Fast Approximate Denial Constraint Discovery

Renjie Xiao
Fudan University
20210240381@fudan.edu.cn

Zijing Tan*
Fudan University
zjtan@fudan.edu.cn

Haojin Wang
Fudan University
21210240334@m.fudan.edu.cn

Shuai Ma
Beihang University
mashuai@buaa.edu.cn

ABSTRACT

We investigate the problem of discovering approximate denial constraints (DCs), for finding DCs that hold with some exceptions to avoid overfitting real-life dirty data and facilitate data cleaning tasks. Different methods have been proposed to address the problem, by following the same framework consisting of two phases. In the first phase a structure called *evidence set* is built on the given instance, and in the second phase approximate DCs are found by leveraging the evidence set. In this paper, we present novel and more efficient techniques under the same framework. (1) We optimize the evidence set construction by first building a condensed structure called *clue set* and then transforming the clue set to the evidence set. The clue set is more memory-efficient than the evidence set and facilitates more efficient bit operations and better cache utilization, and the transformation cost is usually trivial. We further study parallel clue set construction with multiple threads. (2) Our solution to approximate DC discovery from the evidence set is a highly non-trivial extension of the *evidence inversion* method for exact DC discovery. (3) Using a host of datasets, we experimentally verify our approximate DC discovery approach is on average 8.2 and 7.5 times faster than the two state-of-the-art ones that also leverage parallelism, respectively, and our methods for the two phases are up to an order of magnitude and two orders of magnitude faster than the state-of-the-art methods, respectively.

PVLDB Reference Format:
Renjie Xiao, Zijing Tan, Haojin Wang, and Shuai Ma. Fast Approximate Denial Constraint Discovery. PVLDB, 16(2): 269-281, 2022.
doi:10.14778/3565816.3565828

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

1 INTRODUCTION

Data dependencies, *a.k.a.* integrity constraints, state relationships between attributes, and are well employed in schema design [3], data quality management [11, 18], and query optimization [6, 27, 43, 47], among others. The attribute relationship that a dependency can specify depends on the type of the dependency. Denial constraints (DCs) are proposed [8], as a generalization of many other dependencies. DCs can express the attribute relationships that these dependencies can specify, and more beyond them.

Table 1: Relational Instance r

	A	B	C	D	E	F	G
t_1	1	2	3	1	2	apple	fruit
t_2	1	2	4	2	2	apple	vegetable
t_3	7	4	5	3	6	banana	fruit
t_4	7	4	6	4	6	banana	fruit
t_5	6	5	9	6	10	cabbage	vegetable

Example 1: On the relational instance r given in Table 1, (a) a functional dependency (FD) can state that tuples with the same value in attribute A also have the same value in attribute B; (b) a unique column combination (UCC) can state that no distinct tuples can agree on their values in both attributes A and C; and (c) an order dependency (OD) [46, 48] can state that for any two tuples, the tuple with a larger value in attribute C also has a larger value in attribute D. Using the notations of DCs (the formal definition will be reviewed in Section 3), we can specify all the above-mentioned attribute relationships as follows.

(a) $\forall t, s \in r, \neg(t.A = s.A \wedge t.B \neq s.B)$.

(b) $\forall t, s \in r, \neg(t.A = s.A \wedge t.C = s.C)$.

(c) $\forall t, s \in r, \neg(t.C > s.C \wedge t.D \leq s.D)$.

Specifically, DCs support a rich set $\{<, \leq, >, \geq, =, \neq\}$ of comparison operators, and the comparison can be *across* attributes. For example, DCs can specify the following attribute relationship.

(d) $\forall t, s \in r, \neg(t.B = s.D \wedge t.E \neq s.E)$: if the value of attribute B in a tuple is the same as that of attribute D in another tuple, then the two tuples have the same value in attribute E. □

DCs in data are usually unknown. This is why discovery methods for DCs have received much attention [5, 7, 31, 34, 35], aiming at automatically finding DCs. Since DCs generalize many other dependencies, discovering DCs indeed enables us to discover all of them in one pass, instead of discovering them one by one with their specialized methods, *e.g.*, [13, 28, 33, 50, 52] for FDs, [4, 17] for UCCs, and [9, 20–22, 29, 44, 45] for ODs. Compared with other dependencies, the rich expressiveness of DCs comes with a much larger search space for DC discovery. As noted in [7], DC discovery has a search space exponential in the number of *predicates*, in contrast to most other dependencies that have a search space exponential in the number of attributes. As will be illustrated in Section 3, the number of predicates is much larger than that of attributes, since

*Zijing Tan is the corresponding author.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 2 ISSN 2150-8097.
doi:10.14778/3565816.3565828

each predicate can use one of the six comparison operators on a pair of *comparable* attributes, and two different attributes can be comparable. The huge search space makes it very difficult for DC discovery methods to scale well with real-world datasets.

Worse, it is well known real-life data often contain errors. Hence, DCs holding on data with some *exceptions* may be meaningful and valuable, while DCs holding on data may actually *overfit*.

Example 2: All the DCs given in Example 1 hold on the relational instance r. Now consider another DC: $\forall t, s \in r, \neg(t.F = s.F \wedge t.G \neq s.G)$. This DC is violated by two tuple pairs (t_1, t_2) and (t_2, t_1). Intuitively, some measures are needed to judge whether this DC should be regarded as an *approximate* DC holding on r. One of the most common error measures, namely g_1 [7, 26, 31, 35], is defined as the proportion of violating tuple pairs. The DC has the error rate of $2/20 = 0.1$ for g_1. If an error *threshold* $\epsilon = 0.1$ is used, then this DC is a valid approximate DC. This DC is also *minimal*, since the error rates of $\neg(t.F = s.F)$ and $\neg(t.G \neq s.G)$ are greater than ϵ.

Exact DC discovery methods for finding DCs holding on r will miss $\neg(t.F = s.F \wedge t.G \neq s.G)$, but discover, *e.g.*, $\neg(t.F = s.F \wedge t.G \neq s.G \wedge t.D = s.D)$ and $\neg(t.F = s.F \wedge t.G \neq s.G \wedge t.B \neq s.B)$. These DCs *overfit* the instance. $\neg(t.F = s.F \wedge t.G \neq s.G)$ can help identify violations in r; the value of attribute G in t_2 is likely to be "fruit". □

DCs that hold on data with some exceptions are *approximate* DCs. They are, in particular, well employed in data cleaning [8, 10, 14–16, 23, 38]. We study discovering approximate DCs in this paper. Approximate DC discovery subsumes exact DC discovery as a special case, and is necessarily more difficult. In contrast to exact DC discovery that concerns the satisfaction of DCs (decision problem), approximate DC discovery measures the degree of satisfaction of DCs (counting problem). All of the existing works on approximate DC discovery adopt the same two-phase framework proposed in [7]. A structure, namely *evidence set*, is built on the given instance in the first phase, and DC discovery is then conducted by leveraging the evidence set in the second phase. We adopt the same framework, but present novel and more efficient techniques for both phases.

Contributions & Organizations.

(1) We present a novel solution to the evidence set construction, by first building a condensed structure called *clue set* and then transforming the clue set to the evidence set (Section 4). Compared with the evidence set, the smaller memory footprint of the clue set enables more efficient bit operations and better cache utilization, and the transformation from the clue set to the evidence set has a cost irrelevant of the size of the instance. We also propose to partition the instance, which facilitates efficient parallel clue set construction with multiple threads.

(2) We present a novel method to discover the set of approximate DCs from the evidence set (Section 5). Our method is a highly non-trivial extension of the *evidence inversion* method [5] for exact DC discovery, and is combined with several effective pruning rules to improve the efficiency.

(3) We conduct extensive experimental evaluations to verify our approach (Section 6). The results show that our approximate DC discovery method is on average 8.2 and 7.5 times faster than the two state-of-the-art ones that also leverage parallelism, and that our new solutions to evidence set construction and approximate DC discovery from the evidence set are up to an order and two orders of magnitude faster than the state-of-the-art methods, respectively.

In addition, our solution to approximate DC discovery from the evidence set can be easily generalized to solve the problem of approximate set cover (hitting set) enumeration [31]. Set cover enumeration is a fundamental problem in computer science and widely studied in different domains [12, 30].

2 RELATED WORK

Please refer to [1, 2] for surveys of dependency discovery techniques. In this section, we investigate works close to ours.

Approximate DC discovery. To our best knowledge, approximate DC discovery methods are studied in [7, 31, 34, 35]. All these methods are based on the same framework consisting of two phases [7]. In the first phase, a structure, namely *evidence set*, is built on the given instance r, and in the second phase DC discovery on r is recast as the enumeration of *approximate* set covers of the evidence set of r. This framework is suitable for DC discovery, since it avoids DC validations that are costly for inequality comparisons [24, 25, 36, 37, 49].

Existing methods differ in their implementations. For the first phase, [7] gives the baseline method that requires to check every predicate against every tuple pair of r. This method is improved in [34], by using *bit-level* operations to avoid unnecessary tuple comparisons. [35] further uses auxiliary structures to only compute predicates satisfied by tuple pairs, based on the selectivity of predicates. Building evidence set is expensive, with a cost quadratic in $|r|$ (the number of tuples of r). Hence, parallel computation techniques are developed in [7, 35] to further improve the efficiency. The method proposed in [35] is experimentally verified to be the state-of-the-art technique for the evidence set construction.

For the second phase, a depth-first search (DFS) procedure for enumerating all approximate set covers is presented in [7], which is also used in [34, 35] later. Recently, [31] adapts the state-of-the-art hitting set enumeration algorithm MMCS [32] to enumerate all approximate hitting sets. This is possible because set cover enumeration is equivalent to hitting set enumeration [12, 30]. The overall performance of [31] is usually similar to that of [35], because evidence set construction takes most of the time of DC discovery on most datasets, and [31] uses the same method as [35] for evidence set construction in the first phase.

This work differs in the following. (1) We present a novel solution to evidence set construction, by first efficiently building the *clue set* and then transforming the clue set to the evidence set. We also develop methods to partition the data instance r such that clue set construction can be conducted with multiple threads in parallel. (2) We give a novel method for finding approximate DCs from the evidence set. It is a highly non-trivial extension of the technique of *evidence inversion* [5]. Taken together, our approximate DC discovery method significantly outperforms the ones of [31, 35] that also leverage parallelism with multiple threads.

Measure of DC violations. An approximate DC is valid on an instance r, when the measure of violations of the DC is below a given threshold. There are different measures in the literature, namely g_1, g_2 and g_3; they are originally proposed for functional dependencies [26]. Except for [31] that studies all the three measures for DCs, all the other works consider only g_1. As noted in [31], the measure

Table 2: Operator Inverse

op	$=$	$<$	$>$	$\leq$	$\geq$	$\neq$
$\overline{op}$	$\neq$	$\geq$	$\leq$	$>$	$<$	$=$

of g_2 can be too sensitive in practice, and the computation of g_3 for DC is beyond reach (NP-Hard). Hence, we focus on g_1 in this paper.

Ranking DCs. The number of discovered DCs can be large, since DCs subsume many other dependencies. Therefore, ranking functions are studied in [7, 35] to help users select *relevant* DCs. All the ranking functions can be combined with our method, along the same lines as [7, 35]. The measure of *succinctness* is used to prune DCs with (too) many predicates, but it only affects the efficiency of the second phase. All the other ranking functions are typically used in a post-processing step after all DCs are discovered, and improve the effectiveness but not the efficiency of DC discovery methods.

Exact DC discovery. A different line of work is on exact DC discovery, for finding DCs valid on data without exceptions. Exact DC discovery is a special case of approximate DC discovery. Leveraging the evidence set built in the first phase, exact DCs can be found by enumerating (exact) set covers instead of approximate ones in the second phase, as studied in [7, 34, 35]. Our approach can be modified for discovering exact DCs along the same lines.

There are no violations of exact DCs on any part of the instance. Capitalizing on this feature, a hybrid strategy is proposed in [5] that combines DC discovery on small sample with further refinement based on DC violations on the full instance. Similar strategies are also employed to discover other exact dependencies [21, 33, 41]. This strategy does not directly apply to approximate DC discovery, since an approximate DC holding on the whole instance does *not* guarantee to hold on a part of the instance.

3 PRELIMINARIES

In this section, we review basic notations of DCs (Section 3.1) and the framework of approximate DC discovery (Section 3.2).

3.1 Basic definitions

We use the common notations. R denotes a relational schema (an attribute set), r denotes a specific instance (relation) of R, t and s denote tuples from r, and $t.A$ denotes the value of attribute A in a tuple t. We assume each tuple has a distinct identifier. A rich set of comparison operators, *i.e.,* $\{<, \leq, >, \geq, =, \neq\}$, can be used in DCs. We denote by $\overline{op}$ the inverse of an operator op, shown in Table 2.

Recent studies on DC discovery [5, 31, 34, 35] focus on *variable* DCs concerning *two* tuples, formally defined as follows.

Denial constraints (DCs). DCs are defined based on *predicates*. Each predicate p is of the form $t.A_i\ op\ s.A_j$, where $t, s \in r$, $t \neq s$ (different identifiers), $A_i, A_j \in R$, and $op \in \{<, \leq, >, \geq, =, \neq\}$. The comparison is on the same attribute if $i = j$. Otherwise, it is across two attributes. We write $p_1 \sim p_2$ if $p_1.A_i = p_2.A_i$ and $p_1.A_j = p_2.A_j$, *i.e.,* p_1, p_2 are predicates concerning the same attribute pair, and $p_1 \nsim p_2$ otherwise. For $p = t.A_i\ op\ s.A_j$, $\overline{p}$ denotes $t.A_i\ \overline{op}\ s.A_j$.

A DC φ on an instance r is defined as follows: $\forall t, s \in r, \neg (p_1 \wedge \cdots \wedge p_m)$, where $p_1, \ldots, p_m$ are predicates concerning t, s. The tuple quantifiers t, s are omitted when they are clear from the context. We say φ is valid (holds) on r, iff for every tuple pair (t, s) where t, s are from r, at least one of $p_1, \ldots, p_m$ is *not* satisfied. A tuple pair (t, s) violates φ, if the pair satisfies every p_i ($i \in [1, m]$).

Along the same setting as previous works [5, 7, 31, 34, 35], in the sequel we consider DCs that do not contain predicates p_1, p_2 such that $p_1 \sim p_2$. Recall $p_1 \sim p_2$ implies that they are predicates with different operators on the same attribute pair. A DC with such predicates can always be neglected or simplified. For example, (a) a DC with both $t.A > s.A$ and $t.A < s.A$ always holds and is trivial (an inference rule from [7]). (b) A DC φ with both $t.A > s.A$ and $t.A \geq s.A$ is equivalent to the DC φ' that removes $t.A \geq s.A$ from φ.

An *approximate* DC can hold with exceptions, where the exceptions are measured by some criteria.

Error measure g_1. The measure of g_1 is originally proposed for FDs [26], and is extended to DCs [7, 31, 35]. The g_1 value of a given DC φ on an instance r, denoted by $g_1(\varphi, r)$, is defined as the ratio of the number of violating tuple pairs to the total number of tuple pairs of r. That is, $g_1(\varphi, r) = \frac{|\{(t,s) \mid (t,s) \in r^2 \wedge (t,s)\ violates\ \varphi\}|}{|r|^2 - |r|}$

Approximate DC. Given an error threshold ϵ, a DC φ is an *approximate* DC valid on r iff $g_1(\varphi, r) \leq \epsilon$. φ is minimal if there does not exist a distinct φ' such that (a) the set of predicates of φ' is a proper subset of that of φ, and (b) φ' is an approximate DC valid on r. Note the minimality of approximate DCs is defined following the *Augmentation* inference rule of [7].

Approximate DC discovery [7, 31, 34, 35]. Given a relation r of schema R and an error threshold ϵ, the aim of approximate DC discovery is to find the complete set Σ of minimal and valid approximate DCs on r *w.r.t.* ϵ.

3.2 Framework of approximate DC discovery

We review the framework that is employed by all of the existing approximate DC discovery methods [7, 31, 34, 35]. It consists of two phases. In the first phase, a structure, namely *evidence set*, is built on r. In the second phase, approximate DC discovery on r is recast as approximate minimal set cover (hitting set) enumeration of the evidence set of r. We review the definition of *predicate space* first.

Predicate space. For a given instance r, the space of approximate DCs is measured by the *predicate space* $\mathcal{P}$, *i.e.,* the set of all predicates that are allowed on r. As noted in [7, 35], a predicate is meaningful when a proper comparison operator is applied to a pair of comparable attributes. Specifically, (1) all the six operators, *i.e.,* $\{<, \leq, >, \geq, =, \neq\}$ can be used on numerical attributes, *e.g.,* age and salary, but only "=" and "≠" can be used on categorical attributes, *e.g.,* sex and phone number. (2) Two attributes are comparable if (a) they are the same, or (b) they have the same type and at least 30% of common values. Based on the rules, the predicate space $\mathcal{P}$ can be determined in the pre-processing step of DC discovery.

Example 3: In Table 3, we give part of the predicate space $\mathcal{P}$ for the instance r from Table 1. Specifically, (a) all the six operators are used on numerical attribute B; (b) only "=" and "≠" are used on categorical attributes F and G; and (c) all the six operators are used for comparisons across attributes B and D; attributes B and D share more than 30% of common values. □

Table 3: Sample of Predicate Space

p_1:$t[B] \geq s[B]$	p_2:$t[B] > s[B]$
p_3:$t[B] \leq s[B]$	p_4:$t[B] < s[B]$
p_5:$t[B] = s[B]$	p_6:$t[B] \neq s[B]$
p_7:$t[F] \neq s[F]$	p_8:$t[F] = s[F]$
p_9:$t[G] \neq s[G]$	p_{10}:$t[G] = s[G]$
p_{11}:$t[B] \neq s[D]$	p_{12}:$t[B] = s[D]$
p_{13}:$t[B] \geq s[D]$	p_{14}:$t[B] > s[D]$
p_{15}:$t[B] \leq s[D]$	p_{16}:$t[B] < s[D]$

Evidence set [7]. Given a predicate space $\mathcal{P}$, the *evidence* $evi(t,s)$ of a tuple pair (t,s) is the set of predicates from $\mathcal{P}$ satisfied by (t,s), and the *evidence set* evi_r of r is the set of evidences for all (t,s), where t,s are from r. Note different tuple pairs can produce the same evidence. To facilitate approximate DC discovery, a count $cnt(e)$ is associated with each evidence e in evi_r, denoting the number of tuple pairs (t,s) such that $evi(t,s) = e$.

Example 4: For the predicate space in Table 3, we show some evidences of tuple pairs. (a) $evi(t_1,t_2) = \{p_1,p_3,p_5,p_8,p_9,p_{12},p_{13},p_{15}\}$, $evi(t_2,t_1) = \{p_1,p_3,p_5,p_8,p_9,p_{11},p_{13},p_{14}\}$; (b) $evi(t_2,t_3) = evi(t_2,t_4) = evi(t_1,t_5) = \{p_3,p_4,p_6,p_7,p_9,p_{11},p_{15},p_{16}\}$, and hence the count of this evidence is 3 in evi_r. □

In the formal notations, evi_r is a subset family $\{e_1,\ldots,e_k\}$ defined on $\mathcal{P}$. A set cover of evi_r is a subset X of $\mathcal{P}$ such that $X \cap e_i \neq \emptyset$ for every e_i from evi_r. To make $\varphi = \neg(\overline{p_1} \wedge \cdots \wedge \overline{p_m})$ a DC valid on r (without exceptions), the set of $\{p_1,\ldots,p_m\}$ must be a *set cover* of evi_r [7]. That is, every tuple pair from r satisfies at least one p_i ($i \in [1,m]$) and hence does not satisfy at least one $\overline{p_i}$.

The case of approximate DCs is more complicated, since an approximate DC can be violated by some tuple pairs. Given an error threshold ϵ, the following result [7] states the connection between an approximate DC valid on r and evi_r.

Proposition 1: $\neg(\overline{p_1} \wedge \cdots \wedge \overline{p_m})$ is an approximate DC valid on r, iff for all $e \in evi_r$ such that e contains at least one p_i ($i \in [1,m]$),

$$\sum cnt(e) \geqslant (1-\epsilon) \times (|r|^2 - |r|).$$

Example 5: Consider the DC: $\neg(t.F = s.F \wedge t.G \neq s.G)$. The set of *inverse* predicates of this DC is $\{p_7, p_{10}\}$. Among all the evidences (not shown), only $\{p_1,p_3,p_5,p_8,p_9,p_{12},p_{13},p_{15}\}$ and $\{p_1, p_3,p_5,p_8,p_9, p_{11},p_{13},p_{14}\}$, *i.e.,* $evi(t_1,t_2)$ and $evi(t_2,t_1)$, do not intersect with this set, and they both have the count of 1. Hence, the accumulated count of all the evidences that intersect with $\{p_7, p_{10}\}$ = 20 - 2 = 18. If the error threshold ϵ = 0.1, then we know $\neg(\overline{p_7} \wedge \overline{p_{10}})$, *i.e.,* $\neg(t.F = s.F \wedge t.G \neq s.G)$, is a valid approximate DC.

In contrast, $\neg(t.F = s.F \wedge t.G \neq s.G \wedge t.D = s.D)$ is a valid DC without exceptions; the set of inverse predicates of it intersects with every evidence in evi_r. So is $\neg(t.F = s.F \wedge t.G \neq s.G \wedge t.B \neq s.B)$. □

Approximate DC discovery with approximate set cover enumeration. Proposition 1 states that the set of inverse predicates of a valid approximate DC, *i.e.,* $\{p_1,\ldots,p_m\}$, is an *approximate* set cover; it intersects with "enough" (not necessarily all) evidences according to the threshold ϵ and the counts associated with evidences. A valid approximate DC is further minimal, if no proper subset of $\{p_1,\ldots,p_m\}$ is an approximate set cover. The methods for approximate set cover (hitting set) enumeration of the evidence set are studied in [7, 31].

4 BUILD EVIDENCE SET

In this section, we present a novel solution to evidence set construction. We first review a data structure for DC discovery (Section 4.1). We then propose a condensed structure called *clue set*, and develop techniques for clue set construction and transforming the clue set to the evidence set (Section 4.2). We finally study parallel clue set construction with multiple threads (Section 4.3).

4.1 Auxiliary structure

Position list index. Position list index (Pli) used for DCs [5, 35] is an extended version of the original one for FD discovery methods [28, 33] and also shares some common features as the *sorted partition* used in order dependency discovery approaches [21, 29].

Each Pli is built for an attribute from R. We denote the Pli on attribute A by π^A, and the collection of Plis for attributes of R by Π. π^A is a set (or list) of *clusters*. Each cluster is a pair $\langle k,l\rangle$; the key k is a value in attribute A and the value l is the set of all tuple *identifiers* having the same value k in attribute A. For a numerical attribute A, clusters in π^A are further sorted by the key k in descending order.

We define the following operations. Given a parameter k, getEQ() returns the cluster whose key is equal to k, or *null* if no such cluster exists. The operation getLTs() is only defined on numerical Plis. Given a numerical parameter k, getLTs() returns the list of clusters whose keys are less than k, or an empty list if no such cluster exists.

Example 6: For the instance r in Table 1, π^A is a list of clusters. $\pi^A = [\langle 7,\{t_3,t_4\}\rangle, \langle 6,\{t_5\}\rangle, \langle 1,\{t_1,t_2\}\rangle]$. π^A.getEQ(6) = $\langle 6,\{t_5\}\rangle$. π^A.getLTs(6) = $[\langle 1,\{t_1,t_2\}\rangle]$. □

4.2 From clue set to evidence set

As demonstrated in previous works [7, 31, 35], building the evidence set is costly for large datasets. We present a structure, referred to as *clue set*. The computation of the clue set is much faster than that of the evidence set, and the time of transforming the clue set to the evidence set is usually trivial. Taken together, these yield a much more efficient solution to the evidence set construction.

An *evidence* $evi(s,t)$ is the set of predicates satisfied by a tuple pair (s,t), and is implemented with the structure of *bitset* in previous works [5, 35]. Specifically, (a) for a categorical attribute pair, 2 bits are used to encode the 2 predicates with "=" and "≠" respectively; and (b) for a numerical attribute pair, 6 bits are used to encode the 6 predicates (each with an operator from $\{<,\leq,>,\geq,=,\neq\}$). However, we find the evidence is *not* the most efficient way to encode the relationship of s and t, concerning comparable attribute pairs.

Clue and clue set. We present a new structure, referred to as *clue*. The clue $clue(s,t)$ of the pair (s,t) is more condensed than the evidence $evi(s,t)$, and there is a one-to-one relationship between the clue and the evidence. The clue directly encodes the relationship of s and t *w.r.t.* comparable attribute pairs rather than predicates.

A clue is also implemented with bitset, but is more efficient in memory. More specifically, (a) for a categorical attribute pair A,B (or A,A), only 1 bit is used in $clue(s,t)$: "0" if $s.A \neq t.B$ (or $s.A \neq t.A$), or "1" if $s.A = t.B$ (or $s.A = t.A$); and (b) for a numerical attribute pair A,B (or A,A), only 2 bits are used in $clue(s,t)$: "00" if $s.A < t.B$ (or $s.A < t.A$), "01" if $s.A = t.B$ (or $s.A = t.A$), or "10" if $s.A > t.B$ (or $s.A > t.A$). A clue saves 50% to 66.7% memory usage than an evidence.

For all tuple pairs from r^2, we collect their clues to build the clue set $clue_r$ of r. In the same way as building the evidence set, we associate each clue cl in $clue_r$ with a count $cnt(cl)$ denoting the number of tuple pairs that have the clue.

We then present an algorithm to efficiently compute the clue set for a given instance r. It adapts the idea of *presumption and correction* for evidence set construction [35] to clue set construction. Briefly, it first assigns a *default* clue cl_0 to all tuple pairs, and then corrects the clues (with bit operations) for tuple pairs whose clues are different from the default one. The key is that all such tuple pairs can be efficiently identified by leveraging the Plis of r. It is hence much more efficient than the baseline that compares all tuple pairs. The choice of cl_0 also affects the efficiency; a small number of bits are required to be corrected if cl_0 has many same bits as the actual clues. Assuming that most tuples have different values in the attributes [35], we set all bits in cl_0 as "0", *i.e.,* for a tuple pair (s, t) and an attribute pair (A, B), we assume $s.A \neq t.B$ for a categorical attribute pair, and $s.A < t.B$ for a numerical pair.

Algorithm. Algorithm BuildClue (Algorithm 1) builds $clue_r$. It first initializes the clues of all tuple pairs with the default clue cl_0 (line 1), and then corrects clues in bits by considering comparable attribute pairs (lines 2-21). As an example, tuples t, s in the same cluster of π^A have the same value in A, which differs from the presumption of cl_0. Procedure Correct is called to correct (the bits in) the clue of (t, s) if t and s are in the same cluster (lines 2-4). As another example, suppose A, B form a pair of comparable attributes. The clue of (t, s) differs from cl_0 in the bits concerning the pair A, B, if t belongs to the cluster c_1 in π^A, and s belongs to the cluster c_3 in π^B such that the key of c_3 is no larger than the key of c_1. Procedure Correct is called to correct $clue(t, s)$ accordingly (lines 15-21). The $clue_r$ is finally obtained by computing the accumulated count of each distinct clue (line 22).

Example 7: Table 3 concerns 4 comparable attribute pairs: (B, B), (F, F), (G, G) and (B, D). Suppose bits in clues are in the same order as the above attribute pairs. At first $clue(t_1, t_2)$ is "000000", and becomes "011001" after the corrections with BuildClue. □

Remarks. (1) The worst-case complexity of BuildClue is quadratic in $|r|$, but in practice a large proportion of tuple pairs can be skipped in the computations. (2) At most one bit is modified for each attribute pair when correcting a clue. Specifically, a bit of "0" can only be modified to "1" for a categorical attribute pair, and "00" can only be modified to either "01" or "10" for a numerical pair. In contrast, each correction for an evidence [35] may need to modify several bits for predicates on the same attribute pair. Reducing the cost of each correction is crucial to the efficiency, since the number of required corrections can be large. (3) Note each clue may be corrected several times in BuildClue, in a different bit each time. The small memory footprint of the clue set is likely to favor cache hit ratio. That is, a clue is still kept in the cache when the clue is corrected again in different bits. This advantage will be further enhanced with the *sharding* technique that will be proposed in Section 4.3.

From clue set to evidence set. Generating evidences from clues concerns simple bit encoding transformations. The cost of the transformation from the clue set to the evidence set is linear in $|clue_r|$, but is irrelevant of $|r|$ since many tuple pairs can produce the same clue.

Algorithm 1: Build Clue Set (BuildClue)

Input: the Plis Π of r
Output: the clue set $clue_r$ of r

```
1  Clues ← an array of |r| × (|r| − 1) clues that are all equal to cl_0
2  foreach categorical predicate p : t.A = s.A do
3      foreach cluster c ∈ π^A do
4          Correct(p, c, c, Clues)
5  foreach categorical predicate p : t.A = s.B do
6      foreach cluster c_1 ∈ π^A do
7          cluster c_2 = π^B.getEQ(c_1.k)
8          if c_2 ≠ null then Correct(p, c_1, c_2, Clues)
9  foreach numerical predicate p_eq : t.A = s.A do
10     p_gt ← predicate t.A > s.A
11     foreach cluster c_1 ∈ π^A do
12         Correct(p_eq, c_1, c_1, Clues)
13         foreach cluster c_2 ∈ {clusters behind c_1 in π^A} do
14             Correct(p_gt, c_1, c_2, Clues)          // c_1.k > c_2.k
15 foreach numerical predicate p_eq : t.A = s.B do
16     p_gt ← predicate t.A > s.B
17     foreach cluster c_1 ∈ π^A do
18         cluster c_2 = π^B.getEQ(c_1.k)
19         if c_2 ≠ null then Correct(p_eq, c_1, c_2, Clues)
20         foreach cluster c_3 ∈ π^B.getLTs(c_1.k) do
21             Correct(p_gt, c_1, c_3, Clues)          // c_1.k > c_3.k
22 accumulate clues in Clues to get the clue set clue_r
23
24 Procedure Correct(p, c_1, c_2, Clues)
25     foreach tuple t ∈ c_1.l do
26         foreach tuple s ∈ c_2.l do
27             if t ≠ s then correct clue(t, s) in Clues with p
```

This property is desirable, since $|clue_r| \ll |r|^2$, as experimentally verified on a host of datasets (Section 6). We find the transformation time is trivial compared with the time of clue set construction.

4.3 Parallel computation with multiple threads

The space complexity of BuildClue is $|r|^2$, but the space required by $clue_r$ is actually much smaller since only distinct clues are stored. This observation motivates us to compute $clue_r$ *part by part* instead of all at once; the final $clue_r$ can be obtained by merging the partial clue sets one by one. Better, the strategy enables parallel computation of $clue_r$ to further improve the efficiency.

Parallel computations are used to speed up evidence set construction. Specifically, [7] adopts the setting of multiple independent machines, while [35] exploits multithreaded parallelism. In this paper we adopt the same setting as [35]. Dependency discovery methods with multiple threads are shared-memory parallel algorithms [19, 53], and lightweight alternatives to distributed discovery techniques [39, 40, 42]. The results can be easily reproduced since multiple threads are readily supported by modern multi-core CPUs.

[35] adopts the scheme that partitions the set of *tuple pair identifiers* into chunks, and multiple threads are employed to build partial evidence sets of different chunks in parallel. In this paper, we adopt a simple yet effective strategy that partitions *tuple identifiers* into *instance shards* and builds partial clue sets for tuples (a) from the same shard, or (b) from two distinct shards, with multiple threads

Algorithm 2: Build Partial Clue Set (BuildPartialClue)

Input: two different PliShards Π_i of r_i and Π_j of r_j
Output: the partial clue set $clue_{ij}$

1 $Clues \leftarrow$ an array of $|r_i| \times |r_j|$ clues that are all equal to cl_0
2 **foreach** *categorical predicate* $p : t.A = s.A$ **do**
3 **foreach** *cluster* $c_1 \in \pi_i^A$ **do**
4 cluster $c_2 = \pi_j^A$.getEQ($c_1.k$)
5 **if** $c_2 \neq null$ **then** Correct($p, c_1, c_2, Clues$)
6 **foreach** *categorical predicate* $p : t.A = s.B$ **do**
7 **foreach** *cluster* $c_1 \in \pi_i^A$ **do**
8 cluster $c_2 = \pi_j^B$.getEQ($c_1.k$)
9 **if** $c_2 \neq null$ **then** Correct($p, c_1, c_2, Clues$)
10 **foreach** *numerical predicate* $p_{eq} : t.A = s.A$ **do**
11 $p_{gt} \leftarrow$ predicate $t.A > s.A$
12 **foreach** *cluster* $c_1 \in \pi_i^A$ **do**
13 cluster $c_2 = \pi_j^A$.getEQ($c_1.k$)
14 **if** $c_2 \neq null$ **then** Correct($p_{eq}, c_1, c_2, Clues$)
15 **foreach** *cluster* $c_3 \in \pi_j^A$.getLTs($c_1.k$) **do**
16 Correct($p_{gt}, c_1, c_3, Clues$) // $c_1.k > c_3.k$
17 **foreach** *numerical predicate* $p_{eq} : t.A = s.B$ **do**
18 $p_{gt} \leftarrow$ predicate $t.A > s.B$
19 **foreach** *cluster* $c_1 \in \pi_i^A$ **do**
20 cluster $c_2 = \pi_j^B$.getEQ($c_1.k$)
21 **if** $c_2 \neq null$ **then** Correct($p_{eq}, c_1, c_2, Clues$)
22 **foreach** *cluster* $c_3 \in \pi_j^B$.getLTs($c_1.k$) **do**
23 Correct($p_{gt}, c_1, c_3, Clues$) // $c_1.k > c_3.k$
24 accumulate clues in $Clues$ to get the partial clue set $clue_{ij}$

in parallel. Our strategy guarantees each tuple pair is assigned to one and only one thread where the clue of the pair is computed.

Position list index sharding. We implement Pli sharding by partitioning r into several *instance shards* $\{r_0, \ldots, r_k, \ldots\}$ and build PliShards $\{\Pi_0, \ldots, \Pi_k, \ldots\}$ on each of them. Specifically, with a shard size ω, shard r_k contains tuples whose tuple identifiers are in $[k \cdot \omega, (k+1) \cdot \omega)$, and PliShard Π_k are Plis built on tuples in shard r_k. We denote by π_k^A the Pli on attribute A in PliShard Π_k. The accumulated time complexity and space complexity of Pli sharding are almost the same as the whole Plis of r. The operations getEQ() and getLTs() are naturally extended to PliShards.

Build partial clue set. We divide the construction of clue set $clue_r$, to enable parallel computation. Each time we choose instance shards r_i and r_j, and build the partial clue set $clue_{ij}$ leveraging PliShards Π_i and Π_j, *i.e.,* $clue_{ij} = \{clue(t,s) | t \in r_i, s \in r_j\}$. There are two cases for the combination of shards. (1) $i = j$: shards r_i and r_j refer to the same shard of the original instance r. We build $clue_{ij}$ with BuildClue that takes Π_i as the input. (2) $i \neq j$: r_i and r_j are different shards. We present Algorithm BuildPartialClue for this case.

Algorithm. Algorithm BuildPartialClue (Algorithm 2) builds the partial clue set $clue_{ij}$, where $i \neq j$. It is based on BuildClue but differs from it in the following aspects. (1) The size of array $Clues$ is $|r_i| \times |r_j|$ (line 1), since tuples in shard r_i always have different identifiers from those in r_j when $i \neq j$. (2) The two tuples of every tuple pair are always from different shards, and hence visits to different shards are required. For example, consider the predicate $t.A = s.A$. A tuple from a cluster of π_i^A is no longer treated against tuples from the same cluster, but against those from a cluster of π_j^A. This requires an extra calling of getEQ() (line 4). Similarly for the cases of lines 13 and 15, which may also concern getLTs().

Based on the partition strategy and BuildPartialClue (BuildClue), we build the clue set with multiple threads in parallel.

Algorithm 3: Build Merge Clue Set (BuildMergeClue)

Input: Array T of pairs of instance shards
Output: the clue set $clue_r$

1 $n \leftarrow |T|$
2 **if** $n = 0$ **then return** an empty clue set
3 **if** $n = 1$ **then** // suppose the only pair $T[0] = \langle r_i, r_j \rangle$
4 **if** $i = j$ **then**
5 **return** BuildClue (Π_i)
6 **else**
7 **return** BuildPartialClue (Π_i, Π_j)
8 $L \leftarrow$ BuildMergeClue($T[0, ..., n/2)$)
9 $R \leftarrow$ BuildMergeClue($T[n/2, ..., n)$)
10 **return** Merge(L, R)

Algorithm. Algorithm BuildMergeClue (Algorithm 3) constructs the whole clue set $clue_r$, by building and merging partial clue sets in parallel. It follows the parallel computation model of class CountedCompleter from the standard Java library[1], which is well suited to the recursive decomposition and divide-and-conquer strategies of our approach. BuildMergeClue takes as input an array T of pairs of instance shards; each element in T is a pair $\langle i, j \rangle$. It is initially called with all pairs of instance shards. If there is only one element in T, then BuildClue (resp. BuildPartialClue) is called if $i = j$ (resp. $i \neq j$) in lines 3-7. Otherwise, BuildMergeClue recursively calls itself with each half of T (lines 8-9). Function Merge is called to merge the two partial clue sets (line 10). Internally, an available thread from the thread pool is used for each call of BuildClue and BuildPartialClue, and a binary-tree structure is built to maintain the termination of threads.

Note BuildClue and BuildPartialClue only read Plis. The computation of a clue is within a single thread, which favors *data locality.*

5 DISCOVER APPROXIMATE DCS BASED ON EVIDENCE SET

In this section, we present a novel method to discover approximate DCs by leveraging the evidence set. It is based on the technique of *evidence inversion* [5] proposed for exact DC discovery.

Before reviewing the method of evidence inversion, we give some notations first. For a DC $\varphi = \neg(p_1 \wedge \cdots \wedge p_m)$ and an evidence e, we write $\varphi \subseteq e$ if the set of predicates of φ, *i.e.,* $\{p_1, \ldots, p_m\}$ is a subset of e. Recall φ is violated by the tuple pairs that produce e, if $\varphi \subseteq e$. We say φ *covers* (resolves) e if $\varphi \not\subseteq e$, *i.e.,* φ contains at least one predicate not in e and is not violated by the tuple pairs that produce e. For two DCs φ and φ', we write $\varphi \subseteq \varphi'$ if the set of predicates of φ is a subset of that of φ'; φ' is *not* minimal if $\varphi \neq \varphi'$ and φ is valid. We denote by $\varphi \cup \{p\}$ the DC that is obtained by adding a new predicate p to φ. To simplify the notation, we use p_1p_2 as a shorthand for $\{p_1, p_2\}$ when it is clear from the context.

[1] docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountedCompleter.html (last accessed 2022/10/11).

Evidence inversion. Algorithm *evidence inversion* [5] discovers the set Σ of exact DCs, with the evidence set evi_r. Specifically, (1) it initially puts $|\mathcal{P}|$ candidate DCs into Σ, where each DC uses a distinct predicate from $\mathcal{P}$. (2) For each evidence e from evi_r, it checks all candidate DCs in Σ against e. For every DC $\varphi \in \Sigma$ such that $\varphi \subseteq e$, it removes φ from Σ, and then generates candidate DCs based on φ to cover e. To do so, for each predicate p not in e, it forms a new candidate $\varphi' = \varphi \cup \{p\}$ and adds φ' into Σ if there does not exist a DC $\varphi'' \in \Sigma$ such that $\varphi'' \subseteq \varphi'$, *i.e.,* φ' is minimal in terms of Σ. (3) It terminates after all evidences from evi_r are processed.

Example 8: With a given predicate space $\mathcal{P} = \{p_1, p_2, p_3\}$ and $evi_r = \{p_1p_2, p_2p_3, p_1p_3\}$, we illustrate the running of evidence inversion. Note the counts of evidences are irrelevant to exact DC discovery. Initially, $\Sigma = \{\neg(p_1), \neg(p_2), \neg(p_3)\}$; each DC has one predicate from $\mathcal{P}$. Evidences from evi_r are processed one by one. (1) All DCs in Σ are first checked against p_1p_2. We see $p_1 \subseteq p_1p_2$; the tuple pairs that produce p_1p_2 violate $\neg(p_1)$. The DC $\neg(p_1)$ is replaced by $\neg(p_1 \wedge p_3)$, by adding p_3, *i.e.,* a predicate from $\mathcal{P}$ and not in p_1p_2, to $\neg(p_1)$. However, this DC is *not* minimal and is not put into Σ, because $\neg(p_3)$ is in Σ. Similarly for the case of $\neg(p_2)$. $\neg(p_3)$ remains intact since it has the predicate p_3 not in p_1p_2. (2) We have $\Sigma = \{\neg(p_3)\}$ after (1). After checking $\neg(p_3)$ against p_2p_3, $\neg(p_3)$ is replaced by $\neg(p_3 \wedge p_1)$ in Σ. (3) After checking $\neg(p_3 \wedge p_1)$ against p_1p_3, we finally have $\Sigma = \{\neg(p_3 \wedge p_1 \wedge p_2)\}$. □

Intuitively, the evidence inversion enumerates all evidences, and resolves all DC violations *w.r.t.* each evidence by refining the corresponding candidate DCs with a predicate not in the evidence. It is highly non-trivial to adapt the evidence inversion for finding approximate DCs. When generating candidate approximate DCs, it is unnecessary to resolve all the DC violations *w.r.t.* every evidence e. Instead, the count $cnt(e)$ of e should be considered for generating or pruning candidates. This yields a much more complicated strategy to generate or prune candidate approximate DCs, compared with that to generate or prune candidate exact DCs.

Algorithm. AEI (Algorithm 4) discovers the set Σ of minimal valid approximate DCs, with the evidence set evi_r, predicate space $\mathcal{P}$ and error threshold ϵ. It initializes some parameters (lines 1-4), and calls Procedure Inverse (line 5). We illustrate the parameters of Inverse in detail (line 8). Each time Inverse handles one (the i-th) evidence from evi_r, in response to Ψ. Each candidate DC ψ forms a pair with a set *cand* of candidate predicates that can be used to refine ψ later; all such pairs are collected in Ψ. $\mathcal{P}_{add}$ is a set of candidate predicates, which enables efficient pruning of candidate DCs in Inverse. The parameter N is the total *count* of evidences required to be covered. All discovered approximate DCs are collected in Σ.

If $N \leq 0$, then Inverse collects all candidate DCs in Σ if they are minimal, and returns (lines 9-11). Inverse also returns, if (a) all evidences are processed, or (b) no candidate DCs exist, or (c) no candidate predicates exist (line 12). It then processes the i-th evidence e from evi_r. All candidate DCs from Ψ are divided into two parts, according to whether they cover e. The DCs that do not cover e are put into Ψ^-, while the others remain in Ψ (lines 14-15). This is because Inverse can choose not to *cover* the evidence e (lines 16-25), or to cover it (lines 26-37). In the branch where e is *not* covered, only predicates from e can be used afterwards. Hence, the set $\mathcal{P}_{add}$ of candidate predicates is adjusted accordingly (line 16). Function CanCover is called to check whether this branch can lead to valid DCs (line 17). Specifically, CanCover computes the accumulated count of the remaining evidences that can be covered by the remaining predicates in $\mathcal{P}_{add}$, *i.e.,* the upper bound of the accumulated count. The branch fails and is pruned if the value cannot reach N. Otherwise, DCs from Ψ^- are further processed; it suffices to consider Ψ^- in the branch where e is *not* covered. Some

Algorithm 4: Approximate Evidence Inversion (AEI)

Input: evidence set evi_r, predicate space $\mathcal{P}$, and error threshold ϵ
Output: the set Σ of minimal and valid approximate DCs

1 sort evidences e in evi_r by $cnt(e)$, in descending order
2 $\Sigma \leftarrow \emptyset$
3 $\psi \leftarrow$ an empty DC (with no predicates), $cand \leftarrow \mathcal{P}$
4 $N \leftarrow (|r|^2 - |r|) \times (1 - \epsilon)$
5 Inverse$(1, \{\langle \psi, cand \rangle\}, \mathcal{P}, N, \Sigma)$
6 **return** Σ
7
8 **Procedure** Inverse$(i, \Psi, \mathcal{P}_{add}, N, \Sigma)$
9 **if** $N \leq 0$ **then**
10 $\Sigma \leftarrow \Sigma \cup \{\psi | \langle \psi, cand \rangle \in \Psi, \nexists \phi \in \Sigma$ such that $\phi \subseteq \psi\}$
11 **return**
12 **if** $i > |evi_r| \vee \Psi = \emptyset \vee \mathcal{P}_{add} = \emptyset$ **then return**
13 $e \leftarrow evi_r[i]$
14 $\Psi^- \leftarrow \{\langle \psi, cand \rangle \in \Psi | \psi \subseteq e\}$
15 $\Psi \leftarrow \Psi \setminus \Psi^-$
 /* not to cover the evidence e */
16 $\mathcal{P}_{add} \leftarrow \mathcal{P}_{add} \cap e$
17 **if** CanCover$(i + 1, \mathcal{P}_{add}, N)$ **then**
18 **foreach** $\langle \psi, cand \rangle \in \Psi^-$ **do**
19 $cand \leftarrow cand \cap e$
20 **if** $cand = \emptyset$ **then** // no more predicates to use
21 $\Psi^- \leftarrow \Psi^- \setminus \{\langle \psi, cand \rangle\}$
22 **if** $\nexists \phi \in \Sigma$: $\phi \subseteq \psi \wedge$ CanCover$(i + 1, \psi, N)$ **then**
23 $\Sigma \leftarrow \Sigma \cup \{\psi\}$
24 Inverse$(i + 1, \Psi^-, \mathcal{P}_{add}, N, \Sigma)$
25 recover the changes done in lines 16, 19 and 21
 /* to cover the evidence e */
26 **foreach** $\langle \psi, cand \rangle \in \Psi^-$ **do**
27 $addable \leftarrow cand \cap (\mathcal{P} \setminus e)$
28 **foreach** $p \in addable$ **do**
29 $\psi' \leftarrow \psi \cup \{p\}$, $cand' \leftarrow \{p' \in cand | p' \nsim p\}$
30 **if** $cand' \neq \emptyset$ **then**
31 **if** $\nexists \langle \phi, cand \rangle \in \Psi$ *such that* $\phi \subseteq \psi'$ **then**
32 $\Psi \leftarrow \Psi \cup \{\langle \psi', cand' \rangle\}$
33 **else if** $\nexists \phi \in \Sigma : \phi \subseteq \psi' \wedge$ CanCover(i, ψ', N) **then**
34 $\Sigma \leftarrow \Sigma \cup \{\psi'\}$
35 $N \leftarrow N - cnt(e)$
36 Inverse$(i + 1, \Psi, \mathcal{P}_{add}, N, \Sigma)$
37 recover the changes done in lines 32 and 35
38
39 **Function** CanCover(l, ϕ, N)
40 $maxCover \leftarrow 0$
41 **for** $i \leftarrow l$ **to** $|evi_r|$ **do**
42 **if** $\phi \not\subseteq evi_r[i]$ **then**
43 $maxCover \leftarrow maxCover + cnt(evi_r[i])$
44 **if** $maxCover \geq N$ **then return** *True* **else return** *False*

DCs may have an empty set of candidate predicates after the set is adjusted (lines 19-20). Such DCs cannot be further refined (line 21), but they can possibly already be valid approximate DCs. Function CanCover is called for such DCs, and they are put into Σ if they are valid and minimal (lines 22-23). To do so, CanCover computes the accumulated count of the remaining evidences that can be covered by each such DC. At the end of this branch, Inverse is recursively called with the next evidence, adjusted Ψ^- and $\mathcal{P}_{add}$ (line 24).

In the branch where e is covered, the set *addable* of predicates that can be used to cover e is collected for each ψ from Ψ^- (line 27). Using a predicate from *addable* each time, a new candidate DC ψ' is formed, and the set *cand*$'$ is obtained by removing the *related* predicates from *cand* (line 29). Recall $p \sim p'$ if they are predicates with different operators on the same attribute pair and it is safe to neglect DCs with both p, p' in DC discovery (Section 3.1). The candidate DCs are put into Ψ if they are minimal in terms of Ψ and there are still candidate predicates for them (lines 30-32). The DCs with no more candidate predicates are put into Σ if they are already minimal valid approximate DCs (lines 33-34), and are discarded otherwise. After generating all candidate DCs that cover e, N is reduced by $cnt(e)$ and Inverse is recursively called (lines 35-36).

Example 9: In Figure 1, $\mathcal{P} = \{p_1, p_2, p_3\}$ and $evi_r = \{p_1p_2(2), p_2p_3(2), p_1p_3(1)\}$ with the count of each evidence in the bracket. The number of tuple pairs is 2+2+1 = 5, ϵ = 0.4, and hence N = 5×(1-0.4) = 3.

We show the running of Procedure Inverse with the parameters, in a tree structure. One predicate from evi_r is processed at each layer, and two branches are generated from each intermediate node, to cover or not to cover the next evidence. A leaf node is reached when all evidences are processed or all the descendants are pruned.

Consider the node at layer 2 where evidence p_1p_2 is not covered. Only predicates p_1 and p_2 can be used afterwards, to refine the empty DC (with no predicates). (a) On its left branch where p_2p_3 is also not covered, only the predicate p_2 can be used afterwards. The calling of Function CanCover terminates the branch, since the requirement $N = 3$ cannot be met with the only remaining evidence p_1p_3. (b) On its right branch where p_2p_3 is covered, the predicate p_1 is used to generate a new DC $\neg(p_1)$ to replace the empty DC, and only the predicate p_2 can be used afterwards. If the evidence p_1p_3 is covered at the next layer, then a minimal and valid approximate DC $\neg(p_1 \wedge p_2)$ is obtained. It covers evidences p_2p_3 and p_1p_3.

Consider the right-most branch where both p_1p_2 and p_2p_3 are covered. A minimal and valid approximate DC $\neg(p_1 \wedge p_3)$ is obtained. This branch corresponds to the *evidence inversion* for exact DCs, but can terminate before covering all the evidences.

Finally, $\Sigma = \{\neg(p_1 \wedge p_2), \neg(p_2 \wedge p_3), \neg(p_1 \wedge p_3)\}$. The DCs are obtained at different nodes, as shown in Figure 1. □

Proposition 2: Algorithm AEI discovers the complete set Σ of minimal and valid approximate DCs, with the given evidence set evi_r, predicate space $\mathcal{P}$ and error threshold ϵ.

Proof sketch: *Validity.* All DCs in Σ are valid, since the validity is checked for every DC before the DC is added into Σ.

Minimality. All DCs in Σ pass the minimality check (lines 10, 22, 33). It suffices to check the minimality of ϕ by considering all DCs that are added into Σ before ϕ. AEI adopts a DFS strategy to enumerate all combinations of the ways to cover or not to cover each evidence from evi_r, and for each evidence e, the branch where e is not covered

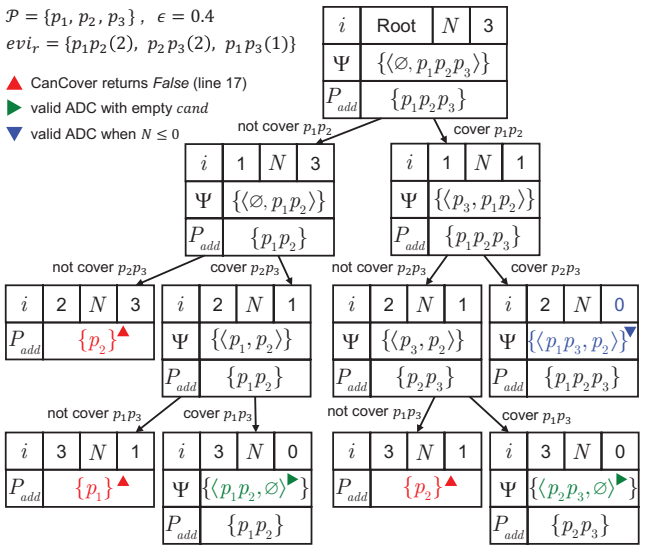


Figure 1: Example 9 for Algorithm 4

is visited before the branch where e is covered. This guarantees for any two candidates ψ and ϕ, ϕ has at least one predicate not in ψ if ϕ is generated after ψ in the DFS. Hence, the DCs generated after ϕ does not affect the minimality of ϕ. Note this also applies to the candidate DCs that are generated based on the same DC (line 29), since a different predicate is used for a candidate each time.

Completeness. AEI uses two branches to cover or not to cover each evidence in evi_r. By considering all evidences in evi_r one by one, it enumerates all combinations of the ways to cover or not to cover each evidence. In each branch, the technique of *evidence inversion* is used to refine candidate DCs. In the branch where an evidence e is not covered, AEI only deals with DCs in Ψ^- (line 24), *i.e.,* the DCs that do not cover e. This does not affect the completeness. All the other DCs, *i.e.,* the DCs that cover e, and the new candidate DCs generated based on DCs from Ψ^-, are processed in the branch where e is covered (line 36). Moreover, note AEI only prunes candidates that cannot be valid or minimal during the traversal. □

Remarks. (1) The complexity of AEI is irrelevant of $|r|$. The worst-case complexity of AEI can be measured by the search space, *i.e.,* the total number of candidate DCs, and is hence exponential in the number $|\mathcal{P}|$ of predicates. AEI uses branches to enumerate all combinations of the ways to cover or not to cover each evidence from evi_r. The last branch where all evidences are covered corresponds to the original *evidence inversion* for exact DCs [5], but the branch may terminate earlier (recall Example 9). In the branch where an evidence e is not covered, only predicates from e and DCs that do not cover e are considered afterwards, which reduces the search space. (2) The running time of AEI (and the enumeration methods of [7, 31, 35]) depends on the actual search space, *i.e.,* the candidate DCs that are really generated and verified during the traversal. The strategy of AEI is to enumerate evidences and use each evidence to refine candidate DCs, as opposed to the strategy adopted in [7, 31, 35] that enumerates combinations of predicates to cover evidences. Although the two strategies lead to the same set

Table 4: Datasets and Execution Statistics (time in seconds, and TL denotes 24^+ hours)

Dataset Properties					Error Threshold ($\epsilon = 0.1$)				Error Threshold ($\epsilon = 0.01$)				Error Threshold ($\epsilon = 0.001$)			
dataset	$\|r\|$	$\|R\|$	$\|\mathcal{P}\|$	$\|evi_r\|$	FastADC	DCFinder	ADCMiner	$\|\Sigma\|$	FastADC	DCFinder	ADCMiner	$\|\Sigma\|$	FastADC	DCFinder	ADCMiner	$\|\Sigma\|$
Airport	55,113	11	32	904	**9.5**	90.8	91.2	44	**9.5**	92.1	91.8	122	**9.3**	91.3	90.6	237
Hospital	114,920	15	30	601	**31.4**	232.2	232.1	33	**32.9**	234.8	234.2	55	**32.1**	238.3	237.7	122
Inspection	229,209	15	40	5,939	**447.8**	1,280	1,321	148	**449.1**	1,302	1,326	128	**448.2**	1,343	1,371	251
NCVoter	675,000	15	38	1,541	**2,743**	22,235	22,238	814	**2,753**	22,246	22,243	1,477	**2,745**	22,114	22,108	620
Tax	500,000	15	62	11,007	**1,165**	11,494	14,269	10,237	**1,174**	13,770	14,270	13,484	**1,197**	16,144	13,781	36,151
SPStock	122,496	7	70	3,023	**182.2**	689.3	647.2	97	**183.5**	690.2	668.3	440	**191.6**	1,018.1	614.4	3,041
Food	200,000	16	56	1,436	**333.1**	3,085	2,772	100	**336.2**	3,926	2,792	169	**332.9**	5,010	2,856	179
Atom	147,067	10	62	614	**102.3**	908.9	850.2	260	**103.4**	984.7	853.8	824	**102.1**	1,237.8	882.9	1,031
Classification	70,859	10	134	6,376	**65.4**	$250 + TL$	$250 + TL$	5,801	**194.4**	$250 + TL$	$250 + TL$	21,987	**782.4**	$250 + TL$	$250 + TL$	74,754

of DCs finally, their actual search spaces can be quite different. Our experimental evaluations in Section 6 verify that AEI significantly outperforms previous approaches [7, 31, 35].

Approximate set cover enumeration. Discovering approximate DCs based on the evidence set is a special case of enumerating approximate set covers [31], with evi_r as the subset family defined on the set $\mathcal{P}$ of elements (each evidence corresponds to a subset). AEI can be employed to solve the general approximate set cover enumeration problem with two modifications. (a) For a subset $\{p_1, \ldots, p_m\}$, use one of p_i ($i \in [1, m]$) rather than $\overline{p_i}$ to cover the subset. (b) Remove only element p_i (but not any other elements) from the set of available elements after using p_i, since all elements from $\mathcal{P}$ are independent of each other in the general setting.

6 EXPERIMENTAL EVALUATIONS

In this section, we conduct an experimental evaluation to verify our approximate DC discovery algorithm, and to provide detailed analyses of our techniques to build the evidence set and discover approximate DCs based on evidence set.

6.1 Experimental settings

Datasets. We use 9 real-life and synthetic datasets, and most of them are also evaluated in previous works [5, 7, 31, 35]. The characteristics of them are given in Table 4. For each dataset, we give the number $|r|$ of tuples, the number $|R|$ of attributes, and the number $|\mathcal{P}|$ of predicates. The predicate space $\mathcal{P}$ is determined by following the rules given in Section 3.2.

Algorithms. All the algorithms are implemented in Java. (1) We compare our approximate DC discovery method, referred to as FastADC, with DCFinder [35] and ADCMiner [31], the two state-of-the-art approximate DC discovery methods. (2) We also evaluate our techniques for the two phases of DC discovery, respectively. (a) We compare our method to build the evidence set (Section 4), referred to as ClueToEvi, with the known fastest method [35], called EviBuild. Note EviBuild is also used in [31]. (b) We compare our method to discover approximate DCs from the evidence set (Section 5), referred to as AEI, with SearchMC used in [7, 34, 35], and ADCEnum used in [31]. The source code of DCFinder is online[2]. We use EviBuild from DCFinder and develop a best-effort implementation of ADCEnum, as the first and second phase of ADCMiner respectively.

Running environment. Unless otherwise stated, all the experiments are run on a machine with an Intel Xeon E-2224 3.4G CPU (4 physical cores), 64GB of memory and CentOS. By default, we set the number of threads as 4 for both ClueToEvi and EviBuild, and the shard size $\omega = 350$ (tuples) for ClueToEvi. Note ClueToEvi and EviBuild (the first phase of DC discovery) leverage parallelism, but AEI, SearchMC and ADCEnum (the second phase) do not.

The qualitative evaluations of DC discovery methods are well studied in [7, 31, 35], concerning (a) the expressiveness of DCs compared with other dependencies, (b) the effectiveness of approximate DC discovery in recalling DCs from dirty data, (c) the effectiveness of ranking functions for helping users select useful DCs, and (d) the effect of the threshold ϵ. The findings apply to this work, since the correctness of FastADC is verified by checking the equivalence of its result and that of DCFinder (ADCMiner). In the sequel, we compare our methods with previous ones in the running time. Each experiment is run 3 times and the average is reported.

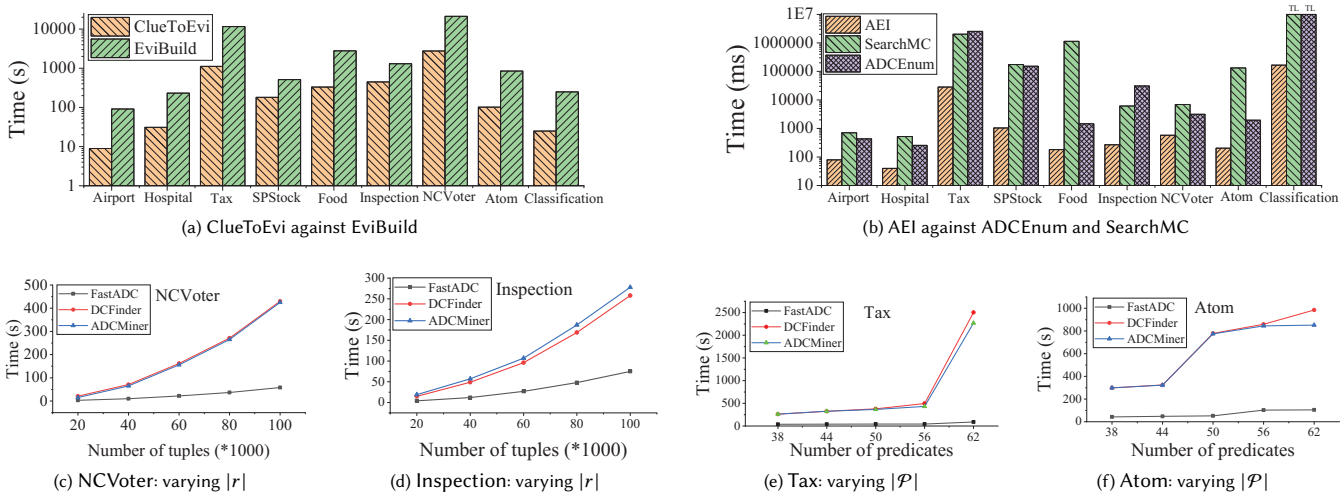
6.2 Experimental results of DC discovery

In this subsection, we experimentally study the methods for approximate DC discovery.

Exp-1: FastADC against DCFinder and ADCMiner. We report running times of all the methods in Table 4, with the error threshold $\epsilon = 0.1$, 0.01 and 0.001, respectively. All the times are shown in seconds. To provide insight into the results, we give the number $|evi_r|$ of elements of evi_r, *i.e.,* the number of distinct evidences. Note evi_r is determined by r and $\mathcal{P}$, but is irrelevant of ϵ. We also show the number $|\Sigma|$ of discovered DCs, which depends on evi_r, $\mathcal{P}$ and ϵ.

We see the following. (a) FastADC is much faster than DCFinder and ADCMiner on all the tested datasets with all the settings of ϵ. Excluding dataset Classification, FastADC is on average 8.2 (resp. 7.5) times and up to 15.1 (resp. 12.1) times faster than DCFinder (resp. ADCMiner). Note DCFinder and ADCMiner *cannot* process Classification within the time limit of 24 hours (250 seconds for the first phase, but more than 24 hours for the second phase). (b) The parameter ϵ does not affect the evidence set construction. When the time of the first phase governs the overall time of DC discovery, the times of all the methods usually vary slightly as ϵ varies. This is especially evident for FastADC, since AEI takes less than 1 second on many datasets. However, the time of DC discovery can vary

[2]https://github.com/HPI-Information-Systems/metanome-algorithms/tree/master/dcfinder (last accessed 2022/10/11).



(a) ClueToEvi against EviBuild

(b) AEI against ADCEnum and SearchMC

(c) NCVoter: varying $|r|$

(d) Inspection: varying $|r|$

(e) Tax: varying $|\mathcal{P}|$

(f) Atom: varying $|\mathcal{P}|$

Figure 2: FastADC **against** DCFinder **and** ADCMiner

considerably as ϵ varies, when the second phase takes a long time. This is because the time of the second phase is very sensitive to ϵ on some datasets. (c) The difference between the running times of DCFinder and ADCMiner is usually small, but on some datasets, *e.g.,* Food and Tax, there are huge gaps between the times of the two methods. Since both DCFinder and ADCMiner use EviBuild for evidence set construction, the gap between their running times is fully determined by the difference between SearchMC and ADCEnum.

We contend that FastADC is a much more efficient solution to approximate DC discovery, compared with previous ones.

Exp-2: Time decomposition. In the same setting as Exp-1, for each method we show the time for evidence set construction in Figure 2a, and the time for DC discovery from the evidence set with $\epsilon = 0.01$ in Figure 2b, respectively. This enables us to compare ClueToEvi with EviBuild (resp. AEI with ADCEnum and SearchMC) in detail. The time for building Plis is omitted, which is trivial and almost the same for all the methods.

The results tell us the following. (a) FastADC consistently beats DCFinder and ADCMiner in both phases. Specifically, ClueToEvi is on average 7.5 times and up to 10.1 times faster than EviBuild. AEI is at least 8.9 times and up to 1,237 times faster than SearchMC, and at least 5.4 times and up to 147 times faster than ADCEnum. Similar results are seen with $\epsilon = 0.1$ and 0.001. (b) Although the first phase of approximate DC discovery takes more time than the second phase on most datasets (the times for the first phase are given in seconds, and those for the second phase are shown in milliseconds), the second phase of DCFinder and ADCMiner can take a (very) long time on some datasets, *e.g.,* more than 24 hours on Classification and more than 30 minutes on Tax. AEI is crucial for the performance of FastADC on these datasets.

Exp-3: The scalability of algorithms. We study the scalability of FastADC and the compared methods by varying parameters. The parameters $|r|$ and $|\mathcal{P}|$ determine $|evi_r|$, and hence affect the times of the first and second phases of DC discovery, while the parameter ϵ only affects the second phase. We set $\epsilon = 0.01$ in this set of experiments, and defer the study of ϵ to Exp-8.

(1) We first study the impact of $|r|$. By varying $|r|$ from 20K to 100K on NCVoter, we report the results in Figure 2c. We see the following. (a) FastADC is on average 7.1 (resp. 6.6) and up to 7.4 (resp. 7.3) times faster than DCFinder (resp. ADCMiner). (b) FastADC takes 3 to 58 seconds, while DCFinder (resp. ADCMiner) takes 21 to 430 seconds (resp. 15 to 425 seconds), as $|r|$ varies from 20K to 100K. The speedup ratio of FastADC increases with the increase of $|r|$, which implies FastADC scales better with $|r|$.

We vary $|r|$ from 20K to 100K on Inspection and show the results in Figure 2d. The results are similar to those found on NCVoter. Specifically, FastADC is on average 3.7 (resp. 4.3) times faster than DCFinder (resp. ADCMiner). The advantage of FastADC becomes more evident with the increase of $|r|$.

(2) We then study the impact of $|\mathcal{P}|$. To vary $|\mathcal{P}|$, we vary $|R|$ and regenerate the predicate space $\mathcal{P}$. Using the first 100K tuples of Tax, we report the results in Figure 2e. As $|\mathcal{P}|$ varies from 38 to 62, all the methods take more time; this is because the number of discovered DCs sharply increases from 460 to 18,102 (not shown). The second phase becomes costly for large $|\Sigma|$. Specifically, as $|\mathcal{P}|$ varies from 38 to 62, AEI takes 0.2 to 42 seconds, as opposed to 2 to 1,868 seconds by ADCEnum and 1 to 2,102 seconds by SearchMC. The increase of $|\mathcal{P}|$ also leads to more time for evidence set construction. Specifically, as $|\mathcal{P}|$ varies from 38 to 62, ClueToEvi takes 39 to 46 seconds, as opposed to 263 to 400 seconds by EviBuild. $|\mathcal{P}|$ has a much stronger impact on the second phase of DC discovery than the first phase, as expected. The search space of DC discovery is exponential in $|\mathcal{P}|$, and hence the time of the second phase is usually very sensitive to $|\mathcal{P}|$. The advantage of AEI becomes more evident with the increase of $|\mathcal{P}|$. This favors the comparison of FastADC against DCFinder and ADCMiner, since the second phase rather than the first phase governs the overall time of DC discovery on Tax with 100K tuples.

Using Atom, we vary $|\mathcal{P}|$ and report the results in Figure 2f. (a) As $|\mathcal{P}|$ varies from 38 to 62, AEI takes 0.042 to 1.1 seconds, as

Table 5: Comparison of Approximate Dependency Discovery Methods (ϵ = 0.01)

Dataset	PYRO (UCC/FD) [28]		DisAOD (OD) [20]		FastADC (DC)		Examples of discovered DCs
	Time (second)	$\|\Sigma\|$	Time (second)	$\|\Sigma\|$	Time (second)	$\|\Sigma\|$	
Airport	1.2	30	3.8	1	9.5	122	$\forall t, s \in r, \neg(t.type = s.type \wedge t.gps\_code = s.gps\_code \wedge t.gps\_code \neq s.local\_code)$
Inspection	3.7	17	14.4	0	449.1	128	$\forall t, s \in r, \neg(t.dbaname = s.akaname \wedge t.address = s.address \wedge t.facilitytype \neq s.facilitytype)$
Tax	5.0	41	209.4	1,347	1,174	13,484	$\forall t, s \in r, \neg(t.state = s.state \wedge t.singleexemp < s.childexemp \wedge t.childexemp > s.childexemp)$

opposed to 0.06 to 2 seconds by ADCEnum and 0.11 to 133 seconds by SearchMC (not shown). The ratios of the increases in the times are high, but the times of AEI and ADCEnum are still trivial. (b) The first phase takes far more time than the second phase on Atom. Specifically, as $|\mathcal{P}|$ varies from 38 to 62, ClueToEvi takes 41 to 102 seconds, as opposed to 287 to 852 seconds by EviBuild. We see ClueToEvi scales better with $|\mathcal{P}|$ than EviBuild.

Exp-4: Comparison of approximate dependency discoveries. We compare FastADC with PYRO [28] and DisAOD [20], the state-of-the-art methods for discovering approximate UCCs/FDs and lexicographical ODs[3]. We show the running time and the number $|\Sigma|$ of discovered dependencies of each method in Table 5. FastADC takes more time than PYRO and DisAOD, as expected. Recall FastADC (resp. PYRO) has a search space exponential in $|\mathcal{P}|$ (resp. $|R|$), and $|\mathcal{P}|$ is much larger than $|R|$ (shown in Table 4). The worst-case complexity of DisAOD is factorial in $|R|$, but the number of valid approximate ODs is usually much smaller than that of DCs, which in practice facilitates the pruning of search space. In Table 5 we show some meaningful discovered DCs [7, 31, 35] that are beyond the expressiveness of FDs and ODs. DCs can indeed specify intricate attribute relationships by using combinations of predicates (possibly across attributes). These discovered DCs can be used in data cleaning and query optimization, as explained in [7, 31, 35].

6.3 In-depth evaluations

In this subsection, we conduct more detailed experiments to evaluate the methods for the two phases.

Exp-5: The choice of shard size ω. We experimentally study the choice of parameter ω, which sets the shard size (Section 4.3). Intuitively, a too small ω may lead to too many shards and hence increase the overhead of maintaining threads and switching between threads. In contrast, a too large ω hinders parallelism and may negatively affect the cache hit ratio of clues. This is because each clue may be corrected several times in BuildClue (BuildPartialClue), and a too large shard is likely to cause clues to be removed from the cache during the processing within a thread.

By varying ω, we run ClueToEvi on all the datasets; 20K tuples are used for Tax and NCVoter. The results are reported in Figure 3a. We see the trends are similar for all the datasets. The time first decreases and then increases as ω increases, and the turning point is roughly within 200 to 500 tuples each shard. It is desirable to have similar turning points for all the datasets, and to find ω within a relatively large range can deliver similar good performance, which facilitates the choice of ω. The experimental results justify our default setting of ω (ω = 350).

Exp-6: The benefit of clue set. We set the number of threads as 1 in this set of experiments, which enables us to compare ClueToEvi and EviBuild *without* parallelism. ClueToEvi still leverages instance sharding technique and EviBuild uses tuple pair partition.

We test all the datasets from Table 4, and use 100K tuples for Tax and NCVoter. Reducing the number of tuples favors EviBuild, since ClueToEvi scales better with $|r|$ than EviBuild (Exp-3). We report the results in Figure 3b. ClueToEvi is on average 6.8 times and up to 8.6 times faster than EviBuild. The results justify the benefit of clue set. The efficiency of clue set construction is high, and is further enhanced with sharding technique. The transformation cost is trivial since $|clue_r| \ll |r|^2$, as shown in Table 4.

Exp-7: Speedup ratios *w.r.t.* the number of threads. We study two kinds of speedup ratio of ClueToEvi, by varying the number of threads. This set of experiments is run on a machine with two Intel Xeon E5-2620 V2 2.1G CPU (6 physical cores each CPU, with hyper-threading disabled) and 64GB of memory. It supports up to 12 threads in parallel, providing more insights into multithreaded parallelism. We use all the tuples of Airport, Hospital, SPStock and Atom, and 100K tuples of the others.

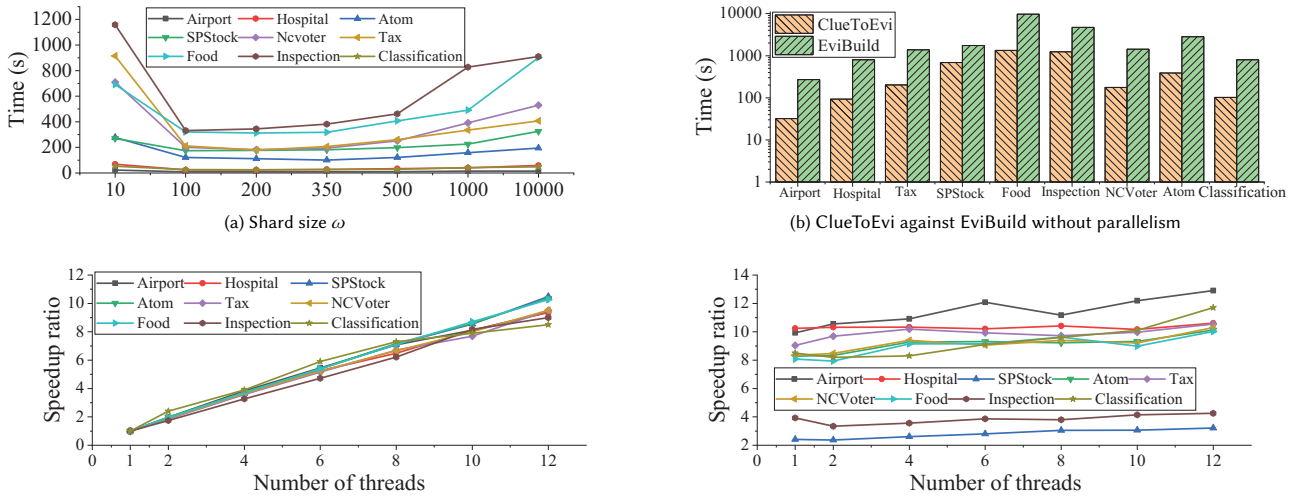
(1) In Figure 3c, we show the speed up ratio of ClueToEvi *w.r.t.* ClueToEvi without parallelism, *i.e.,* the ratio of the time taken by ClueToEvi with 1 thread to that by ClueToEvi with k threads, by varying k. ClueToEvi can well leverage the available threads, and hence almost scales linearly with the number of threads. Specifically, the average speed up ratio of ClueToEvi with 12 threads is 9.6.

(2) In Figure 3d, we show the speedup ratio of ClueToEvi *w.r.t.* EviBuild, *i.e.,* the ratio of the time taken by EviBuild to that by ClueToEvi. In almost all cases, the speedup ratio increases as the number of threads increases; ClueToEvi scales better with the number of threads than EviBuild. Specifically, as the number of threads varies from 1 to 12, on average ClueToEvi extends the lead by 16.7%, based on its advantage over EviBuild with 1 thread.

Exp-8: The impact of ϵ. In this set of experiments, we study the impact of error threshold ϵ in AEI, SearchMC and ADCEnum.

(1) In Figure 4a, we vary ϵ on dataset Airport. (a) AEI consistently outperforms SearchMC and ADCEnum. AEI is on average 9 times and 6.6 times faster than SearchMC and ADCEnum, respectively. (b) As ϵ decreases, all the methods usually take more time. Intuitively, a small ϵ implies more evidences need to be covered, and hence, DCs with more predicates. This negatively affects the efficiency of enumeration methods, in most cases.

[3]The codes are online at https://github.com/HPI-Information-Systems/pyro and https://github.com/chenjixuan20/AOD (last accessed 2022/10/11).



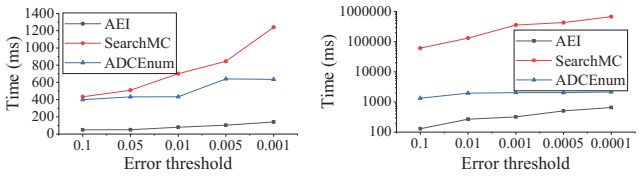
(a) Shard size ω

(b) ClueToEvi against EviBuild without parallelism

(c) The speedup ratio of ClueToEvi *w.r.t.* ClueToEvi without parallelism

(d) The speedup ratio of ClueToEvi *w.r.t.* EviBuild

Figure 3: Shard size ω, ClueToEvi against EviBuild without parallelism, and speedup ratios of ClueToEvi



(a) Airport: varying ϵ

(b) Atom: varying ϵ

Figure 4: AEI, SearchMC **and** ADCEnum ***w.r.t.*** ϵ

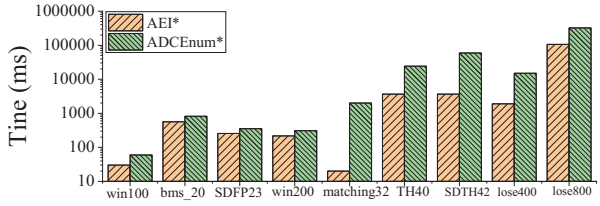


Figure 5: Approximate set cover enumeration

(2) We vary ϵ on dataset Atom in Figure 4b. (a) AEI is much faster than SearchMC and ADCEnum, up to orders of magnitude. (b) The times of all the methods increase as ϵ decreases. This is mainly because the number of discovered DCs significantly increases (not shown), as ϵ decreases from 10^{-1} to 10^{-4}.

Note a smaller ϵ does not necessarily lead to a larger $|\Sigma|$. As shown in Table 4, $|\Sigma|$ decreases on NCVoter when ϵ varies from 0.01 to 0.001. We find many DCs valid for ϵ = 0.01 become invalid for ϵ = 0.001, and most of them fail to produce new valid DCs with all the available predicates. Also note on some datasets the behaviors of different methods may vary in response to the changes of ϵ, due to their different traversal strategies and pruning techniques. In Table 4, AEI and SearchMC take more time but ADCEnum takes less time, as ϵ varies from 0.01 to 0.001 on Tax and SPStock.

Exp-9: Approximate set cover enumeration. We adapt AEI and ADCEnum to solve the general approximate set cover enumeration problem [31], as described in Section 5. We use the benchmark datasets of [32]. The datasets of [32] do not carry weights, while the computation of *approximate* set covers concerns the weights of subsets. In this set of experiments, we assign weights to subsets by following the normal distribution. The results of the two methods (denoted by AEI* and ADCEnum*) are reported in Figure 5. AEI* is far more efficient than ADCEnum*; AEI* is on average 15.6 and up to 100.5 times faster than ADCEnum* on the tested datasets.

7 CONCLUSION

We have investigated discovering approximate DCs. We have improved the evidence set construction by first building the clue set and then transforming the clue set to the evidence set, and studied parallel clue set construction. We have developed a novel method to discover approximate DCs from the evidence set. Extensive experiments have been conducted to verify the efficiency of our methods.

It is very necessary to further improve the precision of DC discovery so as to better leverage the discovered constraints. As shown in [51], user interactions are usually required to select meaningful business rules from the discovered dependencies. We intend to develop practical discovery systems that effectively leverage minimal user interactions. Another topic is to further study inference rules of (approximate) DCs, to enable pruning of DCs in the output.

ACKNOWLEDGMENTS

This work is supported by National Key R&D Program of China 2018YFB1403200 and NSFC 62172102, 61572135, 61925203.

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] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[4] Johann Birnick, Thomas Bläsius, Tobias Friedrich, Felix Naumann, Thorsten Papenbrock, and Martin Schirneck. 2020. Hitting Set Enumeration with Partial Information for Unique Column Combination Discovery. *Proc. VLDB Endow.* 13, 11 (2020), 2270–2283.

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

[6] Qi Cheng, Jarek Gryz, Fred Koo, T. Y. Cliff Leung, Linqi Liu, Xiaoyan Qian, and K. Bernhard Schiefer. 1999. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. In *VLDB*. 687–698.

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

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

[9] Cristian Consonni, Paolo Sottovia, Alberto Montresor, and Yannis Velegrakis. 2019. Discovering Order Dependencies through Order Compatibility. In *EDBT*. 409–420.

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

[12] 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.

[13] Chang Ge, Ihab F. Ilyas, and Florian Kerschbaum. 2019. Secure Multi-Party Functional Dependency Discovery. *PVLDB* 13, 2 (2019), 184–196.

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

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

[16] Amir Gilad, Daniel Deutch, and Sudeepa Roy. 2020. On Multiple Semantics for Declarative Database Repairs. In *SIGMOD*. 817–831.

[17] Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. 2013. Scalable Discovery of Unique Column Combinations. *PVLDB* 7, 4 (2013), 301–312.

[18] Ihab F. Ilyas and Xu Chu. 2019. *Data Cleaning*. ACM.

[19] Joseph F. JáJá. 1992. *An Introduction to Parallel Algorithms*. Addison-Wesley.

[20] Yifeng Jin, Zijing Tan, Weijun Zeng, and Shuai Ma. 2021. Approximate Order Dependency Discovery. In *ICDE*. 25–36.

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

[22] Reza Karegar, Parke Godfrey, Lukasz Golab, Mehdi Kargar, Divesh Srivastava, and Jaroslaw Szlichta. 2021. Efficient Discovery of Approximate Order Dependencies. In *EDBT*. 427–432.

[23] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2015. BigDansing: A System for Big Data Cleansing. In *SIGMOD*. 1215–1230.

[24] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. 2015. Lightning Fast and Space Efficient Inequality Joins. *PVLDB* 8, 13 (2015), 2074–2085.

[25] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. 2017. Fast and scalable inequality joins. *VLDB J.* 26, 1 (2017), 125–150.

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

[27] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *VLDB J.* 31, 1 (2022), 1–22.

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

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

[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] Keisuke Murakami and Takeaki Uno. 2014. Efficient algorithms for dualizing large-scale hypergraphs. *Discret. Appl. Math.* 170 (2014), 83–94.

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

[34] Eduardo H. M. Pena and Eduardo Cunha de Almeida. 2018. BFASTDC: A Bitwise Algorithm for Mining Denial Constraints. In *DEXA*. 53–68.

[35] 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.

[36] Eduardo H. M. Pena, Eduardo Cunha de Almeida, and Felix Naumann. 2022. Fast Detection of Denial Constraint Violations. *PVLDB* 15, 4 (2022), 859–871.

[37] Eduardo H. M. Pena, Edson Ramiro Lucas Filho, Eduardo Cunha de Almeida, and Felix Naumann. 2020. Efficient Detection of Data Dependency Violations. In *CIKM*. 1235–1244.

[38] 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.

[39] Hemant Saxena, Lukasz Golab, and Ihab F. Ilyas. 2019. Distributed Discovery of Functional Dependencies. In *ICDE*. 1590–1593.

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

[41] 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.

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

[43] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. 1996. Fundamental Techniques for Order Optimization. In *SIGMOD*. 57–67.

[44] 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.

[45] 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.

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

[47] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, Wenbin Ma, Weinan Qiu, and Calisto Zuzarte. 2014. Business-Intelligence Queries with Order Dependencies in DB2. In *EDBT*. 750–761.

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

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

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

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

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

[53] C. Xavier and S. Sitharama Iyengar. 1998. *Introduction to parallel algorithms*. Wiley.