Discovery of Approximate Lexicographical Order Dependencies

Yifeng Jin, Zijing Tan, Jixuan Chen, Shuai Ma

Abstract—Lexicographical order dependencies (LODs) specify orders between list of attributes, and are proven useful in optimizing SQL queries with order by clauses. To discover hidden dependencies from dirty data in practice, approximate dependency discoveries are actively studied, aiming at automatically discovering dependencies that hold on data with some exceptions. In this paper we study the discovery of approximate LODs. (1) We adapt two error measures, namely g_1 and g_3, to LODs. We prove their desirable properties, present efficient algorithms for computing the measures and related lower and upper bounds, and study the relationship between the two measures. (2) We present an efficient approximate LOD discovery algorithm that is well suited to the two error measures, with a set of pruning rules, optimization techniques and ranking functions. (3) We study techniques for estimating g_1 by sampling, with high accuracy and far less time. (4) We conduct extensive experiments to verify the effectiveness and scalability of our methods, using both real-life and synthetic data.

Index Terms—Data profiling; Data dependency; Metadata

1 Introduction

Lexicographical order dependencies (LODs) are proposed in [30], [32], which state *lexicographical* ordering specifications. Different from traditional dependencies that are defined on *sets* of attributes, *e.g.,* functional dependencies (FDs) and denial constraints (DCs) [4], LODs are defined on *lists* of attributes. LODs lend themselves to wide applicability, since sorting is one of the most important database operations, and sorting in the *SQL order-by* clause is lexicographical on lists of attributes. Indeed, LODs have been proven useful in query optimizations concerning sorting [30]–[32].

We will review the formal definition of LODs in Section 3, and first give an illustrative example.

Example 1: Consider Table 1, concerning employees in a company (now suppose $t_3.Salary$ = 4500 and $t_6.Salary$ = 8500). We see an employee with a higher salary is at a higher level, or stays at the same level for more years. This is denoted by $\overrightarrow{\text{Salary}}\mapsto\overrightarrow{\text{Level}}\overrightarrow{\text{Year}}$ in the notation of LODs [30], [32]: the ascending order on $Salary$ guarantees the ascending order on $Level$, and the ascending order on $Year$ within each $Level$ group. The lexicographical ordering specification on the left-hand-side (LHS) (resp. right-hand-side (RHS)) of the LOD is consistent with the SQL clause ORDER BY $Salary$ ASC (resp. $Level$ ASC, $Year$ ASC).

LODs are defined on *lists of attributes*, and can have multiple attributes on both LHS and RHS. Observe the following unique features of LODs.

(1) The order of attributes in a list is relevant. For example, $\overrightarrow{\text{Salary}}\mapsto\overrightarrow{\text{Year}}\overrightarrow{\text{Level}}$ does not hold on r.

- *Yifeng Jin, Zijing Tan and Jixuan Chen are with School of Computer Science, Fudan University. Zijing Tan is the corresponding author. E-mail: {18210240103,zjtan,20210240341}@fudan.edu.cn*
- *Shuai Ma is with the SKLSDE Lab, School of Computer Science and Engineering, Beihang University, China. E-mail: mashuai@buaa.edu.cn*

Manuscript received XXXXX; revised XXXXX

TABLE 1
Relation instance r

	Name	Salary	Level	Year	Age
t_1	Alan	4200	1	2	30
t_2	Mark	4300	1	3	31
t_3	Jack	4500→5800	2	2	38
t_4	William	5600	2	3	30
t_5	Steven	8000	3	2	35
t_6	Thomas	8500→8000	4	5	39

(2) The attributes in the LHS and RHS list usually *cannot* be separated. For example, $\overrightarrow{\text{Salary}}\mapsto\overrightarrow{\text{Year}}$ does not hold on r. □

To avoid the error-prone and labor-intensive process of designing dependencies manually, dependency discovery techniques are actively studied; see *e.g.,* [1], [27] for surveys. Recently, LOD discoveries have received an increasing attention [5], [11], [17]. The LOD discovery problem is necessarily very difficult. It has a search space (the total number of candidate LODs) factorial in the number of attributes since LODs concern lists of attributes, in contrast to the search space of FDs that is exponential in the number of attributes.

Data in practice are often dirty, and hence, some of the discovered (exact) dependencies *cannot* correctly express the characteristics of data. It is known that discovered constraints on dirty data may *overfit* [16], [21], [23]. This means too many LHS attributes used in FDs, or too many predicates specified in DCs. Intuitively, this is because the discovered constraints have to be more "specialized" to "tolerate" errors in data. When it comes to LOD discovery, the problem is even more involved. We find LODs discovered from dirty data can not only overfit, but also *underfit*. By *underfit*, we mean too few attributes are specified on the RHS, which implies that the order specification is not fully established. Indeed, the following example shows that a single LOD can overfit on the LHS, and simultaneously, underfit on the RHS.

Example 2: Recall Table 1. Now suppose that there are errors in the data: $t_3.Salary$ = 5800, $t_6.Salary$ = 8000.

We see $\overrightarrow{\mathsf{Salary}}\mapsto\overrightarrow{\mathsf{Level}}\overrightarrow{\mathsf{Year}}$ no longer holds. Specifically, (a) t_4 is before t_3 by $Salary$ ASC, but t_3 is before t_4 by $Level$ ASC, $Year$ ASC. (b) The order of t_5, t_6 is *unspecified* by $Salary$ ASC, but t_5 is before t_6 by $Level$ ASC, $Year$ ASC.

Consequently, an exact LOD discovery algorithm may find $\overrightarrow{\mathsf{Salary}}\overrightarrow{\mathsf{Age}}\mapsto\overrightarrow{\mathsf{Level}}$ that holds on the dirty data. Compared with the original (desirable) LOD $\overrightarrow{\mathsf{Salary}}\mapsto\overrightarrow{\mathsf{Level}}\overrightarrow{\mathsf{Year}}$, it *overfits* on the LHS and *underfits* on the RHS. The attribute $Year$ on the RHS is removed for resolving the violation incurred by t_3, t_4, and the attribute Age is included for resolving the violation caused by t_5, t_6. □

Dirty data in practice motivate the quest for discovering *approximate* dependencies that hold on data with some exceptions. Although desirable, approximate dependency discovery is usually more challenging and expensive than the exact counterpart. Intuitively, exact dependency discoveries concern the *decision* problem of whether a dependency holds or not, while approximate dependency discoveries concern the *counting* problem of measuring the error rate of a dependency. The increasing complexity is well demonstrated in recent studies for approximate FDs [16] and DCs [21], [23].

Contributions. In this paper, we study the discovery of approximate lexicographical order dependencies.

(1) We adapt two error measures to LODs. For a LOD, measure g_1 concerns the number of violating tuple pairs, while g_3 concerns the minimum number of tuples that must be removed such that the LOD is satisfied. We show that both measures have desirable properties and can be efficiently computed. We study the lower and upper bounds for the measures, to enable efficient pruning in approximate LOD discoveries. We also investigate the relationship between the two measures. (Section 5).

(2) We provide an algorithm to discover the complete set of minimal and valid approximate LODs. It traverses the search space of candidate approximate LODs, and computes the error measures and lower/upper bounds of the candidates. A set of pruning rules, optimization techniques and ranking functions are also incorporated into the algorithm, for improving effectiveness and efficiency. (Section 6).

(3) We study techniques for estimating g_1 with high accuracy, by sampling a small portion of tuple pairs. Our sampling strategy is efficient and widely applicable, since the sample size is independent of the instance size, and no assumption is made about data distributions. (Section 7).

(4) Using a host of real-life and synthetic data, we conduct an extensive experimental evaluation. The results demonstrate the effectiveness of approximate LOD discovery in recalling correct LODs from dirty data, and the effectiveness of pruning rules, optimizations, ranking functions and estimation by sampling. (Section 8).

2 Related Work

This work extends our earlier conference version [10]. (1) We have added (a) the study of the relationship between measures g_1 and g_3 (Section 5.3), (b) ranking functions for measuring the interestingness of discovered approximate LODs (Section 6.3), (c) the investigation of the estimation of error measure g_1 by sampling (Section 7), and (d) experimental evaluations concerning the ranking functions and error measure estimation (Exp-6 and Exp-7 in Section 8). (2) We have presented more theoretical results underlying our approach (Propositions 4, 7, 8, 9, 11 and proofs of all the propositions). (3) We have also provided the details of *state compaction* used in Algorithm 5 (Section 6.2).

Theoretical Foundations of Order Dependencies. Unidirectional and bidirectional LODs are proposed in [30], [32], and proven useful in optimizing queries with *order-by* clauses [31]. Different from the list-based LODs, two classes of *set-based* order dependencies (ODs) are also discussed. Set-based *canonical* ODs are proposed in [28], [29]. They are shown to generalize LODs, in the sense that each LOD can be mapped to a set of canonical ODs and any relation instance satisfies the LOD iff it satisfies all canonical ODs in the set. Another set-based ODs [7], [8], known as *pointwise* ODs, further generalize canonical ODs. There is no one-to-one relationship between a set-based OD and a LOD, and to our best knowledge, no techniques exist for transforming set-based ODs to LODs.

We focus on LODs in this paper. They model order specifications in *SQL* and are hence preferable in practice.

Exact Order Dependency Discoveries. There has been an increasing interest in OD discovery techniques. They are studied in [5], [11], [17] for LODs, and in [28], [29], [33] for set-based ODs, aiming to automatically find ODs that hold on the data without exceptions.

In this paper, we study the discovery of approximate LODs, which significantly differs from prior works on exact LOD discovery. Exact LOD discovery algorithms test the satisfaction of each candidate LOD, and a LOD does not hold if a single violation is identified. In contrast, approximate LOD discoveries need to measure the error rate for each candidate approximate LOD. We adapt two error measures to LODs, with both theoretical analyses and efficient computation methods. We also present a discovery algorithm that is well suited to the error measures, with novel pruning rules and optimization techniques for improving efficiency.

Note that it is highly non-trivial, if not impossible, to extend exact LOD discovery techniques to approximate LODs. [5] is based on the observation that each LOD can be divided into an FD and an *order compatibility dependency* (OCD) (refer to Section 5), and the LOD holds iff both FD and OCD hold. This does not apply to approximate LODs. Intuitively, we cannot have the "embedded" FD and OCD in a LOD both hold with exceptions if the LOD holds with exceptions. For example, an approximate FD and an exact OCD can also form an approximate LOD. [11] is experimentally verified to be very efficient, by first discovering LODs on a small subset of data, and then refining LODs on full data in an iterative way. The rationale behind [11] is that any exact LOD that holds on data must hold on any subset of it. This does not apply to approximate LODs. Indeed, the error rate of a LOD may increase after removing some tuples from data, and hence, an approximate LOD valid on full data may be invalid on subsets of it.

Approximate Dependency Discoveries. To cope with dirty data in practice, approximate dependency discoveries are studied for, *e.g.*, FDs [16], CFDs [24], DCs [21], [23] and set-based canonical ODs [12], [29]. Based on the definition of approximation using notions from information theory,

[14] studies implication for approximate dependencies, and discoveries of approximate multi-valued dependencies and then acyclic schemes are investigated in [13].

In this paper, we study the discovery of approximate LODs. The computation of error measures significantly depends on the dependency types, and a completely different strategy is required to generate list-based candidate approximate LODs compared with those set-based dependencies.

A different notion, referred to as *approximate band conditional* LOD, is proposed in [18], [19]. Band LODs relax themselves to hold approximately and conditionally on subsets of data. [18], [19] are complementary to this work. For an approximate LOD discovered by us, the method of [18], [19] can be employed to split the data instance into contiguous segments such that the LOD holds on each segment.

3 Preliminaries

In this section, we review the definition of bidirectional lexicographical order dependencies [30], [32].

Basic notations. $R(A,\ldots)$ denotes a relation schema, r denotes an instance of R, and t, s denote tuples of r. We use *marked attribute*, written as $\overline{A}$, to model the order specifications. $\overline{A}$ is either $\overrightarrow{A}$ or $\overleftarrow{A}$, for A asc or A desc respectively. t_A denotes the value of attribute A in t, and $t_{\overline{A}} = t_A$.

Attribute List. X denotes a list of marked attributes, *i.e.,* $[\overline{A_1},\ldots,\overline{A_k}]$, and $\mathcal{X}$ denotes the set of attributes (without directions) in X. Given a tuple t, t_{X} denotes the list of attribute values on X, *i.e.,* $[t_{A_1},\ldots,t_{A_k}]$.

A non-empty list X can be denoted as $[\overline{A_i}\ |\mathsf{Y}\ ]$, where *head* $\overline{A_i}$ is a single marked attribute, and *tail* Y is the remaining list. For X = $[\overline{A_1},\ldots,\overline{A_k}]$, $prefix(\mathsf{X})$ denotes the set of all possible prefixes of X, *i.e.,* $[\overline{A_1},\ldots,\overline{A_i}]$ for any $i < k$.

Lexicographical Ordering. For a marked attribute $\overline{A}$ and tuples t, s, we write $t \prec_{\overline{A}} s$ iff (a) $\overline{A}$=$\overrightarrow{A}$ and $t_A{<}s_A$; or (b) $\overline{A}$=$\overleftarrow{A}$ and $t_A{>}s_A$. We write $t{=}_{\overline{A}}s$ iff $t_A{=}s_A$.

Given X = $[\overline{A_1},\ldots,\overline{A_k}]$, we write $t \preceq_{\mathsf{X}} s$ iff (a) X = [ ]; or (b) $t \prec_{\overline{A_1}} s$; or (c) X = $[\overline{A_1}\ |\mathsf{Y}\ ]$ such that $t =_{\overline{A_1}} s$ and $t \preceq_{\mathsf{Y}} s$. We write $t{=}_{\mathsf{X}}s$ iff $t{\preceq}_{\mathsf{X}}s$ and $s{\preceq}_{\mathsf{X}}t$, *i.e.,* $t =_{\overline{A_i}} s$ for all $i \in [1,k]$. We write $t{\prec}_{\mathsf{X}}s$ iff $t{\preceq}_{\mathsf{X}}s$ but $s{\not\preceq}_{\mathsf{X}}t$.

Bidirectional Lexicographical Order Dependency [32]. Given two lists X, Y, γ = $\mathsf{X} \mapsto \mathsf{Y}$ denotes a *bidirectional lexicographical order dependency* (abbreviated as LOD). A relation instance r satisfies γ iff for any two tuples $t, s \in r$, $t \preceq_{\mathsf{Y}} s$ if $t \preceq_{\mathsf{X}} s$. If r satisfies γ, then we say γ holds on r.

Remarks. (1) Along the same setting as [11], [17], in the sequel we consider *completely non-trivial* LODs whose LHS and RHS attribute lists (neglecting direction) are disjoint. (2) Each LOD γ has a *symmetry* LOD γ' by reversing all directions [11]. For example, $\overleftarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}$ is the symmetry of $\overrightarrow{\mathsf{A}} \mapsto \overleftarrow{\mathsf{B}}\,\overleftarrow{\mathsf{C}}$: $\overleftarrow{\mathsf{A}}$ (resp. $\overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}$) is the reverse order of $\overrightarrow{\mathsf{A}}$ (resp. $\overleftarrow{\mathsf{B}}\,\overleftarrow{\mathsf{C}}$), and hence, $\overleftarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}$ holds iff $\overrightarrow{\mathsf{A}} \mapsto \overleftarrow{\mathsf{B}}\,\overleftarrow{\mathsf{C}}$ holds. W.l.o.g., we consider LODs with asc on the leftmost attribute in the RHS attribute list, *e.g.,* $\overleftarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}$.

4 Problem Formulation

In this section we give the definition of approximate LODs, and formalize the approximate LOD discovery problem.

Error measures. We use function g to measure the error rate of LODs. Specifically, $g(\gamma, r)$ takes as inputs a LOD γ and a relation instance r. The smaller g value is, the fewer errors *w.r.t.* γ are on r. We first present four criteria for judging whether an error measure function g makes sense, and will study the details of two error functions in Section 5.

(1) $g(\gamma, r)$ is in the range of [0, 1] for any LOD γ on any relation instance r, and $g(\gamma, r)$ = 0 iff γ holds on r.

(2) $g(\mathsf{XA} \mapsto \mathsf{Y}, r) \le g(\mathsf{X} \mapsto \mathsf{Y}, r)$: appending an attribute to the LHS never leads to more errors.

(3) $g(\mathsf{X} \mapsto \mathsf{Y}, r) \le g(\mathsf{X} \mapsto \mathsf{YY}', r)$: appending an attribute to the RHS never removes any errors.

(4) $g(\gamma, r)$ can be efficiently computed, *e.g.,* in polynomial time. This is necessary for a practical setting.

Observe that criteria (2), (3) are consistent with the implication of exact LODs. It is proven in [30], [32] that $\mathsf{X} \mapsto \mathsf{Y}$ *logically implies* $\mathsf{XA} \mapsto \mathsf{Y}$, and $\mathsf{X} \mapsto \mathsf{YY}'$ *logically implies* $\mathsf{X} \mapsto \mathsf{Y}$. Recall that a dependency δ logically implies γ in the sense that every instance that satisfies δ must satisfy γ.

Approximate LOD. Given an error measure function g and an error threshold e, we say that a LOD γ is an *approximate* LOD (abbreviated as AOD) valid on r iff $g(\gamma, r) \le e$.

Note that the introduction of a given threshold is a typical setting adopted in approximate dependency discoveries, *e.g.,* [12], [16], [21], [23], [24], [29].

It is better to discover *minimal* valid dependencies rather than all valid ones [4], [17], [22]. In the sequel we establish the minimality of AODs. Intuitively, an attribute list X is *not minimal* if part of it already imposes the same ordering specification. Inspired by the *reduce order* procedure in [25], we have the following result, and hence the definition of minimal attribute list.

Lemma 1. For a list X, a marked attribute $\overline{B}$, a subset $\mathcal{Y} \subseteq \mathcal{X}$ and two tuples t, s, (a) $t \prec_{\mathsf{X}\overline{\mathsf{B}}} s$ if $t{\prec}_{\mathsf{X}}s$, and (b) if FD $\mathcal{Y} \to B$ holds, then $t =_{\mathsf{X}\overline{\mathsf{B}}} s$ if $t{=}_{\mathsf{X}}s$.

Proof: It is easy to see that (a) holds by the definition of lexicographical ordering. For (b), we have $t =_{\overline{\mathsf{B}}} s$, if $t{=}_{\mathsf{X}}s$ and $\mathcal{X} \to B$. Taken together, $t =_{\mathsf{X}\overline{\mathsf{B}}} s$. □

Minimal Attribute List. An attribute list X is minimal, iff there do not exist (a) a subset $\mathcal{Y}$ of $\mathcal{X}$ and (b) an attribute B in X that is after all attributes in $\mathcal{Y}$, where $\mathcal{Y} \to B$ holds.

Example 3: If $AB \to C$ holds, then we know neither of $\overrightarrow{\mathsf{A}}\,\overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}$, $\overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{D}}\,\overrightarrow{\mathsf{A}}\,\overrightarrow{\mathsf{C}}$, $\overleftarrow{\mathsf{A}}\,\overrightarrow{\mathsf{B}}\,\overleftarrow{\mathsf{C}}$ is minimal; we have the same ordering specification after removing $\overrightarrow{\mathsf{C}}$ ($\overleftarrow{\mathsf{C}}$). Note that the LHS attributes of the FD are not required to be a *sublist* (continuous), and directions of attributes are irrelevant. □

Implication of AODs. As noted earlier, an error measure function g should guarantee that $g(\mathsf{XA} \mapsto \mathsf{Y}, r) \le g(\mathsf{X} \mapsto \mathsf{Y}, r)$ and $g(\mathsf{X} \mapsto \mathsf{Y}, r) \le g(\mathsf{X} \mapsto \mathsf{YY}', r)$. Hence, on any instance r, $\mathsf{XA} \mapsto \mathsf{Y}$ is a valid AOD if $\mathsf{X} \mapsto \mathsf{Y}$ is a valid AOD, and $\mathsf{X} \mapsto \mathsf{Y}$ is a valid AOD if $\mathsf{X} \mapsto \mathsf{YY}'$ is a valid AOD.

We are now ready to define *minimal AODs.*

Minimal AODs. An AOD $\mathsf{X} \mapsto \mathsf{Y}$ is minimal, iff

(1) X and Y are minimal attribute lists; and

(2) for any $\mathsf{X}'{\in}prefix(\mathsf{X})$, $\mathsf{X}' \mapsto \mathsf{Y}$ is not a valid AOD; and

(3) for any non-empty list Y', $\mathsf{X} \mapsto \mathsf{YY}'$ is not a valid AOD.

TABLE 2
Relation instance r

	A	B	C	D	E	F	G	H
t_1	1	2	1	1	1	1	1	2
t_2	1	2	1	1	1	1	2	3
t_3	1	4	3	1	3	2	1	3
t_4	1	1	3	2	2	3	1	4
t_5	2	5	5	2	4	4	1	5
t_6	2	6	5	3	5	5	1	1

Example 4: If $\overleftarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}$ is valid, then $\overleftarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$ is not minimal. $\overleftarrow{\mathsf{A}}\,\overrightarrow{\mathsf{C}} \mapsto \overrightarrow{\mathsf{B}}$ is not minimal if $\overleftarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$ is valid. $\overleftarrow{\mathsf{A}}\,\overrightarrow{\mathsf{B}} \mapsto \overrightarrow{\mathsf{C}}$ is not minimal if $A \rightarrow B$ holds, since $\overleftarrow{\mathsf{A}}\,\overrightarrow{\mathsf{B}}$ is not minimal. □

In the definition of minimal AODs, rule (1) specifies the requirement of minimal attribute lists, and rules (2), (3) follow the principle of logical implication. All AODs satisfying the threshold are considered valid, and minimality check is conducted on valid AODs to return only minimal ones.

Discovery of AODs. Given a relational instance r, an error measure function g and a threshold e, AOD discovery is to find the complete set of minimal valid AODs on r.

5 Error Measures for LODs

In this section, we adapt two error measures to LODs. We show they satisfy the four criteria stated in Section 4, by providing theoretical results and efficient algorithms. We investigate the lower and upper bounds for these measures, to enable early pruning in AOD discoveries. We also study the relationship between the two measures.

5.1 The Percentage of Violating Tuple Pairs

The most common error measure, referred to as g_1, is introduced for FDs [15], [16], and extended to *e.g.,* DCs [4], [21], [23]. The computation of g_1 is closely related to violations of a dependency. We review violations of LODs first.

LOD violations [30], [32]. Violations of a LOD γ = X $\mapsto$Y are categorized into two types: split and swap.

(1) A tuple pair (t, s) incurs a split, if $t =_{\mathsf{X}} s$, $t \neq_{\mathsf{Y}} s$.

(2) A tuple pair (t, s) incurs a swap, if $t \prec_{\mathsf{X}} s$, $s \prec_{\mathsf{Y}} t$.

X $\mapsto$Y has an "embedded" FD $\mathcal{X} \rightarrow \mathcal{Y}$, and (t, s) incurs a split *w.r.t.* X $\mapsto$Y iff t, s violate $\mathcal{X} \rightarrow \mathcal{Y}$. In contrast, a swap is caused by a *swapped* tuple pair (t, s), *i.e.,* t is before s if sorted by X, but s is before t if sorted by Y. This is formalized by *order compatibility dependencies* (OCDs) [5].

Example 5: Recall $\overrightarrow{\mathsf{Salary}} \mapsto \overrightarrow{\mathsf{Level}}\,\overrightarrow{\mathsf{Year}}$ on the dirty instance r in Example 2. (t_3, t_4) is a *swapped* tuple pair, and hence incurs a swap. (t_5, t_6) incurs a split: t_5, t_6 violate FD $Salary \rightarrow Level, Year$, since they have the same value on $Salary$ but different values on $Level$ and $Year$. □

Error measure g_1**.** The g_1 is the ratio of the number of violating tuple pairs to the number of all tuple pairs [15]. By considering split and swap, we extend g_1 to LODs.

$$g_{\mathsf{split}}(\mathsf{X} \mapsto \mathsf{Y}, r) = \frac{|\{(t,s) \in r^2 \mid t =_{\mathsf{X}} s \wedge t \neq_{\mathsf{Y}} s\}|}{|r|^2 - |r|}$$
$$g_{\mathsf{swap}}(\mathsf{X} \mapsto \mathsf{Y}, r) = \frac{|\{(t,s) \in r^2 \mid t \prec_{\mathsf{X}} s \wedge s \prec_{\mathsf{Y}} t\}|}{|r|^2 - |r|}$$
$$g_1(\mathsf{X} \mapsto \mathsf{Y}, r) = g_{\mathsf{split}}(\mathsf{X} \mapsto \mathsf{Y}, r) + 2 \times g_{\mathsf{swap}}(\mathsf{X} \mapsto \mathsf{Y}, r)$$

Herein, $|r|$ denotes the number of tuples in r. We scale up g_{swap} to balance out the fact that split is *symmetric*, but swap is *asymmetric*: (t, s) causes a split iff (s, t) causes a split, but (t, s) *does not* cause a swap if (s, t) causes a swap.

Algorithm 1: Computation of $g_{\mathsf{split}}(\mathsf{X} \mapsto \mathsf{Y}, r)$

```
    Input: sorted partitions $\tau_X$,$\tau_Y$
    Output: $g_{\mathsf{split}}$(X $\mapsto$Y, $r$)
1   $spl \leftarrow 0$;
2   foreach equivalence class $ec$ in $\tau_X$ do
3       $map \leftarrow$ an empty hash table;
4       foreach tuple $t$ in $ec$ do
5           $map[I_Y[t]] \leftarrow map[I_Y[t]] + 1$;
6       foreach $v$ in the value set of $map$ do
7           $spl \leftarrow spl + v \times (|ec| - v)$;
8   return $g_{\mathsf{split}} = \frac{spl}{|r|^2 - |r|}$;
```

It can be verified that $g_1(\gamma, r)$ ranges over [0, 1], and $g_1(\gamma, r)$ = 0 iff γ holds on r. The following proposition shows that g_1 satisfies criteria (2), (3) stated in Section 4.

Proposition 2. (1) $g_1(\mathsf{XA} \mapsto \mathsf{Y}, r) \leq g_1(\mathsf{X} \mapsto \mathsf{Y}, r)$, and (2) $g_1(\mathsf{X} \mapsto \mathsf{Y}, r) \leq g_1(\mathsf{X} \mapsto \mathsf{YY}', r)$.

Proof: (1) We prove $g_1(\mathsf{XA} \mapsto \mathsf{Y}, r) \leq g_1(\mathsf{X} \mapsto \mathsf{Y}, r)$ as follows. (a) Suppose (t, s) causes a split *w.r.t.* XA $\mapsto$Y, *i.e.,* $t =_{\mathsf{XA}} s$, $t \neq_{\mathsf{Y}} s$. It is easy to see that (t, s) also causes a split *w.r.t.* X $\mapsto$Y. (b) Suppose (t, s) causes a swap *w.r.t.* XA $\mapsto$Y, *i.e.,* $t \prec_{\mathsf{XA}} s$, $s \prec_{\mathsf{Y}} t$. We see that both (t, s) and (s, t) lead to a split *w.r.t.* X $\mapsto$Y if $t =_{\mathsf{X}} s$, or that (t, s) causes a swap *w.r.t.* X $\mapsto$Y if $t \prec_{\mathsf{X}} s$.

(2) We then prove $g_1(\mathsf{X} \mapsto \mathsf{Y}, r) \leq g_1(\mathsf{X} \mapsto \mathsf{YY}', r)$. We can see that (t, s) always causes a split (resp. swap) *w.r.t.* X $\mapsto$YY′ if it causes a split (resp. swap) *w.r.t.* X $\mapsto$Y. □

We show g_1 can be efficiently computed by developing such algorithms. We first review an auxiliary data structure.

Sorted Partition. The data structure, referred to as *sorted partition,* is employed in exact LOD discoveries [11], [17]. Given an attribute list X, the sorted partition τ_X on an instance r is a sorted list of *equivalence classes (sets).* Specifically, tuples with the same value on X are put into the same equivalence class, and for tuples t, s with different values on X, the equivalence class of t is before that of s if $t \prec_{\mathsf{X}} s$. It is known that a sorted partition is built in $O(|r|log(|r|))$ on r.

We use $|\tau_X|$ to denote the number (count) of equivalence classes in τ_X, and define the *rank* of a tuple t in τ_X as the sequence number of the equivalence class that t belongs to, denoted by $I_X[t]$. Intuitively, $I_X[t]$ denotes the order of t on X in a compact way.

Example 6: Consider Table 2. $\tau_{\overrightarrow{A}}$ = $[\{t_1, t_2, t_3, t_4\}$,$\{t_5, t_6\}]$, $\tau_{\overrightarrow{B}}$ = $[\{t_4\}$,$\{t_1, t_2\}$,$\{t_3\}$,$\{t_5\}$,$\{t_6\}]$. $I_{\overrightarrow{A}}[t_3]$ = 1; t_3 is in the first equivalence class of $\tau_{\overrightarrow{A}}$. Similarly, $I_{\overrightarrow{B}}[t_3]$ = 3. □

Observe that a split is always incurred by two tuples in the same equivalence class, while two tuples in different equivalence classes can only lead to a swap. We present two algorithms for computing g_{split} and g_{swap} respectively.

Algorithm. Algorithm 1 is provided for computing g_{split}. A tuple pair (t, s) incurs a split *w.r.t.* X $\mapsto$Y, if t, s have the same values on X but different values on Y. Hence, in each equivalence class ec of τ_X, we count the number of tuples for each distinct value on Y using a hash table with $I_Y[t]$ as the key (lines 3-5). A value v in the hash table implies v tuples having the same value on Y, and each of these tuples forms a split with any tuple from the other $|ec| - v$ tuples, *i.e.,*

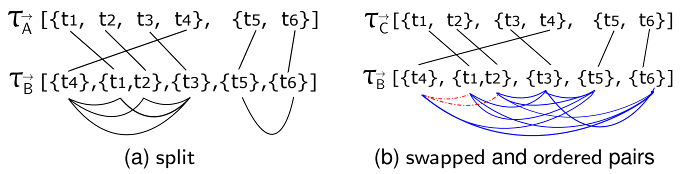

$\tau_{\vec{A}}$ [{t1, t2, t3, t4}, {t5, t6}]
$\tau_{\vec{B}}$ [{t4},{t1,t2},{t3},{t5},{t6}]
(a) split
$\tau_{\vec{C}}$ [{t1, t2}, {t3, t4}, {t5, t6}]
$\tau_{\vec{B}}$ [{t4}, {t1,t2}, {t3}, {t5}, {t6}]
(b) swapped and ordered pairs

Fig. 1. Example 7 for Algorithm 1 and Example 9 for Algorithm 2

tuples in the same equivalence class of τ_X but in different equivalence classes of τ_Y (lines 6-7). Herein, $|ec|$ denotes the number of tuples in the equivalence class ec.

Example 7: Consider $\overrightarrow{A} \mapsto \overrightarrow{B}$ on Table 2. We show the related split violations in Figure 1a. (1) In the first equivalence class $\{t_1, t_2, t_3, t_4\}$ of $\tau_{\vec{A}}$, $I_{\vec{B}}[t_1] = I_{\vec{B}}[t_2] = 2$, $I_{\vec{B}}[t_3] = 3$, and $I_{\vec{B}}[t_4] = 1$. On the hash table, we have $map[2] = 2$, $map[3] = map[1] = 1$. The number of violating tuple pairs is computed as 2×(4-2) + 1×(4-1) + 1×(4-1) = 10. (2) Similarly, we then deal with the second equivalence class $\{t_5, t_6\}$ of $\tau_{\vec{A}}$. □

Time Complexity. Assuming a constant cost for the hash table, Algorithm 1 has a complexity of $O(|r|)$ on τ_X and τ_Y.

Lower and upper bounds of g_1. As will be illustrated in Section 6.1, when traversing the search space of candidate AODs, new candidates following X $\mapsto$Y are in the form of XU $\mapsto$YV (U or V can be empty). It is beneficial if we can obtain a lower bound and an upper bound of g_1 for XU $\mapsto$YV. This is because (1) XU $\mapsto$YV *cannot* be a valid AOD and can be pruned, if its lower bound of g_1 is larger than the error threshold e; and (2) XU $\mapsto$YV is *always* a valid AOD if its upper bound of g_1 is not larger than e.

The bounds can be efficiently computed together with g_{swap}, based on the following observations.

Proposition 3. [17], [32] If (t, s) incurs a swap *w.r.t.* X $\mapsto$Y, then (t, s) also incurs a swap *w.r.t.* XU $\mapsto$YV.

Proposition 4. If $t \prec_X s$ and $t \prec_Y s$, then (t, s) never incurs any LOD violations *w.r.t.* XU $\mapsto$YV.

Proof: We can see that (t, s) does not incur a split *w.r.t.* XU $\mapsto$YV since $t \not=_{XU} s$, and that (t, s) does not incur a swap *w.r.t.* XU $\mapsto$YV, since $t \prec_{XU} s$ and $t \prec_{YV} s$. □

We refer to a tuple pair (t, s) as an *ordered pair* if $t \prec_X s$ and $t \prec_Y s$, and refer to (t, s) as a *swapped pair* if $t \prec_X s$ and $s \prec_Y t$. Intuitively, when appending more attributes to the LHS and RHS attribute lists, Proposition 3 states that swap violations *can never* be resolved, while Proposition 4 tells us that ordered pairs still *never* lead to violations. We hence compute the bounds as follows:

$$LBg_1(\text{XU} \mapsto \text{YV}, r) = 2 \times g_{\text{swap}}(\text{X} \mapsto \text{Y}, r)$$
$$UBg_1(\text{XU} \mapsto \text{YV}, r) = 1 - 2 \times \frac{|\{(t,s) \in r^2 \mid t \prec_X s \wedge t \prec_Y s\}|}{|r|^2 - |r|}$$

The example below shows that both bounds are *tight*.

Example 8: Recall Table 2. For $\overrightarrow{F} \mapsto \overrightarrow{H}$, it can be seen that there are 5 swapped pairs and 8 ordered pairs. We have the lower and upper bounds are 10/30 and 14/30 respectively. It can be verified that $\overrightarrow{F}\overrightarrow{G} \mapsto \overrightarrow{H}$ has g_1 of 10/30 and $\overrightarrow{F} \mapsto \overrightarrow{H}\overrightarrow{G}$ has g_1 of 14/30. □

We present one auxiliary structure to facilitate our algorithm for computing g_{swap} and the number of ordered pairs.

Segment Tree. We employ a simple yet effective data structure, known as segment tree [2]. A segment tree supports various range queries, *e.g.*, range sum/max/min queries, and it takes $O(n)$ to build and $O(log(n))$ to update and query a segment tree built on range $[1, n]$ (integer values). A segment tree also has a space complexity of $O(n)$. As will be seen shortly, the *ranks* of tuples in a sorted partition are well suited for transforming operations on the sorted partition to the corresponding segment tree.

Algorithm 2: Computation of g_{swap} and the number of ordered pairs

```
  Input: sorted partitions $\tau_X$ and $\tau_Y$
  Output: $g_{\text{swap}}$ and the number of ordered pairs for X ↦Y
1 $swap \leftarrow 0$, $ordered \leftarrow 0$;
2 $seg \leftarrow$ an empty segment tree on range $[1, |\tau_Y|]$;
3 foreach equivalence class $ec$ in $\tau_X$ do
4     foreach tuple $t$ in $ec$ do
5         $swap \leftarrow swap + seg.query([I_Y[t] + 1, |\tau_Y|])$;
6         $ordered \leftarrow ordered + seg.query([1, I_Y[t] - 1])$;
7     foreach tuple $t$ in $ec$ do
8         $seg.insert(I_Y[t])$;
9 return $g_{\text{swap}} = \frac{swap}{|r|^2 - |r|}$, $ordered$;
```

Algorithm. Algorithm 2 aims to compute g_{swap} and the number of ordered pairs simultaneously. The application of a segment tree built on range $[1, |\tau_Y|]$ is at the core of this algorithm. On this tree, $insert(key)$ increases the value associated with key by 1, for counting the number of tuples in the same equivalence class of τ_Y (line 8), and $query([a, b])$ performs a range *sum* query on $[a, b]$, for the total number of tuples in several equivalence classes (lines 5-6).

Equivalence classes ec in τ_X are processed one by one in order (line 3), and updates of the segment tree with tuples in ec (line 8) are conducted after the queries concerning these tuples (lines 5-6). Therefore, the number of swap violations *w.r.t.* a tuple t is the number of tuples whose *rank* in τ_Y is larger than that of t (line 5), and the number of ordered pairs *w.r.t.* t is the number of tuples whose *rank* in τ_Y is smaller than that of t (line 6). Recall that a larger (resp. smaller) rank in τ_Y implies a larger (resp. smaller) value on Y.

Example 9: Consider $\overrightarrow{C} \mapsto \overrightarrow{B}$ on Table 2 (shown in Figure 1b). (1) The first equivalence class $\{t_1, t_2\}$ of $\tau_{\vec{C}}$ does not lead to swapped or ordered pairs. After t_1, t_2 are inserted into the segment tree, there are two tuples whose rank in $\tau_{\vec{B}}$ is 2. (2) In the second equivalence class of $\tau_{\vec{C}}$, t_3 leads to 2 ordered pairs (solid lines) since the rank of t_1, t_2 is smaller than $I_{\vec{B}}[t_3] = 3$, while t_4 incurs 2 swapped pairs (dashed lines) since the rank of t_1, t_2 is larger than $I_{\vec{B}}[t_4] = 1$. The segment tree is employed to facilitate an efficient range *sum* query. We then update the tree with t_3, t_4. (3) $\{t_5, t_6\}$ of $\tau_{\vec{C}}$ is processed similarly, which leads to more ordered pairs. □

Time Complexity. The segment tree has a range of $[1, |\tau_Y|]$, and $|\tau_Y|$ equals $|r|$ in the worst case. It hence takes at most $O(|r|)$ to build and $O(log(|r|))$ to update and query the tree. The update and query are conducted for each tuple once. Algorithm 2 has a worst-case complexity of $O(|r|log(|r|))$.

5.2 The Minimum Number of Removed Tuples

Another error measure, referred to as g_3 in literature, is also originally introduced for FDs [15]. It is extended to CFDs [24], set-based canonical LODs [12], [29], comparable

TABLE 3
Relation instance r

	A	B	C	D
t_1	1	2	1	3
t_2	1	1	1	4
t_3	1	3	2	4
t_4	1	4	2	3
t_5	2	2	1	1
t_6	2	1	1	2
t_7	2	3	2	2
t_8	2	4	2	1

dependencies [26] and DCs [21], among others. The computation of g_3 can be very expensive, *e.g., NP-Complete* for comparable dependencies [26] and DCs [21].

Error measure g_3 for LODs. The g_3 measures the minimum number of tuples that must be removed from the given instance such that the dependency is satisfied. Specifically,

$$g_3(\gamma, r) = \frac{|r| - \max\{|r'| \mid r' \subseteq r,\ r' \ satisfies\ \gamma\}}{|r|}$$

Obviously, we have $g_3(\gamma, r) \in [0, 1]$, and $g_3(\gamma, r) = 0$ iff γ holds on r. We then show how to compute g_3, for illustrating the satisfaction of the criteria in Section 4. We present one more definition to facilitate our approach.

OD sequence. For a LOD γ = X $\mapsto$Y, an *OD sequence* on an instance r is a list of tuples t_{a_1}, t_{a_2}, ..., t_{a_k} from r, such that for any $1 \leq i < j \leq k$, (a) $t_{a_i} \preceq_{\mathsf{X}} t_{a_j}$, (b) $t_{a_i} \preceq_{\mathsf{Y}} t_{a_j}$, and (c) $t_{a_i} =_{\mathsf{Y}} t_{a_j}$ if $t_{a_i} =_{\mathsf{X}} t_{a_j}$. It is easy to see that any two tuples in this sequence *cannot* form a LOD violation. We say an OD sequence is the *longest OD sequence*, denoted by $\mathsf{LOS}(\gamma, r)$, whose k is the maximum among all OD sequences (choose an arbitrary one if several ones have the same k).

Example 10: Consider Table 3. For $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{C}}$, we have a LOS $[t_1, t_2, t_5, t_6]$. There can be more than one LOS, *e.g.,* $[t_3, t_4, t_7, t_8]$ or $[t_1, t_2, t_7, t_8]$ is also a LOS. □

We show g_3 can be readily computed from LOS.

Proposition 5. $g_3(\gamma, r) = 1 - \frac{|\mathsf{LOS}(\gamma, r)|}{|r|}$.

Proof: Suppose that g_3 is computed based on a maximum subset of r that satisfies γ, say r'. Let γ = X $\mapsto$Y. We can order all tuples in r' by X, to form a sequence. It can be verified that this is an OD sequence, since any two tuples in r' do not incur a violation. The sequence is also the longest. If we have any longer OD sequence, then all tuples in it form a subset larger than r', which contradicts the assumption. □

Remark. LOS differs from *the longest increasing sequence* in *e.g.,* [6], [9], [20]. This is because LOD violations consist of both swap and split. In an OD sequence, we must have (a) $t_{a_i} \preceq_{\mathsf{Y}} t_{a_j}$ if $t_{a_i} \preceq_{\mathsf{X}} t_{a_j}$, similar to the longest increasing sequence, and (b) $t_{a_i} =_{\mathsf{Y}} t_{a_j}$ if $t_{a_i} =_{\mathsf{X}} t_{a_j}$, which is unique.

We show the monotonicity of g_3, based on Proposition 5.

Proposition 6. (1) $g_3(\mathsf{XA} \mapsto \mathsf{Y}, r) \leq g_3(\mathsf{X} \mapsto \mathsf{Y}, r)$, and (2) $g_3(\mathsf{X} \mapsto \mathsf{Y}, r) \leq g_3(\mathsf{X} \mapsto \mathsf{YY'}, r)$.

Proof: (1) We prove $g_3(\mathsf{XA} \mapsto \mathsf{Y}, r) \leq g_3(\mathsf{X} \mapsto \mathsf{Y}, r)$ by showing that $\mathsf{LOS}(\mathsf{X} \mapsto \mathsf{Y}, r)$ can always be transformed into an OD sequence (not necessarily the longest) for XA $\mapsto$Y. Let $\mathsf{LOS}(\mathsf{X} \mapsto \mathsf{Y}, r)$ = t_{a_1}, t_{a_2}, ..., t_{a_k}, where for any $1 \leq i < j \leq k$, (a) $t_{a_i} \preceq_{\mathsf{X}} t_{a_j}$, (b) $t_{a_i} \preceq_{\mathsf{Y}} t_{a_j}$ and (c) $t_{a_i} =_{\mathsf{Y}} t_{a_j}$ if $t_{a_i} =_{\mathsf{X}} t_{a_j}$. The transformation is done by a linear scan of $\mathsf{LOS}(\mathsf{X} \mapsto \mathsf{Y}, r)$, and we exchange the position of t_{a_i} and $t_{a_{(i+1)}}$, if $t_{a_i} =_{\mathsf{X}} t_{a_{(i+1)}}$ (hence $t_{a_i} =_{\mathsf{Y}} t_{a_{(i+1)}}$) and $t_{a_{(i+1)}} \prec_{\mathsf{XA}} t_{a_i}$. It can be verified that the new sequence must be an OD sequence for XA $\mapsto$Y.

(2) We show $\mathsf{LOS}(\mathsf{X} \mapsto \mathsf{YY'}, r)$ is also an OD sequence (not necessarily the longest) for X $\mapsto$Y. Suppose t_{a_1}, t_{a_2}, ..., t_{a_k} is a LOS for X $\mapsto$YY'. For t_{a_i} and t_{a_j} where $1 \leq i < j \leq k$: (a) If $t_{a_i} =_{\mathsf{X}} t_{a_j}$, then we have $t_{a_i} =_{\mathsf{YY'}} t_{a_j}$ by the definition of LOS, and hence $t_{a_i} =_{\mathsf{Y}} t_{a_j}$. (b) If $t_{a_i} \prec_{\mathsf{X}} t_{a_j}$, then we have $t_{a_i} \preceq_{\mathsf{YY'}} t_{a_j}$ by the definition of LOS. We must have $t_{a_i} \preceq_{\mathsf{Y}} t_{a_j}$, otherwise $t_{a_i} \succ_{\mathsf{YB}} t_{a_j}$ if $t_{a_i} \succ_{\mathsf{Y}} t_{a_j}$, which leads to a contradiction. Putting (a) and (b) together, we know t_{a_1}, t_{a_2}, ..., t_{a_k} is an OD sequence for X $\mapsto$Y.

Recall that LOS is the longest OD sequence and $g_3 = 1 - \frac{|\mathsf{LOS}|}{|r|}$. We have $g_3(\mathsf{XA} \mapsto \mathsf{Y}, r) \leq g_3(\mathsf{X} \mapsto \mathsf{Y}, r)$, and $g_3(\mathsf{X} \mapsto \mathsf{Y}, r) \leq g_3(\mathsf{X} \mapsto \mathsf{YY'}, r)$. □

Example 11: Recall Table 3. (1) $[t_1, t_2, t_5, t_6]$ is a LOS for $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{C}}$. After being transformed into $[t_2, t_1, t_6, t_5]$, it is an OD sequence (but not LOS) for $\overrightarrow{\mathsf{A}}\,\overrightarrow{\mathsf{B}} \mapsto \overrightarrow{\mathsf{C}}$. We see one LOS for $\overrightarrow{\mathsf{A}}\,\overrightarrow{\mathsf{B}} \mapsto \overrightarrow{\mathsf{C}}$ is $[t_2, t_1, t_6, t_5, t_7, t_8]$. (2) $[t_1, t_8]$ is a LOS for $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{C}}\,\overrightarrow{\mathsf{D}}$, and is also an OD sequence (but not LOS) for $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{C}}$. □

Similar to g_1, we aim for the lower and upper bounds of g_3 for XU $\mapsto$YV. We find that they can be computed with two other sequences that slightly differ from LOS.

Strict increasing sequence and non-decreasing sequence. For γ = X $\mapsto$Y, (1) a *strict increasing sequence* on r is a list of tuples $t_{a_1}, \ldots, t_{a_k}$, such that for any $1 \leq i < j \leq k$, (a) $t_{a_i} \prec_{\mathsf{X}} t_{a_j}$, and (b) $t_{a_i} \prec_{\mathsf{Y}} t_{a_j}$. The *longest strict increasing sequence,* denoted by $\mathsf{LSIS}(\gamma, r)$, is a strict increasing sequence with the maximum k.

(2) A *non-decreasing sequence* is a list of tuples $t_{a_1}, \ldots, t_{a_k}$, such that for any $1 \leq i < j \leq k$, (a) $t_{a_i} \preceq_{\mathsf{X}} t_{a_j}$, and (b) $t_{a_i} \preceq_{\mathsf{Y}} t_{a_j}$. The *longest non-decreasing sequence*, denoted by $\mathsf{LNDS}(\gamma, r)$, is a non-decreasing sequence with the maximum k.

We have the following results.

Proposition 7. For a LOD γ and a relation instance r, a strict increasing sequence must be an OD sequence, and an OD sequence must be a non-decreasing sequence.

Proof: These can be directly seen by the definitions. □

Proposition 8. Any two tuples in $\mathsf{LSIS}(\mathsf{X} \mapsto \mathsf{Y}, r)$ do not form a violation (split or swap) *w.r.t.* XU $\mapsto$YV.

Proof: For $t_{a_i}, t_{a_j} \in \mathsf{LSIS}(\mathsf{X} \mapsto \mathsf{Y}, r)$, we know (a) $t_{a_i} \neq_{\mathsf{XU}} t_{a_j}$; and (b) $t_{a_i} \prec_{\mathsf{YV}} t_{a_j}$ if $t_{a_i} \prec_{\mathsf{XU}} t_{a_j}$. This guarantees no split or swap violations *w.r.t.* XU $\mapsto$YV. □

Proposition 9. Tuples in $\mathsf{LNDS}(\mathsf{X} \mapsto \mathsf{Y}, r)$ form a subset r' of r such that (a) any two tuples in r' do not incur a swap *w.r.t.* X $\mapsto$Y; and (b) r' is maximum among all subsets of r that satisfy (a).

Proof: Obviously any two tuples in $\mathsf{LNDS}(\mathsf{X} \mapsto \mathsf{Y}, r)$ do not incur a swap *w.r.t.* X $\mapsto$Y. Let r' be the set of all tuples in $\mathsf{LNDS}(\mathsf{X} \mapsto \mathsf{Y}, r)$. If we can find a tuple $s \in r$ such that (a) $s \notin r'$; and (b) s does not incur a swap *w.r.t.* X $\mapsto$Y against any tuple in r', then s can be inserted into $\mathsf{LNDS}(\mathsf{X} \mapsto \mathsf{Y}, r)$, contradicting the assumption that LNDS is the longest. □

In summary, (1) $\mathsf{LSIS}(\mathsf{X} \mapsto \mathsf{Y}, r)$ must be an OD sequence for XU $\mapsto$YV on r (not necessarily the longest), and hence can serve as a lower bound of $\mathsf{LOS}(\mathsf{XU} \mapsto \mathsf{YV}, r)$. (2) $\mathsf{LNDS}(\mathsf{X} \mapsto \mathsf{Y}, r)$ corresponds to a maximum subset of

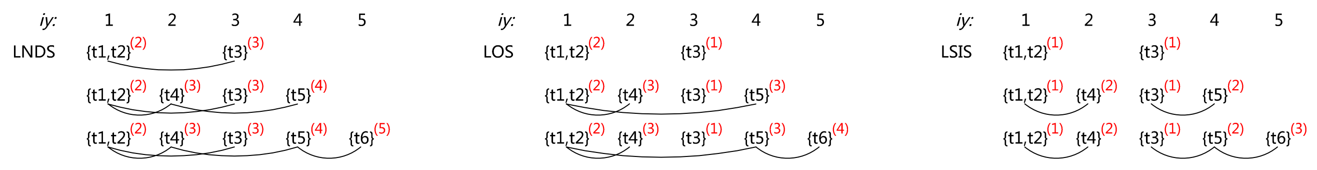

iy: 1 2 3 4 5
LNDS
{t1,t2}(2) {t3}(3)
{t1,t2}(2) {t4}(3) {t3}(3) {t5}(4)
{t1,t2}(2) {t4}(3) {t3}(3) {t5}(4) {t6}(5)
iy: 1 2 3 4 5
LOS
{t1,t2}(2) {t3}(1)
{t1,t2}(2) {t4}(3) {t3}(1) {t5}(3)
{t1,t2}(2) {t4}(3) {t3}(1) {t5}(3) {t6}(4)
iy: 1 2 3 4 5
LSIS
{t1,t2}(1) {t3}(1)
{t1,t2}(1) {t4}(2) {t3}(1) {t5}(2)
{t1,t2}(1) {t4}(2) {t3}(1) {t5}(2) {t6}(3)

Fig. 2. Example 13 for Algorithm 3

```
Algorithm 3: Computations of LOS, LSIS and LNDS
  Input: sorted partitions $\tau_X$,$\tau_Y$
  Output: LOS(X ↦ Y, $r$), LSIS(X ↦ Y, $r$),
          LNDS(X ↦ Y, $r$)
1  $los \leftarrow$ an empty segment tree on range $[1, |\tau_Y|]$;
2  $lsis \leftarrow$ an empty segment tree on range $[1, |\tau_Y|]$;
3  $lnds \leftarrow$ an empty segment tree on range $[1, |\tau_Y|]$;
4  $\tau_{XY} \leftarrow \tau_X$.expand($\tau_Y$);
5  foreach equivalence class $ecx$ in $\tau_X$ do
6      foreach equivalence class $ecxy$ in $\tau_{XY}$ from $ecx$ do
7          $iy \leftarrow I_Y[ecxy[1]]$;
8          $ecxy.los \leftarrow |ecxy| + los.query([1, iy])$;
9          $ecxy.lsis \leftarrow 1 + lsis.query([1, iy-1])$;
10         $ecxy.lnds \leftarrow |ecxy| + lnds.query([1, iy])$;
11         $lnds.insert(iy, ecxy.lnds)$;
12     foreach equivalence class $ecxy$ in $\tau_{XY}$ from $ecx$ do
13         $iy \leftarrow I_Y[ecxy[1]]$;
14         $los.insert(iy, ecxy.los)$;
15         $lsis.insert(iy, ecxy.lsis)$;
16 return $los.query([1, |\tau_Y|])$, $lsis.query([1, |\tau_Y|])$, and
       $lnds.query([1, |\tau_Y|])$;
```

r that is free of swap violations *w.r.t.* X ↦Y. Recall that XU ↦YV can never resolve swap *w.r.t.* X ↦Y. Therefore, LNDS(X ↦ Y, r) is an upper bound of LOS(XU ↦ YV, r).

The following example shows that both bounds are tight.

Example 12: Recall Table 3. For A ↦C, we have a LSIS $[t_1, t_8]$ and a LNDS $[t_1, t_2, t_5, t_6, t_7, t_8]$, which results in a lower bound of 2 and an upper bound of 6 for LOS(AU ↦ CV, r). It can be seen that A ↦CD has a LOS $[t_1, t_8]$, and AB ↦C has a LOS $[t_2, t_1, t_6, t_5, t_7, t_8]$. □

Based on the observation, we are ready to define the lower and upper bounds of g_3 for XU ↦YV.

$$LBg_3(\text{XU} \mapsto \text{YV}, r) = 1 - \frac{|\text{LNDS}(\text{X}\mapsto\text{Y}, r)|}{|r|}$$
$$UBg_3(\text{XU} \mapsto \text{YV}, r) = 1 - \frac{|\text{LSIS}(\text{X}\mapsto\text{Y}, r)|}{|r|}$$

Algorithm. Algorithm 3 is a three-in-one approach to computing LOS, LSIS and LNDS, and we use a segment tree for each of them (lines 1-3). The segment trees here have different operation semantics from those in Algorithm 2, but the complexity of each operation remains unchanged. Specifically, $insert(x, y)$ updates the value associated with key x to y, and $query([a, b])$ returns the max value associated with keys in the range of $[a, b]$. We use segment trees to facilitate our computation in the dynamic programming fashion. The tuple rank in τ_Y is used as the key for querying trees, and the value is the length of the longest sequence (LOS, LSIS or LNDS) that ends with that key (rank).

We first compute τ_{XY} with τ_X and τ_Y (line 4). This is a basic operation on sorted partitions [17]. One equivalence class in τ_X may be divided into several equivalence classes in τ_{XY}, such that tuples in the same equivalence class of τ_{XY} have the same values on both X and Y.

The outer loop (line 5) enumerates equivalence classes ecx in τ_X, ordered by X's value. The first inner loop (lines 6-11) enumerates equivalence classes $ecxy$ in τ_{XY} that are obtained from ecx (with the same X's value), in the order of Y's value. We then identify the rank related to $ecxy$ in τ_Y (all tuples in $excy$ have the same value on Y), denoted by iy in the algorithm (line 7). We use iy as the key for querying and updating segment trees. Note that the Y's values related to keys (equivalence classes in τ_Y) in the range of $[1, iy-1]$ are less than the Y's value related to iy.

Consider the computation of LOS. The LOS that ends with $ecxy$ is obtained by appending all tuples in $excy$ to the *longest* LOS that ends with an equivalence class (a) already inserted into the tree (a smaller value on X) and (b) having a Y's value not larger than that of $ecxy$ (line 8). We save the length of the LOS for $excy$ (line 8), and update the tree with it in the second inner loop (line 14). This is necessary since the same value on Y is required for the same value on X in LOS; a different equivalence class $ecxy'$ from the same ecx *cannot* be combined with $ecxy$. In contrast, we update the tree for LNDS immediately (line 11). This is because $ecxy'$ can contribute to the LNDS of $ecxy$ as long as the Y's value of $ecxy'$ is not larger than that of $excy$. The computation of LSIS differs in the following. Only one (arbitrary) tuple in $excy$ can be appended to LSIS and the range query is conducted on $[1, iy-1]$ (line 9), since an equal value on X or Y is not allowed in LSIS.

Finally, we get LOS, LNDS and LSIS, by querying max values from their corresponding trees (line 16).

Example 13: Consider $\overrightarrow{\text{D}} \mapsto \overrightarrow{\text{E}}$ on Table 2. In Figure 2, we illustrate the process of Algorithm 3, by showing the (length of) LNDS, LOS or LSIS that ends with the rank iy. We have $\tau_{\overrightarrow{D}} = [\{t_1, t_2, t_3\}, \{t_4, t_5\}, \{t_6\}]$ and $\tau_{\overrightarrow{D}\overrightarrow{E}} = [\{t_1, t_2\}, \{t_3\}, \{t_4\}, \{t_5\}, \{t_6\}]$. In the outer loop, we enumerate equivalence classes in $\tau_{\overrightarrow{D}}$.

(1) We deal with $\{t_1, t_2, t_3\}$. In the inner loop, we enumerate equivalence classes in $\tau_{\overrightarrow{D}\overrightarrow{E}}$ that are from $\{t_1, t_2, t_3\}$, *i.e.*, $\{t_1, t_2\}$ and $\{t_3\}$. (a) t_1, t_2 are in the same equivalence class of $\tau_{\overrightarrow{D}\overrightarrow{E}}$; they hence exist (or do not exist) simultaneously in any LOS (resp. LNDS), but only one of them can exist in any LSIS. (b) t_3 can be appended to $\{t_1, t_2\}$ in LNDS, but not in LOS or LSIS.

(2) We handle $\{t_4, t_5\}$ in $\tau_{\overrightarrow{D}}$, which is divided into $\{t_4\}$ and $\{t_5\}$ in $\tau_{\overrightarrow{D}\overrightarrow{E}}$. (a) $\{t_4\}$ can be appended to $\{t_1, t_2\}$ in LOS, LNDS and LSIS. (b) For LNDS, $\{t_5\}$ can be appended to $\{t_1, t_2\}$, $\{t_3\}$ or $\{t_4\}$; in Figure 2 we choose $\{t_4\}$ for the longest sequence (the same for $\{t_3\}$). For LOS and LSIS, $\{t_5\}$ can only be appended to $\{t_1, t_2\}$ or $\{t_3\}$. We must choose $\{t_1, t_2\}$ for LOS, but it is the same to choose $\{t_1, t_2\}$ or $\{t_3\}$ for LSIS, since only one tuple in $\{t_1, t_2\}$ can exist in LSIS.

(3) We deal with $\{t_6\}$ in $\tau_{\overrightarrow{D}}$ (and $\tau_{\overrightarrow{D}\overrightarrow{E}}$). We must choose $\{t_5\}$ for LNDS, but it is the same to choose $\{t_4\}$ or $\{t_5\}$ for

LOS and LSIS. □

Time Complexity. Each segment tree has a range of $[1, |\tau_Y|]$. $|\tau_Y| = |r|$ in the worst case; it takes at most $O(|r|)$ to build and $O(log(|r|))$ to update and query segment trees. The two inner loops are linear in the size of equivalence classes in τ_{XY}. To sum up, Algorithm 2 has a worst-case complexity of $O(|r| log(|r|))$.

5.3 Relationship Between Measures g_1 and g_3

Both g_1 and g_3 of a LOD γ on a relation instance r measure the error rate of γ in r. However, the values of g_1 and g_3 can differ significantly, as illustrated below.

Example 14: Consider an instance $r = \{t^1, t^2, \ldots, t^k\}$ of schema $R(A, B)$ and an AOD $\overrightarrow{A} \mapsto \overrightarrow{B}$. Suppose $t^i{}_A = t^i{}_B = i$ for all $i \in [1, k-1]$, and $t^k{}_A = k$. We consider two cases of $t^k{}_B$: (a) $t^k{}_B = k - 1.5$, or (b) $t^k{}_B = 0$. In both cases, we can remove t^k from r such that $\overrightarrow{A} \mapsto \overrightarrow{B}$ is satisfied, and hence, $g_3 = \frac{1}{k}$. In contrast, we have $g_1^a = \frac{2}{k(k-1)}$ in case (a), but $g_1^b = \frac{2}{k}$ in case (b). The ratios of g_3 to g_1^a and g_1^b to g_1^a can be arbitrarily large, as the number k of tuples increases. □

Formally, we have the following result.

Proposition 10. $\frac{2g_3(\gamma,r)}{|r|-1} \leq g_1(\gamma, r) \leq 2g_3(\gamma, r)$.

Proof: By the definition of g_3, we can find a maximum subset r' of r with the size of $(1 - g_3)|r|$, where any two tuples in r' do not form a violation. For each tuple from the $g_3|r|$ tuples that are removed from r, it violates γ against (a) at most $|r| - 1$ tuples, and (b) at least one tuple from r'. Recall that (1) both (t, s) and (s, t) are counted in g_{split}, and (2) g_{swap} is scaled by 2 when computing g_1. We have:

$$\frac{2g_3|r|}{|r|(|r|-1)} \leq g_1 \leq \frac{2g_3|r|(|r|-1)}{|r|(|r|-1)}$$
$$\frac{2g_3}{|r|-1} \leq g_1 \leq 2g_3$$

□

6 Discovery of AODs

In this section, we first present an AOD discovery algorithm that is suitable to both g_1 and g_3. We then study several optimizations and ranking functions to further improve the efficiency and effectiveness.

6.1 Algorithms for AOD Discovery

Algorithm. DisAOD (Algorithm 4) discovers the complete set Σ of minimal and valid AODs on a given instance r, with a given error measure function g (g_1 or g_3) and a threshold e. DisAOD traverses the search space of AODs by following a depth-first-search (DFS) strategy implemented by recursion. The AOD traversal is organized in a *forest*. Each tree is rooted at an AOD of the form $\overline{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$, *i.e.*, $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$ or $\overleftarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$, where $B \in R$ and $A \in R \backslash B$ (lines 2-3). Recall that it suffices to consider AODs with asc on the leftmost attribute on the RHS due to *symmetry* (Section 3).

For each candidate AOD X $\mapsto$Y, its error measure Vg and lower/upper bound LBg/UBg are computed with the given function g (line 8). If X $\mapsto$Y is valid, then we add it into Σ (lines 9-10). If the UBg is larger than the threshold e, then we further test X $\mapsto$Y$\overline{\mathsf{C}}$ (both X $\mapsto$Y$\overrightarrow{\mathsf{C}}$ and X $\mapsto$Y$\overleftarrow{\mathsf{C}}$) for all $C \in R \backslash$XY, by recursively calling function $Search$ (lines 11-14). As a prerequisite, we check whether Y$\overline{\mathsf{C}}$ is a minimal attribute list by calling function $MinimalAttributelist$ (line 13); candidates with non-minimal attribute lists are directly discarded. By definition (Section 4), it is to check whether $\mathcal{Y} \rightarrow C$. This is done by checking whether appending $\overline{C}$ to Y incurs any changes to τ_Y, *i.e.*, whether $\tau_{Y\overline{C}} = \tau_Y$. It is easy to prove that $\tau_{Y\overline{C}} = \tau_Y$ iff $\mathcal{Y} \rightarrow C$. If Y$\overline{\mathsf{C}}$ is found to be a minimal attribute list, then the checking incurs no extra cost since the computation of $\tau_{Y\overline{C}}$ is originally required. We will further develop optimization techniques for this in Section 6.2.

Algorithm 4: DisAOD

```
  Input: a relation $r$ of schema $R$, an error measure
         function $g$ and a threshold $e$
  Output: the complete set $\Sigma$ of minimal and valid AODs
1 $\Sigma \leftarrow \emptyset$;
2 foreach $B \in R$, $A \in R \backslash B$ do
3     Search($\overline{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$);
4 $\Sigma \leftarrow MinimalAOD(\Sigma)$;
5 return $\Sigma$;
6
7 Function Search(AOD candidate X $\mapsto$Y)
8 $Vg, LBg, UBg \leftarrow Compute$(X $\mapsto$Y, $g$, $r$);
9 if $Vg \leq e$ then
10    $\Sigma \leftarrow \Sigma \cup \{$X $\mapsto$Y $\}$;
11    if $UBg > e$ then
12        foreach $C \in R \backslash$XY do
13            if $MinimalAttributelist$(Y$\overline{\mathsf{C}}$) then
14                Search(X $\mapsto$Y$\overline{\mathsf{C}}$);
15    else
16        $\Sigma \leftarrow \Sigma \cup Extend$(X $\mapsto$Y);
17 else
18    if $LBg \leq e$ then
19        foreach $C \in R \backslash$XY do
20            if $MinimalAttributelist$(X$\overline{\mathsf{C}}$) then
21                Search(X$\overline{\mathsf{C}}$ $\mapsto$Y);
```

If the upper bound UBg is not larger than the threshold e, then AODs of the form X $\mapsto$YV are all valid. We generate minimal ones among them by calling function $Extend$ (line 16). Specifically, this is done by enumerating X $\mapsto$YW, where W is a list on *all* attributes in $R \backslash$XY (the others cannot be minimal) and YW is a minimal attribute list.

If X $\mapsto$Y is invalid and $LBg \leq e$, then we further consider X$\overline{\mathsf{C}}$ $\mapsto$Y (both X$\overrightarrow{\mathsf{C}}$ $\mapsto$Y and X$\overleftarrow{\mathsf{C}}$ $\mapsto$Y) for all $C \in R \backslash$XY, if X$\overline{\mathsf{C}}$ is a minimal attribute list (lines 17-21).

As the final step, we remove non-minimal AODs X $\mapsto$Y if there exists valid AOD X $\mapsto$YU (U is not empty) in Σ, by calling function $MinimalAOD$ (line 4). This is necessary by the definition of minimal AODs (Section 4).

Example 15: Consider $R(A, B, C, D)$. In the tree rooted at $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$, we consider candidates $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\overline{\mathsf{C}}$, $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\overline{\mathsf{D}}$ (resp. $\overrightarrow{\mathsf{A}}\overline{\mathsf{C}} \mapsto \overrightarrow{\mathsf{B}}$, $\overrightarrow{\mathsf{A}}\overline{\mathsf{D}} \mapsto \overrightarrow{\mathsf{B}}$), if $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$ is valid (resp. invalid) and the pruning by lower/upper bounds does not apply to $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$. Before computing the error measure of a candidate, say $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\overrightarrow{\mathsf{C}}$, we check whether $\overrightarrow{\mathsf{B}}\overrightarrow{\mathsf{C}}$ is a minimal attribute list by calling function $MinimalAttributelist$. It is to check whether $B \rightarrow C$. If (a) $\overrightarrow{\mathsf{B}}\overrightarrow{\mathsf{C}}$ is a minimal attribute list, and (b) $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\overrightarrow{\mathsf{C}}$ is valid and the upper bound of error measure is not larger than the threshold e, then AODs obtained by appending more attributes to $\overrightarrow{\mathsf{B}}\overrightarrow{\mathsf{C}}$ are all valid. Function $Extend$ is called to generate minimal ones among them.

Specifically, it enumerates candidates by appending all remaining attributes to the RHS, *e.g.*, $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}\,\overleftarrow{\mathsf{D}}$, and further checks whether $\overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}\,\overleftarrow{\mathsf{D}}$ is a minimal attribute list. If $\overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}\,\overleftarrow{\mathsf{D}}$ is minimal, then $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}\,\overleftarrow{\mathsf{D}}$ is a minimal valid AOD. □

Complexity. AODs concern lists of attributes. In the worst-case, the number of all candidate AODs is $O(|R|!)$, and it takes $O(|r|log(|r|))$ to check each candidate AOD, where $|R|$ (resp. $|r|$) is the number of attributes (resp. tuples). The actual complexity is usually much smaller (Section 8).

Remarks. We highlight the features of DisAOD.

(1) It enumerates all candidate AODs by traversing all trees rooted at $\overline{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$, where $B \in R$ and $A \in R \backslash B$.

(2) It employs pruning rules to remove non-minimal or invalid candidates. (a) It prunes XA ↦Y if X ↦Y is valid. (b) It prunes X ↦YY′ if X ↦Y is invalid. (c) It prunes non-minimal candidates due to non-minimal attribute lists. (d) It prunes non-minimal AOD X ↦Y if AOD X ↦YU is valid.

(3) DisAOD leverages the upper/lower bounds of g_1 and g_3. It employs the lower bound to discard candidates that cannot be valid, and the upper bound to fast generate minimal candidates. These bounds are unique to AODs and experimentally verified to be crucial to the efficiency.

(4) Besides g_1 and g_3, DisAOD is suitable to any error measure function g if g satisfies the criteria stated in Section 4. In case the lower/upper bounds are not available, we can set the lower (resp. upper) bound as 0 (resp. 1).

(5) Different from exact LOD discoveries [5], [17] that follow BFS traversal, DisAOD traverses the space of candidate AODs in DFS. This helps maintain sorted partitions in a more efficient way, resulting in small memory footprint.

6.2 Optimizations

In this subsection, we further develop several optimization techniques for DisAOD.

Incremental computations. Following X ↦Y, we consider new candidate $\mathsf{X}\overline{\mathsf{A}}$ ↦Y or X ↦$\mathsf{Y}\overline{\mathsf{A}}$. Leveraging results of X ↦Y, *incremental* computations not only apply to sorted partitions, but also to error measure functions. We present an "incremental" version of Algorithm 2 with better efficiency.

Recall that a swapped (resp. an ordered) tuple pair *w.r.t.* X ↦Y is still a swapped (resp. an ordered) pair *w.r.t.* $\mathsf{X}\overline{\mathsf{A}}$ ↦Y or X ↦$\mathsf{Y}\overline{\mathsf{A}}$. Hence, the number of swapped (resp. ordered) pairs monotonically increases. Without loss of generality, we consider $\mathsf{X}\overline{\mathsf{A}}$ ↦Y. We see the following.

Proposition 11. If $t \prec_{\mathsf{X}\overline{\mathsf{A}}} s$ and $t \not\prec_{\mathsf{X}} s$, then $t =_{\mathsf{X}} s$.

Proof: We prove this by contradiction. If $s \prec_{\mathsf{X}} t$, then we have $s \prec_{\mathsf{X}\overline{\mathsf{A}}} t$. □

Proposition 11 tells us that if tuples t, s form a new swapped or an ordered pair *w.r.t.* $\mathsf{X}\overline{\mathsf{A}}$ ↦Y, then t, s must have the same value on X, *i.e.*, in the same equivalence class of τ_X. Therefore, we can leverage τ_X that is already computed for X ↦Y, and cope with each equivalence class in τ_X separately, when computing the *incremental* swapped and ordered tuple pairs for $\mathsf{X}\overline{\mathsf{A}}$ ↦Y.

Algorithm. Algorithm 5 *incrementally* computes g_{swap} and the number of ordered tuple pairs for $\mathsf{X}\overline{\mathsf{A}}$ ↦Y, based on the known numbers of swapped and ordered tuple pairs for X ↦Y. As stated earlier, it handles each equivalence class of τ_X one by one (lines 10-21), in contrast to Algorithm 2 that deals with all equivalence classes of τ_X as a whole.

Algorithm 5: Incremental computation of g_{swap} and the number of ordered tuple pairs

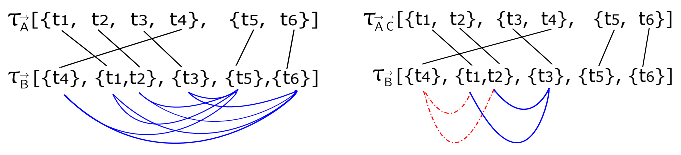
```
   Input: sorted partitions $\tau_X$,$\tau_Y$,$\tau_{\mathsf{X}\overline{\mathsf{A}}}$, the numbers of
          swapped and ordered tuple pairs for X ↦Y
   Output: $g_{\mathsf{swap}}$ and the number of ordered tuple pairs for
           $\mathsf{X}\overline{\mathsf{A}}$ ↦Y
 1 foreach i in $[1, |\tau_X|]$ do
 2     $rank[i] \leftarrow 0$;  $newrank[i] \leftarrow 0$;
 3 foreach equivalence class ecy in $\tau_Y$ do
 4     foreach tuple t in ecy do
 5         if $I_Y(t) > rank[I_X(t)]$ then
 6             $newrank[I_X(t)] \leftarrow newrank[I_X(t)] + 1$;
 7             $rank[I_X(t)] \leftarrow I_Y(t)$;
 8         $m[t] \leftarrow newrank[I_X(t)]$;
 9 swap, ordered ← the numbers of swapped and ordered
     pairs for X ↦Y;
10 foreach equivalence class ecx in $\tau_X$ do
11     $range \leftarrow 0$;
12     foreach tuple t in ecx do
13         $range \leftarrow max(range, m[t])$;
14     seg ← an empty segment tree on range $[1, range]$;
15     foreach equivalence class ecxa in $\tau_{\mathsf{X}\overline{\mathsf{A}}}$ from ecx do
16         foreach tuple t in ecxa do
17             $swap \leftarrow swap + seg.query([m[t]+1, range])$;
18             $ordered \leftarrow ordered + seg.query([1, m[t]-1])$;
19         foreach tuple t in ecxa do
20             $seg.insert(m[t])$;
21 return $g_{\mathsf{swap}} = \frac{swap}{|r|^2-|r|}$, ordered;
```

A segment tree is leveraged for each equivalence class ecx. Recall that the complexity of a segment tree is closely related to the range; it takes $O(n)$ to build and $O(log(n))$ to update and query a segment tree on [1, n]. We use the idea of *state compaction* to build each segment tree on a compact range. We obtain the *local* rank $m[t]$ of a tuple t in ecx based on the original rank $I_Y[t]$, which preserves the order. As an example, suppose there are 5 tuples in ecx whose $I_Y[t]$ values are 3, 5, 5, 1, 10 respectively, their ranks in ecx are 2, 3, 3, 1, 4, respectively.

More specifically, we convert the rank $I_Y[t]$ to $m[t]$ as follows (lines 1-8). We use two auxiliary arrays $rank$ and $newrank$ that are indexed by $I_X(t)$, *i.e.*, the τ_X rank of tuple t. For the ith equivalence class of τ_X, denoted by $\tau_X[i]$, we use $rank[i]$ to store the τ_Y rank of the last visited tuple from $\tau_X[i]$, and $newrank[i]$ to store the *local* τ_Y rank of the last visited tuple from $\tau_X[i]$. All tuples t are processed according to their orders in τ_Y (lines 3-4), and hence $I_Y(t)$ can never decrease during the processing. For t belonging to $\tau_X[i]$ ($I_X(t) = i$), $rank[I_X(t)]$ and $newrank[I_X(t)]$ are updated if t has a larger τ_Y rank than the last visited tuple (Line 5-7), and the *local* rank $m[t]$ is $newrank[I_X(t)]$ (line 8).

The range of the segment tree for the equivalence class ecx is set to be the max value of the new ranks, *i.e.*, the number of distinct $I_Y(t)$ values for all t from ecx (lines 11-14), which reduces from $|\tau_Y|$ to (at most) $|ecx|$ by the state compaction. Note that the sum of the ranges of all segment trees used in Algorithm 5 is at most the number $|r|$ of tuples.

Example 16: On Table 2, $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$ incurs no swapped but 8 ordered pairs (Figure 3a). Now consider $\overrightarrow{\mathsf{A}}\,\overrightarrow{\mathsf{C}} \mapsto \overrightarrow{\mathsf{B}}$. The equivalence class $\{t_1, t_2, t_3, t_4\}$ in $\tau_{\overrightarrow{A}}$ is divided into $\{t_1, t_2\}$


$\tau_{\vec{A}}$[{t1, t2, t3, t4}, {t5, t6}]
$\tau_{\vec{B}}$[{t4}, {t1,t2}, {t3}, {t5},{t6}]
$\tau_{\vec{A}\vec{C}}$[{t1, t2}, {t3, t4}, {t5, t6}]
$\tau_{\vec{B}}$[{t4}, {t1,t2}, {t3}, {t5}, {t6}]

(a) Existing results (b) Incremental results

Fig. 3. Example 16 for Algorithm 5

and $\{t_3, t_4\}$ in $\tau_{\overrightarrow{A}\overrightarrow{C}}$, which incurs two new ordered pairs (solid lines) and two new swapped pairs (dashed lines), shown in Figure 3b. $\square$

Time Complexity. The state compaction for segment trees (lines 1-8 and 11-13) is done in $O(|r|)$, so is the initialization of all segment trees (line 14). Algorithm 5 has the same worst-case complexity as Algorithm 2, but is experimentally verified to be much more efficient in practice.

Index for checking the attribute list minimality. The computation of $\tau_{Y\overline{C}}$ is required for checking whether appending $\overline{C}$ to Y leads to a non-minimal attribute list. To avoid some unnecessary computations, we employ an indexing structure on C, for fetching all $\mathcal{X}$ if we find $\mathcal{X} \to C$ in DisAOD. Specifically, we do the following when $\overline{C}$ is appended to Y.
(1) If $\mathcal{Y}$ is a *superset* of any $\mathcal{X}$ related to C in the index, then we know Y$\overline{\text{C}}$ is not a minimal attribute list.
(2) Otherwise, we compute $\tau_{Y\overline{C}}$. (a) If $\tau_{Y\overline{C}}$ = τ_Y, then Y$\overline{\text{C}}$ is not a minimal attribute list since $\mathcal{Y} \to C$. We update the index with $\mathcal{Y}$, and also remove any $\mathcal{X}$ related to the key C if $\mathcal{X}$ is a superset of $\mathcal{Y}$. (b) If $\tau_{Y\overline{C}} \neq \tau_Y$, then Y$\overline{\text{C}}$ is a minimal attribute list. We continue to the next step of DisAOD with the computed sorted partition $\tau_{Y\overline{C}}$.

Sorted partition cache. Sorted partitions are heavily used in AOD discovery. In the traversal, the same attribute list may occur multiple times (possibly) on different sides. For example, we may generate $\overrightarrow{\text{A}}\,\overrightarrow{\text{C}} \mapsto \overrightarrow{\text{B}}$ from $\overrightarrow{\text{A}} \mapsto \overrightarrow{\text{B}}$, $\overrightarrow{\text{A}}\,\overrightarrow{\text{C}} \mapsto \overrightarrow{\text{D}}$ from $\overrightarrow{\text{A}} \mapsto \overrightarrow{\text{D}}$, and $\overrightarrow{\text{B}} \mapsto \overrightarrow{\text{A}}\,\overrightarrow{\text{C}}$ from $\overrightarrow{\text{B}} \mapsto \overrightarrow{\text{A}}$, all with the list $\overrightarrow{\text{A}}\,\overrightarrow{\text{C}}$, and hence, the same sorted partition $\tau_{\overrightarrow{A}\overrightarrow{C}}$. DisAOD adopts a DFS traversal with a small memory footprint, which enables us to maintain a cache for the created sorted partitions. In addition to the sorted partitions necessary for the DFS traversal, we also use free memory to preserve more sorted partitions for possible reuse. We use a simple LRU (least recently used) strategy when the memory is used up.

6.3 Ranking Functions

The number of minimal valid AODs can be large on some instances. In this subsection, we present two functions to rank AODs, for helping users quickly select a small set of more relevant AODs. Specifically, (a) *ability to sort* is a new measure proposed for AODs, and (b) *succinctness* is a common measure that is adapted to AODs.

Ability to sort. The ability of a list X in terms of sorting can be judged by the sorted partition τ_X. We aim for τ_X that is of a large size (many distinct values on X) and doesn't have very large equivalence classes (tuple values on X are evenly distributed). Specifically, we measure the ability of X as:

$$sort(\text{X}) = \frac{\sum_{ec \in \tau_X} |ec|^2}{|r|^2}$$

In the equation, $|ec|$ denotes the number of tuples in the equivalence class ec of τ_X. It is easy to see that a small value is obtained if tuples are evenly distributed among many equivalence classes. For a LOD X $\mapsto$Y, we define its ability to sort as the average of the abilities of its LHS and RHS.

$$sort(\text{X} \mapsto \text{Y}) = avg(sort(\text{X}), sort(\text{Y}))$$

Succinctness. Based on the minimum description length principle, succinctness is a common criterion for ranking discovered dependencies [4], [23]. We define the succinctness of X $\mapsto$Y as follows and again prefer a small value.

$$succ(\text{X} \mapsto \text{Y}) = \frac{|X|+|Y|}{|R|}.$$

Note that appending attributes to the LHS (RHS) of an AOD harms the *succinctness*, but favors the *ability to sort*. As an add-on feature of AOD discovery, DisAOD can additionally sort all discovered AODs according to the two functions, for *top-k* AODs. We will experimentally study the effectiveness of ranking functions in Section 8.

7 Error Measure Estimation by Sampling

In this section, we first study a method to estimate error measure g_1 by sampling. Based on the method, we then propose an alternative approach to AOD discovery on a sample of the full instance.

It can be very costly to compute error measures on large instances. As explained in [15], [21] for approximate FD and DC discoveries, a good estimation of error measures suffices in many applications. In this paper we focus on the estimation of g_1 .

Estimation of g_1. Consider a LOD γ = X $\mapsto$Y on an instance r. We reorder tuples in r such that for two tuples $t_i, t_j \in r$, $t_i \preceq_{\text{X}} t_j$ if $i < j$. Recall that a tuple pair (t_i, t_j) may incur (a) a split only when $t_i =_{\text{X}} t_j$, or (b) a swap only when $t_i \prec_{\text{X}} t_j$.

Suppose the g_1 value of γ on r is known. When we randomly sample k tuple pairs (t_i, t_j) $(i < j)$ from r, it is to conduct Bernoulli experiments k times with a probability of g_1 each time, in terms of the number of violations (split or swap). Hence, we know that the number $|vio|$ of violations in the sampling follows the binomial distribution:

$$|vio| \sim B(k, g_1)$$

The g_1 can be easily estimated:

$$\hat{g_1} = \frac{|vio|}{k}$$

In practice we can use normal approximation to the binomial distribution, when k is big enough:

$$|vio| \sim N(kg_1, kg_1(1 - g_1))$$

Our aim is to determine the size k of the sampled tuple pairs, for a $\hat{g_1}$ close enough to its true value. Specifically, for a given error threshold e and two more parameters δ, α, we hope the probability that the distance between $\hat{g_1}$ and the true value g_1 is less than δe is at least $1 - \alpha$, *i.e.*,

$$Pr(|\hat{g_1} - g_1| < \delta e) \geqslant 1 - \alpha$$

By normal approximation, we have the following:

$$Pr(|\hat{g_1} - g_1| < \delta e) =$$

$$Pr(\frac{|\hat{g_1}-g_1|}{\sqrt{g_1(1-g_1)/k}} < \frac{\delta e}{\sqrt{g_1(1-g_1)/k}}) =$$
$$2\Phi(\frac{\delta e}{\sqrt{g_1(1-g_1)/k}}) - 1 \geqslant 1 - \alpha$$
$$\Phi(\frac{\delta e}{\sqrt{g_1(1-g_1)/k}}) \geqslant 1 - \frac{\alpha}{2}$$
$$\frac{\delta e}{\sqrt{g_1(1-g_1)/k}} \geqslant \mu_{1-\frac{\alpha}{2}}$$
$$k \geqslant \frac{\mu^2_{1-\frac{\alpha}{2}} g_1(1-g_1)}{\delta^2 e^2}$$

We mainly care about g_1 that is not greater than the given error threshold e and e is typically much smaller than 1. By replacing $(1-g_1)$ with 1 and g_1 with e respectively, we have

$$k \geqslant \frac{\mu^2_{1-\frac{\alpha}{2}}}{\delta^2 e} \quad (1)$$

The value of μ for a specific α in the equation can be obtained by referring to the standard normal distribution table. The theoretical result shows an estimation of high accuracy can be obtained with a k independent of the size $|r|$ of the instance r. For example, using k = 54,103, we have a probability of at least 98% that the gap between $\hat{g_1}$ and g_1 is less than 0.001, *i.e.,* in the setting of e = 0.01, δ = 0.1 and α = 0.02[1]. Note that a very small instance of 1,000 tuples already has 999,000/2 tuple pairs available. Better, the sampling strategy is widely applicable, since it is a uniform random sampling without any assumption on data distributions.

AOD discovery by sampling. We propose a lightweight approach to AOD discovery with g_1. Given an instance r, (1) we randomly choose k tuple pairs (k is determined according to Equation 1), and use all tuples from the pairs to form a sample r'. (2) We run DisAOD with g_1 on r'.

The goal is to perform AOD discovery on r' as a substitution for AOD discovery on r, with high accuracy and far less cost. Our experimental evaluations (Section 8) demonstrate that we can indeed achieve this goal.

8 Experimental Evaluations

In this section, we present an in-depth experimental study. Following the experimental settings, we conduct extensive experiments to (1) demonstrate the efficiency of AOD discoveries and optimization techniques, and to (2) verify the effectiveness of AOD discoveries, the ranking functions and the estimation of g_1 by sampling.

8.1 Experimental Settings.

Datasets. We use a set of real-life and synthetic data from http://metanome.de that are evaluated in OD discoveries [5], [11], [17], [28], [29]. (1) NCV, FLI, Hepa and Atom are real-life data, concerning voters, flights, hepatitis disease and atom sites, respectively. (2) DB, Letter and FDR are synthetic datasets with complicated attribute relationships.

We summarize datasets in Table 4, where $|r|$ denotes the number of tuples, and $|R|$ denotes the number of attributes.

Algorithms. We implement all the algorithms in Java. (1) AOD$_1$ and AOD$_3$, different versions of DisAOD for measures g_1 and g_3 respectively. (2) Some variants of DisAOD, for testing the effectiveness of optimization techniques and ranking functions (details are provided later). (3) FastOD, the algorithm for discovering approximate set-based canonical ODs with measure g_3 [12]. (4) SampleAOD$_1$, the method that performs AOD discovery with g_1 by sampling (Section 7). We provide all the tested datasets and code at https://github.com/chenjixuan20/AOD.

Parameter settings. In addition to $|r|$ and $|R|$, we use one more parameter: the error threshold e. We use random sampling (resp. projection) to vary $|r|$ (resp. $|R|$) when required.

Running environment. We run all experiments on a PC server with an Intel Xeon E-2224 3.4G CPU, 64GB of memory and CentOS 7, and report the average results of 5 runs.

8.2 Efficiency of AOD Discovery

Exp-1: AOD$_1$ and AOD$_3$ on all datasets. We summarize results of AOD$_1$ and AOD$_3$ on all tested data in Table 4, with running times (in seconds) and the number $|AOD|$ of discovered AODs. We use different threshold e for AOD$_1$ and AOD$_3$. Intuitively, an erroneous tuple incurs a g_3 value of $\frac{1}{|r|}$, but a single violation incurs a g_1 value of $\frac{1}{|r|^2-|r|}$. Note that the times of AOD$_1$ and AOD$_3$ cannot be compared, since the sets of discovered AODs by them are quite different.

We are not aware of any existing works on approximate LOD discovery. For reference, we show the experimental results of the state-of-the-art exact LOD discovery algorithm [11] (obtained from the authors). We see approximate discoveries are usually slower than the exact counterpart. This is reasonable since exact discovery is a special case of approximate one, and the method of [11] fully leverages the properties of exact LODs for an efficient pruning. Note that a valid exact LOD is always a valid AOD, but it may be a non-minimal AOD. Hence, the result of approximate discovery does not necessarily contain that of exact discovery. Indeed, the number of exact LODs can be much larger than that of AODs. For example, the exact discovery already finds more than 500K exact LODs on FLI when we manually terminate it. Intuitively, instead of discovering a proper AOD, exact discovery may find many specializations of it to resolve violations that may be actually incurred by errors in the data. We additionally provide results of the memory usage in Table 4. Except for FLI, exact discovery takes less memory on the tested datasets, as expected.

In the sequel we further compare AOD$_1$ (resp. AOD$_3$) against its variants, by varying parameters.

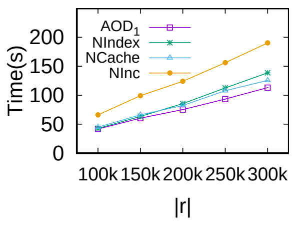
Exp-2: AOD$_1$ against Variants. We implement some variants by disabling an optimization each time (Section 6.2). (1) NInc disables the incremental computation of *swapped* and *ordered* pairs; (2) NIndex disables the index for minimality check; and (3) NCache disables the sorted partition cache. We also test a variant without leveraging lower/upper bounds; it is always orders of magnitude slower (not shown). The results show that pruning rules with bounds are crucial to the efficiency of AOD discovery. We only report the results on FLI, since the results on other datasets are similar.

We set $|r| = 300K$, $|R|$ = 12 and e = 0.001 by default on FLI, and vary $|r|$ from 100K to 300K in Figure 4a, $|R|$ from 8 to 12 in Figure 4b, and e from 0.001 to 0.003 in Figure 4c.

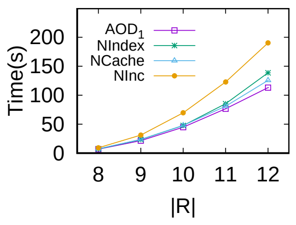
1. $\mu_{0.99}$ = 2.326 in the standard normal distribution table.

TABLE 4
Datasets, execution statistics of AOD_1 and AOD_3

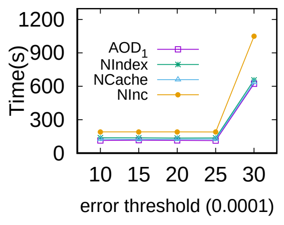
Dataset Properties			AOD_1 (e = 0.001)			AOD_3 (e = 0.01)			Exact LOD		
DataSet	$\|r\|$	$\|R\|$	Time(s)	$\|AOD\|$	Memory	Time(s)	$\|AOD\|$	Memory	Time(s)	$\|LOD\|$	Memory
NCV	930K	15	2,036	184	5,353MB	3,374	102	11,606MB	26	20	985MB
FLI	500K	17	60,684	76,288	11,038MB	4,142	386	5,667MB	$59,857^+$	$526,906^+$	$25,647^+$MB
Hepa	155	20	0.136	0	30MB	0.096	0	31MB	0.163	0	15MB
Atom	33K	10	2	48	100MB	5	126	302MB	0.485	108	253MB
DB	250K	13	21	10	1,705MB	57	2	2,989MB	3	2	224MB
Letter	20K	17	0.833	0	148MB	0.191	0	204MB	0.148	0	17MB
FDR	250K	30	110	10	3,719MB	1,200	40	5,313MB	47	1,917	3,384MB



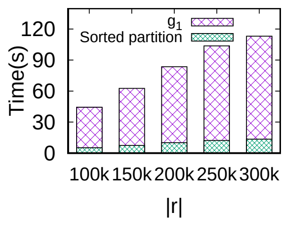
(a) FLI g_1: varying $|r|$



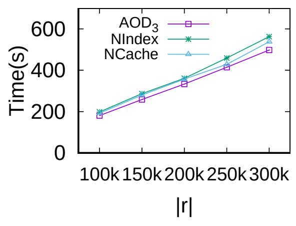
(b) FLI g_1: varying $|R|$



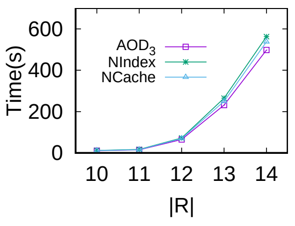
(c) FLI g_1: varying e



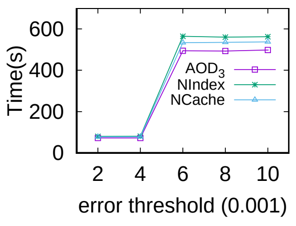
(d) FLI g_1: AOD_1 decomposition



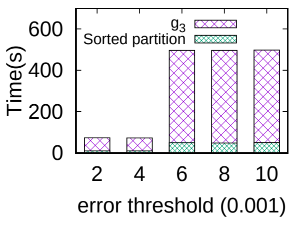
(e) FLI g_3: varying $|r|$



(f) FLI g_3: varying $|R|$



(g) FLI g_3: varying e



(h) FLI g_3: AOD_3 decomposition

Fig. 4. AOD_1, AOD_3 against Variants

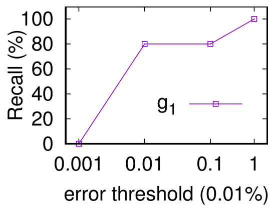
We see the following. (1) AOD_1 scales well with $|r|$, consistent with the complexity analysis. As $|r|$ increases from 100K to 300K, the time increases from 41s to 113s. (2) $|R|$ significantly affects the efficiency, as expected. We find the number of discovered AODs increases from 2 to 56 (not shown), as $|R|$ increases from 8 to 12. However, we see the performance is much better than the worst-case complexity that is factorial in $|R|$. Indeed, the running time is mostly affected by the size of the actual search space, *i.e.*, the number of AODs that are required to be checked (not pruned). Recall that function *Compute* (line 8 of DisAOD) is called to compute the error measures and lower/upper bounds of candidate AODs, and hence, we use the number of calling this function as the size of the actual search space. We experimentally find that only 232 (resp. 4,668) AODs are checked for $|R|$ = 8 (resp. 12). (3) The threshold e can significantly affect the results of AOD_1. Intuitively, a large e value enables AODs with more attributes on the RHS and in turn more attributes on the LHS, while a small e value usually leads to fewer attributes on the RHS. We find the number of discovered AODs almost remains unchanged when e is in the range of [0.001,0.0025], but sharply increases by more than 50 times as e increases to 0.003. The results also demonstrate the benefit of the upper bound. A relatively large e, *e.g.*, 0.003, helps AOD_1 quickly generate valid AODs. Hence, the time increases by less than 6 times as the number of valid AODs increases by more than 50 times.

In terms of the optimizations, we see the following. (1) AOD_1 is up to 1.7 times and on average 61% faster than NInc. The cost of AOD_1 mainly consists of the times for creating sorted partitions and for computing g_1 values. Along the same setting as Figure 4a, we show the two times respectively in Figure 4d. We find the latter governs the overall time, and hence the incremental computation of g_1 significantly improves the efficiency. (2) NIndex and NCache concern computations of sorted partitions. AOD_1 is on average 14% and 10% and up to 23% and 15% faster than NIndex and NCache, respectively.

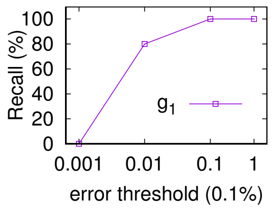
Exp-3: AOD_3 **against Variants.** We compare AOD_3 against variants (excluding NInc). The usage of lower/upper bounds is again experimentally found to be crucial (not shown).

We set $|r| = 300K$, $|R| = 14$ and e = 0.01 by default on FLI, and vary $|r|$ from 100K to 300K in Figure 4e, $|R|$ from 10 to 14 in Figure 4f, and e from 0.002 to 0.01 in Figure 4g.

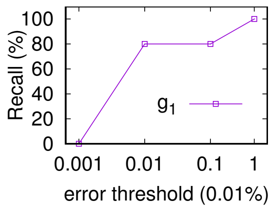
The results tell us the following. (1) AOD_3 scales well with $|r|$. (2) The efficiency of AOD_3 is sensitive to $|R|$. We find the number of discovered AODs increases from 12 to 142, and the size of the actual search space increases from 574 to 9,654 (not shown), as $|R|$ increases from 10 to 14. The actual search space is much smaller than the theoretical worst-case one. (3) The running time significantly increases from 73s to 495s on FLI when e increases from 0.004 to 0.006. This is because the number of discovered AODs increases from 20 to 142. (4) AOD_3 is on average 12% and 7% and up to 18% and 10% faster than NIndex and NCache, respectively. (5) The computation of g_3 on average takes more than 89% of the total time, as shown in Figure 4h (along the same setting as Figure 4g).



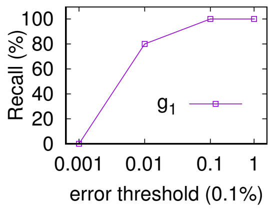
(a) NCV: (#1, 0.1%)



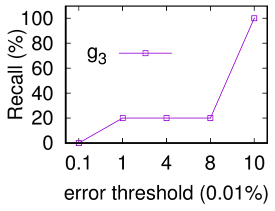
(b) NCV: (#1, 1%)



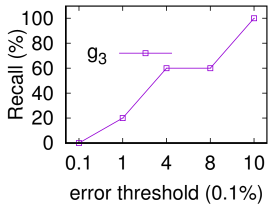
(c) NCV: (#2, 0.1%)



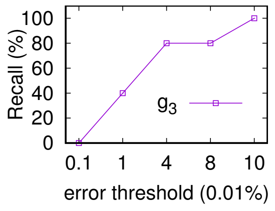
(d) NCV: (#2, 1%)



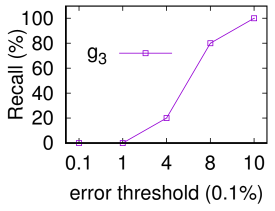
(e) NCV: (#1, 0.1%)



(f) NCV: (#1, 1%)



(g) NCV: (#2, 0.1%)



(h) NCV: (#2, 1%)

Fig. 5. Effectiveness of AOD discovery on NCV

TABLE 5
AOD_3 against FastOD on various datasets

Dataset Properties			AOD_3 ($e = 0.01$)		FastOD ($e = 0.01$)	
DataSet	$\|r\|$	$\|R\|$	Time(s)	$\|AOD\|$	Time(s)	$\|OD\|$
NCV	30K	15	109	138	650	245
FLI	50K	12	42	56	78	82
Hepa	155	20	0.096	0	198	6,862
DB	250K	13	57	2	333	785
FDR	250k	15	96	6	482	1,208

Exp-4: AOD_3 against FastOD. We compare AOD_3 against FastOD [12]. FastOD discovers approximate set-based canonical ODs with measure g_3. Recall that set-based ODs generalize LODs (Section 2), but this does not apply to AODs. Intuitively, a valid AOD does not guarantee that ODs from the corresponding set of canonical ODs are all valid approximate ones. Hence, the connection between approximate LODs and set-based canonical ODs is broken.

The results in Table 5 show AOD_3 is usually much faster than FastOD. We find this is mainly because the number of discovered approximate canonical ODs is typically much larger than that of AODs. Similar results are reported in the exact canonical OD and LOD discoveries [11], [29].

8.3 Effectiveness of AOD Discovery and Ranking

Exp-5: Recall of AOD discovery. In this set of experiments, we verify the effectiveness of AOD discovery by recalling LODs hidden in dirty data. We first manually identify some "golden" LODs verified by domain experts on dataset NCV. We then use 10K tuples of NCV and introduce noises to them; values close to the original correct ones are used as noises, which is common in practice. The introduction of noise is controlled by noise ratio θ and two strategies [21]. #1: on each attribute, each value has a probability of θ to be assigned a new value from the active domain. #2: each tuple has a probability of θ to be selected, and new values from the active domain are assigned to all values of selected tuples. Intuitively, #2 is a setting that favors g_3, since noises are on fewer tuples in #2 than #1.

We run DisAOD on the dirty dataset, and compute the *recall* as the ratio of the number of discovered golden LODs to the total number of golden LODs. From Figure 5a to 5h, we vary the error threshold e and test various settings (#i, θ). We see the following. (a) In contrast to exact LOD discoveries with a recall of 0 in all settings (not shown), both AOD_1 and AOD_3 have a recall of 100% when e is above a threshold. We denote this threshold by e_o. (b) In AOD_1, e_o is much smaller than the noise ratio θ, by up to orders of magnitude. We find NCV has sparse value distributions on some attributes, and hence the introduced new values lead to very few violations. (c) In AOD_3, e_o is very close to the noise ratio θ. This is expected in #2 (Figures 5g and 5h); g_3 concerns the number of violating tuples. We find e_o is also close to θ in #1 (Figures 5e and 5f). This is because there are very few violations in NCV and the number of violating tuples in #1 is similar to that in #2.

Exp-6: Ranking functions. Using FLI, we verify the effectiveness of ranking functions in identifying meaningful AODs. Besides the two ranking functions proposed in Section 6.3, we also rank AODs in terms of their error rates. All discovered AODs have error rates no larger than the given threshold, but their error rates are usually different. AODs with small error rates incur less violations. A discovered AOD is an exact LOD only when its error rate is 0.

We measure ranking functions in terms of *precision*, along the same line as [23]. Specifically, we use 10K tuples of FLI, perform AOD discovery with g_1, obtain top-10 AODs with each ranking function, and ask humans to judge whether the AODs are meaningful or not. The situation is practical, since it is typically much easier for humans to verify discovered dependencies than to manually design dependencies. The precision of each ranking function is the ratio of the number of meaningful AODs (judged by humans) to 10.

The results given in Table 6 show that all the three ranking functions can well help identify meaningful AODs. We find most meaningful AODs on FLI have short LHS and RHS attribute lists (some examples are given in Table 7 with their actual error rates). Note that only AOD discovery rather than exact LOD discovery can find meaningful AODs

TABLE 6
Precision of ranking functions on FLI

threshold e	ability to sort	succinctness	error rate
0.01	1	1	1
0.001	1	1	1
0.0001	0.8	0.8	0.6

TABLE 7
Example AODs on FLI

AOD	error rate
$\overrightarrow{\text{OriginAirportSeqID}}\mapsto\overrightarrow{\text{OriginAirportID}}$	0.000128
$\overrightarrow{\text{DestAirportSeqID}}\mapsto\overrightarrow{\text{DestAirportID}}$	0.0
$\overrightarrow{\text{Origin}}\mapsto\overrightarrow{\text{OriginAirportSeqID}}$	0.000024

when their error rates are not 0.

We test different thresholds. When 0.0001 is used as the error threshold, the precisions of all ranking functions decrease. We find this is because some too "specialized" AODs are discovered instead of meaningful AODs that do not satisfy the threshold. For example, $\overrightarrow{\text{OriginAirportSeqID}}\overrightarrow{\text{FlightDate}}\mapsto\overrightarrow{\text{OriginAirportID}}$ instead of $\overrightarrow{\text{OriginAirportSeqID}}\mapsto\overrightarrow{\text{OriginAirportID}}$ is discovered, but the former one is rejected by human verification. This negatively affects the precisions of ranking functions, since rankings are applied to the discovered AODs. Intuitively, we find that a small set of AODs can be inspected by users to help select a proper threshold.

8.4 Effectiveness of Estimation by Sampling

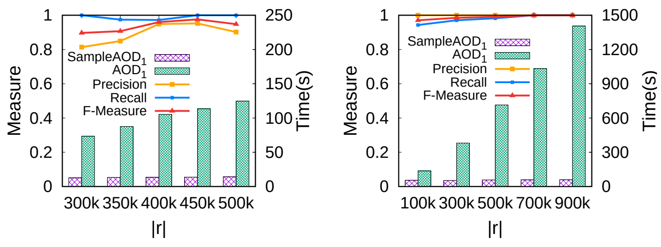
Exp-7: AOD discovery by sampling. We experimentally study SampleAOD$_1$ on FLI ($|R| = 11$ and e = 0.01) and NCV ($|R| = 14$ and e = 0.01) in Figures 6a and 6b, respectively. We also give the running time of AOD$_1$ for comparison. To avoid the bias of random sampling, we report the average of 10 runs. In this set of experiments, the *precision* (P) is the proportion of AODs discovered by SampleAOD$_1$ that are also discovered by AOD$_1$, the *recall* (R) is the proportion of AODs discovered by AOD$_1$ that are also discovered by SampleAOD$_1$, and the *F-measure* (F) is $2 \cdot (P \cdot R)/(P + R)$.

We see the following. (1) SampleAOD$_1$ always achieves high *precision*, *recall* and *F-measure*, *i.e.*, $P \geq 0.81$, $R \geq 0.97$ and $F \geq 0.90$ on FLI and $P \geq 0.99$, $R \geq 0.94$ and $F \geq 0.97$ on NCV. The values of P, R and F are not affected by the increase of $|r|$, consistent with our theoretical analysis. (2) The running time of SampleAOD$_1$ almost remains unchanged on each dataset, since the sample size is independent of $|r|$. Hence, SampleAOD$_1$ is much faster than AOD$_1$, up to 8.8 times on FLI and 24.7 times on NCV.

The results demonstrate that SampleAOD$_1$ can deliver results of high accuracy in far less time.

9 Conclusions

We have formalized the AOD discovery problem, developed efficient algorithms and optimizations for measures g_1, g_3, related lower/upper bounds and AOD discoveries. We have theoretically analyzed the properties of measures g_1, g_3 and stated their relationship. We have studied techniques for estimating g_1 by sampling. We have also experimentally verified the benefits of our methods.



(a) FLI: varying $|r|$ (b) NCV: varying $|r|$

Fig. 6. SampleAOD$_1$ against AOD$_1$

We intend to further study the workflow of AOD discovery. There are some technical issues that need to be addressed, *e.g.*, the method to help users determine and adjust the error threshold e when required. We also intend to further study the complexity in the framework of *fixed-parameter tractable*, along the same lines as [3].

Acknowledgments

This work is supported in part by National Key R&D Program of China 2018YFB1700403, NSFC 62172102 & 61572135 & 61925203.

References

[1] Z. Abedjan, L. Golab, and F. Naumann. Data profiling: A tutorial. In *SIGMOD 2017*, pages 1747–1751, 2017.
[2] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.
[3] T. Bläsius, T. Friedrich, and M. Schirneck. The parameterized complexity of dependency detection in relational databases. In *IPEC*, volume 63, pages 6:1–6:13, 2016.
[4] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.
[5] C. Consonni, P. Sottovia, A. Montresor, and Y. Velegrakis. Discovering order dependencies through order compatibility. In *EDBT*, pages 409–420, 2019.
[6] M. L. Fredman. On computing the length of longest increasing subsequences. *Discret. Math.*, 11(1):29–35, 1975.
[7] S. Ginsburg and R. Hull. Order dependency in the relational model. *Theor. Comput. Sci.*, 26:149–195, 1983.
[8] S. Ginsburg and R. Hull. Sort sets in the relational model. *J. ACM*, 33(3):465–488, 1986.
[9] L. Golab, H. J. Karloff, F. Korn, A. Saha, and D. Srivastava. Sequential dependencies. *PVLDB*, 2(1):574–585, 2009.
[10] Y. Jin, Z. Tan, W. Zeng, and S. Ma. Approximate order dependency discovery. In *ICDE*, pages 25–36, 2021.
[11] Y. Jin, L. Zhu, and Z. Tan. Efficient bidirectional order dependency discovery. In *ICDE*, pages 61–72, 2020.
[12] R. Karegar, P. Godfrey, L. Golab, M. Kargar, D. Srivastava, and J. Szlichta. Efficient discovery of approximate order dependencies. In *EDBT*, pages 427–432, 2021.
[13] B. Kenig, P. Mundra, G. Prasaad, B. Salimi, and D. Suciu. Mining approximate acyclic schemes from relations. In *SIGMOD*, pages 297–312, 2020.
[14] B. Kenig and D. Suciu. Integrity constraints revisited: From exact to approximate implication. In *ICDT*, pages 18:1–18:20, 2020.
[15] J. Kivinen and H. Mannila. Approximate dependency inference from relations. In *ICDT*, pages 86–98, 1992.
[16] S. Kruse and F. Naumann. Efficient discovery of approximate dependencies. *PVLDB*, 11(7):759–772, 2018.
[17] P. Langer and F. Naumann. Efficient order dependency detection. *VLDB J.*, 25(2):223–241, 2016.
[18] P. Li, M. H. Böhlen, J. Szlichta, and D. Srivastava. Discovery of band order dependencies. *CoRR*, abs/1905.11948, 2019.
[19] P. Li, J. Szlichta, M. H. Böhlen, and D. Srivastava. Discovering band order dependencies. In *ICDE*, pages 1878–1881, 2020.

[20] D. Liben-Nowell, E. Vee, and A. Zhu. Finding longest increasing and common subsequences in streaming data. *J. Comb. Optim.*, 11(2):155–175, 2006.

[21] E. Livshits, A. Heidari, I. F. Ilyas, and B. Kimelfeld. Approximate denial constraints. *PVLDB*, 13(10):1682–1695, 2020.

[22] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *SIGMOD*, pages 821–833, 2016.

[23] E. H. M. Pena, E. C. de Almeida, and F. Naumann. Discovery of approximate (and exact) denial constraints. *PVLDB*, 13(3):266–278, 2019.

[24] J. Rammelaere and F. Geerts. Revisiting conditional functional dependency discovery: Splitting the "c" from the "fd". In *ECML PKDD*, pages 552–568, 2018.

[25] D. E. Simmen, E. J. Shekita, and T. Malkemus. Fundamental techniques for order optimization. In *SIGMOD*, pages 57–67, 1996.

[26] S. Song, L. Chen, and P. S. Yu. Comparable dependencies over heterogeneous data. *VLDB J.*, 22(2):253–274, 2013.

[27] S. Song, F. Gao, R. Huang, and C. Wang. Data dependencies extended for variety and veracity: A family tree. *IEEE Trans. Knowl. Data Eng. Accepted for publication. doi: 10.1109/TKDE.2020.3046443.*

[28] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. Effective and complete discovery of order dependencies via set-based axiomatization. *PVLDB*, 10(7):721–732, 2017.

[29] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. Effective and complete discovery of bidirectional order dependencies via set-based axioms. *VLDB J.*, 27(4):573–591, 2018.

[30] J. Szlichta, P. Godfrey, and J. Gryz. Fundamentals of order dependencies. *PVLDB*, 5(11):1220–1231, 2012.

[31] J. Szlichta, P. Godfrey, J. Gryz, W. Ma, W. Qiu, and C. Zuzarte. Business-intelligence queries with order dependencies in DB2. In *EDBT*, pages 750–761, 2014.

[32] J. Szlichta, P. Godfrey, J. Gryz, and C. Zuzarte. Expressiveness and complexity of order dependencies. *PVLDB*, 6(14):1858–1869, 2013.

[33] Z. Tan, A. Ran, S. Ma, and S. Qin. Fast incremental discovery of pointwise order dependencies. *PVLDB*, 13(10):1669–1681, 2020.

PLACE
PHOTO
HERE

Yifeng Jin received the master degree in computer science from Fudan University, in 2021. He is currently a software engineer in Alibaba Group. His current research interests include databases on modern hardware and distributed computations.

PLACE
PHOTO
HERE

Zijing Tan is an associate professor in the School of Computer Science, Fudan University, China. He obtained his PhD degree from Fudan University, and was a visiting researcher of University of Edinburgh. His current research interests include data profiling and data quality management.

PLACE
PHOTO
HERE

Jixuan Chen is a master student in the School of Computer Science, Fudan University, China. He obtained his BS degree in software engineering from Huazhong University of Science and Technology, in 2020. His research focuses on data profiling and distributed computations.

PLACE
PHOTO
HERE

Shuai Ma is a professor at the School of Computer Science and Engineering, Beihang University, China. He obtained his PhD degrees from University of Edinburgh in 2011, and from Peking University in 2004, respectively. He is a recipient of the best paper award for VLDB 2010. He is an Associate Editor of VLDB Journal since 2017 and IEEE Transactions On Big Data and Knowledge and Information Systems since 2020. His current research interests include database theory and systems, and big data.