J. Karloff, Flip Korn, Avishek Saha, and Divesh Srivastava. 2009. Sequential Dependencies. *PVLDB* 2, 1 (2009), 574–585.

[21] 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.

[22] 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.

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

[24] Youri Kaminsky, Eduardo H. M. Pena, and Felix Naumann. 2023. Discovering Similarity Inclusion Dependencies. *Proc. ACM Manag. Data* 1, 1 (2023), 75:1–75:24.

[25] Nick Koudas, Avishek Saha, Divesh Srivastava, and Suresh Venkatasubramanian. 2009. Metric Functional Dependencies. In *ICDE*. 1275–1278.

[26] Ioannis K. Koumarelas, Thorsten Papenbrock, and Felix Naumann. 2020. MDedup: Duplicate Detection with Matching Dependencies. *Proc. VLDB Endow.* 13, 5 (2020), 712–725.

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

[28] Selasi Kwashie, Jixue Liu, Jiuyong Li, and Feiyue Ye. 2014. Mining Differential Dependencies: A Subspace Clustering Approach. In *ADC (Lecture Notes in Computer Science)*, Vol. 8506. 50–61.

[29] Selasi Kwashie, Jixue Liu, Jiuyong Li, and Feiyue Ye. 2015. Efficient Discovery of Differential Dependencies Through Association Rules Mining. In *ADC (Lecture Notes in Computer Science)*, Vol. 9093. 3–15.

[30] Li Lin and Yunfei Jiang. 2003. The computation of hitting sets: Review and new algorithms. *Inf. Process. Lett.* 86, 4 (2003), 177–184.

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

[32] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. 2000. Efficient Discovery of Functional Dependencies and Armstrong Relations. In *EDBT*. 350–364.

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

[34] 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.

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

[36] 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.

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

[38] Nataliya Prokoshyna, Jaroslaw Szlichta, Fei Chiang, Renée J. Miller, and Divesh Srivastava. 2015. Combining Quantitative and Logical Data Cleaning. *Proc. VLDB Endow.* 9, 4 (2015), 300–311.

[39] Abdulhakim Ali Qahtan, Nan Tang, Mourad Ouzzani, Yang Cao, and Michael Stonebraker. 2020. Pattern Functional Dependencies for Data Cleaning. *Proc. VLDB Endow.* 13, 5 (2020), 684–697.

[40] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (2017), 1190–1201.

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

[42] 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.

[43] Nuhad Shaabani and Christoph Meinel. 2019. Incrementally updating unary inclusion dependencies in dynamic data. *Distributed Parallel Databases* 37, 1 (2019), 133–176.

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

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. 2012. Fundamentals of Order Dependencies. *PVLDB* 5, 11 (2012), 1220–1231.

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

[51] Zijing Tan, Ai Ran, Shuai Ma, and Sheng Qin. 2020. Fast Incremental Discovery of Pointwise Order Dependencies. *PVLDB* 13, 10 (2020), 1669–1681.

[52] Saravanan Thirumuruganathan, Laure Berti-Équille, Mourad Ouzzani, Jorge-Arnulfo Quiané-Ruiz, and Nan Tang. 2017. UGuide: User-Guided Discovery of FD-Detectable Errors. In *SIGMOD*. 1385–1397.

[53] Fabian Tschirschnitz, Thorsten Papenbrock, and Felix Naumann. 2017. Detecting Inclusion Dependencies on Very Many Tables. *ACM Trans. Database Syst.* 42, 3 (2017), 18:1–18:29.

[54] Yihan Wang, Shaoxu Song, Lei Chen, Jeffrey Xu Yu, and Hong Cheng. 2017. Discovering Conditional Matching Rules. *TKDD* 11, 4 (2017), 46:1–46:38.

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

[56] Ziheng Wei, Uwe Leck, and Sebastian Link. 2019. Discovery and Ranking of Embedded Uniqueness Constraints. *Proc. VLDB Endow.* 12, 13 (2019), 2339–2352.

[57] Ziheng Wei and Sebastian Link. 2018. DataProf: Semantic Profiling for Iterative Data Cleansing and Business Rule Acquisition. In *SIGMOD*. 1793–1796.

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

[59] 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*.

[60] Renjie Xiao, Zijing Tan, Haojin Wang, and Shuai Ma. 2022. Fast Approximate Denial Constraint Discovery. *Proc. VLDB Endow.* 16, 2 (2022), 269–281.

[61] 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.