Effective and Efficient Lexicographical Order Dependency Discovery

Jixuan Chen, Yifeng Jin, Yihan Li, Zijing Tan, Weidong Yang, and Shuai Ma

***Abstract*—Lexicographical order dependencies state relationships of order between lists of attributes. They naturally model the order-by clauses in SQL queries, and are proven useful in query optimizations concerning sorting. Despite their importance, order dependencies on a dataset are typically unknown and are too costly, if not impossible, to design or discover manually. Techniques for automatic order dependency discovery are recently studied. It is challenging for order dependency discovery to scale well, since it is by nature factorial in the number m of attributes and quadratic in the number n of tuples. In this article, we adopt a strategy that decouples the impact of m from that of n, and that still finds all minimal and valid lexicographical order dependencies. We present carefully designed data structures, a host of algorithms and optimizations, and an enhanced strategy combined with multithreaded parallelism, for an efficient implementation. Using a host of real-life and synthetic datasets, we experimentally verify our approach is up to orders of magnitude faster than the state-of-the-art methods, and can deliver better results with an improved definition of minimal attribute lists.**

***Index Terms*—Data dependency, data profiling, metadata.**

I. Introduction

SORTING is one of the most basic but important database operations. In a SQL statement, order for tuples is specified in the *order-by* clause. For example, the order specification *order by year asc, month desc* suggests sorting tuples by year in ascending order, and then breaking ties by month in descending order. This is known as a bidirectional lexicographical ordering, given in a list of attributes with directionality operators (asc or desc). (Bidirectional) lexicographical order dependencies (ODs) are recently proposed in [22], [24], to state a lexicographical ordering between two lists of attributes. The goal is to lay a logical foundation for order specifications. This enables formal discussions about ODs, to fulfill their potential in query optimization [11], [18], [22], [23], [24] and data quality management [3], [6], among others.

Manuscript received 10 March 2022; revised 5 February 2023; accepted 21 February 2023. Date of publication 24 February 2023; date of current version 8 August 2023. The work of Zijing Tan was supported by NSFC under Grant 62172102. The work of Weidong Yang was supported by NSFC under Grant U2033209. The work of Shuai Ma was supported by NSFC under Grants 61925203 and U22B2021. Recommended for acceptance by Dr. B. Glavic. *(Corresponding author: Zijing Tan.)*

Jixuan Chen, Yifeng Jin, Yihan Li, Zijing Tan, and Weidong Yang are with the School of Computer Science, Fudan University, Shanghai 200433, China, and also with the Shanghai Key Laboratory of Data Science, Shanghai 200433, China (e-mail: 20210240341@fudan.edu.cn; 18210240103@fudan.edu.cn; yihanli21@m.fudan.edu.cn; zjtan@fudan.edu.cn; wdyang@fudan.edu.cn).

Shuai Ma is with the SKLSDE Lab, School of Computer Science and Engineering, Beihang University, Beijing 100191, China (e-mail: mashuai@buaa.edu.cn).

Digital Object Identifier 10.1109/TKDE.2023.3248780

TABLE I
A Sample Instance

	No	Year	Month	Date	Amount
t_1	1120	2020	10	12	10K
t_2	2306	2020	10	17	11K
t_3	3100	2021	8	19	9K
t_4	3508	2022	2	15	10K

Example 1: Table I shows a relational instance. Each tuple denotes a piece of information with the following attributes: tax serial number (No), tax year (Year), month (Month), date (Date) and the tax amount (Amount). Suppose No is a unique column and values in it always increase as time goes by. When tuples are sorted by No in ascending order, we know they are also sorted by Year, Month and Date in ascending order. In the notion of OD, this is written as $\overrightarrow{\text{No}} \mapsto \overrightarrow{\text{Year}}, \overrightarrow{\text{Month}}, \overrightarrow{\text{Date}}$ (will be formalized in Section III).

This knowledge of the valid OD can be used in various data management tasks. For example, as shown in [22], [23], [24], the clause "*order by* Year *asc,* Month *asc,* Date *asc*" in a SQL query can be rewritten as "*order by* No *asc*," to well leverage an index built on No and be more efficiently evaluated. In addition, this OD can be employed to detect tuples violating the OD and guide data cleaning operations to repair error values in the tuples [3], [6].

Attribute relationships are commonly unknown in practice, which motivates the quest for dependency discovery methods, to find hidden dependencies from datasets. Discovery methods typically address two subproblems. The first is to traverse the space of all dependency candidates, and the second is to validate each candidate on the dataset. For list-based OD discovery, its search space is inevitably of permutations of attributes. It generally requires pairwise comparison of all tuple pairs to validate an OD, and hence incurs a cost quadratic in the number of tuples. Taken these together, OD discovery is by nature factorial in the number of attributes and quadratic in the number of tuples, and is hence very challenging to scale well on real-world datasets.

Inspired by recent advances in dependency discovery [1], [13], [15], we adopt a strategy to overcome the scalability limitation, by decoupling the number m of attributes from the number n of tuples. Briefly, we first perform OD discovery on a small sample r' extracted from the whole dataset r for the set Σ' of minimal valid ODs on r', with a complexity independent of n. ODs in Σ' are then validated on r, with a complexity independent of m. If some ODs are invalid on r, then the violating tuples are collected to enrich r'. We further perform discovery again on the

updated r'. The overall algorithm terminates when Σ' converges to the OD set Σ on r, i.e., when all ODs in Σ' are valid on r.

Contributions. This article presents efficient techniques for discovering lexicographical bidirectional ODs.

(1) We propose to approximate the set of ODs on a small dataset first, and then to refine the set for final complete one on the whole dataset, in an iterative way. We formalize and justify this strategy for OD discovery (Section IV).

(2) We provide a set of algorithms and optimizations underlying our approach. Specifically, we present techniques for sampling r' following the data characteristics of r (Section V-B). We give methods to discover the complete set of minimal valid ODs on r', with a strategy for traversing the space of OD candidates, a data structure ODTree for encoding our OD traversal and passing results back and forth between different phases, and techniques for efficiently checking the minimality of attribute lists to prune the search space (Section VI-C). We also develop methods for efficient batch validation of ODs on r (Section VII-A).

(3) We present an enhanced strategy to switch between OD discovery and validation in a flexible way. Combining the strategy with multithreaded parallelism, we further significantly improve the efficiency of our approach (Section VIII).

(4) We exhaustively experimentally evaluate our approach, against state-of-the-art OD discovery methods (Section IX). The results show our approach is not only up to orders of magnitude faster, but also delivers better results with an improved definition of minimal attribute lists.

II. Related Work

This work extends our conference version [9], and improves both effectiveness and efficiency of the former approach. (1) We have refined the definition of minimal attribute lists by considering valid functional dependencies (FDs) (Section III), which improves the effectiveness and does not affect the completeness. We have added several techniques to facilitate the approach. Specifically, we have justified our approach that employs FDs to prune non-minimal attribute lists (Section IV), and given methods to identify related FDs from sample r' on demand (Section VI-D) and to validate the identified FDs on instance r (Section VII-B). (2) We have added a strategy to switch between OD discovery and validation in a flexible way, and enhanced the strategy with multithreaded parallelism (Section VIII). (3) We have added completely new sets of experiments to verify the benefits of the pruning of non-minimal attribute lists with valid FDs and our flexible strategy (Section IX).

Order Dependency Foundations. A lexicographical order dependency (OD) states an ordering relationship between lists of attributes. [22] presents unidirectional ODs, in which all attributes are ordered in the same direction (ascending or descending). They are extended to bidirectional ODs in [24], for bidirectional ordering. ODs are well studied in theory and in practice. Axiomatization problems for unidirectional and bidirectional ODs are discussed in [22] and [24], respectively. ODs lend themselves as an effective solution in query optimizations with *order-by* clauses [11], [23].

Dependencies concerning *set-based* order definitions are also discussed in the literature. There are two *set-based* order dependencies, known as set-based canonical form ODs [19], [20] and pointwise ODs [4], [5], respectively. It is proved in [19] that canonical ODs generalize lexicographical ODs, in the sense that each lexicographical OD can be mapped to a set of canonical ODs and the lexicographical OD is valid iff all canonical ODs from the set are valid. Pointwise ODs further generalize canonical ODs.

Order Dependency Discovery. [12] presents the first algorithm for discovering unidirectional ODs. It enumerates permutations of attributes for OD candidates, and proposes OD validation techniques and several pruning rules to reduce the search space. The algorithm in [12] does not scale well with the number of attributes, as experimentally verified in [2], [9], [19], [20]. Another method to discover unidirectional ODs is proposed in [2]. The idea is to divide each OD into an FD and an *order compatible dependency* (OCD), and to present techniques for discovering OCDs and transforming OCDs back to ODs. However, the pruning rules of [2] may lead to incompleteness, as noted in [9], [21].

To avoid the factorial complexity of list-based ODs, [19], [20] study discovery techniques for set-based canonical form ODs, and [16] considers set-based canonical OD discovery in the distributed setting. Set-based ODs exhibit a search space exponential in the number of attributes, but the number of valid set-based ODs on a relation is often much larger than that of valid list-based ODs. This is because each list-based OD $X \mapsto Y$ is mapped to $|X|\cdot|Y|$ set-based ODs, where $|X|$ (resp. $|Y|$) is the number of attributes in list X (resp. Y). In this article we consider discovering list-based ODs, since they naturally model the lexicographical orders employed in SQL order-by clauses.

A different line of research is on *approximate* OD discovery, for ODs holding on data with some exceptions. Such techniques are presented for lexicographical ODs [7], [8] and set-based canonical ODs [10], [20], respectively. In this article, we consider discovering (exact) ODs holding on data (without exceptions). In particular, our strategy that combines OD discovery on sample data with OD validation on the whole instance applies to only exact ODs, which usually makes (exact) OD discovery far more efficient than the approximate counterpart [7].

Hybrid Approaches to Dependency Discovery. This work follows a hybrid approach to dependency discovery. The idea of approximating dependencies on sample data for further refinement has been applied for functional dependency discovery [13], [25], denial constraint discovery [1] and matching dependency discovery [15], among others. This work clearly differs from these ones. We consider bidirectional lexicographical ODs that are list-based and may have multiple RHS attributes. We develop a completely different traversal strategy, constraint validation techniques, pruning rules and optimization techniques for OD discovery.

III. Preliminaries

In this section, we review basic notations of lexicographical ODs [20], [22], [24]. We then formalize minimal ODs. In

particular, we improve the definition of minimal attribute lists from [12], by considering pruning with valid FDs.

Relation and Marked Attribute. We use common basic notations. R denotes a relational schema, and A, B, C are attributes of R. r denotes an instance of schema R, and t, s denote tuples in r. We use marked attribute $\overrightarrow{A}$ ($\overrightarrow{A}$ or $\overleftarrow{A}$), to model A asc or A desc in the SQL order-by clauses. t_A denotes the value of attribute A in t, and let $t_{\overrightarrow{A}} = t_A$.

Sets and Lists. (1) $\mathcal{X}$, $\mathcal{Y}$ denote sets of attributes, while X, Y denote lists of marked attributes. $\{\}$ (resp. [ ]) denotes the empty set (resp. empty list). $\mathcal{X}\,\mathcal{Y}$ is a shorthand for $\mathcal{X} \cup \mathcal{Y}$, and X Y is a shorthand for the concatenation of X and Y. For a list X, set $\mathcal{X}$ denotes the set of elements in X.

(2) For a list of marked attributes $\mathsf{X} = [\overrightarrow{A}, \overrightarrow{B}, \dots]$, we denote by t_X the projection of tuple t on X, i.e., $[t_A, t_B, ...]$.

(3) A non-empty list X can be expressed as $[\overrightarrow{A}\,|\mathsf{Y}\,]$, where *head* $\overrightarrow{A}$ is a single marked attribute, and *tail* Y is the remaining list by removing $\overrightarrow{A}$ from X.

(4) For $\mathsf{X} = [\overrightarrow{A_1}, \dots, \overrightarrow{A_k}]$, we use $prefix(\mathsf{X})$ to denote the set of all possible prefixes of X, i.e., $[\overrightarrow{A_1}, \dots, \overrightarrow{A_i}]$ for any $i < k$.

Order Specification. List X specifies an order between tuples, denoted as $\preceq_\mathsf{X}$. For two tuples t and s, $t \preceq_\mathsf{X} s$ iff

1) $\mathsf{X} = [\,]$; or
2) $\mathsf{X} = [\overrightarrow{A}\,|\mathsf{Y}\,]$, and $t_A < s_A$; or
3) $\mathsf{X} = [\overleftarrow{A}\,|\mathsf{Y}\,]$, and $t_A > s_A$; or
4) $\mathsf{X} = [\overrightarrow{A}\,|\mathsf{Y}\,]$, $t_A = s_A$, and $t \preceq_\mathsf{Y} s$.

$<, >, =$ are comparative operators on, e.g., numerical values, date, and time. We write $t \prec_\mathsf{X} s$ iff $t \preceq_\mathsf{X} s$ but $s \not\preceq_\mathsf{X} t$.

Bidirectional Order Dependency [20], [24]. For attributes lists X, Y, $\gamma = \mathsf{X} \mapsto \mathsf{Y}$ denotes a *bidirectional order dependency* (OD). 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 is valid on r and γ holds on r interchangeably. We write $\mathsf{X} \not\mapsto \mathsf{Y}$ if $\mathsf{X} \mapsto \mathsf{Y}$ is invalid, and write $\mathsf{X} \leftrightarrow \mathsf{Y}$ if $\mathsf{X} \mapsto \mathsf{Y}$ and $\mathsf{Y} \mapsto \mathsf{X}$.

Remarks. (1) Unidirectional OD [22] is a simplified version where all attributes are marked as asc or desc, and is considered in existing works on OD discovery [2], [12]. (2) Each bidirectional OD γ has a *symmetry* OD γ^{sym} by reversing all directions, and γ is valid iff γ^{sym} is valid, e.g., $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\,\overleftarrow{\mathsf{C}}$ and $\overleftarrow{\mathsf{A}} \mapsto \overleftarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}$. To avoid the redundancy, we always consider bidirectional ODs with asc on the leftmost attribute in the RHS attribute list, e.g., $\overrightarrow{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}\,\overleftarrow{\mathsf{C}}$.

In this article we consider the discovery of bidirectional "disjoint" ODs whose LHS and RHS attribute lists (neglecting direction) are disjoint, rather than all bidirectional ODs. The same restriction is applied to unidirectional ODs in [12]. ODs that fall into this subclass have been proven effective in query optimization [11], [22]. Arbitrarily allowing the same attributes on LHS and RHS lists may lead to less meaningful ODs, and significantly complicates OD discovery due to a much larger search space. We allow ODs with multiple attributes on both LHS and RHS. It would be too restricted to only consider ODs with an attribute on the LHS or RHS, since a list of attributes can be used in *order-by* clauses.

Violations of Order Dependency [22], [24]. Two kinds of OD violations may exist for an OD $\gamma = \mathsf{X} \mapsto \mathsf{Y}$:

TABLE II
RELATION INSTANCE r

	A	B	C	D
t_1	1	1	5	3
t_2	2	1	3	2
t_3	2	1	3	1
t_4	3	2	4	5

1) split is incurred by tuples t, s, such that $t_\mathsf{X} = s_\mathsf{X}$ but $t_\mathsf{Y} \neq s_\mathsf{Y}$.
2) swap is incurred by tuples t, s, such that $t \prec_\mathsf{X} s$ but $s \prec_\mathsf{Y} t$.

Example 2: Consider $\overleftarrow{\mathsf{C}} \mapsto \overrightarrow{\mathsf{D}}$ on Table II . Tuples t_2, t_3 have the same value in C but different values in D; they incur a split. Compared to t_1, t_2 has a smaller value in C and a smaller value in D; this leads to a swap.

It is stated in [2], [22], [24] that $\mathsf{X} \mapsto \mathsf{Y}$ can be decomposed into two dependencies $\mathcal{X} \rightarrow \mathcal{Y}$ and $\mathsf{X} \sim \mathsf{Y}$. $\mathcal{X} \rightarrow \mathcal{Y}$ is an FD; it holds if no split exists. $\mathsf{X} \sim \mathsf{Y}$ is referred to as an *order compatible dependency*; it holds if no swap exists. Obviously, $\mathsf{X} \mapsto \mathsf{Y}$ is valid iff both $\mathcal{X} \rightarrow \mathcal{Y}$ and $\mathsf{X} \sim \mathsf{Y}$ are valid.

Discovery methods usually find *minimal* valid dependencies, rather than all valid ones; see e.g., [1], [13]. A formal definition of minimal ODs concerns the *minimality* of attribute lists. Our definition of minimal attribute lists is based on the following theoretical result.

Lemma 1. (1) MYNWO $\leftrightarrow$ MYNO if $\mathcal{Y} \rightarrow \mathcal{W}$. (2) MWYO $\leftrightarrow$ MYO if $\mathsf{Y} \mapsto \mathsf{W}$ [22], [24].

Note $\mathcal{Y} \rightarrow \mathcal{W}$ in rule (1) is an FD. The correctness of (1) can be easily inferred from the *reduce order* procedure in [17]. If the values in $\mathcal{Y}$ determine those in $\mathcal{W}$, then W after Y in a list does *not* enhance the order specification of the list.

Minimal Attribute List. An attribute list X is minimal, if X is *not* pruned by either of the following two rules.

1) Forward FD rule: X is *not* minimal, if there exist disjoint subsets $\mathcal{Y}$, $\mathcal{W}$ of $\mathcal{X}$, such that in X every attribute of $\mathcal{W}$ is after all attributes of $\mathcal{Y}$, and $\mathcal{Y} \rightarrow \mathcal{W}$.
2) Backward OD rule: X is *not* minimal, if there exist disjoint sub-lists Y and W in X, such that Y directly follows W and $\mathsf{Y} \mapsto \mathsf{W}$.

Example 3: (1) Forward FD rule: if $AB \rightarrow C$, then none of $\overrightarrow{\mathsf{A}}\,\overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{C}}$, $\overrightarrow{\mathsf{B}}\,\overrightarrow{\mathsf{D}}\,\overrightarrow{\mathsf{A}}\,\overrightarrow{\mathsf{C}}$, $\overleftarrow{\mathsf{A}}\,\overleftarrow{\mathsf{D}}\,\overrightarrow{\mathsf{B}}\,\overleftarrow{\mathsf{C}}$ is minimal.

(2) Backward OD rule: if $\overleftarrow{\mathsf{A}}\,\overrightarrow{\mathsf{C}} \mapsto \overrightarrow{\mathsf{B}}$, then $\overrightarrow{\mathsf{B}}\,\overleftarrow{\mathsf{A}}\,\overrightarrow{\mathsf{C}}$ is not minimal. Note that $\overleftarrow{\mathsf{B}}\,\overrightarrow{\mathsf{A}}\,\overleftarrow{\mathsf{C}}$ is also not minimal in this case, because $\overleftarrow{\mathsf{B}}\,\overrightarrow{\mathsf{A}}\,\overleftarrow{\mathsf{C}}$ is minimal iff $\overrightarrow{\mathsf{B}}\,\overleftarrow{\mathsf{A}}\,\overrightarrow{\mathsf{C}}$ is minimal.

Remarks. (1) Our definition of minimal attribute lists improves that proposed in [12] (adopted in [9], [12]). We use the same backward OD rule, but replace the original forward OD rule with FD rule. The forward OD rule from [12] states that an attribute list X is *not* minimal, if there exist disjoint sub-lists Y and W in X, such that W follows (maybe not directly) Y and $\mathsf{Y} \mapsto \mathsf{W}$. It is easy to see $\mathcal{Y} \rightarrow \mathcal{W}$ if $\mathsf{Y} \mapsto \mathsf{W}$, and hence our forward FD rule can prune all non-minimal lists that can be pruned by the forward OD rule, but the reverse is not true. Better still, since FD is set-based, attributes of $\mathcal{Y}$ (resp. $\mathcal{W}$) are not required to be a *sublist* (continuous), and directions of attributes are also irrelevant (recall Example 3).

(2) Our experimental evaluations (Section IX) demonstrate the forward FD rule is far more applicable in practice, and typically helps prune a large number of ODs. This significantly improves the effectiveness and *does not* affect the completeness of OD discovery. Intuitively, if a valid OD is pruned due to non-minimal attribute lists, then it is guaranteed that another valid OD exists with more concise attribute lists and exactly the same order specification.

Minimal OD [12]. An OD $X \mapsto Y$ is minimal, iff

1) for any list $X' \in prefix(X)$, $X' \not\mapsto Y$,
2) for any non-empty list Y' such that YY' is a minimal attribute list, $X \not\mapsto YY'$, and
3) X and Y are minimal attribute lists.

IV. Solution Framework

In this section, we first formalize and justify our framework for OD discovery, and then present the main algorithm.

OD Discovery. We adopt the following framework for finding the complete set Σ of minimal valid ODs on a relation instance r, consisting of three phases in an iterative way.

1) Sample a small subset $r' \subseteq r$.
2) Find the complete set Σ' of minimal valid ODs and the set Ψ of valid FDs on r'. An FD belongs to Ψ iff it is employed to prune non-minimal attribute lists according to the forward FD rule, in the process of computing Σ'.
3) If all ODs from Σ' and all FDs from Ψ are valid on r, then $\Sigma = \Sigma'$, is the complete set of minimal valid ODs on r, and the OD discovery terminates. Otherwise, choose a non-empty subset $\triangle r \subseteq r \backslash r'$, add $\triangle r$ to r', i.e., $r' = r' \cup \triangle r$, goto (2) and continue with the updated r'.

Observe that this approach always terminates within a finite number of rounds. Since the size of r' monotonically increases in the iteration, the termination is guaranteed in the worst case when r' grows to r. The correctness can be seen from the following proposition.

Proposition 2. Suppose $r' \subseteq r$, Σ' is the complete set of minimal valid ODs and Ψ is a set of valid FDs on r'. Ψ contains all and only FDs that are used to prune non-minimal attribute lists according to the forward FD rule, in the process of computing Σ'. If all ODs from Σ' and all FDs from Ψ are valid on r, then Σ' is the complete set of minimal valid ODs on r.

Proof. (1) *Validity*. It is known all ODs in Σ' are valid on r.

(2) *Minimality*. Suppose an OD $\gamma \in \Sigma'$ is valid but not minimal on r. We can always find a minimal valid OD δ on r that logically implies γ, according to the three cases in the definition of minimal ODs (Section III). δ is also valid on $r' \subseteq r$, and hence γ is not minimal on r'. This contradicts the assumption that Σ' contains minimal ODs on r'.

(3) *Completeness*. Suppose γ is minimal and valid on r, but $\gamma \notin \Sigma'$. Since Σ' is the complete set of minimal valid ODs on r' and γ is valid on $r' \subseteq r$, we know γ is not minimal on r' if $\gamma \notin \Sigma'$. γ *cannot* have non-minimal attribute lists, because all ODs and FDs that are employed to prune non-minimal attribute lists on r' are all valid on r. Just as case (2) stated above, we can find δ that is minimal and valid, and that logically implies γ on r'. δ is also valid on r, since Σ' is complete and all ODs from Σ' are valid on r. This contradicts the assumption that γ is minimal on r.

Algorithm 1: FindOD.

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

```
1 Σ' ← ∅, Ψ ← ∅;
2 r' ←Sample (r);
3 while true do
4 |   Σ', Ψ ← Discover (r');
5 |   △r ←Validate (r, Σ', Ψ);
6 |   if △r = ∅ then
7 |   |   return Σ';
8 |   r' ←r' ∪ △r;
```

Algorithm. We present our OD discovery algorithm FindOD (Algorithm 1). It first samples r' from r by Algorithm Sample (line 2). The complete set Σ' of minimal valid ODs and the set Ψ of FDs used to prune non-minimal attribute lists are discovered on r' by Algorithm Discover (line 4). ODs from Σ' and FDs from Ψ are validated on the whole relation r by Algorithm Validate (line 5). If no violation exists, then the complete set Σ of minimal valid ODs on r is found (lines 6-7). Otherwise, some violating tuples are added to r' (line 8), and the next round of iteration starts.

As will be illustrated in the following sections, performing OD discovery with the hybrid strategy introduces many new challenges, due to the completely different search space traversal strategy and dependency validation method. We present a new sampling method for OD discovery, perform a batch validation of ODs from upper levels to enable pruning as early as possible, label the nodes already validated on r in our search tree to avoid redundant computation, and discover ODs on the refined sample from scratch to guarantee the correctness and completeness of the results. We also combine the validation of FDs used by the forward FD rule in our hybrid strategy. All of these operations are completely different from those for other dependencies.

V. Sampling

In this section, we present techniques to sample a dataset $r' \subseteq r$, on which the primitive set Σ' of ODs is discovered. We expect a computationally inexpensive sampling method, and r' should adapt to the data characteristics of r, such that Σ' can quickly converge to the OD set Σ on r. In our method, we model data characteristics as *violations*, i.e., swap and split. The principle behind this is that the search space of ODs can be effectively pruned by collecting in r' those tuples that cause violations. To this end, we first present methods to efficiently identify OD violations (Section V-A), and then give our sampling method (Section V-B).

A. Validation of an OD

To check the validity of an OD, we use the data structure of *sorted partition* [12]. A sorted partition τ_X on an attribute list X is a sorted list of *equivalence classes* based on tuple values in X. Tuples in the same equivalence class agree on their values

on X, while for tuples t, s in different equivalence classes, $t \prec_X s$ if the equivalence class with t is before the equivalence class with s in the sorted partition. To reduce memory footprint, only tuple *ids* are stored in the sorted partition. For example, in Table II we have $\tau_{\overrightarrow{A}} = [\{1\}, \{2,3\}, \{4\}]$ and $\tau_{\overleftarrow{C}} = [\{1\}, \{4\}, \{2,3\}]$.

At first, $\tau_{\overline{A}}$ is built for each attribute A and an auxiliary structure is built for each $\tau_{\overline{A}}$ that maps the tuple id of every tuple to the index (id) of the equivalence class in $\tau_{\overline{A}}$ containing the tuple. It takes $O(n \cdot \log(n))$ to build $\tau_{\overline{A}}$ and $O(n)$ to build the mapping structure, respectively.

Expand and Check are operations on sorted partitions.

Expand**.** When a marked attribute $\overline{A}$ is appended to an attribute list X, τ_X is expanded to $\tau_{X\overline{A}}$. Tuples from the same equivalence class of τ_X may be divided into several equivalence classes in $\tau_{X\overline{A}}$ according to their values in A, while the order of tuples in different equivalence classes of τ_X is not affected. $\tau_{X\overline{A}}$ can be efficiently computed from τ_X leveraging the mapping structure built for $\tau_{\overline{A}}$. For tuples t, s in the same equivalence class of τ_X, in $\tau_{X\overline{A}}$ the equivalence class with t is before that with s, if the equivalence class id of t is smaller than that of s in $\tau_{\overline{A}}$. In this way visiting the values of A is avoided in the computation of $\tau_{X\overline{A}}$.

Example 4: Consider $\overrightarrow{\mathsf{B}} \mapsto \overrightarrow{\mathsf{A}}$ on Table II. $\tau_{\overrightarrow{B}} = [\{1,2,3\},\{4\}]$ and $\tau_{\overrightarrow{A}} = [\{1\},\{2,3\},\{4\}]$. For $\overrightarrow{\mathsf{B}}\overleftarrow{\mathsf{C}} \mapsto \overrightarrow{\mathsf{A}}$, $\tau_{\overrightarrow{B}}$ is expanded to $\tau_{\overrightarrow{B}\overleftarrow{C}} = [\{1\},\{2,3\},\{4\}]$.

Complexity. A sorted partition on a list can be created from scratch by a series of Expand, each time for a single attribute from left to right in the list. Expand has a worst-case complexity of $O(n \cdot \log(n))$; n is the number of tuples.

Check**.** Sorted partitions τ_X, τ_Y can be used to validate the state of an OD $\mathsf{X} \mapsto \mathsf{Y}$ efficiently. The state is one of valid, swap, or split (split and no swap violation).

Algorithm. Check (Algorithm 2) is given to validate $\mathsf{X} \mapsto \mathsf{Y}$, for its state (valid, split or swap). For each tuple, we first store in an auxiliary array m[] its class index in τ_Y (lines 3-5). For each equivalence class i in τ_X, we then find two tuples with the maximal and minimal class index in τ_Y (choose the first tuple if equal) in lines 7-8. If these two tuples are not the same, then there is a split (lines 9-10). Check continues to find possible swap violations. As will be seen in Section VI-C, this is because swap outperforms split in the pruning power for OD discovery. Check identifies a swap violation, if the minimal value of some class i is smaller than the maximal value of class i-1 (lines 12-13). When there are violations, Check collects two tuples leading to split in line 11, which may be replaced by two tuples leading to swap in line 14.

Example 5: Suppose we have $\tau_X = [\{1,2\},\{3,4\},\{5\}]$, $\tau_Y = [\{1,2,3,5\},\{4\}]$. We can see tuples 1, 2 are in the same equivalence class of τ_Y. Therefore, the first class of τ_X does not introduce any violations. In the second class of τ_X, max[2] = m[4] = 2, while min[2] = m[3] = 1; we get a split caused by tuples 3, 4. In the third class of τ_X, we find (min[3] = m[5] = 1) < (max[2] = m[4] = 2), which indicates a swap. Since swap outperforms split, the final state is set to be swap and we get two violating tuples 4 and 5.

Complexity. Given sorted partitions τ_X,τ_Y, Check has a complexity linear in n, where n is the number of tuples.

Algorithm 2: Check.

Input: sorted partition τ_X,τ_Y
Output: state of $\mathsf{X} \mapsto \mathsf{Y}$, and two tuples leading to swap or split (when no swap) if $\mathsf{X} \mapsto \mathsf{Y}$ is invalid

```
1  state ← valid;
2  vioTuples ← ∅;
3  for i=1 to τ_Y.size do
4      foreach t in τ_Y [i] do
5          m[t]←i;
6  for i=1 to τ_X.size do
7      max[i]← t, m[t] is maximal among all t in τ_X[i];
8      min[i]← t, m[t] is minimal among all t in τ_X[i];
9      if state=valid and max[i]≠min[i] then
10         state ← split;
11         vioTuples ← {max[i], min[i]};
12     if min[i]<max[i-1] then
13         state ← swap;
14         vioTuples ← {min[i], max[i-1]};
15         break ;
16 return (state, vioTuples);
```

B. Sampling Method

Algorithm. Sample adopts a mixed strategy of random sampling and focused sampling. (1) It randomly chooses a small number of tuples from the full dataset, denoted as r_1. (2) On r_1, Sample checks the validity of all ODs in the form of $\overline{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$ with Algorithm Check (Section V-A), where $A, B \in R$. Sample collects as r' all violating tuples returned by Check. Recall that Check returns two tuples for each violated OD.

Complexity. The complexity of Sample is $O(|R|^2 \cdot |r_1| \cdot \log(|r_1|))$, where $|R|$ is the number of attributes and $|r_1|$ is the number of tuples in sample r_1. The number of ODs in the form of $\overline{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$ is $2 \cdot |R| \cdot (|R| - 1)$, and it takes $|r_1| \cdot log(|r_1|)$ to check a single OD with sorted partitions.

In our implementation, we randomly pick 1% of r as r_1 (we increase the ratio on those very small datasets to collect at least 50 tuples for r_1). Our experimental studies show that the sampling strategy works sufficiently well.

VI. Order Dependency Discovery

In this section, we perform OD discovery on the sample instance r'. We first give an efficient traversal strategy for generating OD candidates (Section VI-A), which is implemented with a data structure, namely ODTree (Section VI-B). We then present an algorithm to find the set Σ' of minimal valid ODs on r' (Section VI-C), with techniques for the minimality check of attribute lists, leveraging FDs and ODs (Section VI-D).

A. Traversal of OD Candidates

OD traversal is to enumerate all OD candidates, a necessary step for discovery algorithms. Suppose we want to enumerate more OD candidates, after handling an OD $\gamma = \mathsf{X} \mapsto \mathsf{Y}$. The baseline method is to generate ODs following γ by appending $\overline{A}$ ($\overrightarrow{\mathsf{A}}$ or $\overleftarrow{\mathsf{A}}$) to either X or Y for all $A \in R \backslash XY$, i.e., $\mathsf{X} \mapsto \mathsf{Y}\overline{\mathsf{A}}$ or $\mathsf{X}\overline{\mathsf{A}} \mapsto \mathsf{Y}$. As an example, with schema $R(ABCD)$, there

are 8 ODs following $\overrightarrow{B} \mapsto \overrightarrow{A}$: $\overrightarrow{B}\,\overrightarrow{C} \mapsto \overrightarrow{A}$, $\overrightarrow{B}\,\overleftarrow{C} \mapsto \overrightarrow{A}$, $\overrightarrow{B}\,\overrightarrow{D} \mapsto \overrightarrow{A}$, $\overrightarrow{B}\,\overleftarrow{D} \mapsto \overrightarrow{A}$, $\overrightarrow{B} \mapsto \overrightarrow{A}\,\overrightarrow{C}$, $\overrightarrow{B} \mapsto \overrightarrow{A}\,\overleftarrow{C}$, $\overrightarrow{B} \mapsto \overrightarrow{A}\,\overrightarrow{D}$ and $\overrightarrow{B} \mapsto \overrightarrow{A}\,\overleftarrow{D}$.

We aim to find minimal valid ODs. According to the validation of $X \mapsto Y$ (recall the result is one of valid, swap, and split (split and no swap violation)), some OD candidates *cannot* be valid or minimal, as shown in the following rules:

1) If $X \mapsto Y$ is valid, then $XZ \mapsto Y$ is not minimal. This follows from the definition of minimal ODs (Section III).
2) If $X \mapsto Y$ causes a split, then $X \mapsto YZ$ is invalid. For tuples t, s that lead to a split *w.r.t.* $X \mapsto Y$, $t_X = s_X$, but $t_Y \neq s_Y$. Obviously $t_{YZ} \neq s_{YZ}$, and t, s cause a split *w.r.t.* $X \mapsto YZ$.
3) If $X \mapsto Y$ causes a swap, then $XU \mapsto YZ$ is invalid [12], [24]. For tuples t, s that lead to a swap *w.r.t.* $X \mapsto Y$, $t \prec_X s$ but $s \prec_Y t$. Obviously, $t \prec_{XU} s$ but $s \prec_{YZ} t$, and hence t, s cause a swap *w.r.t.* $XU \mapsto YZ$.

Algorithm. ExtendOD takes as input an OD $\gamma = X \mapsto Y$, and generates OD candidates with an attribute $\overline{A}$ ($\overrightarrow{A}$ or $\overleftarrow{A}$) appended to either X or Y for all $A \in R \backslash XY$, according to the validation of $X \mapsto Y$ as follows:

1) valid: it generates all OD candidates of the form $X \mapsto Y\overline{A}$.
2) split and no swap: it generates all OD candidates of the form $X\overline{A} \mapsto Y$.
3) swap: it generates no ODs.

Example 6: Consider r shown in Table II. (1) $\overrightarrow{B} \mapsto \overrightarrow{A}$ leads to a split (no swap) on r. ExtendOD generates four ODs from it: $\overrightarrow{B}\,\overrightarrow{C} \mapsto \overrightarrow{A}$, $\overrightarrow{B}\,\overleftarrow{C} \mapsto \overrightarrow{A}$, $\overrightarrow{B}\,\overrightarrow{D} \mapsto \overrightarrow{A}$ and $\overrightarrow{B}\,\overleftarrow{D} \mapsto \overrightarrow{A}$. (2) Consider $\overrightarrow{B}\,\overleftarrow{C} \mapsto \overrightarrow{A}$ valid on r. ExtendOD generates two ODs from it: $\overrightarrow{B}\,\overleftarrow{C} \mapsto \overrightarrow{A}\,\overrightarrow{D}$ and $\overrightarrow{B}\,\overleftarrow{C} \mapsto \overrightarrow{A}\,\overleftarrow{D}$. (3) ExtendOD generates no ODs from $\overrightarrow{C} \mapsto \overrightarrow{A}$, due to swap violations on r.

We call OD δ a *child* of OD γ, if δ is obtained by calling ExtendOD with γ as the input. We say δ is generated from γ, if there is a list of ODs $[\beta_0 = \gamma, ..., \beta_k = \delta]$ such that β_i is a child of β_{i-1}, where $i \in [1, k]$. Recall Example 6. $\overrightarrow{B}\,\overleftarrow{C} \mapsto \overrightarrow{A}\,\overrightarrow{D}$ is a child of $\overrightarrow{B}\,\overleftarrow{C} \mapsto \overrightarrow{A}$, which is in turn a child of $\overrightarrow{B} \mapsto \overrightarrow{A}$. Hence, we say $\overrightarrow{B}\,\overleftarrow{C} \mapsto \overrightarrow{A}\,\overrightarrow{D}$ is generated from $\overrightarrow{B} \mapsto \overrightarrow{A}$.

Traverse the Space of OD Candidates. From an OD with a single attribute on the LHS (resp. RHS), i.e., $\overline{A} \mapsto \overrightarrow{B}$ ($\overline{A} \mapsto \overleftarrow{B}$ is ignored due to symmetry (Section III)), ODs of the form $\overline{A}X \mapsto \overrightarrow{B}Y$ are generated. Obviously, all OD candidates are enumerated, by generating ODs from all ODs of the form $\overline{A} \mapsto \overrightarrow{B}$ ($B \in R$, $A \in R \backslash B$).

Distinct ODs are generated from $\overline{A} \mapsto \overrightarrow{B}$ and $\overline{A'} \mapsto \overrightarrow{B'}$ if $\overline{A}$ differs from $\overline{A'}$ in attributes or directions, and (or) B and B' are different. Without loss of generality, in the sequel we consider ODs generated from $\overline{A} \mapsto \overrightarrow{B}$. Before studying properties of ExtendOD in detail, we present several notations.

Ancestor ODs and a Layered Structure. Given $\gamma = \overline{A}X \mapsto \overrightarrow{B}Y$, we call ODs of the form $\overline{A}X' \mapsto \overrightarrow{B}Y'$ ancestor ODs of γ, if (1) X' is a prefix of X, and Y' is a prefix of Y or Y; or (2) Y' is a prefix of Y, and X' is a prefix of X or X. OD γ and its ancestors form a *layered structure*, denoted by L_γ. We say $\overline{A}X' \mapsto \overrightarrow{B}Y'$ is at layer i if $|X'|+|Y'| = i$, where $|X'|$ (resp. $|Y'|$) is the number of attributes in list X' (resp. Y'). As an example, we show the layered structure of $\overrightarrow{A}\,\overrightarrow{C} \mapsto \overrightarrow{B}\,\overrightarrow{D}\,\overrightarrow{E}\,\overrightarrow{F}$ and its ancestors in Fig. 1.

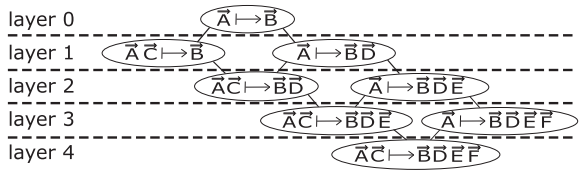


Fig. 1. A layered structure.

ExtendOD is indeed efficient, in the sense that it never generates *duplicate* ODs.

Proposition 3. If $\gamma = \overline{A}X \mapsto \overrightarrow{B}Y$ is generated, then there is a unique list [$\delta_0 = \overline{A} \mapsto \overrightarrow{B}$,..., $\delta_k = \overline{A}X \mapsto \overrightarrow{B}Y$], such that δ_i is a child of δ_{i-1}, where $i \in [1, k]$ and $k = |X|+|Y|$.

Example 7: $\overrightarrow{A}\,\overrightarrow{C} \mapsto \overrightarrow{B}\,\overrightarrow{D}\,\overrightarrow{E}$ in Fig. 1 may be generated on an instance as follows: $\overrightarrow{A} \mapsto \overrightarrow{B}$ is valid; $\overrightarrow{A} \mapsto \overrightarrow{B}\,\overrightarrow{D}$ causes a split; $\overrightarrow{A}\,\overrightarrow{C} \mapsto \overrightarrow{B}\,\overrightarrow{D}$ is valid. Note that $\overrightarrow{A}\,\overrightarrow{C} \mapsto \overrightarrow{B}\,\overrightarrow{D}\,\overrightarrow{E}$ *cannot* be generated from $\overrightarrow{A} \mapsto \overrightarrow{B}\,\overrightarrow{D}\,\overrightarrow{E}$ on the instance. This is because $\overrightarrow{A} \mapsto \overrightarrow{B}\,\overrightarrow{D}$ cannot simultaneously have $\overrightarrow{A}\,\overrightarrow{C} \mapsto \overrightarrow{B}\,\overrightarrow{D}$ and $\overrightarrow{A} \mapsto \overrightarrow{B}\,\overrightarrow{D}\,\overrightarrow{E}$ as children on any instance.

The following proposition states the completeness of ExtendOD. Please refer to [9] for the detailed proof.

Proposition 4. If $\gamma = \overline{A}X \mapsto \overrightarrow{B}Y$ is a candidate of minimal valid OD, then γ is generated from $\overline{A} \mapsto \overrightarrow{B}$.

B. Data Structure ODTree

We present an effective tree data structure, referred to as ODTree, for encoding OD candidates in ExtendOD.

ODTree. Each node of ODTree has the following fields:

1) a marked attribute;
2) state: valid, or swap, or split (split and no swap violation);

ODTree is implemented as a prefix tree, and each node p represents an OD candidate, denoted by γ_p. In what follows, we interchangeably use *node* p and its corresponding OD candidate γ_p when they are clear from the context. γ_p is constructed following the path from the root to p, as follows:

1) the root node corresponds to a trivial OD [ ]$\mapsto$[ ];
2) for a node p with node p' as its parent node,
   a) if the state of p' is valid, then γ_p is obtained by appending the marked attribute in p to the RHS list of $\gamma_{p'}$;
   b) if the state of p' is split, then γ_p is obtained by appending the marked attribute in p to the LHS list of $\gamma_{p'}$.

An ODTree is initialized as follows: (1) the root has state valid. (2) For each $B \in R$, a node is generated with $\overrightarrow{B}$ as its marked attribute and state split at level 2 (child of the root). This node denotes an OD [ ] $\mapsto \overrightarrow{B}$; each attribute at level 2 becomes the first RHS attribute since the state of root is valid. Recall that we consider ODs whose leftmost attributes on RHS are labeled asc, due to symmetry. (3) For each node p with $\overrightarrow{B}$ at level 2 and each attribute $A \in R \backslash B$, two child nodes of p are generated, with $\overrightarrow{A}$ and $\overleftarrow{A}$ as their marked attributes respectively, denoting

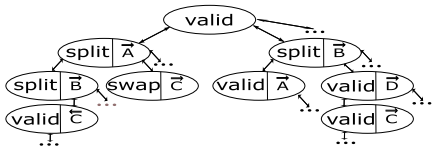


Fig. 2. Example ODTree.

Algorithm 3: Discover.

Input: a sample relation r' of schema R, ODTree *tree*
Output: ODTree *tree*

```
1  q ← an empty queue;
2  q.enqueue(tree.root);
3  while q ≠ ∅ do
4  |  parent ← q.dequeue();
5  |  create children of parent by ExtendOD;
6  |  foreach child in parent.children do
7  |  |  if not CheckMinimality (child) then
8  |  |  |  continue;
9  |  |  τ_X,τ_Y ← sorted partitions of LHS, RHS of child;
10 |  |  child.state ← Check (τ_X,τ_Y);
11 |  |  if child.state ≠ swap then
12 |  |  |  q.enqueue(child);
13 |  |  |  add child to tree, as a child of parent;
14 return tree;
```

$\overrightarrow{A} \mapsto \overrightarrow{B}$ and $\overleftarrow{A} \mapsto \overrightarrow{B}$. Each attribute at level 3 becomes the first LHS attribute, since states of all nodes at level 2 are split.

Example 8: We show a partial ODTree in Fig. 2. Path $\overrightarrow{A}\,\overrightarrow{B}$ denotes $\overrightarrow{B} \mapsto \overrightarrow{A}$, which causes a split on r in Table II. Path $\overrightarrow{A}\,\overrightarrow{B}\,\overleftarrow{C}$ denotes $\overrightarrow{B}\,\overleftarrow{C} \mapsto \overrightarrow{A}$ valid on r. Path $\overrightarrow{A}\,\overrightarrow{C}$ denotes $\overrightarrow{C} \mapsto \overrightarrow{A}$; it has no children due to a swap violation on r.

C. Algorithm for OD Discovery

We present Algorithm Discover for the complete set Σ' of minimal valid ODs, by following our OD traversal strategy and employing techniques to prune non-minimal ODs.

Algorithm. Discover (Algorithm 3) puts the root of ODTree into an empty queue to start a breadth-first-search (BFS) traversal of OD candidates (lines 1-2). For each node (OD) in the queue, Discover creates its children by calling ExtendOD (Section VI-A) in line 5 (ODs of the form [ ] $\mapsto \overrightarrow{B}$ at level 2 and $\overline{A} \mapsto \overrightarrow{B}$ at level 3 in ODTree are also generated). Algorithm CheckMinimality is employed to check the minimality of LHS and RHS attribute lists of each child (line 7), with the forward FD rule and backward OD rule. More details of CheckMinimality will be given in Section VI-D. Non-minimal ODs are skipped (lines 7-8). Based on the sorted partitions of both sides of an OD, the OD is checked with Algorithm Check (Section V-A) (lines 9-10). All nodes except swap nodes are enqueued and encoded in the ODTree (lines 11-13).

Proposition 5. Discover finds the complete set Σ' of minimal valid ODs on r'.

Proof. ExtendOD is employed to build the ODTree, for generating all candidates of minimal valid ODs (Proposition 4). Discover prunes non-minimal ODs due to non-minimal attribute lists. If $X \mapsto Y$ contains a non-minimal attribute list, i.e., X or Y, then neither $X \mapsto Y\overline{A}$ nor $X\overline{A} \mapsto Y$ is minimal. Hence, the pruning does not affect the completeness. In the ODTree, ODs in Σ' are the nodes (a) labeled with valid, and (b) having *no* valid children. This is because (a) all nodes labeled with valid in the ODTree are validated to be valid on r'; and (b) a valid node $X \mapsto Y$ only has children of the form $X \mapsto Y\overline{A}$, and $X \mapsto Y$ is *not* minimal if $X \mapsto Y\overline{A}$ is minimal and valid (case (2) in the definition of minimal ODs).

Complexity. The search space of ODs whose LHS and RHS lists are disjoint is $O(|R|!)$, where $|R|$ is the number of attributes. The worst-case complexity of Discover is $O(|R|! \cdot |r'| \cdot log(|r'|))$, where $|r'|$ is the number of tuples in r'.

The worst-case complexity of a dependency discovery method is usually measured by the search space [1], [12], [13], [19], [20], i.e., the total number of candidate dependencies. The efficiency of a dependency discovery method, however, is mostly determined by the actual search space, i.e., the number of dependencies that are really validated during the discovery, since in practice many (maybe most) of the candidate dependencies can be pruned during the discovery. Algorithm Discover itself significantly improves the OD discovery method given in [12], with more effective pruning strategy, efficient Check algorithm and novel techniques for checking the minimality of attribute lists.

D. Fast Checking of Minimal Attribute Lists

We give details of CheckMinimality, which checks the minimality of the LHS and RHS attribute lists of each candidate OD in Discover, with the forward FD rule and backward OD rule. An attribute $\overline{A}$ is appended to either LHS or RHS, for generating a new candidate OD. If the new LHS or RHS list is no longer minimal, then it is incurred by $\overline{A}$. We study the cases of FD and OD rules respectively.

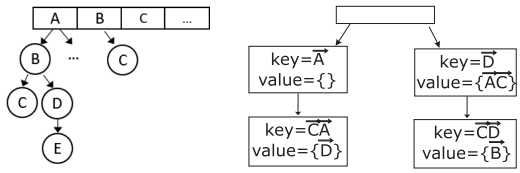
Checking With Forward FD Rule. This rule states that X is *not* minimal, if there exist disjoint subsets $\mathcal{Y}$, $\mathcal{W}$ of $\mathcal{X}$, such that in X every attribute of $\mathcal{W}$ is after all attributes of $\mathcal{Y}$, and $\mathcal{Y} \rightarrow \mathcal{W}$. Hence, when $\overline{A}$ is appended to Z, it suffices to check whether $\mathcal{Z} \rightarrow \mathcal{A}$ for this rule. This is done as follows.

Proposition 6. $\tau_{Z\overline{A}} = \tau_Z$, iff $\mathcal{Z} \rightarrow \mathcal{A}$.

Recall that the computation of $\tau_{Z\overline{A}}$ is originally required in Discover and is very efficient. When $\mathcal{Z} \rightarrow \mathcal{A}$ is identified to prune non-minimal attribute lists, we maintain the set Ψ of valid FDs as follows. (a) If there does not exist $\mathcal{Z}' \rightarrow \mathcal{A}$ in Ψ where $\mathcal{Z}' \subseteq \mathcal{Z}$, then we add $\mathcal{Z} \rightarrow \mathcal{A}$ into Ψ and remove all FDs of the form $\mathcal{Z}'' \rightarrow \mathcal{A}$ from Ψ where $\mathcal{Z} \subset \mathcal{Z}''$. (b) Otherwise, we neglect $\mathcal{Z} \rightarrow \mathcal{A}$. That is, all FDs in Ψ are minimal in terms of Ψ.

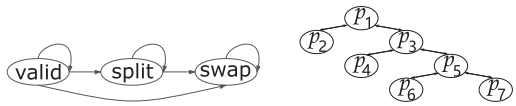
$\mathcal{Z} \rightarrow \mathcal{A}$ is *not* required to be a minimal valid FD on r'; there may exist a proper subset $\mathcal{Z}'$ of $\mathcal{Z}$ such that $\mathcal{Z}' \rightarrow \mathcal{A}$ is valid on r' but is not identified. Moreover, note our goal is to identify the set Ψ of valid FDs that are employed to prune non-minimal attribute lists, rather than all valid FDs on r'. These are a departure from the traditional FD discoveries.

We present a data structure, namely FDMap, to encode FDs in Ψ and facilitate the validation of Ψ on r later.



(a) Example FDMap (b) Example Index

Fig. 3. Examples for minimality check.



(a) state diagram (b) ODtree to be validated

Fig. 4. Examples for Algorithm ValidateOD.

FDMap. FDs in FDMap are clustered based on their RHS attributes, and all FDs with the same RHS attribute are further organized into a prefix tree. Suppose an order η is defined for all attributes from schema R, e.g., $ABCDE$. For $\mathcal{Z} \rightarrow \mathcal{A}$ in Ψ, in the tree structure we store all prefixes of $\mathcal{Z}$ by treating attributes in $\mathcal{Z}$ following the order of η; the leaf node denotes $\mathcal{Z} \rightarrow \mathcal{A}$. For example, two FDs $\mathcal{BC} \rightarrow \mathcal{A}$ and $\mathcal{BDE} \rightarrow \mathcal{A}$ are in the same cluster on A and organized into a tree (shown in Fig. 3(a)). This enables us to share computations among FD validations. Since neither $\mathcal{B}\ \mathcal{C} \rightarrow \mathcal{A}$ nor $\mathcal{BDE} \rightarrow \mathcal{A}$ guarantees to be a minimal FD on r, the validations of them can be safely skipped if $\mathcal{B} \rightarrow \mathcal{A}$ is proven valid on r later. More details will be discussed in Section VII-B.

Checking With Backward OD Rule. The rule states that X is *not* minimal, if there exist disjoint sub-lists Y and W in X, such that Y directly follows W and Y $\mapsto$W. When $\overline{A}$ is appended to Z, it suffices to consider those sublists of Z$\overline{\text{A}}$ ended with $\overline{A}$, for checking the minimality of Z$\overline{\text{A}}$.

Example 9: Consider $\gamma = \overrightarrow{\text{A}}\,\overrightarrow{\text{B}}\,\overrightarrow{\text{C}}\,\overrightarrow{\text{D}} \mapsto \overrightarrow{\text{E}}\,\overrightarrow{\text{F}}\,\overrightarrow{\text{G}}$, where $\overrightarrow{\text{D}}$ is the new attribute. Suppose we have discovered so far three ODs: $\overrightarrow{\text{C}}\,\overrightarrow{\text{A}} \mapsto \overrightarrow{\text{D}}$, $\overrightarrow{\text{C}}\,\overrightarrow{\text{D}} \mapsto \overrightarrow{\text{B}}$ and $\overrightarrow{\text{D}} \mapsto \overrightarrow{\text{A}}\,\overrightarrow{\text{C}}$. To check the minimality of γ, we consider its LHS attribute list $\text{Z} = \overrightarrow{\text{A}}\,\overrightarrow{\text{B}}\,\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$, with target sublists $\overrightarrow{\text{D}}$, $\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$, $\overrightarrow{\text{B}}\,\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$ and $\overrightarrow{\text{A}}\,\overrightarrow{\text{B}}\,\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$.

1) $\overrightarrow{\text{D}}$ is the LHS attribute list of $\overrightarrow{\text{D}} \mapsto \overrightarrow{\text{A}}\,\overrightarrow{\text{C}}$, and hence we check whether $\overrightarrow{\text{A}}\,\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$ is a sublist in Z.
2) $\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$ is the LHS attribute list of $\overrightarrow{\text{C}}\,\overrightarrow{\text{D}} \mapsto \overrightarrow{\text{B}}$, and hence we check whether $\overrightarrow{\text{B}}\,\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$ is a sublist in Z.

It is worth mentioning that ODs discovered so far are used in the backward OD rule. This is why this work and former lexicographical OD discovery algorithms [2], [12] take BFS traversal to discover ODs. With BFS, all ODs with total l attributes on LHS and RHS are handled, before ODs with total l+1 attributes. This enables the checking of minimal attribute lists with backward OD rule. Note this work presents a novel and efficient method (ExtendOD) for OD candidates generation and performs OD discovery on r' rather than r, which significantly differs from [2], [12].

Index Structure. We present a key-value based index to make checking with backward OD rule efficient. Each index node has a key field and a value field; the value field is a set of attribute lists. If a valid OD X $\mapsto$Y is discovered, then we update the index with a node that takes X as its key, and Y as its value. If a node with the same key X already exists, then we add Y to its value field. We further organize all key values in a suffix tree. Index node p is the parent of node p', if the key value of p is a suffix of the key value of p'.

Example 10: We show the index in Fig. 3(b), for the three ODs $\overrightarrow{\text{C}}\,\overrightarrow{\text{A}} \mapsto \overrightarrow{\text{D}}$, $\overrightarrow{\text{C}}\,\overrightarrow{\text{D}} \mapsto \overrightarrow{\text{B}}$ and $\overrightarrow{\text{D}} \mapsto \overrightarrow{\text{A}}\,\overrightarrow{\text{C}}$ from Example 9. To check the minimality of attribute list $\text{Z} = \overrightarrow{\text{A}}\,\overrightarrow{\text{B}}\,\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$, we consider sublists $\overrightarrow{\text{D}}$, $\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$, $\overrightarrow{\text{B}}\,\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$ and $\overrightarrow{\text{A}}\,\overrightarrow{\text{B}}\,\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$. Specifically, in the index we first fetch the node with $\overrightarrow{\text{D}}$ as its key, and identify $\overrightarrow{\text{A}}\,\overrightarrow{\text{C}}$ as its value. We do not find sublist $\overrightarrow{\text{A}}\,\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$ in Z. We then fetch the node with key $\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$, a child of the node with key $\overrightarrow{\text{D}}$ in the index. We identify $\overrightarrow{\text{B}}$ in its value field, and find $\overrightarrow{\text{B}}\,\overrightarrow{\text{C}}\,\overrightarrow{\text{D}}$ in Z. Hence, Z is not minimal.

VII. Dependency Validations

In this section, we validate the discovery results of r' on r. We present methods to validate ODs (Section VII-A) and FDs (Section VII-B), respectively.

A. Batch Validation of ODs From Σ'

The baseline approach is to only validate ODs from Σ' on r. We find this strategy becomes inefficient when a large portion of ODs are found to be invalid on r. The following observation motivates us to adopt a different strategy.

Example 11: Fig. 4(a) shows possible ways the state of a node p on r' changes to its state on r. A valid node p remains intact, or changes to a split (resp. swap) node, if split but no swap (resp. swap) violation is detected on r. When p = X $\mapsto$Y is valid on r', recall we generate ODs X $\mapsto$Y$\overline{\text{A}}$ as children of p. If X $\mapsto$Y is not valid on r, then all these children cannot be valid. For the case the state of p is split on r' but swap on r, the state of all children of p must be swap on r.

If the state of p changes on r, then we can not only update the state of p, but also prune all *descendants* of p from the ODTree. Since our approach works in an iterative way, the children of p will be regenerated in the next round of Discover based on the new state of p. For each node n in the ODTree that denotes an OD in Σ', we propose to validate not only n, but also the nodes in the path from the root to n. This batch validation of ODs starts at upper levels, which helps prune the ODTree as early as possible. Combined with a careful plan of node order in the validation, our (depth-first-search) DFS validation approach also facilitates a more efficient way to leverage sorted partitions.

Better, this strategy enables us to pass results back to Discover from Validate. Specifically, after a node is validated on r, we update its state and add one field "confirmed" to the node in ODTree, indicating this node has been validated on r. This ODTree is then passed to Discover as the input in the next round. Instead of starting from an empty ODTree each time, Discover keeps in ODTree the nodes that are confirmed in former rounds to avoid checking them again, and generates children of them

Algorithm 4: ValidateOD.

```
   Input: relation r, ODTree tree
   Output: a set vioSet of violating tuples, ODTree tree
1  vioSet ← ∅;
2  odList ← valid but not confirmed nodes in DFS order;
3  visited ← {tree.root};
4  foreach node n in odList do
5      p ← n;
6      while p ∉ visited do
7          visited ← visited ∪ {p};
8          p ← p.parent;
9      n.cacheNode ← p;
10 foreach node n in odList do
11     foreach p in the path from n.cacheNode to n do
12         τ_X,τ_Y ← sorted partitions of LHS and RHS of p;
13         if p is not confirmed then
14             p.confirmed ← true;
15             result ← Check (τ_X,τ_Y);
16             if result.state ≠ p.state then
17                 p.state ← result.state;
18                 vioSet ← vioSet ∪ result.vioTuples;
19                 delete descendants of p;
20                 break ;
21 return (vioSet, tree);
```

Algorithm 5: ValidateFD.

```
   Input: relation r, FDMap fdMap
   Output: a set vioSet of violating tuples, FDMap fdMap
1  vioSet ← ∅;
2  foreach RHS attribute A in fdMap do
3      q ← an empty queue;
4      q.enqueue(the root of the FD tree for A in fdMap);
5      while q ≠ ∅ do
6          node ← q.dequeue();
7          if node.state = valid then
8              continue;
9          if node.state = unknown then
10             fdResult ← checkFD(node);
11             node.state ← fdResult.state;
12             if node.state = valid then
13                 prune the descendants of node from fdMap;
14             else
15                 if isLeaf(node) then
16                     vioSet ← vioSet ∪ fdResult.vioTuples;
17         if node.state = split then
18             foreach child in node.children do
19                 q.enqueue(child);
20     pruneTree(the FD tree for A in fdMap);
21 return (vioSet, fdMap);
```

based on their "confirmed" states on r. This is possible since a node is validated after its ancestors, which well preserves the tree structure.

Algorithm. ValidateOD (Algorithm 4) takes as inputs r and Σ' on r' encoded in ODTree *tree*. It outputs a set of violating tuples on r *w.r.t.* Σ' as additions to r', and the updated *tree*. ValidateOD first collects in a list *odList* new valid nodes (valid but not confirmed), in DFS order of the ODTree (line 2). It then plans the validation order for these nodes (lines 3-9). For each node n in *odList*, ValidateOD finds a node p whose sorted partition is computed before n and can be leveraged for n. To do so, it uses a set, referred to as *visited*, to maintain all nodes whose sorted partitions are available. The root is initially added to *visited* (line 3). For node n, ValidateOD backtracks its path to the root, until a node p in *visited* is identified (lines 5-8). It saves p as the position, namely *cacheNode*, where the validation for n starts (line 9), and collects all nodes in the path from p to n (line 7).

As an example, suppose four nodes p_4, p_5, p_6, p_7 are to be validated, whose topology is shown in Fig. 4(b). For p_4, we follow the path p_4–p_3–p_1, and reach a "visited" node p_1, i.e., the root node. p_3 and p_4 are also put into the set *visited* in this backtracking. For p_5, since p_3 is already "visited," we just follow path p_5–p_3. For p_6 and p_7, their validations will start from node p_5 already in the set *visited*.

The aim here is to make a validation plan for each node, and no visit to r is required in this phase. This plan also helps decide whether intermediate sorted partitions should be saved or not; sorted partitions can be discarded to save memory if they will not be used later according to the plan.

For each node n in *odList*, ValidateOD checks all nodes p in the path from the *cacheNode* of n to n (lines 10-20). This follows a DFS traversal of the ODTree, in accordance with the way to build sorted partitions. If p is not confirmed, then ValidateOD employs Check to validate it (line 15). Recall that a node is confirmed after its validation on r. Hence, every OD candidate is validated on r at most once in all iterations. If the state of p differs from its state on r', then ValidateOD updates the state of p, collects two tuples leading to violations, and deletes all descendants of p from ODTree and *odList* (lines 16-19). ValidateOD collects in a set *vioSet* all violating tuples returned by Check, which will be added into r' for the next round. Recall that Check returns only two tuples for each violated OD. That is, the goal is to keep r' as small as possible. We adopt this strategy to favor Discover the most in terms of the complexity *w.r.t.* data size.

B. Batch Validation of FDs From Ψ

The violation of an FD $\mathcal{Z} \rightarrow \mathcal{A}$ corresponds to the split violation of the OD $\mathsf{Z} \mapsto \mathsf{A}$: there are tuples t, s, such that $t_{\mathsf{Z}} = s_{\mathsf{Z}}$ but $t_A \neq s_A$ (Section III). Hence, Algorithm Check (Section V-A) can be easily adapted (simplified) for checking FD violations and returning two tuples leading to FD violations. The data structure of *sorted partition* can also be simplified: split concerns tuples in the same equivalence class, and the order of different equivalence classes is irrelevant. Such data structure is known as position list index (Pli) in FD discoveries [13]. Pli for an attribute A is built by clustering tuples on their values in A, and Pli can be further built for an attribute set $\mathcal{Z} \subseteq R$. It takes $O(n)$ to build a Pli with hashing, and $O(n)$ to check the validity of $\mathcal{Z} \rightarrow \mathcal{A}$ leveraging Plis for $\mathcal{Z}$ and $\mathcal{A}$, where n is the number of tuples.

Batch Validation. Based on the FDMap that encodes FDs from Ψ, we propose to validate FDs with the same RHS attribute, by following the tree structure from the root. For example, recall the FDs shown in Fig. 3(a). We first validate $\mathcal{B} \rightarrow \mathcal{A}$, i.e., the root of the tree. We can skip all descendant nodes if this FD is valid, since they are all valid if the FD is valid. Recall that in the tree

structure, FDs in Ψ, i.e., $\mathcal{B}\ \mathcal{C} \rightarrow \mathcal{A}$ and $\mathcal{BDE} \rightarrow \mathcal{A}$, are always leaf nodes. Therefore, this strategy enables early termination of the FD validations. Moreover, this also enables an efficient manipulation of Plis.

Algorithm. ValidateFD (Algorithm 5) takes as inputs r and the FDMap *fdMap* produced by Discover, and returns a set *vioSet* of tuples violating FDs from *fdMap*, and the modified *fdMap*. To avoid checking the same FDs on r multiple times in the iterative approach, we additionally associate each node (FD) with a state. The state is one of valid, split and unknown. A new valid FD identified on r' has the state of unknown. After a node is validated on r with Function checkFD (line 10), it has the state of valid (resp. split) if it is valid (resp. invalid) on r. For a valid node, all descendants of it are pruned (lines 12-13). If the validation fails on a leaf node, i.e., the node denoting an FD from Ψ, then the two tuples returned by checkFD are collected (lines 15-16). For a split node, the validation then continues on its child nodes in the BFS way (lines 17-19). After processing the whole FD tree, split nodes are pruned unless they have valid descendants (Function pruneTree in line 20).

The modified FDMap will be passed back to Discover in the next iteration of Algorithm 1. In later rounds, when a valid FD identified in Discover is inserted into FDMap, the insertion is skipped if the FD is already in the tree or has a parent that is already proven valid on r.

VIII. A Flexible Strategy With Parallelism

In this section, we present a strategy that switches between Discover and Validate in a flexible way, and further enhance the strategy with multithreaded parallelism.

As noted in Section VII-A, the state of a node p in the ODTree may be *inconsistent* between r' and r, and this inconsistency makes the partial ODTree generated from p useless. We tackle the problem by starting OD validations at upper levels in Section VII-A. A better solution is to enable *early start* of Validate, since enriching r' with violating tuples found on r can help r' adapt to the characteristics of r. The basic strategy that enables *early start* of Validate is as follows. After a predefined number of ODs are discovered, we suspend Discover and start Validate with the ODTree and FDMap discovered so far as inputs. If some violations are detected on r, then we terminate former Discover and directly start the next round with the updated r', ODTree and FDMap. Otherwise, we resume the suspended Discover.

We further enhance the strategy with multithreaded parallelism, to run Discover and Validate in parallel.

The Multithreaded Parallelism. We extend OD discovery to run with multiple threads. It is a lightweight alternative to distributed dependency discovery methods [14], [16].

Multithreaded parallelism is suitable to our solution, since Discover and Validate deal with r' and r, respectively. We further leverage multiple threads in Discover (resp. Validate), to enable the internal operations of Discover (resp. Validate) to run in parallel as well. Specifically, we perform parallel computation layer by layer; recall $\overline{\mathsf{A}}\mathsf{X}' \mapsto \overrightarrow{\mathsf{B}}\mathsf{Y}'$ is at layer i if $|\mathsf{X}'|+|\mathsf{Y}'| = i$ (Section VI-A). Multiple threads are employed to generate ODs at the same layer in Discover, and after all ODs at one layer are generated, they and the related valid FDs are validated by multiple threads in Validate. A thread pool is used for Discover (resp. Validate), to limit the total number of threads for Discover (resp. Validate), while the new *Distributor* module works as a single thread.

The framework of our multithreaded approach is shown in Fig. 5. We give more details as follows.

(1) *Discovery.* With a concurrent queue q, we implement the BFS traversal of Discover with parallelism. A concurrent queue guarantees that pushing and pulling elements are thread-safe. Initially, all nodes in q are ODs at layer 0, e.g., $\overline{\mathsf{A}} \mapsto \overrightarrow{\mathsf{B}}$. Multiple threads are employed to expand the ODTree in parallel. Specifically, each thread pulls one node from q, generates its children and adds them to ODTree. The updates to ODTree in a thread start from the node it handles, without incurring write-write conflicts, and reading the same node by different threads is thread-safe. Each thread then collects all the generated ODs and valid FDs used in minimality check, and sends them to the *Distributor*. After all the nodes in q are pulled, their children are pushed into q to start the discovery of ODs at the next layer.

(2) *Distributor.* The *Distributor* collects discovered ODs that are at the same layer into a set and pushes the set into a concurrent queue w, to be validated by Validate. It also inserts the newly received FDs into FDMap. FDMap is constructed in the *Distributor*, since different threads of Discover can identify valid FDs with the same RHS attribute.

(3) *Validation.* Validate pulls an element (a set of ODs) from the queue w, and assigns the ODs to available threads. For ODs that are assigned to the same thread, validation plans are made for them, along the same lines as Section VII-A. FD validations are also conducted with multiple threads in parallel. For every attribute $A \in R$, all FDs of the form $\mathcal{Z} \rightarrow \mathcal{A}$ are validated in one thread. This enables us to share computations among FD validations, since all FDs of the form $\mathcal{Z} \rightarrow \mathcal{A}$ are organized into a tree structure in FDMap. To fully leverage the structure, it is more efficient to validate a large number of FDs at a time. We hence choose to postpone FD validations (more details will be given in (4) below). (4) *Iterative approach.* Discover and Validate still run in an iterative way; Discover needs to restart if violating tuples are found on r in Validate. It is better to restart Discover after enough tuples are collected to enrich r'. A threshold μ for the accumulated number of violating tuples is used to control the restart. After all ODs at one layer are validated, we will restart Discover if μ is reached. Before the restart, we perform FD validation with multiple threads, collect violating tuples returned by FD validation, and add all the violating tuples from OD and FD validations to r'. We then restart Discover with the updated r' and the modified ODTree and FDMap. If μ has not been reached yet, then Validate pulls another set of ODs from the queue w to process. This continues until μ is reached, or Discover terminates on r'. After the termination of Discover, Validate will process all the remaining ODs and FDs, and Discover will restart if violating tuples are found on r.

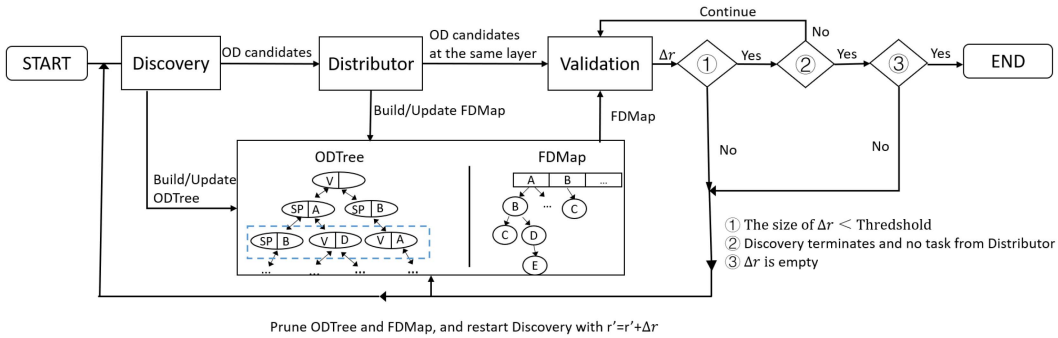


Fig. 5. The framework of multithreaded parallelism.

TABLE III
DATASETS AND EXECUTION STATISTICS FOR ORDER, UOD, UOD_m, AND UOD^*

Dataset Properties			ORDER [12]		UOD		UOD_m		UOD^*	
Dataset	$\|r\|$	$\|R\|$	Time (ms)	$\|OD\|$	Time (ms)	$\|OD\|$	Time (ms)	$\|OD\|$	Time (ms)	$\|OD\|$
NCV	1K	19	332	18	98	18	78	18	82	18
NCV	900K	16	–	–	15,262	12	8,145	12	15,357	12
NCV	500K	9	798,400	12	6,074	12	2,387	12	2,122	12
FLI	1K	14	–	–	31,749	152,704	20,130	152,704	138	32
FLI	500K	17	–	–	–	–	–	–	5,298	32
DB	250K	30	–	–	13,846	83	11,335	83	30,680	14
DB	250K	9	1,033,810	80	8,923	80	7,953	80	7,506	11
FEB	500K	14	–	–	324,890	3,396	118,189	3,396	24,321	78
ATOM	32K	25	1,734	384	1,533	384	955	384	907	384
WP	20K	7	2,386	81	350	81	335	81	298	19

Communication. Discover sends the identified valid ODs and FDs to the *Distributor* via a concurrent queue. The *Distributor* collects all ODs at the same layer in a set, and sends the set to Validate via another concurrent queue.

Thread-Safe Operations. Our approach is a shared-memory parallel algorithm. Discover, *Distributor* and Validate visit the same ODTree and FDMap, and hence the key is to make the visits *thread-safe*. (1) When OD candidates at one layer are validated in Validate, Discover can discover ODs at the next layer. The visits to the ODTree in different threads of Discover are thread-safe since the same OD nodes are never generated in different threads. Threads for different ODs in Validate, however, may visit the same ancestor node if the ancestor is shared by the ODs assigned to the threads. If one thread prunes the ancestor, then other thread that tries to visit the node may fail. To this end, instead of directly pruning nodes from the ODTree, the threads of Validate save the nodes that should be pruned in a thread-safe set. As will be illustrated shortly, these nodes will be pruned from the ODTree right before the restart of Discover.

(2) The identified valid FDs are continuously inserted into FDMap in *Distributor*, while FDMap is only visited in Validate right before the restart of Discover or before the termination of the whole algorithm (if no violating tuples of ODs and FDs are found). The FDs with the same RHS attribute are validated in the same thread of Validate, which makes visits to the FDMap thread-safe in different threads of Validate.

(3) A single thread is leveraged to restart Discover, by terminating all current threads of Discover, adding $\triangle r$ to r', updating the ODTree by pruning the saved nodes, and restarting Discover with the updated r' and ODTree.

IX. EXPERIMENTAL EVALUATIONS

In this section, we conduct extensive experiments to compare our methods with other OD discovery techniques, and to analyze the performance of our algorithms in detail.

A. Experimental Setting

Datasets. Our evaluations are conducted on a set of real-life and synthetic data; most of them are also used in previous works [2], [9], [12], [19], [20]. Details of all the experimental datasets are summarized in Table III. We denote by $|r|$ the number of tuples, and by $|R|$ the number of attributes. To favor algorithms that do not scale well, we also use some small versions with fewer $|r|$ and $|R|$.

Algorithms. Our algorithms are implemented in Java. (1) BOD [9]: the algorithm for discovering bidirectional lexicographical ODs, (2) BOD_m: the algorithm that adds multithreaded parallelism to BOD (Section VIII), and (3) BOD^*: the algorithm that further enhances BOD_m by pruning non-minimal ODs with the forward FD rule (Section III). To compare with algorithms that only discover unidirectional ODs, we implement versions of our algorithms for unidirectional ODs, denoted by UOD, UOD_m and UOD^* respectively.

We compare our algorithms against OD discovery methods [12], [16], [19], [20][1]. (1) ORDER [12] discovers unidirectional lexicographical ODs. (2) FastOD [19], [20] discovers set-based canonical ODs. (3) DistOD [16] is a distributed

[1]The results of [2] are omitted. [2] may mistakenly prune minimal valid ODs and fail to discover the complete OD set, as noted in [9], [21].

set-based canonical OD discovery algorithm that leverages a cluster of computing nodes. We use the implementations of ORDER, FastOD and DistOD that are available online[2].

Running Environment. Unless otherwise stated, all experiments are run on a machine with an Intel Xeon E-2224 3.4 G CPU (4 physical cores), 64 GB of memory and CentOS. For BOD_m and BOD^*, we set the total number of threads as 4 by default, and the threshold $\mu = 10$ (the number of violating tuples to control the restart of Discover in Section VIII). Each experiment is run 5 times and we report the average here.

B. Experimental Results

Exp-1. We show the experimental results of methods for discovering lexicographical ODs in Table III. For each method, we report its running time in milliseconds and the number of discovered ODs ($|OD|$). We abort experiments after 3 hours and denote the results as "−" in the table.

(1) UOD is much faster than ORDER, up to orders of magnitude. For example, ORDER can process NCV ($|R| = 16$) in 15 seconds, and DB ($|R| = 30$) in 13 seconds, while ORDER cannot terminate within 3 hours. UOD always discovers the same set of ODs as ORDER on the datasets that ORDER can finish, which demonstrates the correctness of UOD.

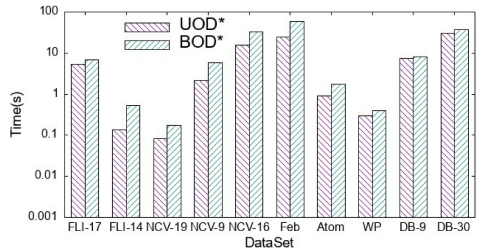
(2) UOD_m further improves the efficiency of UOD and always discovers the same set of ODs as UOD. Specifically, UOD_m is on average 1.69 times and up to 2.75 times faster than UOD on the tested datasets. The flexible strategy combined with multithreaded parallelism reduces the running time and does not affect the correctness.

(3) The number of ODs discovered by UOD^* is no larger than that of UOD_m, because the number of non-minimal ODs pruned by the forward FD rule is no less (and can be much larger) than that pruned by the forward OD rule. Therefore, UOD^* is very likely to improve the effectiveness of OD discovery and does not affect the completeness (will be further studied in Exp-5). The enhanced pruning power can also significantly improve the efficiency on some datasets. For example, neither UOD nor UOD_m can handle FLI ($|R| = 17$) within the time limit; we find more than 50 K ODs are already discovered at the termination. In contrast, UOD^* discovers only 32 ODs in less than 6 seconds. UOD^* is slower than UOD_m on some datasets. We find this occurs when the forward FD rule does not help prune more ODs on NCV ($|R| = 16$), or when a huge number of valid FDs are identified for pruning non-minimal ODs on sample data, but most of them are proven invalid on the whole of DB ($|R| = 30$).

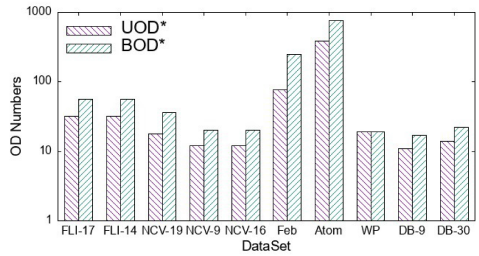
Exp-2. We experimentally study the differences between unidirectional and bidirectional OD discovery methods in terms of the number of discovered ODs and running time.

(1) We report results in Fig. 6(a) and (b), respectively. BOD^* necessarily takes more time than UOD^*, since the number of bidirectional ODs is larger than that of unidirectional ODs. However, the numbers of discovered ODs are typically greatly reduced by the forward FD rule, for both UOD^* and BOD^*.

[2]https://hpi.de/naumann/projects/repeatability/data-profiling/fds.html.

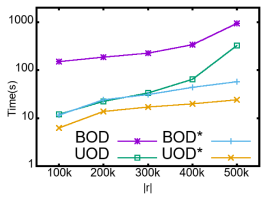


(a) Running Time

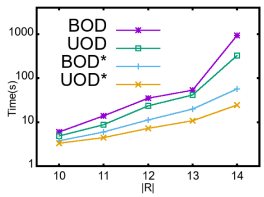


(b) Number of ODs

Fig. 6. Comparison of UOD^* and BOD^* on All the datasets.



(a) FEB: Varying $|r|$



(b) FEB: Varying $|R|$

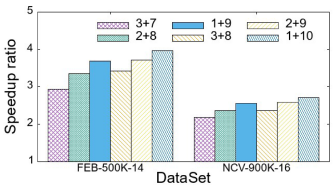
Fig. 7. Comparison of Unidirectional and Bidirectional OD Discoveries.

The results show that even BOD^* can process all of the tested datasets within 60 seconds.

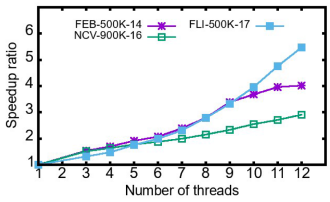
(2) We further conduct in-depth analyses by varying $|r|$ and $|R|$. We set $|r| = 500$ K and $|R| = 14$ on FEB by default, vary $|r|$ from 100 K to 500 K in Fig. 7(b), and vary $|R|$ from 10 to 14 in Fig. 7(a), respectively. The results tell us that BOD^* and UOD^* scale better than BOD and UOD. In particular, BOD^* consistently beats UOD on FEB ($|r| = 500$ K) as $|R|$ increases.

Exp-3. We conduct more experiments with BOD^*. A machine with two Intel Xeon E5-2620 V2 2.1 G CPU (6 physical cores each CPU) and 64 GB of memory is used in this set of experiments. When only one thread is used, BOD^* exploits the strategy that enables early start of Validate *without* parallelism. When more threads are available (at least three threads are required to enable parallelism), we report the speedup ratio *w.r.t.* the single-threaded setting.

(1) We study different strategies for thread allocation among Discover, the *Distributor* and Validate. One thread is sufficient for the *Distributor* due to the light workload. The workload of Validate is typically much heavier than that of Discover, since Discover deals with the small sample but Validate processes the whole instance. Therefore, more threads are required for Validate. For example in Fig. 8(a), we denote by "3+7" the setting in which 3 (resp. 7) threads are assigned to Discover



(a) Thread Allocations



(b) Speedup Ratio *w.r.t.* # of Threads

Fig. 8. Experiments on multithreaded parallelism.

TABLE IV
EXECUTION STATISTICS FOR FastOD, DistOD AND BOD*

Dataset Properties	FastOD [19], [20]		DistOD [16]		BOD*	
Dataset	Time(ms)	$\|OD\|$	Time(ms)	$\|OD\|$	Time(ms)	$\|OD\|$
NCV-900K-16	–	–	6,421,615	2,526	32,578	20
NCV-500K-9	–	–	44,611	105	5,852	20
FLI-500K-17	6,374,436	556	977,710	556	6,756	56
DB-250K-30	–	–	311,814	90,330	37,894	22
FEB-500K-14	–	–	755,389	419	57,531	246
ATOM-32K-25	9,035	68	17,624	68	1,787	768

(resp. Validate). The results show that with a fixed total number of threads for Discover and Validate, the more threads Validate uses, the more efficient BOD* is. When the total number of threads for Discover and Validate increases from 10 to 11, the speedup ratio always increases with one additional thread for Validate.

(2) We study the speedup ratio in terms of the number of threads in Fig. 8(b). The ratio consistently increases as the number of threads increases, and is about 3 to 5.9 for 12 threads (10 threads for Validate) on the tested datasets. The relatively low speedup ratio is mainly due to the inherent complexity of OD discovery: valid ODs with fewer attributes are used in the backward OD rule to check the minimality of ODs with more attributes. Hence, recall only ODs at the same layer are validated in parallel, and the validation for a layer can start only after all the threads for the previous layer terminate. Another reason is that the sorted partitions computed in one thread cannot be reused by the other threads running in parallel. Hence, the total computation cost of validating ODs may increase as the number of threads increases, because more sorted partitions are repeatedly computed in different threads.

Exp-4. We compare BOD* against FastOD and DistOD, as shown in Table IV. For DistOD, we use a cluster with 12 computing nodes (virtual machines) in this set of experiments. The cluster is equipped with three Intel Xeon E5-2683 v3@ 2.00 GHz CPUs, and the Spark version is 2.4.0-cdh6.3.2. Each computing node is equipped with 8-16 GB of memory.

FastOD and its distributed version DistOD discover set-based ODs. This explains why $|OD|$ of FastOD (DistOD) is different. Specifically, FastOD discovers ODs of the form of $\mathcal{X}$: [ ] $\mapsto_{cst}$ $\mathcal{A}$ and $\mathcal{X}$: $A \sim B$. The former one is an FD, and the latter one is an order compatible dependency (Section III).

In Table IV, we denote the results by "–" if FastOD runs out of memory or cannot terminate within 3 hours. The performance of DistOD can be worse than FastOD on small datasets, e.g., ATOM, due to the overhead of Spark. DistOD significantly outperforms FastOD on large datasets by leveraging parallelism, as expected. BOD* is much faster than DistOD on all tested datasets, up to orders of magnitude. The better efficiency of BOD* is mainly due to its hybrid discovery strategy; DistOD (FastOD) follows the column based strategy. Another reason is that the $|OD|$ of DistOD is usually much larger than that of BOD*. Recall each list-based OD $X \mapsto Y$ is mapped to $|X|\cdot|Y|$ set-based ODs. A lexicographical OD is valid only when the set-based canonical ODs from the lexicographical OD are all valid. The results reported here are consistent with those in [2], [19], [20], showing that the running time of a discovery algorithm is closely related to the actual number of discovered dependencies.

Exp-5. We show some discovered ODs and their applications in query optimization in Table V, to verify the effectiveness of OD discovery. With $\overrightarrow{\text{Year}} \mapsto \overrightarrow{\text{WorldPopulation}}$ discovered on WP, the condition of "WorldPopulation asc" can be removed from the order by clause, for a more efficient query. As another example, based on the two ODs on FLI, an index on (Carrier, Airport) instead of the two indexes on (UniqueCarrier, OriginAirportID) and (UniqueCarrier, OriginAirportSeqID) can be employed to support the two queries given in Table V, since the two queries are equivalent to the query with "order by Carrier asc, Airport asc". This optimization helps reduce the index space.

Exp-6. We verify that the forward FD rule can help prune non-minimal ODs, without affecting the completeness. We give some examples from the tested datasets in Table VI.

There are three columns WorldPopulation, Year and YearlyChange in the dataset WP, where WorldPopulation is the total population of the world of a year, and YearlyChange is the percentage increase in population compared to the previous year. The world population keeps rising with each year. BOD discovers an OD $\overrightarrow{\text{Year}} \mapsto \overrightarrow{\text{WorldPopulation}}, \overrightarrow{\text{YearlyChange}}$ (the last line of Table VI). This OD is minimal for BOD since $\overrightarrow{\text{WorldPopulation}} \mapsto \overrightarrow{\text{YearlyChange}}$ does not hold; the percentage increase in population does not guarantee to increase as WorldPopulation increases. However, the FD WorldPopulation $\rightarrow$ YearlyChange holds because WorldPopulation $\rightarrow$Year and Year $\rightarrow$ YearlyChange. Leveraging the FD, BOD* can prune $\overrightarrow{\text{Year}} \mapsto \overrightarrow{\text{WorldPopulation}}, \overrightarrow{\text{YearlyChange}}$, and discover $\overrightarrow{\text{Year}} \mapsto \overrightarrow{\text{WorldPopulation}}$. Note $\overrightarrow{\text{Year}} \mapsto \overrightarrow{\text{WorldPopulation}}$ is not minimal for BOD because $\overrightarrow{\text{Year}} \mapsto \overrightarrow{\text{WorldPopulation}}, \overrightarrow{\text{YearlyChange}}$ is valid. Similarly for the case of $\overrightarrow{\text{Year}} \mapsto \overrightarrow{\text{WorldPopulation}}, \overrightarrow{\text{UrbanPop}}$ in Table VI. Finally, BOD* discovers one OD $\overrightarrow{\text{Year}} \mapsto \overrightarrow{\text{WorldPopulation}}$ instead of the two ODs of BOD. This OD imposes the same order specification as the other two and is more concise and applicable, e.g., for building indexes [11], [23].

TABLE V
SOME DISCOVERED ODS AND THEIR APPLICATIONS IN QUERY OPTIMIZATION

Dataset	Discovered ODs	Original query	New query after optimization
WP	$\overrightarrow{\text{Year}} \mapsto \overrightarrow{\text{WorldPopulation}}$	select * from WP order by Year asc, WorldPopulation asc	select * from WP order by Year asc
FLI	$\overrightarrow{\text{Carrier}}, \overrightarrow{\text{Airport}} \mapsto \overrightarrow{\text{UniqueCarrier}}, \overrightarrow{\text{OriginAirportID}}$ $\overrightarrow{\text{Carrier}}, \overrightarrow{\text{Airport}} \mapsto \overrightarrow{\text{UniqueCarrier}}, \overrightarrow{\text{OriginAirportSeqID}}$	select * from FLI order by UniqueCarrier asc, OriginAirportID asc select * from FLI order by UniqueCarrier asc, OriginAirportSeqID asc	select * from FLI order by Carrier asc, Airport asc

TABLE VI
THE EFFECTIVENESS OF PRUNING NON-MINIMAL ODS WITH THE FORWARD FD RULE

Dataset	ODs that are pruned	Valid FDs for pruning ODs	Discovered ODs
FLI	$\overrightarrow{\text{Carrier}}, \overrightarrow{\text{AirlineID}}, \overrightarrow{\text{OriginAirportSeqID}} \mapsto \overrightarrow{\text{UniqueCarrier}}, \overrightarrow{\text{OriginAirportID}}$	Carrier → AirlineID	$\overrightarrow{\text{Carrier}}, \overrightarrow{\text{OriginAirportSeqID}} \mapsto \overrightarrow{\text{UniqueCarrier}}, \overrightarrow{\text{OriginAirportID}}$
FEB	$\overrightarrow{\text{DestAirportID}} \mapsto \overrightarrow{\text{Dest}}, \overrightarrow{\text{DestAirportSeqID}}$	Dest → DestAirportSeqID	$\overrightarrow{\text{DestAirportID}} \mapsto \overrightarrow{\text{Dest}}$
WP	$\overrightarrow{\text{Year}} \mapsto \overrightarrow{\text{WorldPopulation}}, \overrightarrow{\text{UrbanPop}}$	WorldPopulation → UrbanPop	$\overrightarrow{\text{Year}} \mapsto \overrightarrow{\text{WorldPopulation}}$
WP	$\overrightarrow{\text{Year}} \mapsto \overrightarrow{\text{WorldPopulation}}, \overrightarrow{\text{YearlyChange}}$	WorldPopulation → YearlyChange	$\overrightarrow{\text{Year}} \mapsto \overrightarrow{\text{WorldPopulation}}$

X. CONCLUSION

We have studied the problem of lexicographical OD discovery. We have given efficient algorithms for data sampling, OD discovery and batch validations, developed a host of data structures and optimizations, and exploited multithreaded parallelism. Our experimental studies have demonstrated that our approach significantly outperforms existing methods in terms of effectiveness and efficiency.

We intend to extend our method for discovering ODs with the same attributes on LHS and RHS attribute lists. Another topic is to study discovery methods with multithreaded parallelism for other dependencies, to explore the potentials of modern multicore CPUs.

REFERENCES

[1] T. Bleifuß, S. Kruse, and F. Naumann, "Efficient denial constraint discovery with hydra," *Proc. VLDB Endowment*, vol. 11, no. 3, pp. 311–323, 2017.

[2] C. Consonni, P. Sottovia, A. Montresor, and Y. Velegrakis, "Discovering order dependencies through order compatibility," in *Proc. Int. Conf. Extending Database Technol.*, 2019, pp. 409–420.

[3] W. Fan and F. Geerts, *Foundations of Data Quality Management*. San Rafael, CA, USA: Morgan & Claypool, 2012.

[4] S. Ginsburg and R. Hull, "Order dependency in the relational model," *Theor. Comput. Sci.*, vol. 26, pp. 149–195, 1983.

[5] S. Ginsburg and R. Hull, "Sort sets in the relational model," *J. ACM*, vol. 33, no. 3, pp. 465–488, 1986.

[6] I. F. Ilyas and X. Chu, "Trends in cleaning relational data: Consistency and deduplication," *Foundations Trends Databases*, vol. 5, no. 4, pp. 281–393, 2015.

[7] Y. Jin, Z. Tan, J. Chen, and S. Ma, "Discovery of approximate lexicographical order dependencies," *IEEE Trans. Knowl. Data Eng.*, early access, Nov. 23, 2021, doi: 10.1109/TKDE.2021.3130227.

[8] Y. Jin, Z. Tan, W. Zeng, and S. Ma, "Approximate order dependency discovery," in *Proc. IEEE Int. Conf. Data Eng.*, 2021, pp. 25–36.

[9] Y. Jin, L. Zhu, and Z. Tan, "Efficient bidirectional order dependency discovery," in *Proc. IEEE Int. Conf. Data Eng.*, 2020, pp. 61–72.

[10] R. Karegar, P. Godfrey, L. Golab, M. Kargar, D. Srivastava, and J. Szlichta, "Efficient discovery of approximate order dependencies," in *Proc. Int. Conf. Extending Database Technol.*, 2021, pp. 427–432.

[11] J. Kossmann, T. Papenbrock, and F. Naumann, "Data dependencies for query optimization: A survey," *VLDB J.*, vol. 31, no. 1, pp. 1–22, 2022.

[12] P. Langer and F. Naumann, "Efficient order dependency detection," *VLDB J.*, vol. 25, no. 2, pp. 223–241, 2016.

[13] T. Papenbrock and F. Naumann, "A hybrid approach to functional dependency discovery," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 821–833.

[14] H. Saxena, L. Golab, and I. F. Ilyas, "Distributed implementations of dependency discovery algorithms," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1624–1636, 2019.

[15] P. Schirmer, T. Papenbrock, I. K. Koumarelas, and F. Naumann, "Efficient discovery of matching dependencies," *ACM Trans. Database Syst.*, vol. 45, no. 3, pp. 13:1–13:33, 2020.

[16] S. Schmidl and T. Papenbrock, "Efficient distributed discovery of bidirectional order dependencies," *VLDB J.*, vol. 31, no. 1, pp. 49–74, 2022.

[17] D. E. Simmen, E. J. Shekita, and T. Malkemus, "Fundamental techniques for order optimization," in *Proc. Int. Conf. Manage. Data*, 1996, pp. 57–67.

[18] S. Song, F. Gao, R. Huang, and C. Wang, "Data dependencies extended for variety and veracity: A family tree," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 10, pp. 4717–4736, Oct. 2022.

[19] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava, "Effective and complete discovery of order dependencies via set-based axiomatization," *Proc. VLDB Endowment*, vol. 10, no. 7, pp. 721–732, 2017.

[20] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava, "Effective and complete discovery of bidirectional order dependencies via set-based axioms," *VLDB J.*, vol. 27, no. 4, pp. 573–591, 2018.

[21] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava, "Erratum for discovering order dependencies through order compatibility," in *Proc. Int. Conf. Extending Database Technol.*, 2020, pp. 659–663.

[22] J. Szlichta, P. Godfrey, and J. Gryz, "Fundamentals of order dependencies," *Proc. VLDB Endowment*, vol. 5, no. 11, pp. 1220–1231, 2012.

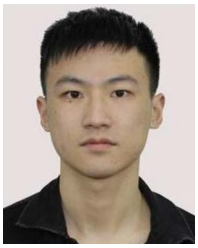
[23] J. Szlichta, P. Godfrey, J. Gryz, W. Ma, W. Qiu, and C. Zuzarte, "Business-intelligence queries with order dependencies in DB2," in *Proc. Int. Conf. Extending Database Technol.*, 2014, pp. 750–761.

[24] J. Szlichta, P. Godfrey, J. Gryz, and C. Zuzarte, "Expressiveness and complexity of order dependencies," *Proc. VLDB Endowment*, vol. 6, no. 14, pp. 1858–1869, 2013.

[25] Z. Wei and S. Link, "Discovery and ranking of functional dependencies," in *Proc. IEEE Int. Conf. Data Eng.*, pp. 1526–1537, 2019.

Jixuan Chen received the BS degree in software engineering from the Huazhong University of Science and Technology, in 2020. He is currently working toward the master's degree with the School of Computer Science, Fudan University, China. His research interests include data profiling and distributed computations.

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



Yihan Li received the BS degree in software engineering from the Huazhong University of Science and Technology, in 2020. He is currently working toward the master's degree with the School of Computer Science, Fudan University, China. His research interests include data profiling and machine learning for data quality management.

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

Weidong Yang received the PhD degree from Xidian University. He is a professor with the School of Computer Science, Fudan University, China. He is also the director of Big Data and Intelligent Computing Laboratory, Zhuhai Fudan Innovation Institute. His current research interests include data lake, time series data, knowledge graph, intelligent software development, and point cloud.

Shuai Ma received the PhD degrees from the University of Edinburgh, in 2011, and from Peking University, in 2004. He is a professor with the School of Computer Science and Engineering, Beihang University, China. He is a recipient of the best paper award for VLDB 2010. He is an associate editor of *VLDB Journal* since 2017 and *IEEE Transactions on Big Data and Knowledge and Information Systems* since 2020. His current research interests include database theory and systems, and Big Data.