Approximate Order Dependency Discovery

Yifeng Jin [1,2] Zijing Tan [1,2] † Weijun Zeng [1,2] Shuai Ma [3,4]

[1]*School of Computer Science, Fudan University, Shanghai, China* [2]*Shanghai Key Laboratory of Data Science*
[3]*SKLSDE Lab, Beihang University, China* [4]*Beijing Advanced Innovation Center for Big Data and Brain Computing*

***Abstract*—Lexicographical order dependencies (ODs) specify orders between list of attributes, and are proven useful in optimizing SQL queries with *order by* clauses. To find hidden ODs from dirty data in practice, in this paper we make a first effort to study the approximate OD discovery problem, aiming at automatically discovering ODs that hold on the data with some exceptions. (1) We adapt two error measures to ODs, prove their desirable properties, and present efficient algorithms for computing the measures and related lower and upper bounds. (2) We present an efficient approximate OD discovery algorithm that is well suited to the two error measures, with a set of pruning rules and optimization techniques. (3) We conduct extensive experiments to verify the effectiveness and scalability of our methods, using real-life and synthetic data.**

***Index Terms*—Algorithms; Data profiling; Data dependency**

I. Introduction

Lexicographical order dependencies (ODs) are proposed in [26], [28], 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) [3], ODs are defined on *lists* of attributes. ODs lend themselves to wide applicability, since sorting is one of the most important database operations. We first give an example to illustrate the features of ODs.

Example 1: Consider the relation instance r in Table I, about 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 $\overrightarrow{\mathsf{Salary}} \mapsto \overrightarrow{\mathsf{Level}}\overrightarrow{\mathsf{Year}}$ in the notation of OD [26], [28]: 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 OD is consistent with the *SQL* clause ORDER BY $Salary$ ASC (resp. $Level$ ASC, $Year$ ASC).

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

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

(2) The attributes in the LHS and RHS list usually *cannot* be separated. For example, $\overrightarrow{\mathsf{Salary}} \mapsto \overrightarrow{\mathsf{Year}}$ 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 [1] for a survey. Recently, OD discoveries have received an increasing attention [4], [9], [14].

† Zijing Tan is the corresponding author.

TABLE I
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

The OD discovery problem is necessarily very difficult: it has a search space (the total number of candidate ODs) factorial in the number of attributes since ODs concern lists of attributes.

Worse, 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* [13], [19]. 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 the data. When it comes to OD discovery, the problem is even more involved. ODs 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. Interestingly, the following example shows that a single OD may overfit on the LHS, and simultaneously, underfit on the RHS.

Example 2: Recall Table I. 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 OD discovery algorithm may find $\overrightarrow{\mathsf{Salary}}\overrightarrow{\mathsf{Age}} \mapsto \overrightarrow{\mathsf{Level}}$ that holds on the dirty data. Compared with $\overrightarrow{\mathsf{Salary}} \mapsto \overrightarrow{\mathsf{Level}}\overrightarrow{\mathsf{Year}}$, 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 the 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.

This increasing complexity is well demonstrated in recent studies for approximate FDs [13] and DCs [17], [19].

Contributions. In this paper, we make a first effort to study the problem of approximate OD discovery.

(1) We adapt two error measures to approximate ODs. For an OD, 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 OD is satisfied. We show both measures have desirable properties and can be efficiently computed. We also study the lower and upper bounds for these measures, to enable pruning in approximate OD discovery (Section V).

(2) We provide an algorithm for discovering the complete set of minimal and valid approximate ODs. It traverses the search space of approximate ODs, computes the error measures and related lower/upper bounds of candidate ODs, and employs a host of pruning rules and optimization techniques for improving efficiency (Section VI).

(3) Using a host of real-life and synthetic data, we conduct extensive experiments to verify our approach. The results show the effectiveness of approximate OD discovery in recalling ODs from dirty data, and the effectiveness of our pruning rules and optimization techniques. (Section VII).

II. Related Work

Theoretical Foundations of Order Dependencies. Unidirectional and bidirectional lexicographical ODs are proposed in [26], [28], which are proven useful in optimizing queries with *order-by* clauses [27]. Different from these list-based lexicographical ODs, two classes of *set-based* order dependencies are also discussed. Set-based *canonical* ODs are proposed in [24], [25], and it is proven that they generalize lexicographical ODs. Another set-based ODs [6], [7], known as *pointwise* ODs, further generalize canonical ODs. There is no one-to-one relationship between a list-based OD and a set-based OD, and to our best knowledge, no techniques exists for transforming set-based ODs to list-based ODs.

In this paper, we consider lexicographical bidirectional ODs that 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 [4], [9], [14] for list-based (bidirectional) ODs, and in [24], [29] for set-based ODs, aiming to automatically find ODs that hold on the data without exceptions.

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

It is highly non-trivial, if not impossible, to extend exact OD discovery techniques to approximate ODs. [4] is based on the observation that each OD can be divided into an FD and an *order compatibility dependency* (OCD) (refer to Section V), and the OD holds iff both FD and OCD hold. This does not apply to approximate ODs. Intuitively, we cannot have the "embedded" FD and OCD in an OD both hold with exceptions if the OD holds with exceptions. For example, an approximate FD and an exact OCD may also form an approximate OD.

[9] is experimentally verified to be very efficient, by discovering ODs on a small sample (subset) data first, and then refining ODs on full data in an iterative way. The rationale behind [9] is that any exact OD that holds on data must hold on any subset of it. This does not apply to approximate ODs. Indeed, the error rate of an OD may increase after removing some tuples from data, and hence an approximate OD that holds on data may not hold on subsets of it.

Approximate Dependency Discoveries. To cope with dirty data in practice, approximate dependency discoveries are studied for *e.g.,* FDs [13], CFDs [21], DCs [17], [19] and set-based canonical ODs [25]. Based on definition of approximation that is given by using notions from information theory, [11] studies implication for approximate dependencies, and discoveries of approximate multi-valued dependencies and then acyclic schemes are investigated in [10].

In this paper, we study approximate OD discovery. The computation of error measures significantly depends on the dependency types, and a completely different strategy is required for generating candidate approximate list-based ODs compared with the other set-based dependencies.

A different notion, referred to as *approximate band conditional* OD, is proposed in [15]. Band ODs relax themselves to hold approximately with some exceptions and conditionally on subsets of data. Different from our work, [15] does not study how to discover approximate ODs. The method in [15] is complementary to ours. For an approximate OD discovered by us, [15] can be employed to split the data instance into contiguous segments such that the OD holds on each segment.

III. Preliminaries

In this section, we review basic notations and the definition of bidirectional lexicographical ODs [25], [26], [28].

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 $\mathsf{X} = [\overline{A_1}, \ldots, \overline{A_k}]$, $prefix(\mathsf{X})$ 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 $\mathsf{X} = [\overline{A_1}, \ldots, \overline{A_k}]$, we write $t \preceq_{\mathsf{X}} s$ iff (a) X = [ ]; or (b) $t \prec_{\overline{A_1}} s$; or (c) $\mathsf{X} = [\overline{A_1}\ |\mathsf{Y}\ ]$ 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 Order Dependency [28]. Given two lists X, Y, $\gamma = \mathsf{X} \mapsto \mathsf{Y}$ denotes a *bidirectional order dependency*. 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.

Example 3: If $\mathsf{X} \mapsto \mathsf{Y}$ holds, then we know tuples are ordered by Y if they are ordered by X, both in lexicographical ordering. ODs in Example 1 and Example 2 satisfy the definition. □

Remarks. (1) Along the same setting as [9], [14], in the sequel we consider *completely non-trivial* ODs whose LHS and RHS attribute lists (neglecting direction) are disjoint.

(2) Each OD γ has a *symmetry* OD γ' by reversing all directions [9]. As an example, we can see that $\overleftarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}$ is the symmetry of $\overrightarrow{\mathsf{A}} \mapsto \overleftarrow{\mathsf{B}}\,\overleftarrow{\mathsf{C}}$: $\overleftarrow{\mathsf{A}}$ is the reverse order of $\overrightarrow{\mathsf{A}}$, $\overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}$ is the reverse order of $\overleftarrow{\mathsf{B}}\,\overleftarrow{\mathsf{C}}$, 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. Without loss of generality, in the sequel we only consider ODs with asc on the leftmost attribute in the RHS attribute list, *e.g.,* $\overleftarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}$.

IV. Problem Formulation

In this section we present the definition of approximate ODs, and formalize the approximate OD discovery problem.

Error measures. We use a function g to measure the errors of ODs. Specifically, $g(\gamma, r)$ returns a value by taking as inputs an OD γ and a relation instance r. The smaller g value is, the fewer errors *w.r.t.* γ are on r. Here we present four criteria for judging whether an error measure function g makes sense, and will study the details of error measures in Section V.

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

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

(3) $g(\mathsf{X} \mapsto \mathsf{Y}, r) \leq g(\mathsf{X} \mapsto \mathsf{YY'}, r)$: 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 ODs. It is proven in [26], [28] 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 OD. Given an error measure function g and an error threshold e, we say that an OD γ is an *approximate* OD (abbreviated as AOD) valid on r iff $g(\gamma, r) \leq e$.

It is usually better to discover *minimal* valid dependencies rather than all valid ones, for a more concise result set without losing informative ones [3], [14], [18]. 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 [22], we have the following result, and hence the definition of minimal attribute list.

Proposition 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 $\mathcal{Y} \rightarrow B$ holds, then $t =_{\mathsf{X}\overline{\mathsf{B}}} s$ if t=$_{\mathsf{X}}s$.

Minimal Attribute List. We say 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 $\mathcal{Y} \rightarrow B$ holds.

Example 4: If $AB \rightarrow C$ 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 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 $g(\mathsf{XA} \mapsto \mathsf{Y}, r) \leq g(\mathsf{X} \mapsto \mathsf{Y}, r)$ and $g(\mathsf{X} \mapsto \mathsf{Y}, r) \leq g(\mathsf{X} \mapsto \mathsf{YY'}, r)$. Hence, on any instance r, we know $\mathsf{XA} \mapsto \mathsf{Y}$ 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.

Putting together the observations, we 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 $\mathsf{Y'}$, $\mathsf{X} \mapsto \mathsf{YY'}$ is not a valid AOD.

Example 5: If $\overleftarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}$ is valid, then $\overleftarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$ is not minimal. In this case the *minimality* does not concern fewer attributes. $\overleftarrow{\mathsf{A}}\,\overrightarrow{\mathsf{C}} \mapsto \overrightarrow{\mathsf{B}}$ 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. □

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.

V. Error Measures for Approximate ODs

In this section, we adapt two error measures to AODs. We show they satisfy the four criteria stated in Section IV, by providing theoretical results and efficient algorithms. We also study the lower and upper bounds for these measures.

A. The Percentage of Violating Tuple Pairs

The most common error measure, referred to as g_1, is introduced for FDs [12], [13], and further extended to *e.g.,* DCs [3], [19]. The computation of g_1 is closely related to violations of a dependency. We review violations of ODs first.

OD violations [26], [28]. Violations of an OD $\gamma = \mathsf{X} \mapsto \mathsf{Y}$ 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 $t \prec_{\mathsf{X}} s$, $s \prec_{\mathsf{Y}} t$.

Indeed, $\mathsf{X} \mapsto \mathsf{Y}$ has an "embedded" FD $\mathcal{X} \rightarrow \mathcal{Y}$, and (t, s) incurs a split *w.r.t.* $\mathsf{X} \mapsto \mathsf{Y}$ 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) [4].

TABLE II
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 6: 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 for ODs. The g_1 is measured as the ratio of the number of violating tuple pairs to the total tuple pairs [12]. By considering split and swap, we extend g_1 to ODs.

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

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

$$g_1(\mathsf{X} \mapsto \mathsf{Y},\, r) = g_{\mathsf{split}}(\mathsf{X} \mapsto \mathsf{Y},\, r) + 2 \times g_{\mathsf{swap}}(\mathsf{X} \mapsto \mathsf{Y},\, r)$$

Observe that we need to scale up the second number 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.

It is easy to see 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 IV.

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

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

Sorted Partition. The data structure, referred to as *sorted partition*, is employed in exact OD discoveries [9], [14]. 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.

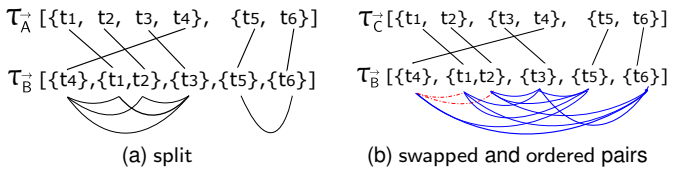


Fig. 1. Example 8 for Algorithm 1 and Example 10 for Algorithm 2

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

Input: sorted partitions τ_X,τ_Y
Output: $g_{\mathsf{split}}(\mathsf{X} \mapsto \mathsf{Y},\, r)$

```
1 spl ← 0;
2 foreach equivalence class ec in τ_X do
3     map ← an empty hash table;
4     foreach tuple t in ec do
5         map[I_Y[t]] ← map[I_Y[t]] + 1;
6     foreach v in the value set of map do
7         spl ← spl + v × (|ec| − v);
8 return g_split = spl / (|r|²−|r|);
```

Example 7: Consider Table II. $\tau_{\overrightarrow{A}} = [\{t_1, t_2, t_3, t_4\}, \{t_5, t_6\}]$, $\tau_{\overrightarrow{B}} = [\{t_4\}, \{t_1, t_2\}, \{t_3\}, \{t_5\}, \{t_6\}]$. $I_{\overrightarrow{A}}[t_3] = 1$; 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 may only lead to a swap. We hence present two algorithms for computing g_{split} and g_{swap} respectively.

Algorithm. Algorithm 1 is provided for computing g_{split}. Recall that a tuple pair (t, s) incurs a split *w.r.t.* $\mathsf{X} \mapsto \mathsf{Y}$, if t, s have the same value 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.,* 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 8: Consider $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$ on Table II. We show the related split violations in Figure 1a. (1) In the first equivalence class $\{t_1, t_2, t_3, t_4\}$ of $\tau_{\overrightarrow{A}}$, $I_{\overrightarrow{B}}[t_1] = I_{\overrightarrow{B}}[t_2] = 2$, $I_{\overrightarrow{B}}[t_3] = 3$, and $I_{\overrightarrow{B}}[t_4] = 1$. 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 $\tau_{\overrightarrow{A}}$. □

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 VI-A, when traversing the search space of candidate AODs, new candidates following $\mathsf{X} \mapsto \mathsf{Y}$ are in the form of $\mathsf{XU} \mapsto \mathsf{YV}$ (U or V can be empty). It is beneficial if we can obtain a lower bound and an upper bound of g_1 for $\mathsf{XU} \mapsto \mathsf{YV}$. This is because (1) $\mathsf{XU} \mapsto \mathsf{YV}$ *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) $\mathsf{XU} \mapsto \mathsf{YV}$ 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.

(1) If (t, s) incurs a swap *w.r.t.* $\mathsf{X} \mapsto \mathsf{Y}$, then (t, s) also incurs a swap *w.r.t.* $\mathsf{XU} \mapsto \mathsf{YV}$ (U or V can be empty) [14], [28];

Algorithm 2: Compute g_{swap} and # of ordered pairs

Input: sorted partitions τ_X and τ_Y
Output: g_{swap} and # of ordered pairs for $\mathsf{X} \mapsto \mathsf{Y}$

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* τ_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** $g_{\mathsf{swap}} = \frac{swap}{|r|^2-|r|}$, $ordered$;

(2) If $t \prec_{\mathsf{X}} s$ and $t \prec_{\mathsf{Y}} s$, then (t, s) never incurs OD violations *w.r.t.* $\mathsf{XU} \mapsto \mathsf{YV}$.

We refer to tuple pairs in (2) as *ordered pairs*, in contrast to *swapped pairs* in (1). Intuitively, when appending more attributes to the LHS and (or) RHS attribute list, (1) states that swap violations *can never* be resolved; and (2) shows that ordered pairs still *never* lead to violations. We hence compute the bounds as follows:

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

We show both bounds are *tight* with the following example.

Example 9: Recall Table II. For $\overrightarrow{F} \mapsto \overrightarrow{H}$, there are 5 swapped pairs and 8 ordered pairs, so 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 $O(n)$. The *ranks* of tuples in a sorted partition are well suited for transforming operations on the sorted partition to a segment tree.

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). Note that a larger (resp. smaller) rank in τ_Y implies a larger (resp. smaller) value on Y.

Example 10: Consider $\overrightarrow{\mathsf{C}} \mapsto \overrightarrow{\mathsf{B}}$ on Table II (shown in Figure 1b). (1) The first equivalence class $\{t_1, t_2\}$ of $\tau_{\overrightarrow{C}}$ 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 $\tau_{\overrightarrow{B}}$ is 2. (2) In the second equivalence class of $\tau_{\overrightarrow{C}}$, t_3 leads to 2 ordered pairs (solid lines) since the rank of t_1, t_2 is smaller than $I_{\overrightarrow{B}}[t_3]$ = 3, while t_4 incurs 2 swapped pairs (dashed lines) since the rank of t_1, t_2 is larger than $I_{\overrightarrow{B}}[t_4]$ = 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 $\tau_{\overrightarrow{C}}$ 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 $O(|r|log(|r|))$.

B. The Minimum Number of Removed Tuples

Another error measure function, referred to as g_3 in literature, is also originally introduced for FDs [12]. This g_3 measure is further extended to CFDs [20], set-based canonical ODs [25], comparable dependencies [23] and DCs [17], among others. The computation of g_3 can be very expensive. For example, it is quadratic in the number of tuples to compute g_3 for set-based canonical ODs [25], and even becomes *NP-Complete* for comparable dependencies and DCs.

Error measure g_3 for ODs. 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, $g_3(\gamma, r) \in [0, 1]$, 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 IV. We present one more definition to facilitate our approach.

OD sequence. For an OD $\gamma = \mathsf{X} \mapsto \mathsf{Y}$, 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 violation. We say an OD sequence is a *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 value).

Example 11: Consider Table III. For $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{C}}$, we have a LOS $[t_1, t_2, t_5, t_6]$. LOS may not be unique, *e.g.,* $[t_3, t_4, t_7, t_8]$ or $[t_1, t_2, t_7, t_8]$ is also a LOS. □

The following proposition tells us that g_3 can be readily computed from LOS.

Proposition 3: $g_3(\gamma, r) = 1 - \frac{|\mathsf{LOS}(\gamma, r)|}{|r|}$.

Proof sketch: g_3 is computed based on a maximum subset of r that satisfies γ, say r'. We can order all tuples in r' to form an OD sequence, and the sequence is also the longest. ■

TABLE III
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

Remark. LOS differs from *the longest increasing sequence* in *e.g.,* [5], [8], [16]. This is because OD violations consist of both swap and split. It is required in an OD sequence that (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.

The following proposition shows the monotonicity of g_3.

Proposition 4: (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 sketch: We prove (1) by showing that $\mathsf{LOS}(\mathsf{X} \mapsto \mathsf{Y}, r)$ can always be transformed into an OD sequence (not necessarily the longest) for $\mathsf{XA} \mapsto \mathsf{Y}$, and prove (2) by showing that $\mathsf{LOS}(\mathsf{X} \mapsto \mathsf{YY}', r)$ is also an OD sequence (not necessarily the longest) for $\mathsf{X} \mapsto \mathsf{Y}$. ■

Example 12: Recall Table III. (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 (not LOS) for $\overrightarrow{\mathsf{A}}\,\overrightarrow{\mathsf{B}} \mapsto \overrightarrow{\mathsf{C}}$. 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 (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 $\mathsf{XU} \mapsto \mathsf{YV}$. We find that they can be computed with two other sequences that slightly differ from LOS.

Strict increasing sequence and non-decreasing sequence. For $\gamma = \mathsf{X} \mapsto \mathsf{Y}$, (1) a *strict increasing sequence* on r is a list of tuples t_{a_1}, t_{a_2}, ..., 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) $t_{a_i} \prec_{\mathsf{Y}} t_{a_j}$. A *longest strict increasing sequence*, denoted by $\mathsf{LSIS}(\gamma, r)$, is a strict increasing sequence with a maximum k. (2) A *non-decreasing sequence* is a list of tuples t_{a_1}, t_{a_2}, ..., 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) $t_{a_i} \preceq_{\mathsf{Y}} t_{a_j}$. A *longest non-decreasing sequence*, denoted by $\mathsf{LNDS}(\gamma, r)$, is a non-decreasing sequence with a maximum k.

According to the definitions, it is easy to see that a strict increasing sequence is an OD sequence, and an OD sequence is a non-decreasing sequence. We also see the following.

Proposition 5: Any two tuples in $\mathsf{LSIS}(\mathsf{X} \mapsto \mathsf{Y}, r)$ do not form a violation (split or swap) *w.r.t.* $\mathsf{XU} \mapsto \mathsf{YV}$.

Proposition 6: 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.* $\mathsf{X} \mapsto \mathsf{Y}$; and (b) r' is maximum among all subsets of r that satisfy (a).

Intuitively, (1) $\mathsf{LSIS}(\mathsf{X} \mapsto \mathsf{Y}, r)$ must be an OD sequence for $\mathsf{XU} \mapsto \mathsf{YV}$ 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 r that is free of swap violations *w.r.t.* $\mathsf{X} \mapsto \mathsf{Y}$. Recall that $\mathsf{XU} \mapsto \mathsf{YV}$ can never resolve swap *w.r.t.* $\mathsf{X} \mapsto \mathsf{Y}$. Therefore, $\mathsf{LNDS}(\mathsf{X} \mapsto \mathsf{Y}, r)$ is an upper bound of $\mathsf{LOS}(\mathsf{XU} \mapsto \mathsf{YV}, r)$.

Algorithm 3: Compute LOS, LSIS and LNDS

Input: sorted partitions τ_X,τ_Y
Output: $\mathsf{LOS}(\mathsf{X} \mapsto \mathsf{Y}, r)$, $\mathsf{LSIS}(\mathsf{X} \mapsto \mathsf{Y}, r)$, $\mathsf{LNDS}(\mathsf{X} \mapsto \mathsf{Y}, r)$

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* τ_X **do**
6 **foreach** $ecxy$ *in* τ_{XY} *that is 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** $ecxy$ *in* τ_{XY} *that is 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 $lnds.query([1, |\tau_Y|])$;

Both bounds are tight, as shown in the following example.

Example 13: Recall Table III. For $\mathsf{A} \mapsto \mathsf{C}$, 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 $\mathsf{LOS}(\mathsf{AU} \mapsto \mathsf{CV}, r)$. It can be seen that $\mathsf{A} \mapsto \mathsf{CD}$ has a LOS $[t_1, t_8]$, and $\mathsf{AB} \mapsto \mathsf{C}$ has a LOS $[t_2, t_1, t_6, t_5, t_7, t_8]$. □

Following this, we are ready to define the lower and upper bounds of g_3 for $\mathsf{XU} \mapsto \mathsf{YV}$.

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

Algorithm. Algorithm 3 is a three-in-one approach to computing LOS, LSIS and LNDS. We again use segment trees, with one tree for each of LOS, LSIS and LNDS (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 computations in a dynamic programming fashion. On the trees, the tuple rank in τ_Y is used as the key, 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 [14]. 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 value on both X and Y.

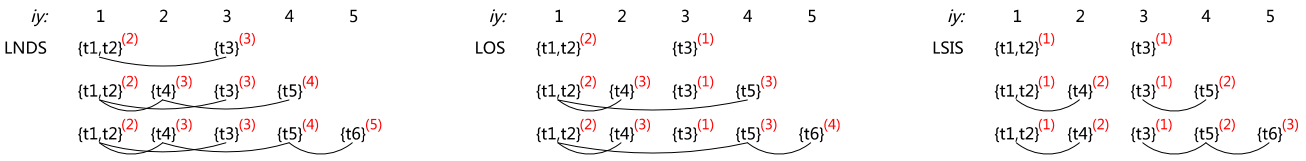


Fig. 2. Example 14 for Algorithm 3

The outer loop (line 5) enumerates equivalence class ecx in τ_X, ordered by X's value. The first inner loop (lines 6-11) enumerates equivalence class $ecxy$ 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 the results of LOS, LNDS and LSIS, by querying *max* values from their related trees (line 16).

Example 14: Consider $\overrightarrow{\mathsf{D}}\mapsto\overrightarrow{\mathsf{E}}$ on Table II. 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 process $\{t_1,t_2,t_3\}$. 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 not) simultaneously in any LOS (resp. LNDS), but only one of them exists in any LSIS. (b) t_3 can be appended to $\{t_1,t_2\}$ in LNDS, but not in LOS or LSIS.

(2) We process $\{t_4,t_5\}$ 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 process $\{t_6\}$ 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 $[1,|\tau_Y|]$, where $|\tau_Y|$ equals $|r|$ in the worst case. It takes at most $O(|r|)$ to build and $O(log(|r|))$ to update and query segment trees. It takes $O(|r|log(|r|))$ to compute τ_{XY} in line 4. The two inner loops are linear in the size of equivalence classes in τ_{XY}. Algorithm 2 has a worst-case complexity of $O(|r|log(|r|))$.

VI. Discovery of Approximate ODs

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 to further improve the efficiency.

A. Algorithms for Approximate OD 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 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 $\overrightarrow{\mathsf{A}}\mapsto\overrightarrow{\mathsf{B}}$, *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 III).

For each AOD candidate $\mathsf{X}\mapsto\mathsf{Y}$, we compute its error measure value Vg and lower/upper bound value LBg/UBg on instance r, with the given function g (line 8). If $\mathsf{X}\mapsto\mathsf{Y}$ is valid, then we add it into Σ (lines 9-10). If the UBg is larger than the threshold e, then we further test $\mathsf{X}\mapsto\mathsf{Y}\overline{\mathsf{C}}$ (both $\mathsf{X}\mapsto\mathsf{Y}\overrightarrow{\mathsf{C}}$ and $\mathsf{X}\mapsto\mathsf{Y}\overleftarrow{\mathsf{C}}$) for all $C\in R\backslash\mathsf{XY}$, by recursively calling function $Search$ (lines 11-14). As a prerequisite, we check whether $\mathsf{Y}\overline{\mathsf{C}}$ is a minimal attribute list; non-minimal AODs due to non-minimal attribute list are directly discarded. By definition (Section IV), it suffices to consider the new attribute C (line 13), *i.e.,* whether $\mathcal{Y}\rightarrow C$. Indeed, it is to check 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 $\mathsf{Y}\overline{\mathsf{C}}$ 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 further develop optimization techniques for this in Section VI-B.

If the upper bound UBg is not larger than the threshold e, then AODs of the form $\mathsf{X}\mapsto\mathsf{YV}$ are all valid. We generate minimal ones among them by calling function $Extend$ (line

Algorithm 4: DisAOD

Input: a relation r of schema R, an error measure function g and a threshold e
Output: the complete set Σ of minimal and valid AODs on r

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** Σ;
6
7 **Function Search**(AOD candidate $\mathsf{X} \mapsto \mathsf{Y}$)
8 Vg, LBg, $UBg \leftarrow Compute(\mathsf{X} \mapsto \mathsf{Y}, g, r)$;
9 **if** $Vg \leq e$ **then**
10 | $\Sigma \leftarrow \Sigma \cup \{\mathsf{X} \mapsto \mathsf{Y}\}$;
11 | **if** $UBg > e$ **then**
12 | | **foreach** $C \in R \backslash \mathsf{XY}$ **do**
13 | | | **if** $MinimalAttributelist(\mathsf{Y}\overline{\mathsf{C}})$ **then**
14 | | | | **Search**($\mathsf{X} \mapsto \mathsf{Y}\overline{\mathsf{C}}$);
15 | **else**
16 | | $\Sigma \leftarrow \Sigma \cup Extend(\mathsf{X} \mapsto \mathsf{Y})$;
17 **else**
18 | **if** $LBg \leq e$ **then**
19 | | **foreach** $C \in R \backslash \mathsf{XY}$ **do**
20 | | | **if** $MinimalAttributelist(\mathsf{X}\overline{\mathsf{C}})$ **then**
21 | | | | **Search**($\mathsf{X}\overline{\mathsf{C}} \mapsto \mathsf{Y}$);

16). It suffices to consider $\mathsf{X} \mapsto \mathsf{YW}$, where W is a list on *all* attributes in $R \backslash \mathsf{XY}$ (the others cannot be minimal), and exclude AODs with non-minimal attribute lists.

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

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

Correctness&Time Complexity. DisAOD finds the complete set of minimal valid AODs: it enumerates all possible candidates and only prunes non-minimal or invalid ones. Besides g_1 and g_3, DisAOD is suitable to any error measure function g if g satisfies the criteria stated in Section IV. In case the lower/upper bounds are not available, we can set the lower (resp. upper) bound as 0 (resp. 1).

DisAOD has a worst-case complexity of $O(|R|!)$ (the size of the search space of AOD discovery) in the number $|R|$ of attributes, and $O(|r|log(|r|))$ in the number $|r|$ of tuples.

Remarks. We highlight the differences between DisAOD and existing works on exact OD discovery [4], [9], [14].

(1) DisAOD computes error measures for each candidate OD, in contrast to exact OD discoveries that perform OD validations. We adapt two measures g_1 and g_3 to AOD discovery, with desirable properties and efficient computations.

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

Input: sorted partitions τ_X,τ_Y,$\tau_{\mathsf{X}\overline{\mathsf{A}}}$, # of swapped and ordered tuple pairs for $\mathsf{X} \mapsto \mathsf{Y}$
Output: g_{swap} and # of ordered tuple pairs for $\mathsf{X}\overline{\mathsf{A}} \mapsto \mathsf{Y}$

1 $swap$, $ordered \leftarrow$ # of swapped and ordered tuple pairs for $\mathsf{X} \mapsto \mathsf{Y}$;
2 **foreach** *equivalence class* ecx *in* τ_X **do**
3 | $range \leftarrow 0$;
4 | **foreach** *tuple* t *in* ecx **do**
5 | | $m[t] \leftarrow$ # of distinct $I_Y[p] \leq I_Y[t]$ for all $p \in ecx$;
6 | | $range \leftarrow max(range, m[t])$;
7 | $seg \leftarrow$ an empty segment tree on range $[1, range]$;
8 | **foreach** *equivalence class* $ecxa$ *in* $\tau_{\mathsf{X}\overline{\mathsf{A}}}$ *from* ecx **do**
9 | | **foreach** *tuple* t *in* $ecxa$ **do**
10 | | | $swap \leftarrow swap + seg.query([m[t]+1, range])$;
11 | | | $ordered \leftarrow ordered + seg.query([1, m[t]-1])$;
12 | | **foreach** *tuple* t *in* $ecxa$ **do**
13 | | | $seg.insert(m[t])$;
14 **return** $g_{\mathsf{swap}} = \frac{swap}{|r|^2-|r|}$, $ordered$;

(2) DisAOD employs novel pruning rules, based on the introduction of upper/lower bounds of g_1 and g_3. These rules are crucial to the efficiency. Without them, DisAOD becomes orders of magnitude slower in our experimental evaluations.

(3) A set of novel optimizations is introduced to DisAOD for further improving efficiency (Section VI-B).

B. Optimizations

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

Incremental computations. Following $\mathsf{X} \mapsto \mathsf{Y}$, we consider new candidate $\mathsf{X}\overline{\mathsf{A}} \mapsto \mathsf{Y}$ or $\mathsf{X} \mapsto \mathsf{Y}\overline{\mathsf{A}}$. Leveraging results of $\mathsf{X} \mapsto \mathsf{Y}$, *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.* $\mathsf{X} \mapsto \mathsf{Y}$ is still a swapped (resp. an ordered) pair *w.r.t.* $\mathsf{X}\overline{\mathsf{A}} \mapsto \mathsf{Y}$ or $\mathsf{X} \mapsto \mathsf{Y}\overline{\mathsf{A}}$. Hence, the number of swapped (resp. ordered) pairs monotonically increases. Without loss of generality, we consider $\mathsf{X}\overline{\mathsf{A}} \mapsto \mathsf{Y}$. If tuples t, s form a new swapped or an ordered pair *w.r.t.* $\mathsf{X}\overline{\mathsf{A}} \mapsto \mathsf{Y}$, then the order of them on $\mathsf{X}\overline{\mathsf{A}}$ must be different from that on X. This implies that t, s have the same value on X, *i.e.,* in the same equivalence class of τ_X; otherwise we know $t \prec_{\mathsf{X}\overline{\mathsf{A}}} s$ if $t \prec_{\mathsf{X}} s$. Therefore, we can leverage τ_X that is already computed for $\mathsf{X} \mapsto \mathsf{Y}$, and cope with each equivalence class in τ_X separately, when computing the *incremental* swapped and ordered tuple pairs for $\mathsf{X}\overline{\mathsf{A}} \mapsto \mathsf{Y}$.

Algorithm. Algorithm 5 *incrementally* computes g_{swap} and the number of ordered tuple pairs for $\mathsf{X}\overline{\mathsf{A}} \mapsto \mathsf{Y}$, based on the known number of swapped and ordered tuple pairs for $\mathsf{X} \mapsto \mathsf{Y}$. As stated earlier, it handles each equivalence class of τ_X one by

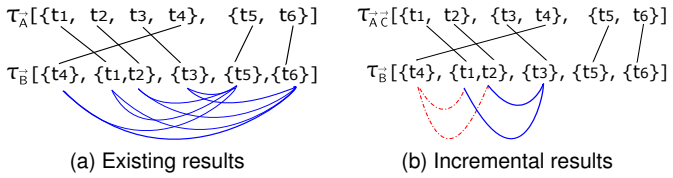


Fig. 3. Example 15 for Algorithm 5

one, in contrast to Algorithm 2 that deals with all equivalence classes of τ_X as a whole.

A segment tree is leveraged for each equivalence class ecx. The complexity of a segment tree is closely related to the range; recall that 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 (line 5). As an example, for 5 tuples with $I_Y[t]$ values of {3, 5, 5, 1, 10}, their ranks in ecx are {2, 3, 3, 1, 4}. The range of the segment tree is adjusted as the max value of new ranks (line 6), 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 15: On Table II, $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$ incurs no swapped but 8 ordered pairs (Figure 3a). 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 $\{t_1, t_2\}$ 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. □

Time Complexity. State compaction is done in $O(|r|)$, so is the initialization of all segment trees. 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 $\mathcal{X} \rightarrow C$ 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 $\mathsf{Y}\overline{\mathsf{C}}$ is not a minimal attribute list.

(2) Otherwise, we compute $\tau_{Y\overline{C}}$. (a) If $\tau_{Y\overline{C}} = \tau_Y$, then $\mathsf{Y}\overline{\mathsf{C}}$ is not a minimal attribute list since $\mathcal{Y} \rightarrow C$. 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 $\mathsf{Y}\overline{\mathsf{C}}$ 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 $\overrightarrow{\mathsf{A}}\,\overrightarrow{\mathsf{C}} \mapsto \overrightarrow{\mathsf{B}}$ from $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$, $\overrightarrow{\mathsf{A}}\,\overrightarrow{\mathsf{C}} \mapsto \overrightarrow{\mathsf{D}}$ from $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{D}}$, and $\overrightarrow{\mathsf{B}} \mapsto \overrightarrow{\mathsf{A}}\,\overrightarrow{\mathsf{C}}$ from $\overrightarrow{\mathsf{B}} \mapsto \overrightarrow{\mathsf{A}}$, all with the list $\overrightarrow{\mathsf{A}}\,\overrightarrow{\mathsf{C}}$, 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.

TABLE IV
DATASETS, EXECUTION STATISTICS OF AOD_1 AND AOD_3

Dataset Properties			AOD_1 (e = 0.001)		AOD_3 (e = 0.01)	
DataSet	$\|r\|$	$\|R\|$	Time(s)	$\|AOD\|$	Time(s)	$\|AOD\|$
NCV	1K	19	8	36	17	124
NCV	930K	17	35,235	222	254,861	433
FLI	500K	14	26,829	479	8,655	386
DB	250K	16	117	59	1,026	180
Letter	20K	17	2	0	0.552	0
Hepa	155	20	6	0	0.191	0
Horse	300	26	11	47	10	40
Atom	33k	11	68	325	131	310

VII. EXPERIMENTAL EVALUATIONS

In this section, we present an experimental study. Following the experimental settings, we conduct extensive experiments to (1) demonstrate the efficiency of AOD discovery and optimization techniques, and to (2) verify the effectiveness of AOD discovery from dirty data.

A. Experimental setting.

Datasets. We use a set of real-life and synthetic data that are evaluated in OD discoveries [4], [9], [14], [24], [25] (available online http://metanome.de). (1) NCV, FLI, Hepa and Atom are real-life data, concerning voters, flights, hepatitis disease and atom sites, respectively. (2) DB, Letter and Horse are synthetic datasets with complicated attribute relationships. We summarize datasets in Table IV, where $|r|$ denotes the number of tuples, and $|R|$ denotes the number of attributes.

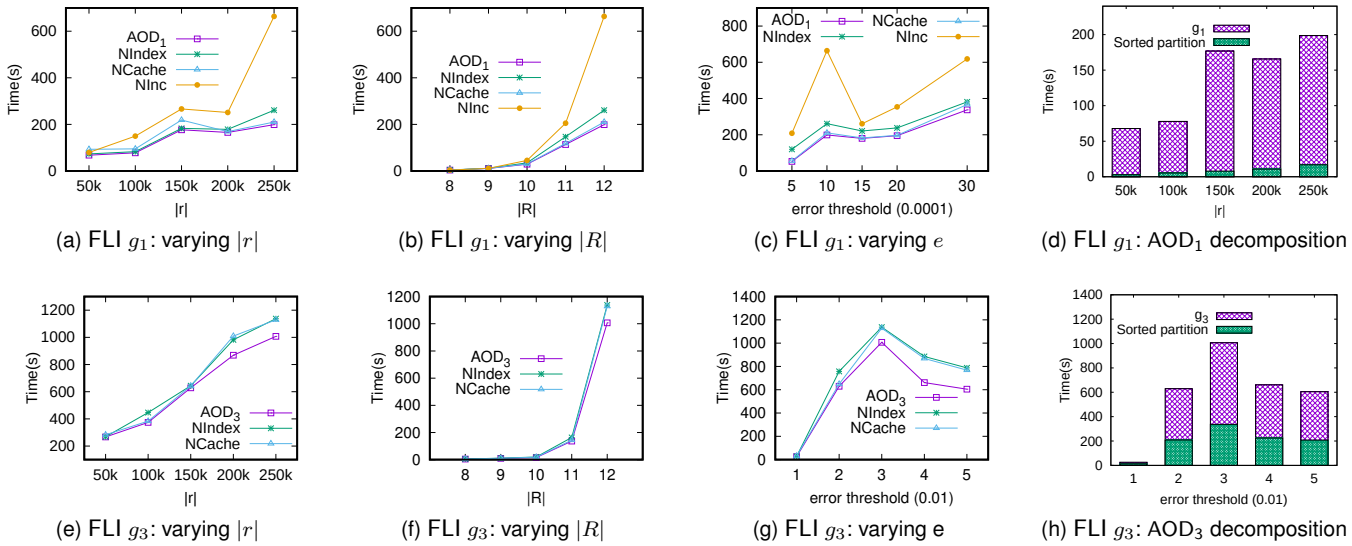
Algorithms. We implement all algorithms in Java. (1) AOD_1 and AOD_3, different versions of DisAOD for measure g_1 and g_3 respectively. (2) Some variants of DisAOD, for testing the effectiveness of optimizations (details are provided later). (3) FastAOD, the algorithm for discovering approximate set-based canonical ODs with measure g_3 [25].

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 with an Intel Core(TM) i5 1.8GHz CPU, 8GB of memory and Windows, and report the average results of 5 runs.

B. 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 IV, 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 $\frac{1}{|r|}$,



(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

but a single violation incurs a g_1 value of $\frac{1}{|r|^2-|r|}$. Therefore, an effective error threshold for g_1 is typically much smaller than that for g_3. The times of AOD_1 and AOD_3 cannot be compared, since the sets of discovered AODs are different.

We are not aware of any existing works on lexicographical AOD discovery, and it is beyond the scope of this paper to extend recent exact OD discoveries to approximate ODs (explained in Section II). In the sequel we 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 VI-B). (1) NInc disables the incremental computation of *swapped* and *ordered* pais; (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 experimental results on data FLI, since they are similar on other datasets.

We set $|r| = 250K$, $|R| = 12$ and e = 0.001 by default on FLI, and vary $|r|$ from 50K to 250K in Figure 4a, $|R|$ from 8 to 12 in Figure 4b, and e from 0.0005 to 0.003 in Figure 4c.

We see the following. (1) AOD_1 scales well with $|r|$, consistent with the complexity analysis. As $|r|$ increases from 50K to 250K, the time increases from 68s to 199s. (2) $|R|$ significantly affects the efficiency; recall that the search space of AODs is factorial in $|R|$. We find the number of discovered AODs increases from 0 to 1094, as $|R|$ increases from 8 to 12. (3) The threshold e affects the results of AOD_1. Intuitively, a large e value leads to *general* AODs, with few attributes on the LHS and more attributes on the RHS, while a small e value leads to *specialized* AODs (recall Example 2). We find on FLI the number of discovered AODs almost remains unchanged when e is in the range of [0.0005,0.002], but increases by more than 4 times as e increases to 0.003. The results also show that e affects the effectiveness of upper/lower bounds. A relatively small e, *e.g.,* 0.0005, helps AOD_1 prune invalid candidates more efficiently and leads to far less running time. A relatively large e, *e.g.,* 0.003, helps AOD_1 quickly generate valid AODs. Hence, the time only increases by about 60% as the number of AODs increases by more than 4 times.

In terms of the optimizations, we see the following. (1) AOD_1 is faster than NInc by up to 4 times and on average 118%. The cost of AOD_1 mainly consists of the times for creating sorted partitions and for computing g_1 values; Algorithms 1 and 2 take sorted partitions as inputs. 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 efficiency. (2) NIndex and NCache only concern computations of sorted partitions. We see AOD_1 is on average faster than NIndex and NCache by 22% and 8%, 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 $|r| = 250K$, $|R| = 12$ and e = 0.03 by default on FLI, and vary $|r|$ from 50K to 250K in Figure 4e, $|R|$ from 8 to 12 in Figure 4f, and e from 0.01 to 0.05 in Figure 4g.

We see the following. (1) AOD_3 scales well with $|r|$. (2) As expected, the efficiency of AOD_3 is sensitive to $|R|$; the number of discovered AODs increases from 2 to 886 as $|R|$ increases. (3) We find the number of discovered AODs almost remain unchanged when $e > 0.02$ on FLI. When $e > 0.03$, the upper-bound technique helps generate AODs more efficiently, and hence the time decreases. (4) AOD_3 is on average faster than NIndex and NCache by 13% and 10%, respectively. (5) The computation of g_3 takes more than 65% of the total time, as shown in Figure 4h (along the same setting as Figure 4g).

Exp-4: AOD_3 against FastAOD. We compare AOD_3 against FastAOD [25]. Different from lexicographical ODs considered in this paper, FastAOD discovers approximate set-based

TABLE V
$\mathsf{AOD_3}$ AGAINST FastAOD ON VARIOUS DATASETS

Dataset Properties			$\mathsf{AOD_3}$ (e = 0.01)		FastAOD (e = 0.01)	
DataSet	$\lvert r\rvert$	$\lvert R\rvert$	Time(s)	$\lvert AOD\rvert$	Time(s)	$\lvert AOD\rvert$
NCV	10K	12	2	12	2,276	482
FLI	10K	12	4	20	607	99
Hepa	155	20	0.191	0	139	58,028
Horse	300	26	10	40	142	175,118
Atom	33k	11	131	310	3,177	25

canonical ODs with measure g_3. Canonical ODs are suggested as alternatives to lexicographical ODs in [24], [25] (Section II).

The results in Table V show $\mathsf{AOD_3}$ is faster than FastAOD by orders of magnitude (results on DB, Letter are omitted since FastAOD cannot terminate within 6 hours).

The reason is mainly two-fold. (1) The number of discovered canonical ODs is usually much larger than lexicographical ODs, as shown in Table V. Similar results are seen in the comparison of exact canonical and lexicographical OD discoveries [9]. Although the canonical OD discovery has a smaller theoretical search space than the lexicographical one, its huge result set negatively affects efficiency. (2) It is quadratic in $|r|$ for FastAOD to compute g_3 [25], which hinders the scalability.

C. Effectiveness of AOD discovery

Exp-5: Recall of AOD discovery. We show the effectiveness of AOD discovery by finding ODs from dirty data. We add some attributes to FLI and NCV, and populate these attributes with real-life data, for more interesting and complex ODs. We then manually identify some "golden" ODs verified by domain experts (some example ODs are shown in Table VI).

For each dataset, we use a sample of 10K tuples. We introduce noise to data, which is controlled by the noise ratio θ and two different strategies [17]. #1: on each attribute, each value has a probability of θ to be assigned a new value. #2: each tuple has a probability of θ to be selected, and new values 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 datasets, and compute the *recall* as the ratio of the number of discovered golden ODs to the total number of golden ODs.

(1) From Figure 5a to 5h, we vary the error threshold e and test various settings (#i, θ) on NCV. In this set of experiments, we use values close to the original correct ones as new values, which is common in practice.

We see the following. (a) In contrast to exact OD discoveries with a recall of 0 in all settings (not shown), both $\mathsf{AOD_1}$ and $\mathsf{AOD_3}$ have a recall of 100% when e is above a threshold. We denote this threshold by e_o. (b) In $\mathsf{AOD_1}$, 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. Recall that g_1 concerns the ratio of the number of violating tuple pairs to $|r|^2$. (c) In $\mathsf{AOD_3}$, 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.

(2) We report results on FLI from Figure 5i to 5p. In this set of experiments, the maximum/minimum values in the domain are used as new values, to maximize violations.

We see the following. (a) As expected, the required threshold e_o for a recall of 100% increases significantly and is larger than the noise ratio θ in most cases. This becomes very evident when relatively more noises are distributed among more tuples, *i.e.,* the setting of (#1, 1%), as shown in Figures 5j, 5n. (b) $\mathsf{AOD_3}$ still guarantees a recall of 100% in #2, when the threshold equals θ (shown in Figures 5o and 5p). (c) Using the same threshold, we usually get a larger recall in the setting of #2 than #1; see *e.g.,* Figure 5i against 5k. The reason is that noises introduced in #2 are on a smaller set of tuples, compared with #1. (d) We find a relatively large threshold is required to recall ODs with multiple LHS (RHS) attributes (not shown). Intuitively, such ODs concern more new values and are hence more likely to be involved in violations.

TABLE VI
SAMPLE ODS

$\overrightarrow{\mathsf{Rank}} \mapsto \overrightarrow{\mathsf{FreeLuggage}}$	free luggage allowance increases with customer rank
$\overrightarrow{\mathsf{SeqNo}} \mapsto \overrightarrow{\mathsf{YearMonthDay}}$	Sequence No is an auto-increment number
$\overrightarrow{\mathsf{Birthday}} \mapsto \overleftarrow{\mathsf{Age}}$	a late birthday implies a small age
$\overrightarrow{\mathsf{Salary}} \mapsto \overrightarrow{\mathsf{Tax}}$	tax increases with salary (in NCV all person are in the same state)

VIII. CONCLUSION

We have formalized the AOD discovery problem, developed efficient algorithms and optimizations for error measures, related lower/upper bounds and AOD discovery. We have also experimentally verified the benefits of our methods.

There is naturally more to be done. As shown in our experimental evaluations, the number of minimal valid AODs can be large on some instances. We intend to study ranking functions for measuring the *interestingness* of AODs, so as to help users quickly select a small set of more relevant AODs. We also intend to study further optimizations for AOD discovery, *e.g.,* by leveraging sampling techniques [9].

Acknowledgments. This work is supported in part by National Key R&D Program of China 2018YFB1700403, NSFC 61572135 and NSFC 61925203. We thank anonymous reviewers for their valuable suggestions.

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Profiling relational data: a survey. *VLDB J.*, 24(4):557–581, 2015.
[2] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.
[3] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.
[4] Cristian Consonni, Paolo Sottovia, Alberto Montresor, and Yannis Velegrakis. Discovering order dependencies through order compatibility. In *EDBT*, pages 409–420, 2019.

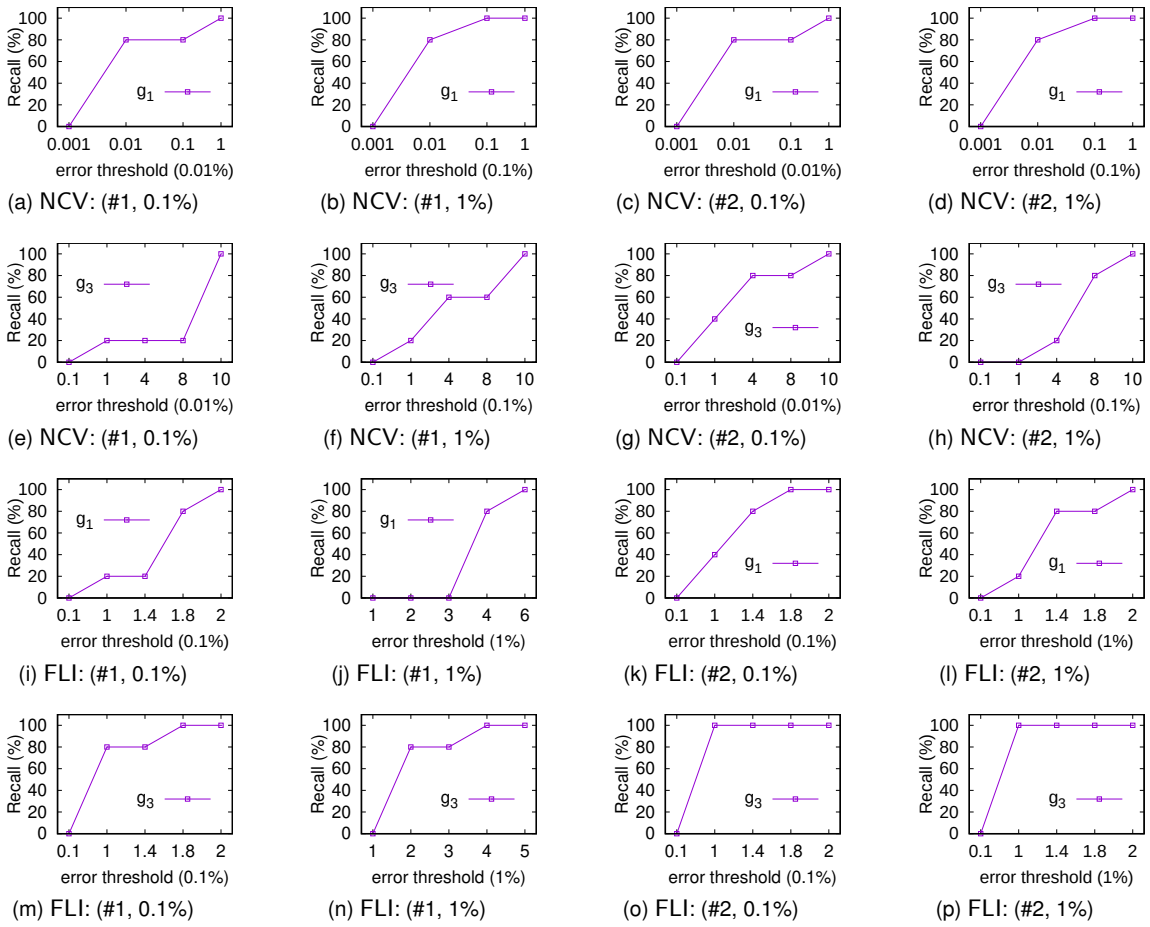


Fig. 5. Effectiveness of AOD discovery

[5] Michael L. Fredman. On computing the length of longest increasing subsequences. *Discret. Math.*, 11(1):29–35, 1975.

[6] Seymour Ginsburg and Richard Hull. Order dependency in the relational model. *Theor. Comput. Sci.*, 26:149–195, 1983.

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

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

[9] Yifeng Jin, Lin Zhu, and Zijing Tan. Efficient bidirectional order dependency discovery. In *ICDE*, pages 61–72, 2020.

[10] Batya Kenig, Pranay Mundra, Guna Prasaad, Babak Salimi, and Dan Suciu. Mining approximate acyclic schemes from relations. In *SIGMOD*, pages 297–312, 2020.

[11] Batya Kenig and Dan Suciu. Integrity constraints revisited: From exact to approximate implication. In *ICDT*, pages 18:1–18:20, 2020.

[12] Jyrki Kivinen and Heikki Mannila. Approximate dependency inference from relations. In *ICDT*, pages 86–98, 1992.

[13] Sebastian Kruse and Felix Naumann. Efficient discovery of approximate dependencies. *PVLDB*, 11(7):759–772, 2018.

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

[15] Pei Li, Jaroslaw Szlichta, Michael H. Böhlen, and Divesh Srivastava. Discovering band order dependencies. In *ICDE*, pages 1878–1881, 2020.

[16] David Liben-Nowell, Erik Vee, and An Zhu. Finding longest increasing and common subsequences in streaming data. *J. Comb. Optim.*, 11(2):155–175, 2006.

[17] Ester Livshits, Alireza Heidari, Ihab F. Ilyas, and Benny Kimelfeld. Approximate denial constraints. *PVLDB*, 13(10):1682–1695, 2020.

[18] Thorsten Papenbrock and Felix Naumann. A hybrid approach to functional dependency discovery. In *SIGMOD*, pages 821–833, 2016.

[19] Eduardo H. M. Pena, Eduardo Cunha de Almeida, and Felix Naumann. Discovery of approximate (and exact) denial constraints. *PVLDB*, 13(3):266–278, 2019.

[20] Joeri Rammelaere and Floris Geerts. Explaining repaired data with cfds. *PVLDB*, 11(11):1387–1399, 2018.

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

[22] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *SIGMOD*, pages 57–67, 1996.

[23] Shaoxu Song, Lei Chen, and Philip S. Yu. Comparable dependencies over heterogeneous data. *VLDB J.*, 22(2):253–274, 2013.

[24] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. Effective and complete discovery of order dependencies via set-based axiomatization. *PVLDB*, 10(7):721–732, 2017.

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

[26] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. Fundamentals of order dependencies. *PVLDB*, 5(11):1220–1231, 2012.

[27] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, Wenbin Ma, Weinan Qiu, and Calisto Zuzarte. Business-intelligence queries with order dependencies in DB2. In *EDBT*, pages 750–761, 2014.

[28] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. Expressiveness and complexity of order dependencies. *PVLDB*, 6(14):1858–1869, 2013.

[29] Zijing Tan, Ai Ran, Shuai Ma, and Sheng Qin. Fast incremental discovery of pointwise order dependencies. *PVLDB*, 13(10):1669–1681, 2020.