Efficient Set-based Order Dependency Discovery with a Level-wise Hybrid Strategy

Yihan Li [1,2] Ruifeng Li [1,2] Zijing Tan [1,2†] Weidong Yang [1,2] Shuai Ma [3]

[1]*Fudan University, China* [2]*Shanghai Key Laboratory of Data Science, China* [3]*SKLSDE Lab, Beihang University, China*

***Abstract*—Order dependencies (ODs) state ordering specifications between attributes, and have been proven effective in query optimization for sorting operations. In this paper we investigate the problem of set-based OD discovery, for automatically finding hidden ODs from data. We tackle the problem with a novel level-wise hybrid strategy. With a given relational instance r, we discover ODs from a sample (subset) of r, validate the discovered ODs on r and refine the sample by leveraging the validation, in a level-by-level manner according to the lattice of set-based ODs. This process continues until the discovery result on the sample converges to that on r. We prove that a dynamic sample whose size keeps growing can be used in the process without affecting the correctness and completeness of the discovery result, and present techniques to incrementally refine the sample on demand. We also enhance our method with multi-threaded parallelism. On a host of datasets, our method is faster than the state-of-the-art method up to orders of magnitude even when the parallelism of our approach is disabled, and achieves up to a 4.5x self-relative parallel speedup with 6 threads.**

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

I. Introduction

Understanding inherent relationships between attributes is crucial for query optimization [7], [16], [17], [31], [35]–[37], and in the database community attribute relationships are usually stated as data dependencies. Ordered attributes, *e.g.,* time, weight and distance, are very common in real-life data, and sorting tuples by ordered attributes with *order-by* clauses, is one of the most important building blocks of SQL queries. Therefore, *order dependencies* (ODs) have been actively studied in the literature [8], [9], [32], [33], [35], [37], so as to fulfill their potentials in query optimization.

Lexicographical bidirectional ordering is adopted in *order-by* clauses. For example, with the order specification "*order by A asc, B desc*", tuples are first sorted by A in ascending order, and then by B in descending order to break ties. This observation motivates the research on lexicographical (list-based) ODs [35], [37] (the formal definitions will be reviewed in Section III). Discovery techniques for lexicographical ODs are also studied [6], [12], [19], for automatically finding hidden lexicographical ODs from data. Such techniques are necessary in practice, since it is usually prohibitively expensive, if not impossible, to design dependencies manually [1], [2].

As noted in [29], [32], [33], the discovery of lexicographical ODs suffers from its high theoretical complexity, while the existing effort to reduce the complexity may lead to the loss of many useful ODs. Specifically, recall the complexity of a dependency discovery method is usually measured by the search space, *i.e.,* the total number of candidate dependencies. Since lexicographical ODs are list-based by nature, the search space of lexicographical OD discovery is factorial in the number $|R|$ of attributes, much larger than the exponential complexity of discovery methods for set-based dependencies, *e.g.,* functional dependencies (FDs). Hence, existing discovery methods [12], [19] only consider ODs whose left-hand-side (LHS) and right-hand-side (RHS) attribute lists are disjoint, reducing the search space from $O((|R|!)^2)$ to $O(|R|!)$. However, this restriction prevents the discovery of many lexicographical ODs useful in query optimization, as shown in the following example.

TABLE I
Relation instance r

	M	W	DT	AT	ET	MTOW	CAP	Type	MC
t_1	1	2	7:34	10:22	168	233	253	A330	FR
t_2	1	2	7:34	8:46	72	233	253	A330	FR
t_3	1	3	7:34	8:46	72	233	253	A330	FR
t_4	1	3	9:05	12:10	185	242	293	A330	FR
t_5	1	4	6:22	7:17	55	89	204	B738	USA
t_6	2	6	10:10	11:40	90	83	185	A321	FR

Example 1: Table I shows a relation instance r extracted from real-life data. Each tuple denotes a flight, with month (M), week (W), departure time (DT), arrival time (AT), elapsed time (ET) in minutes, maximum take-off weight (MTOW) in tons, capacity (CAP), plane model (Type) and manufacture country (MC). The following lexicographical ODs hold on r. (1) $\overrightarrow{\text{MTOW}} \mapsto \overrightarrow{\text{CAP}}$: this lexicographical OD states that if tuples in r are sorted by attribute MTOW in ascending order, then they are also sorted by CAP in ascending order ($\overrightarrow{\text{MTOW}}$ denotes MTOW asc). This OD holds, because (a) flights with the same MTOW also have the same CAP; and (b) if a flight t_i has a larger MTOW than another flight t_j, then the CAP of t_i is no less than that of t_j.

As shown in [35]–[37], the statement "*order by* CAP *asc,* MTOW *asc*" can be optimized to "*order by* MTOW *asc*" according to the valid OD, and be more efficiently evaluated. (2) $\overrightarrow{\text{Type}} \mapsto \overrightarrow{\text{Type}}\overrightarrow{\text{MC}}$: it can be verified that this lexicographical OD is equivalent to the FD Type $\rightarrow$MC. Every FD can be mapped to an equivalent OD by prefixing LHS attributes onto the RHS, as proved in [35], [37].

This OD (FD) is useful in optimizing SQL queries [31], [35], [37]. A query with "*order by* Type, MC" can be rewritten as an equivalent query with "*order by* Type", since tuples with the same value in Type also have the same value in MC.

† Zijing Tan is the corresponding author.

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00059

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00059

(3) $\overrightarrow{\mathsf{M}}\overrightarrow{\mathsf{W}} \leftrightarrow \overrightarrow{\mathsf{W}}\overrightarrow{\mathsf{M}}$: tuples that are sorted by M in ascending order and then by W in ascending order to break ties, are also sorted by W in ascending order and then by M in ascending order, and vice versa. This is because *week* never decreases as *month* increases. The OD is also referred to as an *order compatible dependency* [6], [35], [37]. Note that $\overrightarrow{\mathsf{M}} \mapsto \overrightarrow{\mathsf{W}}$ does *not* hold, since one *month* contains several *weeks*, *e.g.,* 4 weeks in the first month.

Based on $\overrightarrow{\mathsf{M}}\overrightarrow{\mathsf{W}} \leftrightarrow \overrightarrow{\mathsf{W}}\overrightarrow{\mathsf{M}}$, a composite index on (M, W) can be used for both "*order by* M, W" and "*order by* W, M".

(4) $\overrightarrow{\mathsf{DT}}\overrightarrow{\mathsf{ET}} \mapsto \overrightarrow{\mathsf{DT}}\overrightarrow{\mathsf{AT}}$: This OD holds, because for two flights with the same departure time, (a) they have the same arrival time if they have the same elapsed time, and (b) a flight with more elapsed time always arrives later. This OD specifies an ordering specification with a *context*: the ordering relationship between ET and AT only applies to flights with the same DT. Note neither $\overrightarrow{\mathsf{ET}} \mapsto \overrightarrow{\mathsf{AT}}$ nor $\overrightarrow{\mathsf{DT}}\overrightarrow{\mathsf{ET}} \mapsto \overrightarrow{\mathsf{AT}}$ holds on r.

Based on the OD, a query with "*where* DT = *const order by* AT" can be readily rewritten as "*where* DT = *const order by* ET" to take advantage of an index built on ET.

Except for the OD in (1), ODs in (2), (3), (4) *cannot* be discovered by methods of [12], [19], since repeated attributes occur in the LHS and RHS attribute lists of these ODs. □

To avoid the high complexity of lexicographical OD discovery, the notion of set-based ODs is proposed [32], [33]. Set-based ODs generalize lexicographical ODs: there is a polynomial mapping from every lexicographical OD to a set of set-based ODs, and the lexicographical OD holds iff set-based ODs from the set all hold. That is, set-based ODs suffice to encode all ordering specifications that can be stated by lexicographical ODs. For example, all lexicographical ODs in Example 1 can be mapped to set-based ODs (shown in Example 3 in Section III). With a given query, lexicographical ODs useful in query optimization can be identified and the validity of each of them can be checked via valid set-based ODs. Hence, the discovery of set-based ODs enables all query optimizations that leverage lexicographical ODs.

The expressiveness of set-based ODs is far beyond that of lexicographical ODs considered in [12], [19]. Better, the search space of set-based OD discovery is exponential in $|R|$, even much smaller than that of [12], [19]. Intuitively, this is because mappings of different lexicographical ODs can share the same set-based ODs. The baseline discovery methods for set-based ODs are presented in [32], [33], for automatically finding set-based ODs from data.

This work aims to develop an efficient set-based OD discovery method with the *hybrid* strategy. The hybrid strategy combines dependency discovery on sample data with dependency validation on the whole instance, and is proven effective in the discovery of other dependencies [3], [4], [12], [22], [27], [44]. We find performing a hybrid set-based OD discovery introduces new challenges, due to relationships between the two *subtypes* of set-based ODs that must be resolved during the discovery (to be illustrated in Section III). Such challenges are unique to set-based ODs, and motivate us to develop a novel *level-wise* hybrid strategy.

Contributions. This paper presents an efficient hybrid approach to set-based OD discovery.

(1) We address the problem of set-based OD discovery on a given relational instance r with a novel *level-wise hybrid* strategy (Section IV). We discover ODs from sample data of r, validate the discovered ODs on r, and refine the sample according to the detected violations in a *level-by-level* manner, until the discovery result on the sample converges to that on r. We show a dynamic sample whose size keeps growing can be used in the process, without affecting the correctness and completeness of the discovery. We present techniques to refine the sample on demand, based on OD validations on r.

(2) We present efficient algorithms and optimizations underlying our approach (Section V). We develop techniques for switching between discovery and validation, and for validating ODs on r and fetching violating tuples when necessary. We further enhance our method with multi-threaded parallelism.

(3) Using a host of real-life and synthetic datasets, we empirically verify the efficiency of our approach (Section VI). Our method is up to orders of magnitude faster than the state-of-the-art method [33] by exploiting the hybrid strategy, and the multi-threaded parallelism further delivers a speed-up ratio of up to 4.5x with 6 threads.

II. Related Work

Dependency discovery methods have been actively studied in the literature; see, *e.g.,* [4], [5], [10], [13], [18], [21]–[28], [30], [40]–[42], [44]–[46]. Please refer to [1], [2] for surveys. In this section, we investigate works close to ours.

Order dependency and its variants. There is a long history of studies on dependencies to specify ordering between tuples. The first kind of order dependency (OD), known as *pointwise* OD, and its theoretical foundations, are studied in [8], [9]. To be consistent with ordering specifications adopted in *order by* clauses, unidirectional and bidirectional lexicographical ODs are studied in [35] and [37], respectively. In unidirectional ODs all attributes are ordered in the same direction, while bidirectional ODs allow a mixture of the two directions. The inference problem of lexicographical ODs, and the relationship between FDs and lexicographical ODs, are also investigated [35], [37]. Specifically, lexicographical ODs generalize FDs. More recently, the notion of set-based canonical ODs is proposed [32], [33], together with a sound and complete set of inference rules. Set-based ODs generalize lexicographical ODs [32], [33], in the sense that every lexicographical OD can be mapped to a set of set-based ODs and the lexicographical OD holds iff the set-based ODs in the set all hold.

Discovery of pointwise and lexicographical ODs. A method for incrementally discovering pointwise ODs in response to tuple insertions is given in [38]. The first lexicographical OD discovery method [19] is developed for unidirectional ODs. A more efficient method for discovering bidirectional lexicographical ODs is presented in [12], equipped with a

hybrid strategy and new techniques for traversing the space of candidate lexicographical ODs and validating ODs. [12], [19] consider a subclass of lexicographical ODs, in which no repeated attributes can occur in LHS and RHS attribute lists. As noted earlier, many meaningful ODs are hence missed. In particular, lexicographical ODs no longer generalize FDs under this restriction. Another method to discover unidirectional lexicographical ODs is proposed in [6]. However, due to some flaws in its pruning rules, [6] fails to discover the complete set of ODs, as noted in [12], [34].

Discovery of set-based ODs. The methods for discovering unidirectional and bidirectional set-based ODs are given in [32] and [33], respectively. They enjoy the exponential worst-case complexity in $|R|$, and hence traverse the set-containment lattice to enumerate candidate ODs. Recently, [29] develops a *distributed* set-based OD discovery method that runs on multiple computing nodes in parallel. Ranking functions for set-based ODs are also studied in [29], [33], to help users select ODs that are more likely to be meaningful or relevant.

Other works on OD discovery. A different line of work is on discovering *approximate* ODs, for ODs holding on (dirty) data with some exceptions. Such techniques are developed for lexicographical ODs [11], [20] and set-based ODs [14], [33]. Another work [15] aims to discover *implicit* orders, which are potential domain orders that are not known yet.

This work considers the discovery of bidirectional set-based ODs (holding on data without exceptions), in the same setting as [29], [33]. This work differs from [29], [33] in the following. (1) We perform set-based OD discovery with the *hybrid strategy* that combines discovery on sample data with validation on the whole instance. Although the hybrid strategy is employed to discover other dependencies, *e.g.,* FDs [22], [44], unique column combinations (UCCs) [3], denial constraints (DCs) [4], matching dependencies [27], and lexicographical ODs [12], this is the first effort to apply this strategy to set-based OD discovery. We develop a set of novel techniques to facilitate the hybrid strategy for set-based OD discovery, since dependency discovery techniques, *e.g.,* the traversal strategy of the search space and the dependency validation method, heavily depend on the dependency types. In particular, we present a novel *level-wise* hybrid strategy and a novel *dynamic* sampling scheme, to correctly handle the relationship between the two subtypes of set-based ODs (to be illustrated in Section III), enable pruning rules and guarantee the correctness and completeness of our discovery. (2) We enhance our method with shared-memory parallelism implemented by multiple threads. Results of multi-threaded parallelism can be easily reproduced, while the distributed method [29] necessarily depends on more computing resources and sophisticated environment settings.

III. Preliminaries

In this section, we review basic notations of lexicographical ODs [35], [37] and set-based ODs [32], [33].

R denotes a relational schema, and A, B are attributes of R. r denotes an instance of R, and t, s denote tuples in r. t_A denotes the value of attribute A in t. We use marked attribute $\overline{A}$ ($\overrightarrow{A}$ or $\overleftarrow{A}$) to denote A asc or A desc in SQL clauses, and let $t_{\overline{A}} = t_A$. X denotes a list of marked attributes, *e.g.,* $[\overline{A_1}, \ldots, \overline{A_k}]$, and $\mathcal{X}$ denotes the set of attributes (without directions) in X. With a given tuple t, $t_{\mathsf{X}} = [t_{A_1}, \ldots, t_{A_k}]$. A non-empty list X can be denoted as $[\overline{A_1} \,|\mathsf{Y}\,]$, where Y is the remaining list after removing the first marked attribute $\overline{A_1}$ from X.

Lexicographical Ordering. For a given $\overline{A}$ and tuples t and s, we write $t \prec_{\overline{A}} s$ if (a) $\overline{A}{=}\overrightarrow{A}$ and $t_A{<}s_A$; or (b) $\overline{A}{=}\overleftarrow{A}$ and $t_A{>}s_A$. With a given $\mathsf{X} = [\overline{A_1}, \ldots, \overline{A_k}]$, we write $t \preceq_{\mathsf{X}} s$ if (a) $\mathsf{X} = [\,]$; or (b) $t \prec_{\overline{A_1}} s$; or (c) $\mathsf{X} = [\overline{A_1} \,|\mathsf{Y}\,]$ such that $t_{A_1} = s_{A_1}$ and $t \preceq_{\mathsf{Y}} s$. We write $t{\prec_{\mathsf{X}}}s$ if $t{\preceq_{\mathsf{X}}}s$ but $s{\not\preceq_{\mathsf{X}}}t$.

Lexicographical Order Dependency [35], [37]. Given two lists X, Y, $\gamma = \mathsf{X} \mapsto \mathsf{Y}$ denotes a lexicographical order dependency (OD). An 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 is valid (holds) on r. If $\mathsf{X} \mapsto \mathsf{Y}$ is valid on an instance r, then we know tuples of r are sorted by Y if they are sorted by X.

Violations of ODs [35], [37]. Given $\mathsf{X} \mapsto \mathsf{Y}$ and r, two kinds of violations may exist if $\mathsf{X} \mapsto \mathsf{Y}$ is *not* valid on r.

(1) A split violation occurs, when there are tuples t and s in r, such that $t_{\mathsf{X}}{=}s_{\mathsf{X}}$ but $t_{\mathsf{Y}}{\neq}s_{\mathsf{Y}}$.

(2) A swap violation occurs, when there are tuples t and s in r, such that $t{\prec_{\mathsf{X}}}s$ but $s{\prec_{\mathsf{Y}}}t$.

Example 2: Consider $\overrightarrow{\mathsf{W}}{\mapsto}\overrightarrow{\mathsf{DT}}$ on Table I.

(1) Tuples t_3 and t_4 have the same value in W but different values in DT, which incurs a split violation. Intuitively, there is no guarantee that t_3 is before t_4 if tuples are sorted by $\overrightarrow{\mathsf{W}}$.

(2) Tuple t_4 has a smaller value in W and a larger value in DT than t_5, which incurs a swap violation. Tuple t_4 is before t_5 when tuples are sorted by $\overrightarrow{\mathsf{W}}$, but t_4 is after t_5 when tuples are sorted by $\overrightarrow{\mathsf{DT}}$. □

To avoid the high complexity of lexicographical OD discovery and still keep the full expressiveness of ordering specifications, the notion of set-based ODs is introduced [32], [33]. It is based on the separation between split and swap violations, and is enhanced by combining the idea of *context*.

Set-based ODs [32], [33]. A set-based OD λ is given in the form of $\mathcal{X}$: [ ] $\mapsto \overline{A}$ or $\mathcal{X}$: $\overline{A} \sim \overline{B}$, where $\mathcal{X}$ is a set of attributes, called *context*, and [ ] denotes an empty list. Set-based ODs in the form of $\mathcal{X}$: [ ] $\mapsto \overline{A}$ or $\mathcal{X}$: $\overline{A} \sim \overline{B}$ are referred to as *constant* ODs or *order compatible* ODs, respectively. With a given instance r, all tuples of r are divided into *equivalence classes w.r.t.* $\mathcal{X}$, where tuples with the same values in all the attributes of $\mathcal{X}$ are in the same equivalence class. We denote an equivalence class *w.r.t.* $\mathcal{X}$ by $EC_{\mathcal{X}}$.

(1) $\mathcal{X}$: [ ] $\mapsto \overline{A}$ holds, if no split violations *w.r.t.* $\mathsf{X} \mapsto \overline{A}$ exist. That is, all the tuples in the same $EC_{\mathcal{X}}$ have the same value in A, *i.e.,* $\mathcal{X} \rightarrow A$ is a valid FD. Note the direction of $\overline{A}$, *i.e.,* $\overrightarrow{A}$ or $\overleftarrow{A}$, is irrelevant in the definition.

(2) $\mathcal{X}$: $\overline{A} \sim \overline{B}$ holds, if within every $EC_{\mathcal{X}}$, no swap violations

w.r.t. $\overline{A} \mapsto \overline{B}$ exist. That is, for any two tuples t and s in the same $EC_{\mathcal{X}}$, we have $t \preceq_{\overline{B}} s$ if $t \prec_{\overline{A}} s$.

We write $r \models \lambda$ if the (constant or order compatible) OD λ is valid (holds) on an instance r.

Please note the following. (1) An FD $\mathcal{X} \rightarrow A$ is equivalent to a constant OD $\mathcal{X}$: [ ] $\mapsto \overline{A}$. Hence, set-based ODs subsume FDs. (2) $\mathcal{X}$: $\overline{A} \sim \overline{B}$ denotes two ODs $\mathcal{X}$: $\overrightarrow{A} \sim \overrightarrow{B}$ and $\mathcal{X}$: $\overleftarrow{A} \sim \overleftarrow{B}$, while $\mathcal{X}$: $\overrightarrow{A} \sim \overleftarrow{B}$ (resp. $\mathcal{X}$: $\overleftarrow{A} \sim \overrightarrow{B}$) is neglected since it is equivalent to $\mathcal{X}$: $\overrightarrow{A} \sim \overrightarrow{B}$ (resp. $\mathcal{X}$: $\overrightarrow{A} \sim \overleftarrow{B}$). (3) $\mathcal{X}$: $\overline{A} \sim \overline{B}$ is equivalent to $\mathcal{X}$: $\overline{B} \sim \overline{A}$. If we have $t \preceq_{\overline{B}} s$ if $t \prec_{\overline{A}} s$, then we also have $t \preceq_{\overline{A}} s$ if $t \prec_{\overline{B}} s$. In what follows we only consider ODs in the form of $\mathcal{X}$: $\overline{A} \sim \overline{B}$.

The following result [32], [33] states the connection between lexicographical and set-based ODs.

Proposition 1: There is a polynomial mapping from a lexicographical OD to a set of set-based ODs; the lexicographical OD is valid iff set-based ODs in the set are all valid.

Proposition 1 is proved by providing a method to map an arbitrary lexicographical OD to set-based ODs [32], [33]. For example, $\overrightarrow{A}\overrightarrow{B} \mapsto \overrightarrow{C}\overrightarrow{D}$ is mapped to a set of set-based ODs: { $\{A, B\}$: [ ] $\mapsto \overrightarrow{C}$, $\{A, B\}$: [ ] $\mapsto \overrightarrow{D}$, $\{\}$: $\overrightarrow{A} \sim \overrightarrow{C}$, $\{A\}$: $\overleftarrow{B} \sim \overrightarrow{C}$, $\{C\}$: $\overrightarrow{A} \sim \overrightarrow{D}$, $\{A, C\}$: $\overleftarrow{B} \sim \overrightarrow{D}$ }. In what follows we showcase the mappings for lexicographical ODs in Example 1.

Example 3: (1) $\overrightarrow{\mathsf{MTOW}} \mapsto \overrightarrow{\mathsf{CAP}}$. It is mapped to a constant OD $\{\mathsf{MTOW}\}$: [ ] $\mapsto \overrightarrow{\mathsf{CAP}}$ and an order compatible OD $\{\}$: $\overrightarrow{\mathsf{MTOW}} \sim \overrightarrow{\mathsf{CAP}}$.

(2) $\overrightarrow{\mathsf{Type}} \mapsto \overrightarrow{\mathsf{TypeMC}}$. As proved in [35], [37], it is mapped to a constant OD: $\{\mathsf{Type}\}$: [ ] $\mapsto \overrightarrow{\mathsf{MC}}$, *i.e.,* the FD $\mathsf{Type} \rightarrow \mathsf{MC}$.

(3) $\overrightarrow{\mathsf{M}}\overrightarrow{\mathsf{W}} \leftrightarrow \overrightarrow{\mathsf{W}}\overrightarrow{\mathsf{M}}$. It is mapped to an order compatible OD: $\{\}$: $\overrightarrow{\mathsf{M}} \sim \overrightarrow{\mathsf{W}}$, as shown in [35], [37].

(4) $\overrightarrow{\mathsf{DT}}\overrightarrow{\mathsf{ET}} \mapsto \overrightarrow{\mathsf{DT}}\overrightarrow{\mathsf{AT}}$. This lexicographical OD is mapped to a constant OD $\{\mathsf{DT}, \mathsf{ET}\}$: [ ] $\mapsto \overrightarrow{\mathsf{AT}}$ and an order compatible OD $\{\mathsf{DT}\}$: $\overrightarrow{\mathsf{ET}} \sim \overrightarrow{\mathsf{AT}}$. □

According to Example 3, we can enable all query optimizations stated in Example 1 by checking the validity of lexicographical ODs via their related set-based ODs.

Along the same lines as [29], [33], we aim to discover the *complete* set of *minimal* and *valid* set-based ODs.

Trivial and minimal set-based ODs [32], [33].

(1) $\mathcal{X}$: [ ] $\mapsto \overline{A}$ is trivial if $A \in \mathcal{X}$, and $\mathcal{X}$: $\overline{A} \sim \overline{B}$ is trivial if $A \in \mathcal{X}$, or $B \in \mathcal{X}$, or $\overline{A} = \overline{B}$.

(2) $\mathcal{X}$: [ ] $\mapsto \overline{A}$ is minimal if (a) it is non-trivial, and (b) there is no context $\mathcal{Y} \subset \mathcal{X}$ such that $\mathcal{Y}$: [ ] $\mapsto \overline{A}$ is valid.

$\mathcal{X}$: $\overline{A} \sim \overline{B}$ is minimal if (a) it is non-trivial, (b) there is no context $\mathcal{Y} \subset \mathcal{X}$ such that $\mathcal{Y}$: $\overline{A} \sim \overline{B}$ is valid, (c) $\mathcal{X}$: [ ] $\mapsto \overline{A}$ is not valid, and (d) $\mathcal{X}$: [ ] $\mapsto \overline{B}$ is not valid.

Note the minimality of $\mathcal{X}$: $\overline{A} \sim \overline{B}$ concerns the validity of $\mathcal{X}$: [ ] $\mapsto \overline{A}$ and $\mathcal{X}$: [ ] $\mapsto \overline{B}$. If within each $EC_{\mathcal{X}}$ all tuples have the same value in A (or B), then $\mathcal{X}$: $\overline{A} \sim \overline{B}$ always holds since no swap violations can exist. This relationship between the two subtypes of set-based ODs is unique, and significantly complicates the development of a hybrid set-based OD discovery method, as will be illustrated in Section IV.

TABLE II
INSTANCE r

	A	B	C
t_1	1	4	6
t_2	1	5	6
t_3	1	6	7
t_4	2	7	6
t_5	3	7	6
t_6	4	7	7
t_7	5	6	3
t_8	6	5	7

TABLE III
SAMPLE r_1

	A	B	C
t_5	3	7	6
t_6	4	7	7
t_8	6	5	7

TABLE IV
SAMPLE r_2

	A	B	C
t_1	1	4	6
t_2	1	5	6
t_4	2	7	6
t_5	3	7	6

TABLE V
SAMPLE r_3

	A	B	C
t_7	5	6	3
t_8	6	5	7

TABLE VI
SAMPLE r_4

	A	B	C
t_1	1	4	6
t_7	5	6	3
t_8	6	5	7

IV. FOUNDATION OF OUR APPROACH

In this section, we illustrate the challenge of discovering set-based ODs with the hybrid strategy, and present our solution.

A. Analyses of hybrid set-based OD discovery

This paper aims to perform set-based OD discovery with the *hybrid* strategy. In hybrid discovery methods of other dependencies, *e.g.,* FDs [22], [44], an initial set of dependencies is first discovered from a sample (subset) of the given instance, and is then validated on the whole instance. The dependencies valid on the instance remain unchanged, while invalid ones are further refined for valid dependencies on the instance. It is guaranteed that the *complete* set of dependencies can be obtained in this manner. However, we find performing hybrid set-based OD discovery in the same way *cannot* guarantee *completeness*, as shown below.

Example 4: Consider r and r_1 in Table II and Table III, where r_1 is a sample of r, with three tuples drawn from r. In a hybrid discovery method, suppose we first perform OD discovery on r_1. After finding a minimal and valid OD $\{A\}$: [ ] $\mapsto \overline{C}$, we use it to prune *non-minimal* candidate ODs, *e.g.,* $\{A, B\}$: [ ] $\mapsto \overline{C}$ and $\{A\}$: $\overrightarrow{B} \sim \overrightarrow{C}$ during the discovery on r_1. However, $\{A\}$: [ ] $\mapsto \overline{C}$ is proven invalid when we later switch to the whole instance r. Worse still, $\{A, B\}$: [ ] $\mapsto \overline{C}$ and $\{A\}$: $\overrightarrow{B} \sim \overrightarrow{C}$ are minimal valid ODs on r. To guarantee the completeness, mistakenly pruned ODs must be recovered. $\{A, B\}$: [ ] $\mapsto \overline{C}$ can be found by adding more attributes to the context of $\{A\}$: [ ] $\mapsto \overline{C}$, *i.e.,* by refining an OD discovered on r_1 but invalid on r. In contrast, $\{A\}$: $\overrightarrow{B} \sim \overrightarrow{C}$ cannot be found in this way; note $\{\}$: $\overrightarrow{B} \sim \overrightarrow{C}$ is invalid on r_1. □

From Example 4, we can see the reason for losing completeness. If a constant OD valid on a sample is used to *prune* "non-minimal" order compatible ODs on the sample but is later proven invalid on the whole instance, then the order compatible ODs may be *mistakenly pruned* but they cannot be found out by refining the constant OD. To address this intricate

problem that is unique to set-based ODs, the hybrid strategy must be carefully redesigned.

B. A level-wise hybrid strategy

To correctly discover all minimal valid set-based ODs, we develop our method with a novel *level-wise* hybrid strategy.

Levels of ODs. Along the same lines as [32], [33], we say a constant OD in the form of $\mathcal{X}$: [ ] $\mapsto \overline{A}$ is at the *level* $|\mathcal{X}|$, and an order compatible OD in the form of $\mathcal{X}$: $\overline{A} \sim \overline{B}$ is at the *level* $|\mathcal{X}| + 1$, where $|\mathcal{X}|$ denotes the number of attributes in $\mathcal{X}$. The reason for different ways to define levels of constant and order compatible ODs lies in the observation that the minimality of a constant OD λ_c can be determined by checking every valid constant OD whose context is a proper subset of the context of λ_c, but determining the minimality of an order compatible OD λ_o needs to consider not only every valid order compatible OD whose context is a proper subset of the context of λ_o, but also valid constant ODs having the same context as λ_o.

Example 5: To determine the minimality of $\{E, C\}$: [ ] $\mapsto \overrightarrow{A}$, we need to check the validity of $\{\}$: [ ] $\mapsto \overrightarrow{A}$, $\{C\}$: [ ] $\mapsto \overrightarrow{A}$ and $\{E\}$: [ ] $\mapsto \overrightarrow{A}$. To determine the minimality of $\{E\}$: $\overrightarrow{A} \sim \overleftarrow{B}$, we must consider the validity of $\{\}$: $\overrightarrow{A} \sim \overleftarrow{B}$, $\{E\}$: [ ] $\mapsto \overrightarrow{A}$ and $\{E\}$: [ ] $\mapsto \overleftarrow{B}$. □

Intuitively, *levels* of ODs are proposed to facilitate our algorithm design, since for every (constant or order compatible) OD λ, checking valid ODs whose levels are lower than the level of λ suffices to determine the minimality of λ.

We present several theoretical results that guide the design of our discovery method. Suppose r is the given instance for OD discovery, and r' is a sample drawn from r, *i.e.,* $r' \subseteq r$. Note all the constant and order compatible ODs valid on r are also valid on r', but the reverse is not always true. In what follows, we denote by $C_k(r')$ (resp. $O_k(r')$) the complete set of minimal and valid constant (resp. order compatible) ODs at the level k on r', and by $C_k(r)$ (resp. $O_k(r)$) the complete set of minimal and valid constant (resp. order compatible) ODs at the level k on r, respectively.

ODs on the sample and ODs on the whole instance. We can see the following results concerning ODs at levels 0 and 1.

(1) $C_0(r) = \{\lambda_c \mid \lambda_c \in C_0(r') \wedge r \models \lambda_c\}$.

(2) If $C_0(r) = C_0(r')$, then $O_1(r) = \{\ \lambda_o \mid \lambda_o \in O_1(r') \wedge r \models \lambda_o\}$ and $C_1(r) = \{\ \lambda_c \mid \lambda_c \in C_1(r') \wedge r \models \lambda_c\}$.

All the non-trivial constant ODs at the level 0 are minimal, but the non-trivial ODs at the level 1 may be not minimal due to valid constant ODs at the level 0. Our results show if the valid constant ODs at the level 0 on r' are all valid on r, then we will not miss any minimal valid ODs at the level 1 on r, by checking the corresponding results on r', *i.e.,* the complete set of minimal valid constant (resp. order compatible) ODs at the level 1 on r is a subset of that on r'. The results can be naturally extended to ODs at any level k.

Proposition 2: (1) If $C_i(r') = C_i(r)$ for all $i < k$, then $C_k(r) = \{\ \lambda_c \mid \lambda_c \in C_k(r') \wedge r \models \lambda_c\ \}$.

(2) If $C_i(r') = C_i(r)$ and $O_i(r') = O_i(r)$ for all $i < k$, then $O_k(r) = \{\ \lambda_o \mid \lambda_o \in O_k(r') \wedge r \models \lambda_o\ \}$.

Proof: All minimal ODs at the level k on r' are minimal on r, since $C_i(r') = C_i(r)$ and $O_i(r') = O_i(r)$ for all $i < k$. The only difference between $C_k(r)$ and $C_k(r')$ (resp. $O_k(r)$ and $O_k(r')$) is that some ODs valid on r' may be invalid on r. □

Proposition 2 shows that all minimal valid ODs at the level k on r can be obtained by removing ODs invalid on r from the complete set of minimal valid ODs at the level k on r', if the correctness and completeness of the discovery results on r' are guaranteed for all the levels lower than k. Note the correctness and completeness of the discovery result at every level must be explicitly stated in Proposition 2, because we do not always have $C_{i-1}(r') = C_{i-1}(r)$ if $C_i(r') = C_i(r)$, or $O_{i-1}(r') = O_{i-1}(r)$ if $O_i(r') = O_i(r)$.

Example 6: Consider r and r_2 shown in Table II and Table IV. It can be verified that at the level 1, $C_1(r_2) = C_1(r) = \phi$, but at the level 0, $C_0(r) = \phi$ and $C_0(r_2) = \{\ \{\}$: [ ] $\mapsto \overline{C}\ \}$. Note $\{A\}$: [ ] $\mapsto \overline{C}$ and $\{B\}$: [ ] $\mapsto \overline{C}$ *do not* belong to $C_1(r_2)$; they are not minimal since $\{\}$: [ ] $\mapsto \overline{C}$ is valid on r_2. □

In practice it is very difficult to directly find a sample r' that meets the requirement of Proposition 2. Instead, we propose to adopt a *level-by-level* strategy to *incrementally* construct the sample. Our strategy is based on two key observations.

(1) During the discovery process, we can exploit a *dynamic* sample whose size keeps growing, rather than a static sample of the instance. This *incremental* sampling scheme is a departure from the common one-time random or focused sampling methods [4], [22].

(2) We can combine OD discovery on the sample with OD validation on the whole instance, to find the sample that meets the requirement of Proposition 2 for each level i.

Sampling tuples incrementally. Suppose at a level i, the current sample r' does not meet the requirement of Proposition 2, and we refine r' by including more tuples from r. Slightly abusing notations, we denote this new version of r' by r''. If we have $C_i(r'') = C_i(r)$ and $O_i(r'') = O_i(r)$, then we hope to use r'' as the sample on which the subsequent discovery is performed, since r'' enables Proposition 2 at the current level i. This incremental sampling scheme implies that we perform OD discovery on an ever-growing sample, and ODs at different levels may be discovered on different versions of the sample. Our observation is that this scheme does not affect the correctness of our method.

Proposition 3: Suppose $r' \subset r'' \subseteq r$. With a given $i \geq 1$, we have the following results.

(1) $C_j(r'') = C_j(r)$ for all $j < i$, if $C_j(r') = C_j(r)$ for all $j < i$.

(2) $O_j(r'') = O_j(r)$ for all $j < i$, if $C_j(r') = C_j(r)$ and $O_j(r') = O_j(r)$ for all $j < i$.

Proposition 3 states that r'' guarantees to preserve properties of r' in terms of the correctness and completeness of the discovery at all the levels $j < i$, if r'' is obtained by adding some new tuples to r' at the level i. Therefore, instead of finding a static sample for Proposition 2, we can use a dynamic sample whose size keeps growing, for the same purpose.

Algorithm 1: HyOD

Input: a relation r of schema R
Output: the complete set Σ of minimal valid set-based ODs on r

1 $\Sigma \leftarrow \emptyset$, $l \leftarrow 0$
2 $context_0{}^+ \leftarrow \{ \{A\} \mid A \in R\}$
3 $r' \leftarrow$ randomly draw some tuples from r
4 **while** $context_l{}^+ \neq \emptyset$ **do**
5 $\quad O_l(r'), C_l(r') \leftarrow$ Discover(r', l, $context_l{}^+$)
6 $\quad O_l(r), C_l(r), \Delta r \leftarrow$ Validate($O_l(r')$,$C_l(r')$, r)
7 $\quad \Sigma \leftarrow \Sigma \cup O_l(r) \cup C_l(r)$
8 $\quad r' \leftarrow r' \cup \Delta r$
9 $\quad context_{l+1}{}^+ \leftarrow$ NextLevel (l, $O_l(r)$, $C_l(r)$, $context_l{}^+$)
10 $\quad l \leftarrow l + 1$
11 **return** Σ

Refining the sample according to OD validation. A remaining issue concerns the method to refine r' at the level i. If $O_i(r') \neq O_i(r)$ and (or) $C_i(r') \neq C_i(r)$, then we can have the set V of ODs that are valid on r' but invalid on r, after validating ODs from $O_i(r')$ and $C_i(r')$ on r. Our observation is that if we select at least one violating tuple pair for each OD in V and add into r' the new tuples from the selected pairs, then we can obtain a new version r'' of r', such that $O_i(r'') = O_i(r)$ and $C_i(r'') = C_i(r)$. That is, r'' is a sample that enables Proposition 2, and is obtained by refining r' according to the validation result of $O_i(r')$ and $C_i(r')$ on r.

Example 7: Consider the instances r and r_3 in Table II and Table V. No valid constant ODs at the level 0 are discovered on r_3, and $\{\}$: $\overrightarrow{A} \sim \overrightarrow{C}$ is the only minimal valid order compatible OD at the level 1 on r_3. This OD is invalid on r, and (t_1, t_7) is a violating tuple pair *w.r.t.* the OD. We get a new sample r_4 (shown in Table VI) after adding t_1 to r_3. It can be verified that $O_1(r_4) = O_1(r)$. There are more violating tuple pairs *w.r.t.* the OD on r, *e.g.,* (t_3, t_5), but it suffices to pick only one violating tuple pair and add the new tuple(s) to r_3. □

Proposition 4: Suppose $r' \subseteq r'' \subseteq r$, and $C_j(r') = C_j(r)$ and $O_j(r') = O_j(r)$ for all $j < i$. We have $C_i(r'') = C_i(r)$ (resp. $O_i(r'') = O_i(r)$), if for every λ_c (resp. λ_o) in $C_i(r')$ (resp. $O_i(r')$) such that $r \not\models \lambda_c$ (resp. λ_o), there are tuples $t, s \in r''$ such that (t, s) is a violating tuple pair *w.r.t.* λ_c (resp. λ_o).

Proof: According to Proposition 3, $C_j(r'') = C_j(r)$ and $O_j(r'') = O_j(r)$ for all $j < i$. Hence, all minimal ODs at the level i on r'' are minimal on r. All ODs valid on r'' are also valid on r, because all ODs valid on r'' are valid on r', and for every minimal OD λ valid on r' but invalid on r, there are violations of λ in r''. □

V. A HYBRID APPROACH TO SET-BASED OD DISCOVERY

In this section, we first present our discovery method, and then enhance it with multi-threaded parallelism.

A. A hybrid set-based OD discovery algorithm

Algorithm. HyOD (Algorithm 1) finds the complete set of minimal valid set-based ODs on a given instance r. It starts with a sample r' with some tuples randomly selected from r (line 3), and adopts a hybrid strategy that switches between discovery on r' (line 5) and validation on r (line 6) in a level-by-level manner. According to Proposition 2, $O_l(r)$ and $C_l(r)$ are obtained by removing ODs invalid on r from $O_l(r')$ and $C_l(r')$. The ODs discovered at the current level l are collected in the set Σ (line 7). The sample r' is updated by including violating tuples found during the validation on r (line 8), and is used for subsequent OD discovery, by following Propositions 3 and 4. Candidate ODs at the level $l + 1$ are generated based on the discovery result at the level l (line 9).

Algorithm 2: Discover

Input: the sample r', the current level l, and the set $context_l{}^+$ for the current level
Output: $O_l(r')$ and $C_l(r')$

1 $O_l(r') \leftarrow \emptyset$, $C_l(r') \leftarrow \emptyset$
2 **foreach** $\mathcal{X} \in context_l{}^+$ **do**
3 $\quad$ **if** l = *0* **then** $\mathcal{C}_c^+(\mathcal{X}) \leftarrow R$, $\mathcal{C}_o^+(\mathcal{X}) \leftarrow \emptyset$
4 $\quad$ **else**
5 $\quad\quad \mathcal{C}_c^+(\mathcal{X}) \leftarrow \cap_{A \in \mathcal{X}} \mathcal{C}_c^+(\mathcal{X} \backslash A)$
6 $\quad\quad$ **if** l = *1* **then**
7 $\quad\quad\quad \forall_{A,B \in R^2,\ A \neq B}\ \mathcal{C}_o^+(AB) \leftarrow \{\overrightarrow{A}, \overrightarrow{B}\}$ /* $\overrightarrow{B} \in \{\overrightarrow{B}, \overrightarrow{B}\}$ */
8 $\quad\quad$ **else**
9 $\quad\quad\quad \mathcal{C}_o^+(\mathcal{X}) \leftarrow \{\{\overrightarrow{A}, \overrightarrow{B}\} \in \bigcup_{C \in \mathcal{X}} \mathcal{C}_o^+(\mathcal{X} \backslash C) \mid \forall_{D \in \mathcal{X} \backslash AB} \{\overrightarrow{A}, \overrightarrow{B}\} \in \mathcal{C}_o^+(\mathcal{X} \backslash D)\ \}$
10 **foreach** $\mathcal{X} \in context_l{}^+$ **do**
11 $\quad$ **foreach** $A \in \mathcal{X} \cap \mathcal{C}_c^+(\mathcal{X})$ **do**
12 $\quad\quad \lambda \leftarrow \mathcal{X} \backslash A$: [] $\mapsto A$
13 $\quad\quad$ **if** Check *(r', λ)* **then** $O_l(r') \leftarrow O_l(r') \cup \{\lambda\}$
14 $\quad$ **foreach** $\{\overrightarrow{A}, \overrightarrow{B}\} \in \mathcal{C}_o^+(\mathcal{X})$ **do**
15 $\quad\quad$ **if** $A \notin \mathcal{C}_c^+(\mathcal{X} \backslash B)$ *or* $B \notin \mathcal{C}_c^+(\mathcal{X} \backslash A)$ **then**
16 $\quad\quad\quad$ remove $\{\overrightarrow{A}, \overrightarrow{B}\}$ from $\mathcal{C}_o^+(\mathcal{X})$
17 $\quad\quad$ **else**
18 $\quad\quad\quad \lambda \leftarrow \mathcal{X} \backslash AB$: $\overrightarrow{A} \sim \overrightarrow{B}$
19 $\quad\quad\quad$ **if** Check *(r', λ)* **then** $O_l(r') \leftarrow O_l(r') \cup \{\lambda\}$
20 **return** ($C_l(r')$,$O_l(r')$)

In the sequel we illustrate components of HyOD.

Algorithm. Discover (Algorithm 2) computes $C_l(r')$ and $O_l(r')$ for the level l on r'. Along the similar lines as [10], [32], [33], it uses a set $context_l{}^+$ to store attribute sets from which contexts of ODs at the level l are generated. For each $\mathcal{X}$ in $context_l{}^+$, a set $\mathcal{C}_c^+(\mathcal{X})$ (resp. $\mathcal{C}_o^+(\mathcal{X})$) is further used to save RHS attributes (resp. attribute pairs) for candidate constant (resp. order compatible) ODs. Discover enumerates $\mathcal{X}$ in $context_l{}^+$, and generates candidate constant (resp. order compatible) ODs based on $\mathcal{C}_c^+(\mathcal{X})$ (resp. $\mathcal{C}_o^+(\mathcal{X})$) in line 12 (resp. 18). As an example, for $context_l{}^+ = \{\{BC\}\}$ and $\mathcal{C}_c^+(\{BC\}) = \{A, B, C, D\}$, candidates $\{B\}$: [] $\mapsto C$, and $\{C\}$: [] $\mapsto B$ are generated following line 12. Algorithm Check is called for validating candidates, and valid ones are collected in $O_l(r')$ or $C_l(r')$ (lines 13 and 19).

The computation of $context_l{}^+$ and the related $\mathcal{C}_c^+$ and $\mathcal{C}_o^+$ is the key part [10], [32], [33]. (1) At the level 0, $\forall A \in R$, $\{A\}$ belongs to $context_0{}^+$ (line 2 of HyOD), and $\mathcal{C}_c^+(\{A\}) = R$ and $\mathcal{C}_o^+(\{A\}) = \emptyset$ (line 3 of Discover). In this way, constant ODs of the form $\{\}$: [] $\mapsto A$ are initially generated for every $A \in R$ in Discover. (2) At the level 1, all the attribute pairs are enumerated for $\mathcal{C}_o^+$ (lines 6-7), resulting in order compatible

Algorithm 3: NextLevel

Input: l, $O_l(r)$, $C_l(r)$, and $context_l{}^+$
Output: $context_{l+1}{}^+$

1 $context_{l+1}{}^+ \leftarrow \emptyset$
2 **foreach** $\lambda \in C_l(r) \cup O_l(r)$ **do**
3 **if** λ *is in the form of* $\mathcal{X}$: [ ] $\mapsto \overline{A}$ **then**
4 remove A from $\mathcal{C}_c^+(\mathcal{X}A)$
5 remove all $B \in R \backslash \mathcal{X}A$ from $\mathcal{C}_c^+(\mathcal{X}A)$
6 **if** λ *is in the form of* $\mathcal{X}$: $\overrightarrow{A} \sim \overrightarrow{B}$ **then**
7 remove $\{\overrightarrow{A}, \overrightarrow{B}\}$ from $\mathcal{C}_o^+(\mathcal{X}AB)$
8 **foreach** $\mathcal{X} \in context_l{}^+$ **do**
9 **if** $l \geq 1$ *and* $\mathcal{C}_c^+(\mathcal{X}) = \emptyset$ *and* $\mathcal{C}_o^+(\mathcal{X}) = \emptyset$ **then**
10 $context_l{}^+ \leftarrow context_l{}^+ \backslash \mathcal{X}$
11 **foreach** $\mathcal{Y}B, \mathcal{Y}C \in context_l{}^+$ **do**
12 $\mathcal{X}' \leftarrow \mathcal{Y}BC$
13 add $\mathcal{X}'$ to $context_{l+1}{}^+$, if $\mathcal{X} \in context_l{}^+$ for every $\mathcal{X} \subset \mathcal{X}'$ such that $|\mathcal{X}|+1 = |\mathcal{X}'|$
14 **return** $context_{l+1}{}^+$

ODs of the form $\{\}$: $\overrightarrow{A} \sim \overrightarrow{B}$ in Discover. (3) In all the other cases, $context_l{}^+$ and the related $\mathcal{C}_c^+$ and $\mathcal{C}_o^+$ are computed based on $context_{l-1}{}^+$, so as to prune the search space.

Specifically, for any $\mathcal{X}$ in $context_l{}^+$, there must exist $\mathcal{Y}$ in $context_{l-1}{}^+$ such that $\mathcal{Y} \subset \mathcal{X}$ and $|\mathcal{Y}| + 1 = |\mathcal{X}|$. Note an attribute A belongs to $\mathcal{C}_c^+(\mathcal{Y})$ only when the validity of the corresponding constant OD, *i.e.,* $\mathcal{Y} \backslash A$: [] $\mapsto A$, is unknown, and should be removed from $\mathcal{C}_c^+(\mathcal{Y})$ if the OD is verified to be valid on r. This is because *no* minimal constant ODs can be produced by adding more attributes to $\mathcal{Y}$, *e.g.,* $\mathcal{Y}B \backslash A$: [] $\mapsto A$ is not minimal if $\mathcal{Y} \backslash A$: [] $\mapsto A$ is valid. Hence, $A \in \mathcal{C}_c^+(\mathcal{X})$ iff $A \in \mathcal{C}_c^+(\mathcal{Y})$ for all $\mathcal{Y}$, where $\mathcal{Y} \subset \mathcal{X}$ and $|\mathcal{Y}| + 1 = |\mathcal{X}|$ (line 5). Similarly for order compatible ODs and $\mathcal{C}_o^+(\mathcal{X})$ (line 9). Moreover, $A \notin \mathcal{C}_c^+(\mathcal{X}\backslash B)$ implies that $\mathcal{X}\backslash AB$: [] $\mapsto \overline{A}$ is valid, and hence $\mathcal{X}\backslash AB$: $\overrightarrow{A} \sim \overrightarrow{B}$ is not minimal by definition. Similarly for the case of $B \notin \mathcal{C}_c^+(\mathcal{X}\backslash A)$ (lines 15-16).

Just as $context_l{}^+$ is computed based on $context_{l-1}{}^+$, $context_l{}^+$ are updated to facilitate the computation of $context_{l+1}{}^+$ after ODs at the level l are discovered. Different from [32], [33], updates to $context_l{}^+$ have to be conducted after $O_l(r)$ and $C_l(r)$ are computed, since some ODs discovered on r' may be invalid on r. That is, we must update $context_l{}^+$ according to ODs valid on r. The computation of $O_l(r)$ and $C_l(r)$ is done in Algorithm Validate (line 6 of HyOD), and updates to $context_l{}^+$ are in Algorithm NextLevel (line 9 of HyOD). For ease of understanding, we first present and discuss NextLevel, since it is closely related to Discover.

Algorithm. NextLevel (Algorithm 3) is given to compute $context_{l+1}{}^+$ based on $O_l(r)$, $C_l(r)$ and $context_l{}^+$, aiming at pruning the search space by leveraging known valid ODs at the level l. Since our incremental sampling scheme guarantees the correctness and completeness of discovery results on r' for all levels $i < l$ (not only the level l), we can use all pruning rules of [10], [32], [33], as if we perform the discovery on r.

NextLevel removes attributes and attribute pairs from $\mathcal{C}_c^+$ and $\mathcal{C}_o^+$ of $context_l{}^+$ to facilitate the computation of $context_{l+1}{}^+$. We illustrate pruning rules in detail [10], [32], [33]. (1) If $\mathcal{X}$: [ ] $\mapsto \overline{A}$ is valid, then $\overline{A}$ should not be used as

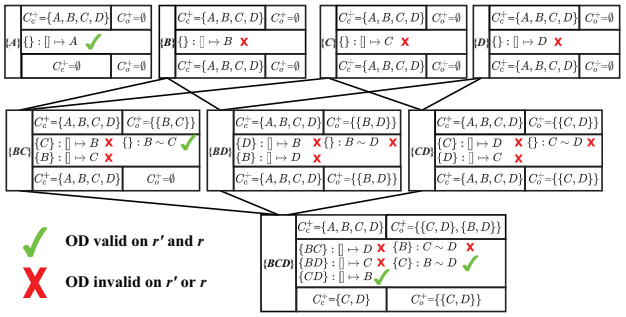


Fig. 1. Example 8 for Discover and NextLevel

the RHS attribute for any constant OD with a superset of $\mathcal{X}$ as the context (lines 3-4), since all such ODs are not minimal, *e.g.,* $\mathcal{X}\mathcal{Y}$: [ ] $\mapsto \overline{A}$ is not minimal if $\mathcal{X}$: [ ] $\mapsto \overline{A}$ is valid. Similarly for the case of order compatible OD (lines 6-7). (2) $\mathcal{X}A\backslash B = \mathcal{X}A$ for any $B \in R\backslash\mathcal{X}A$. No minimal constant ODs exist with a superset of $\mathcal{X}A$ as the context and B as the RHS attribute, if $\mathcal{X}$: [ ] $\mapsto \overline{A}$ is valid (line 5), *e.g.,* $\mathcal{X}\mathcal{Y}A$: [ ] $\mapsto \overline{B}$ *cannot* be minimal if $\mathcal{X}$: [ ] $\mapsto \overline{A}$ is valid. (3) If $\mathcal{C}_c^+(\mathcal{X}) = \emptyset$ and $\mathcal{C}_o^+(\mathcal{X}) = \emptyset$, then $\mathcal{C}_c^+(\mathcal{X}') = \emptyset$ and $\mathcal{C}_o^+(\mathcal{X}') = \emptyset$ for every superset $\mathcal{X}'$ of $\mathcal{X}$. Such $\mathcal{X}$ can be directly removed from $context_l{}^+$ (lines 8-10). (4) $\mathcal{X}' \in context_{l+1}{}^+$, iff $\mathcal{X} \in context_l{}^+$ for every $\mathcal{X} \subset \mathcal{X}'$ such that $|\mathcal{X}| + 1 = |\mathcal{X}'|$ (lines 11-13).

Example 8: We show the running of Discover and NextLevel on an instance of $R(ABCD)$ in Figure 1. All operations for a $\mathcal{X} \in context_l{}^+$ are presented in a block, and blocks are organized according to levels. For example, in the block for $\{A\} \in context_0{}^+$ (left upper corner), (1) In Discover, $\mathcal{C}_c^+(\{A\})$ = R, $\mathcal{C}_o^+(\{A\}) = \emptyset$ (row 1). (2) The constant OD $\{\}$: [] $\mapsto \overline{A}$ is generated as a candidate. It is first validated on r' in Discover, and is further validated on r in Algorithm Validate if it is valid on r' (row 2). (3) Suppose the OD is valid on r. $\mathcal{C}_c^+(\{A\})$ and $\mathcal{C}_o^+(\{A\})$ are updated accordingly in NextLevel (row 3). Blocks at level l+1 are generated and processed after all operations for blocks at level l are completed.

We illustrate some algorithm steps in detail with the figure. (1) In Algorithm NextLevel, the validity of $\{\}$: [] $\mapsto \overline{A}$ on r leads to (a) $\mathcal{C}_c^+(\{A\}) = \emptyset$ (lines 4-5), (b) the removal of $\{A\}$ from $context_0{}^+$ afterwards (lines 9-10), and (c) the absence of $\{AB\}$, $\{AC\}$ $\{AD\}$ in $context_1{}^+$ (lines 12-13). (2) In Algorithm Discover, $\mathcal{C}_c^+(\{BC\})$ in $context_1{}^+$ is computed as the intersection of $\mathcal{C}_c^+(\{B\})$ and $\mathcal{C}_c^+(\{C\})$ (line 5). $\mathcal{C}_o^+(\{BC\})$ is obtained by enumerating all possible combinations (lines 6-7). The OD $\{\}$: $\overrightarrow{B} \sim \overrightarrow{C}$ is generated as a candidate and checked for its validity; this is because $C \in \mathcal{C}_c^+(\{B\})$ and $B \in \mathcal{C}_c^+(\{C\})$ (lines 14-19). □

We then present methods for OD validations. HyOD employs two algorithms for OD validations on r' and r respectively, a departure from [32], [33]. Specifically, Algorithm Check (not shown) is called to validate every candidate OD during the discovery on r' (lines 13 and 19 of Discover), while Validate (Algorithm 4) is used to validate $C_l(r') \cup O_l(r')$ on r and generate $C_l(r)$ and $O_l(r)$ (line 6 of HyOD). The key

Algorithm 4: Validate

Input: $O_l(r')$, $C_l(r')$ and r
Output: $O_l(r)$, $C_l(r)$, and the set Δr of violating tuples *w.r.t.* $O_l(r') \cup C_l(r')$

1 $\Delta r \leftarrow \emptyset$, $C_l(r) \leftarrow \emptyset$, $O_l(r) \leftarrow \emptyset$
2 **foreach** $\lambda \in C_l(r') \cup O_l(r')$ **do**
3 *vioPairHeap* $\leftarrow$ an empty minHeap
4 *vioPair* $\leftarrow \emptyset$, $\Delta r' \leftarrow \emptyset$
5 **if** λ *is in the form of* $\mathcal{X}$*:* $[\ ] \mapsto \overline{A}$ **then**
6 **foreach** $EC_{\mathcal{X}}$ *eqc* $\in$ StriP$_{\mathcal{X}}$ **do**
7 *vioPair* $\leftarrow$ *vioPair*$\cup\{(t, t^0) \mid t \in eqc, t_A \neq t^0_A\}$
 `/* t^0 is a random tuple in eqc */`
8 **if** λ *is in the form of* $\mathcal{X}$*:* $\overrightarrow{A} \sim \overrightarrow{B}$ **then**
9 **foreach** $EC_{\mathcal{X}}$ *eqc* $\in$ StriP$_{\mathcal{X}}$ **do**
10 SortedP$_A \leftarrow$ *sort* (*eqc*, $\overrightarrow{A}$)
11 **for** i = *1 to* SortedP$_A$*.size* $-$ *1* **do**
12 $max_i \leftarrow t_i$, such that $t_i \in$ SortedP$_A[i]$ and $s \preceq_{\overline{B}} t_i$ for every $s \in$ SortedP$_A[i]$
13 **foreach** $t' \in$ SortedP$_A[i+1]$ **do**
14 **if** $t' \prec_{\overline{B}} max_i$ **then**
15 *vioPair* $\leftarrow$ *vioPair* $\cup \{(max_i, t')\}$
16 **if** *vioPair* $\neq \emptyset$ **then**
17 **foreach** *(t, s)* $\in$ *vioPair* **do**
18 *len* $\leftarrow 0$
19 **foreach** $A \in R$ **do**
20 **if** $t_A = s_A$ **then** *len* $\leftarrow$ *len* + 1
21 *vioPairHeap.push*((*t*, *s*, *len*))
22 **if** *vioPairHeap.size* $> n$ **then**
 vioPairHeap.pop()
23 **foreach** *(t, s, len)* $\in$ *vioPairHeap* **do**
24 $\Delta r' \leftarrow \Delta r' \cup \{\ t, s\ \}$
25 **if** $\Delta r' \neq \emptyset$ **then** $\Delta r \leftarrow \Delta r \cup \Delta r'$
26 **else** add λ to $C_l(r)$ or $O_l(r)$ `// λ is valid on r`
27 **return** ($O_l(r)$, $C_l(r)$, Δr);

difference between them is that Check only *checks* the validity of an OD and terminates as soon as one violation *w.r.t.* the OD is identified, while Validate also collects some violating tuples for every invalid OD, so as to refine r' according to Proposition 4. Hence, Check can be regarded as a "simplified" version of Validate. In the sequel we discuss Validate in detail.

Algorithm. Validate (Algorithm 4) is called for the validation of $C_l(r') \cup O_l(r')$ on r, and for collecting some violating tuples for invalid ODs. According to Proposition 4, collecting tuples of a single violating tuple pair for each invalid OD suffices. However, we also aim at effectively refining r', to prune invalid ODs and make the discovery result converge to the final result on r as quickly as possible. To this end, we heuristically select several violating tuples for each invalid OD.

(1) Recall a constant OD is equivalent to an FD. We adapt techniques for FD validations to validate constant ODs and collect violating tuples in Algorithm Validate. We use a data structure called *stripped partition* [10], [18], [22]. A stripped partition StriP$_{\mathcal{X}}$ *w.r.t.* $\mathcal{X}$ is a *set* of *equivalence classes*, from which singleton equivalence classes are removed (a singleton equivalence class has only one tuple). Since each OD violation concerns two tuples in the same equivalence class, it is safe to discard singleton equivalence classes. We denote an equivalence class *w.r.t.* $\mathcal{X}$ by $EC_{\mathcal{X}}$. For a constant OD $\lambda = \mathcal{X}$: $[\ ] \mapsto \overline{A}$, we randomly pick a tuple, say t^0, in each $EC_{\mathcal{X}}$ of StriP$_{\mathcal{X}}$. A violation occurs if there exists a tuple t in the same $EC_{\mathcal{X}}$ such that $t^0_A \neq t_A$. In each $EC_{\mathcal{X}}$ we collect all violating tuple pairs with t^0, as violating pairs *w.r.t.* λ (lines 5-7).

(2) For the validation of order compatible ODs, we further sort tuples in the same equivalence class of StriP$_{\mathcal{X}}$, resulting in a refined version of *sorted partitions* [12], [19]. Specifically, a sorted partition SortedP$_A$ is a *list* of equivalence classes, where equivalence classes in the list are sorted by $\overrightarrow{A}$; the equivalence class with t is before that with s if $t_A < s_A$. For an order compatible OD $\lambda = \mathcal{X}$: $\overrightarrow{A} \sim \overrightarrow{B}$, we build SortedP$_A$ for each equivalence class of StriP$_{\mathcal{X}}$ (lines 9-10). Tuples in the same $EC_{\mathcal{X}}$ of StriP$_{\mathcal{X}}$ may have different values in A, and hence each $EC_{\mathcal{X}}$ may be divided into several equivalence classes in SortedP$_A$. For any two successive equivalence classes SortedP$_A[i]$ and SortedP$_A[i+1]$ in SortedP$_A$, we identify the tuple t_i with the maximal value *w.r.t.* $\overline{B}$ in each SortedP$_A[i]$, and collect all violating tuple pairs (t_i, t') where $t' \in$ SortedP$_A[i+1]$ and $t' \prec_{\overline{B}} t_i$ (lines 11-15).

Note for each set-based OD, the number of collected violating tuple pairs is at most (and almost always much smaller than) the number $|r|$ of tuples of r. Collecting all violating tuple pairs is unnecessary and can be costly.

We then rank tuple pairs and select tuples from *top* pairs (lines 16-24). We rank every pair according to the number of attributes in which the two tuples have the same values (lines 18-20). The reason is that ODs only apply to tuples in the same equivalence class, and tuples with the same values in more attributes are more likely to be in the same equivalence classes *w.r.t.* different contexts. We use a parameter n to limit the number of tuple pairs selected for each OD. This is because selecting too many violating tuples for one OD may be redundant in their abilities to prune invalid ODs, and performing OD discovery on a small sample is more likely to be efficient. A structure of min-Heap is used to efficiently keep at most n tuple pairs (lines 21-22).

Example 9: Recall the instance r shown in Table II. StriP$_{\{B\}}$ = $\{\{t_2, t_8\}, \{t_3, t_7\}, \{t_4, t_5, t_6\}\}$; the singleton equivalence class with only t_1 is discarded. To check $\{B\}$: $[\ ] \mapsto \overline{A}$, we randomly pick a tuple from each of the three equivalence classes. Suppose we pick t_4 from $\{t_4, t_5, t_6\}$. t_4 and t_5 (resp. t_6) lead to a violation since they have different values in A. To check $\{B\}$: $\overrightarrow{A} \sim \overrightarrow{C}$, we further build sorted partitions on the three equivalence classes according to $\overrightarrow{A}$. For $\{t_3, t_7\}$, SortedP$_A$ = $[\{t_3\}, \{t_7\}]$. The value of t_3 in C is larger than that of t_7 in C, which incurs a violation. □

Complexity. Recall HyOD switches between Discover and Validate in a level-by-level manner. Each time Discover is called to handle ODs at one level, so is Validate. Both Discover and Validate have the worst-case complexity of $O(2^{|R|} \cdot |r| \cdot log(|r|))$ after processing all levels; at worst all the possible candidate ODs are enumerated in Discover and validated in both Discover and Validate, and $r' = r$. This complexity is the same as that of the set-based OD discovery method proposed in [33], known as FastOD. In practice, however, HyOD is experimentally verified to be much more

efficient than FastOD. Below we give a detailed analysis to provide insight into the efficiency of our hybrid method.

In practice Discover and Validate take $O(K' \cdot |r'| \cdot \log(|r'|))$ and $O(K \cdot |r| \cdot log(|r|))$ respectively,where $|r'|$ is the number of tuples of the final r', and K' (resp. K) is the number of ODs validated on r' (resp. r). Note $K \leq K' \leq 2^{|R|}$. Specifically, building a stripped partition (resp. a sorted partition) on r takes $O(|r|)$ (resp. $O(|r| \cdot log(|r|))$), and so are the same operations on r'. With stripped or sorted partitions, checking the validity of an OD on r' takes at most $O(|r'|)$, and validating an OD and collecting violating tuples on r take $O(|r|)$. In the same measurement, FastOD takes $O(K' \cdot |r| \cdot log(|r|))$. Note the ODs validated on r in FastOD are just the ODs validated on r' in HyOD, since our level-wise hybrid strategy and incremental sampling scheme guarantee the same pruning power as FastOD. Therefore, HyOD can significantly beat FastOD if both $\frac{|r'|}{|r|}$ and $\frac{K}{K'}$ are very small, which will be experimentally verified in Section VI (Exp-3).

Remarks. We highlight the novelty of HyOD. (1) Different from set-based OD discovery methods [29], [32], [33] directly performing discovery on r, HyOD combines discovery on r' and validation on r in a novel level-wise hybrid strategy. HyOD also presents novel techniques for selecting preferred violating tuples to enrich r'. (2) Different from hybrid methods of other dependencies that combine discovery on r' and validation on r in a simple serial manner, HyOD switches between discovery and validation and incrementally maintains r', in a level-by-level manner. HyOD also employs the traversal strategy and validation methods specific to set-based ODs.

B. Multi-threaded parallelism

We enhance our method with multi-threaded parallelism to further improve the efficiency.

To guarantee the correctness and completeness, we still follow the level-wise strategy in our parallel running mode; only operations on ODs at the same level are conducted in parallel. Breaking the limit of levels may lead to mistakenly pruned ODs, and correcting the problem often requires costly recomputation. We find OD validations on r take most of the running time, and hence benefit most from parallelism. When ODs are validated one by one (without parallelism), the stripped partition $\mathsf{StriP}_{\mathcal{X}}$ built for an OD with the context $\mathcal{X}$ can be directly reused for other ODs with the same context. With multi-threaded parallelism, using a stripped partition in one thread becomes complicated if the stripped partition is built in another thread running in parallel, due to possible read-write conflicts. In light of this, we prefer to assign candidate ODs with the same context to the same thread. Specifically, our parallel version of HyOD works as follows.

(1) The main thread enumerates all candidate ODs at the current level l, and saves them in a key-value structure with the context of each OD as the key. Note candidate ODs with *different* contexts are generated for a $\mathcal{X} \in context_l{}^+$.

(2) For each key in the key-value structure, all ODs with the key are assigned to a thread from the thread pool. In case of skewed distribution, a thread can additionally pull ODs from other threads to process after finishing all ODs assigned to it. In each thread candidate ODs are first validated on r', and only valid ones are then validated on r. Valid ODs on r and violating tuples identified in different threads are put into $O_l(r)$ (or $C_l(r)$) and Δr, with thread-safe operations.

(3) After all threads that process ODs at the level l terminate, the main thread refines r', generates $context_{l+1}{}^+$ with Algorithm NextLevel, and goes to the next level.

TABLE VII
DATASETS AND EXECUTION STATISTICS OF ALL THE METHODS (TL DENOTES THE RUNNING TIME IS BEYOND THE LIMIT OF 8 HOURS, AND ML DENOTES RUNNING OUT OF THE JVM HEAP SPACE OF 80 GB)

Dataset Properties			Number of ODs		Running Time (in seconds)			
Dataset	$\|r\|$	$\|R\|$	#const	#order	FastOD	FastOD$^+$	HyOD	HyOD$^+$
DB	250K	30	89,571	759	4,708	ML	**2,175**	**510**
FEB	500K	15	220	199	TL	TL	**232**	**72**
FLI	500K	17	77	479	ML	ML	**245**	**54**
NCV	500K	9	60	45	254	95	**33**	**13**
ALP	20K	17	0	1,414	TL	TL	**3,381**	**2,529**
FDR	250K	15	4,022	4	391	211	**201**	**103**
SAL	150K	9	25	55	42	26	**11**	**6**
WP	21K	7	24	18	1.82	0.96	**0.63**	**0.41**
Letter	20K	17	61	2,427	TL	TL	**2,539**	**579**
Adult	32K	15	78	1,405	5,105	ML	**202**	**58**
Plista	1K	35	5,235	72,861	ML	ML	**2,194**	**964**
Fuel	22K	6	2	5	3.98	2.05	**0.81**	**0.69**
NUT	90K	15	234	839	375	167	**91**	**43**
ATH	380K	12	20	142	1,360	855	**44**	**31**
EQ	610K	12	92	909	8,475	ML	**508**	**264**
VEH	30K	16	0	1,613	ML	ML	**4,667**	**1,133**

VI. EXPERIMENTAL EVALUATIONS

In this section, we conduct an experimental study to verify our approach. For reproducibility, all tested datasets and code are provided at the site https://github.com/jgszxlyh/HyOD.

A. Experimental settings

Datasets. We use a host of real-life and synthetic datasets from the repeatability page of Hasso Plattner Institute and kaggle website. The datasets from HPI can be found on the website[1], and detailed URLs for the datasets from kaggle are given on our github website[2]. Their properties are shown in Table VII; $|r|$ (resp. $|R|$) denotes the number of tuples (resp. attributes).

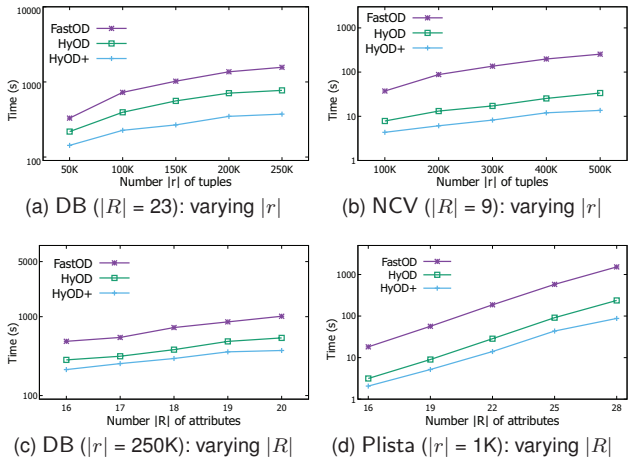
Algorithms. All the algorithms are implemented in Java.

(1) HyOD and HyOD$^+$: our set-based OD discovery method and its multi-threaded version. We randomly draw 100 tuples as the initial sample r', and set n = 10 in Algorithm Validate (at most 10 tuple pairs are selected for each violated OD). The parameters will be further studied in Exp-4 and Exp-5.

(3) FastOD [33]: the state-of-the-art set-based OD discovery method is available online[3]. In the same way as performing OD validations in parallel with HyOD$^+$, we implement a parallel version of FastOD, referred to as FastOD$^+$.

Environment. Unless otherwise stated, all the experiments are run on a machine with an Intel Xeon Bronze 3204 1.90G CPU

[1]https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html
[2]https://github.com/jgszxlyh/HyOD/tree/main/Data
[3]https://git.io/fastodbid.



(a) DB ($|R|$ = 23): varying $|r|$ (b) NCV ($|R|$ = 9): varying $|r|$

(c) DB ($|r|$ = 250K): varying $|R|$ (d) Plista ($|r|$ = 1K): varying $|R|$

Fig. 2. Scalability of all the methods

(6 physical cores), 128GB of memory and CentOS. HyOD$^+$ and FastOD$^+$ run with 6 threads by default. Each experiment is run 3 times and the average is reported.

B. Experimental results

Exp-1: Running time. The results are reported in Table VII. We terminate an algorithm after a time limit of 8 hours, and FastOD and FastOD$^+$ may fail after running out of the JVM heap space of 80 GB. We also show the number of discovered constant ODs (#const) and that of discovered order compatible ODs (#order). We see the following.

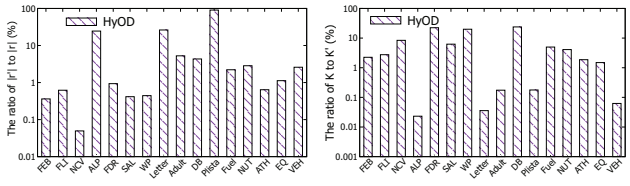
(1) HyOD significantly outperforms FastOD on all the tested datasets, up to two orders of magnitude. HyOD performs better on datasets with a wide range of $|R|$ and $|r|$, and can well accommodate datasets that differ significantly in the numbers of constant and order compatible ODs. See, *e.g.,* DB with a very large #const and Plista with a very large #order. HyOD is also more memory efficient than FastOD; FastOD runs out of the memory limit on three datasets. We conclude that HyOD is a much more efficient solution to set-based OD discovery than the state-of-the-art method FastOD.

(2) HyOD$^+$ further improves the efficiency. The speedup ratio with 6 threads is on average 2.7 times and up to 4.5 times. We will study speedup ratios of HyOD$^+$ in more detail in Exp-6.

(3) Multithreading accelerates FastOD$^+$, but also causes it to run out of memory on more datasets because multithreading typically requires more memory usage. The speedup obtained by FastOD$^+$ relative to FastOD is similar to that obtained by HyOD$^+$ relative to HyOD. However, FastOD$^+$ still cannot match HyOD on many datasets, indicating that the hybrid strategy usually brings greater efficiency gains than parallelization.

Exp-2: Scalability. In this set of experiments, we study the scalability of all methods by varying parameters $|r|$ and $|R|$. We use datasets NCV and DB since they were also employed for scalability test in previous works [12], [33], and use Plista for its largest $|R|$ and result set among all the tested datasets.

(1) We first study the impact of $|r|$ on the running time. By



(a) the ratio of $|r'|$ to $|r|$ (b) the ratio of K to K'

Fig. 3. Insights into the efficiency of HyOD

varying $|r|$ from 50K to 250K on DB with $|R|$ = 23, we report the results in Figure 2a. HyOD takes 218 to 772 seconds, as opposed to 330 to 1,565 seconds by FastOD. HyOD scales much better with $|r|$ than FastOD. HyOD$^+$ takes 143 to 372 seconds as $|r|$ increases, further improving the scalability.

We vary $|r|$ from 100K to 500K on NCV ($|R|$ = 9) and show the results in Figure 2b. Trends of the scalability are similar to those in Figure 2a. Specifically, the speedup ratio of HyOD (resp. HyOD$^+$) *w.r.t.* FastOD increases from 4.8 (resp. 8.6) to 8.0 (resp. 18.7), as $|r|$ increases.

(2) We then study the impact of $|R|$. We report the results on DB ($|r|$ = 250K) in Figure 2c by varying $|R|$. As $|R|$ increases from 16 to 20, the number of discovered ODs increases from 6,639 to 15,895, which necessarily leads to more time for all the methods. Specifically, HyOD takes 283 to 538 seconds, as opposed to 487 to 1,012 seconds by FastOD, and 213 to 373 seconds by HyOD$^+$. HyOD scales better with $|R|$ than FastOD, and HyOD$^+$ achieves similar scalability as HyOD.

Using Plista, we vary $|R|$ from 16 to 28 and report the results in Figure 2d. As $|R|$ increases, the number of discovered ODs sharply increases from 494 to 22,857. Consequently, HyOD takes 3.1 to 238 seconds, as opposed to 18 to 1,526 seconds by FastOD, and 2.1 to 88 seconds by HyOD$^+$. HyOD outperforms FastOD in terms of the scalability, and HyOD$^+$ further significantly enhances the scalability.

Exp-3: Insights into the efficiency of HyOD. For all the datasets, we show the ratio of $|r'|$ to $|r|$ and the ratio of K to K' in Figures 3a and 3b respectively, where K' (resp. K) is the number of ODs that are validated on r' (resp. r) and $|r'|$ is the number of tuples of the final r' in HyOD. As illustrated in Section V-A, these two ratios are crucial to the comparison of HyOD and FastOD. Note the ratio of K to K' also applies to datasets that FastOD fails to process within the time or memory limit, because the ODs that are (required to be) validated in FastOD are just the ODs validated on r' in HyOD.

We see the following. (1) Excluding Plista, the ratio of $|r'|$ to $|r|$ is at most 26% (on Letter) and is as low as 0.0494% (on NCV). $|r'|$ does *not* depend on $|r|$. For example, on NCV with 500K tuples, the final sample r' has only 247 tuples. The results imply that the OD discovery on r' can be much faster than that on r since r' is usually only a small fraction of r. The small ratios verify the effectiveness of our method to refine r' based on OD validations on r, which helps r' quickly adapt to the data characteristics of r. As an exception, the ratio on Plista is about 90%. This is because Plista is a small dataset with $|r|$

= 1K and has a relatively large $|R|$. We find the OD discovery on Plista reaches up to 14 levels. Nevertheless, we find more than 90% ODs discovered on Plista have already been found when the ratio of $|r'|$ to $|r|$ is less than 20% (not shown) and the ratio of K to K' is very small (shown in Figure 3b). Taken together, HyOD still significantly beats FastOD on Plista.

(2) The ratios of K to K' are in the range of [0.023%, 23.80%]. That is, for ODs validated in FastOD (ODs validated on r' in HyOD), usually a small fraction of them are validated on r in HyOD. The number of reduced validations can be dramatically large. For example on Letter, among more than 10 million ODs that are validated on r', only less than 3.6K ODs are validated on r. The small ratios also explain why HyOD is more memory-efficient than FastOD, since auxiliary structures built to validate ODs on r govern the memory usage.

Exp-4: The initial sample. HyOD employs random sampling. In Figures 4a and 4b we experimentally study different initial sample sizes. We also implement a variant leveraging focused sampling, which extracts initial sample tuples by considering data features. Specifically, it validates all ODs in the form of $\{\}$: $[] \mapsto \overline{A}$ and $\{\}$: $\overline{A} \sim \overline{B}$ on r, and terminates the validation of an OD upon finding the first violating tuple pair. It constructs a sample comprising all the identified violating tuples.

We see the following. (a) The methods randomly sampling 10 or 100 tuples and the method leveraging focused sampling perform similarly, usually with differences of less than 5%. Focused sampling does not bring performance improvements, because our incremental sampling technique also makes r' quickly adapt to the data characteristics of r. (b) The performance degrades with a relatively large sample, *e.g.,* 1,000 tuples, which justifies our strategy. It is more effective to start with a small sample and enrich it later (see Exp-5 below) than to randomly draw more tuples at the beginning. (c) Since datasets differ significantly in $|r|$, we conclude that we can use a sample size irrelevant of $|r|$, which is desirable.

Exp-5: The method to refine the sample. We verify Validate (Algorithm 4) that refines r' based on OD validations on r.

(1) In this set of experiments, we justify our strategy that ranks and selects *top* violating tuple pairs for each violated OD. We compare our strategy (denoted as *top-10*) against the strategy that selects the first violating tuple pair (denoted as *random-1*), and the strategy that randomly selects 10 violating tuple pairs (denoted as *random-10*). For all the strategies, we show the running time in Figure 4c, and the ratio of K to K' and that of $|r'|$ to $|r|$ in Figure 4d and Figure 4e, respectively. We see the following. (a) The strategy of *top-10* always achieves the best performance. The additional cost of ranking tuple pairs brings significant improvement, as shown in the comparison of strategies of *top-10* and *random-10*. The results verify the effectiveness of our method that ranks tuple pairs according to the number of attributes in which the two tuples have the same values. (b) The strategy of *random-1* aims to minimize $|r'|$. However, it falls short of the ability to prune invalid ODs, usually incurs more "false positive" ODs, *i.e.,* ODs valid on r' but invalid on r, and results in a larger ratio of K to

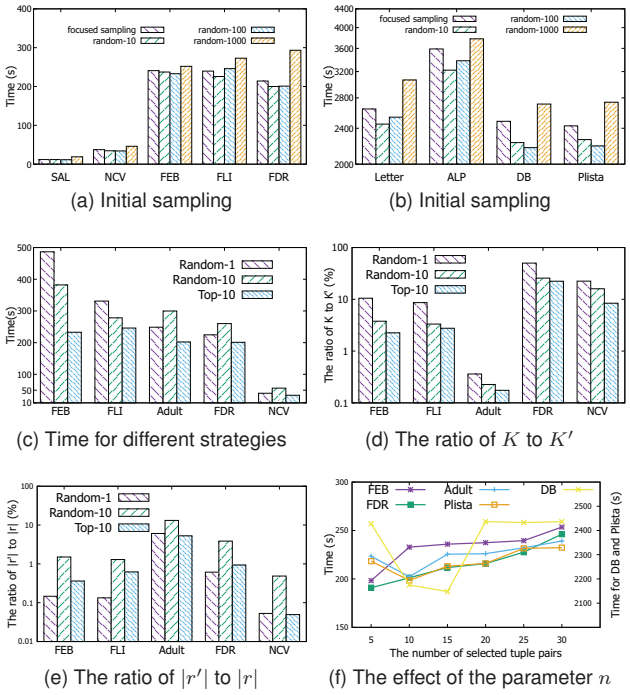


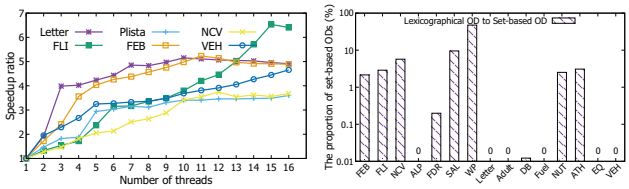
Fig. 4. Parameters and optimizations in HyOD

K' than the other two strategies. Due to the large number of "false positive" ODs, selecting one violating tuple pair for each violated OD on r and adding violating tuples to r' even cause larger ratios of $|r'|$ to $|r|$ than those incurred by the strategy of *top-10* on Adult and NCV.

(2) In this set of experiments, we study the parameter n that limits the number of violating tuple pairs selected for each OD. We vary n and report the results in Figure 4f. Note we use a different y-axis on the right for the time of DB and Plista. The results tell us that the best performance can be achieved with $n \leq 15$ on the tested datasets, and the performance starts to degrade as n further increases after the turning point. Intuitively, this is because the improvement in the ability to prune "false positive" ODs becomes less evident after many violating tuples are added to r', while the increase in $|r'|$ harms the efficiency of OD discovery on r'. We find setting n = 10 is a balanced choice, according to our experimental evaluations.

Exp-6: The speedup ratio of HyOD$^+$. We use a machine with two Intel Xeon E5-2609 V4 1.7G CPU (8 physical cores each CPU) and 64GB of memory in this set of experiments. In Figure 5a, we show the speedup ratio of HyOD$^+$ relative to HyOD as the number of threads varies from 1 to 16.

We see the following. (a) The maximum speedup ratio varies on different datasets. For example, the maximum ratio is 6.5 on FLI, and is 3.6 on Plista. (b) HyOD$^+$ reaches the maximum parallelization on some datasets before the number of threads reaches 16. The maximum parallelization mainly depends on the parallelizable part, *i.e.,* OD validations on r' and r. Since HyOD$^+$ prefers to assign ODs with the same context at the



(a) The speedup ratio of HyOD$^+$ (b) Set-based ODs for encoding lexicographical ODs

Fig. 5. Experimental results

same level to a thread, the distribution of computations among threads concerns the number of different contexts at a level. In addition, the workload of a thread is mostly determined by the number of assigned ODs that are required to be validated on r and the cost of validating each OD. (c) Speedup ratios on some datasets are relatively low. We find some forced garbage collections occur on these datasets, which hinders parallelism.

Exp-7: Set-based ODs against lexicographical ODs. We denote by T the set of set-based ODs found by HyOD, and by S the subset of T that can encode all ordering specifications stated by lexicographical ODs found by [12]. To compute S, we employ the method of [12][4] to discover lexicographical ODs, and then map them into set-based ODs with the technique of [33]. For each non-minimal set-based OD λ acquired from the mapping, we replace λ with an OD from T that prunes λ in the minimality check, *i.e.,* an OD generalizes λ. Many set-based ODs acquired from the mapping are not minimal, and some ODs are not minimal due to the same OD from T. For example, 22,434 lexicographical ODs are discovered on NUT. They are mapped to 5,910 set-based ODs, and only 27 among them are minimal. These 27 set-based ODs indeed generalize all the 5,910 ODs acquired from the mapping, and hence form the set S. The results show that set-based ODs can be more concise than lexicographical ODs [33].

The ratios of $|S|$ to $|T|$ are reported in Figure 5b, as a comparison of the expressiveness of set-based ODs found by HyOD and that of lexicographical ODs found by [12]. The ratios are usually very small and are 0 when no lexicographical ODs are found. The results show that set-based OD discovery is far more complete in finding valid ordering specifications.

Exp-8: Examples of discovered set-based ODs. In Table VIII, we showcase a few interesting order compatible ODs, and combinations of constant and order compatible ODs that can form lexicographical ODs. The output of HyOD contains all minimal valid FDs since set-based ODs subsume FDs. We do not focus on the effectiveness of constant ODs (FDs), which has been well studied in works on FD discovery [22], [44].

In Table VIII, the OD #1 on FLI states that for flights with the same *AirLineID*, *FlightNumber* and delay time, the fly time is order compatible with the actual elapsed time (the elapsed time never decreases as the fly time increases). As another example, the OD #3 on WP states that for a given country, the population is order compatible with the density of population. A lexicographical OD $\overrightarrow{Year} \mapsto \overrightarrow{WP}$ can be identified by combining ODs #4 and #5, and this lexicographical OD can be discovered by the methods of [12], [19]. In contrast, *Y, M, ES* $\mapsto$ *Y, M, CYS* acquired by combining ODs #6 and #7 *cannot* be discovered by [12], [19], since it has repeated attributes in the LHS and RHS attribute lists. It is interesting, which states an order relationship between the engine size and cylinders of vehicles of the same model produced in the same year.

[4]We use the implementation at https://github.com/chenjixuan20/BOD.

TABLE VIII
EXAMPLE SET-BASED ODS (ALL DIRECTIONS ARE IN ASCENDING ORDER)

Dataset		Discovered set-based ODs
FLI	**Abbr.**	ALI-AirLineID; FN-FlightNumber
	OD:	(1) { *ALI, FN, Delay* } : *AirTime* $\sim$ *ActualElapsedTime* (2) { *ALI* } : *Distance* $\sim$ *DistanceGroup*
WP	**Abbr.**	WP-WorldPopulation;
	OD:	(3) { *Country* } : *Population* $\sim$ *PopulationDensity* (4) { *Year* } : [] $\mapsto$ *WP* (5) { } : *Year* $\sim$ *WP* } $\Rightarrow$ *Year* $\mapsto$ *WP*
Fuel	**Abbr.**	Y-Year; M-Model; ES-EngineSize; CYS-Cylinders;
	OD:	(6) { *Y, M, ES* } : [] $\mapsto$ *CYS* (7) { *Y, M* } : *ES* $\sim$ *CYS* } $\Rightarrow$ *Y, M, ES* $\mapsto$ *Y, M, CYS*

Exp-9: Use cases. We showcase the usage of set-based ODs in query optimization. Suppose we have a user query "*Select* $*$ *From* Fuel *Where Year = 2023 and Model = GLS Order By EngineSize, Cylinders*". It can be rewritten to "... *Order By EngineSize*" and be more efficiently evaluated, according to the validity of *Y, M, ES* $\mapsto$ *Y, M, CYS* (shown in Table VIII).

It is worth mentioning that there is no need to find all valid lexicographical ODs in advance. Instead, the task can be performed *on demand*, similar in spirit to [16]. For a given query, we can identify every lexicographical OD that can help query optimization and check its validity via discovered set-based ODs, with a trivial cost irrelevant of the instance size. In this way, we can enable all query optimizations that exploit lexicographical ODs (and constant or order compatible ODs) [17], [31], [35]–[37].

VII. CONCLUSION

We have presented a set-based OD discovery method with a novel level-wise hybrid strategy. We have given the theoretical foundation of our approach, developed efficient algorithms and optimizations underlying it, enhanced it with multi-threaded parallelism, and conducted a detailed experimental study.

Query optimizations can exploit all discovered ODs, while other applications, *e.g.,* integrity maintenance, often focus on only *meaningful* or *relevant* ODs. As studied in [39], [43], selecting such dependencies from all discovered ones often ultimately needs user involvement. We intend to combine our discovery method with necessary user interactions, to meet the diverse needs of user applications.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China 62172102, 61925203 and U22B2021.

REFERENCES

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

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

[3] Johann Birnick, Thomas Bläsius, Tobias Friedrich, Felix Naumann, Thorsten Papenbrock, and Martin Schirneck. Hitting set enumeration with partial information for unique column combination discovery. *Proc. VLDB Endow.*, 13(11):2270–2283, 2020.

[4] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. Efficient denial constraint discovery with hydra. *PVLDB*, 11(3):311–323, 2017.

[5] Loredana Caruccio, Vincenzo Deufemia, Felix Naumann, and Giuseppe Polese. Discovering relaxed functional dependencies based on multi-attribute dominance. *IEEE Trans. Knowl. Data Eng.*, 33(9):3212–3228, 2021.

[6] Cristian Consonni, Paolo Sottovia, Alberto Montresor, and Yannis Velegrakis. Discovering order dependencies through order compatibility. In *EDBT*, pages 409–420, 2019.

[7] Marius Eich, Pit Fender, and Guido Moerkotte. Faster plan generation through consideration of functional dependencies and keys. *Proc. VLDB Endow.*, 9(10):756–767, 2016.

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

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

[10] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: an efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

[11] Yifeng Jin, Zijing Tan, Weijun Zeng, and Shuai Ma. Approximate order dependency discovery. In *ICDE*, pages 25–36, 2021.

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

[13] Youri Kaminsky, Eduardo H. M. Pena, and Felix Naumann. Discovering similarity inclusion dependencies. *Proc. ACM Manag. Data*, 1(1):75:1–75:24, 2023.

[14] Reza Karegar, Parke Godfrey, Lukasz Golab, Mehdi Kargar, Divesh Srivastava, and Jaroslaw Szlichta. Efficient discovery of approximate order dependencies. In *EDBT*, pages 427–432, 2021.

[15] Reza Karegar, Melicaalsadat Mirsafian, Parke Godfrey, Lukasz Golab, Mehdi Kargar, Divesh Srivastava, and Jaroslaw Szlichta. Discovering domain orders via order dependencies. In *ICDE*, pages 1098–1110, 2022.

[16] Jan Kossmann, Felix Naumann, Daniel Lindner, and Thorsten Papenbrock. Workload-driven, lazy discovery of data dependencies for query optimization. In *CIDR*, 2022.

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

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

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

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

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

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

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

[24] Eduardo H. M. Pena, Fábio Porto, and Felix Naumann. Fast algorithms for denial constraint discovery. *PVLDB*, 16(4):684–696, 2022.

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

[26] Hemant Saxena, Lukasz Golab, and Ihab F. Ilyas. Distributed implementations of dependency discovery algorithms. *PVLDB*, 12(11):1624–1636, 2019.

[27] Philipp Schirmer, Thorsten Papenbrock, Ioannis K. Koumarelas, and Felix Naumann. Efficient discovery of matching dependencies. *ACM Trans. Database Syst.*, 45(3):13:1–13:33, 2020.

[28] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. Dynfd: Functional dependency discovery in dynamic datasets. In *EDBT*, pages 253–264, 2019.

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

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

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

[32] 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.

[33] 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.

[34] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. Erratum for discovering order dependencies through order compatibility (EDBT 2019). In *EDBT*, pages 659–663, 2020.

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

[36] 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.

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

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

[39] Saravanan Thirumuruganathan, Laure Berti-Équille, Mourad Ouzzani, Jorge-Arnulfo Quiané-Ruiz, and Nan Tang. Uguide: User-guided discovery of fd-detectable errors. In *SIGMOD*, pages 1385–1397, 2017.

[40] Fabian Tschirschnitz, Thorsten Papenbrock, and Felix Naumann. Detecting inclusion dependencies on very many tables. *ACM Trans. Database Syst.*, 42(3):18:1–18:29, 2017.

[41] Yihan Wang, Shaoxu Song, Lei Chen, Jeffrey Xu Yu, and Hong Cheng. Discovering conditional matching rules. *TKDD*, 11(4):46:1–46:38, 2017.

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

[43] Ziheng Wei and Sebastian Link. Dataprof: Semantic profiling for iterative data cleansing and business rule acquisition. In *SIGMOD*, pages 1793–1796, 2018.

[44] Ziheng Wei and Sebastian Link. Discovery and ranking of functional dependencies. In *ICDE*, pages 1526–1537, 2019.

[45] Renjie Xiao, Zijing Tan, Haojin Wang, and Shuai Ma. Fast approximate denial constraint discovery. *PVLDB*, 16(2):269–281, 2022.

[46] Renjie Xiao, Yong'an Yuan, Zijing Tan, Shuai Ma, and Wei Wang. Dynamic functional dependency discovery with dynamic hitting set enumeration. In *ICDE*, pages 286–298, 2022.