Incremental Discovery of Order Dependencies on Tuple Insertions

Lin Zhu[1,2], Xu Sun[1,2], Zijing Tan[1,2(✉)], Kejia Yang[3], Weidong Yang[1,2], Xiangdong Zhou[1,2], and Yingjie Tian[4]

[1] School of Computer Science, Fudan University, Shanghai, China
zjtan@fudan.edu.cn
[2] Shanghai Key Laboratory of Data Science, Shanghai, China
[3] Ann Arbor EECS Department, University of Michigan, Ann Arbor, USA
[4] State Grid Shanghai Municipal Electric Power Company, Shanghai, China

Abstract. Order dependencies (ODs) are recently proposed to describe a relationship of ordering between lists of attributes. It is typically too costly to design ODs manually, since the number of possible ODs is of a factorial complexity in the number of attributes. To this end, automatic discovery techniques for ODs are developed. In practice, data is frequently updated, especially with tuple insertions. Existing techniques do not lend themselves well to these situations, since it is prohibitively expensive to recompute all ODs from scratch after every update. In this paper, we make a first effort to investigate incremental OD discovery techniques in response to tuple insertions. Given a relation D, a set Σ of valid and minimal ODs on D, and a set $\triangle D$ of tuple insertions to D, it is to find, changes $\triangle\Sigma$ to Σ that makes $\Sigma \oplus \triangle\Sigma$ a set of valid and minimal ODs on $D + \triangle D$. Note that $\triangle\Sigma$ contains both new ODs to be added to Σ and outdated ODs to be removed from Σ. Specifically, (1) We formalize the incremental OD discovery problem. Although the incremental discovery problem has a same complexity as its batch (non-incremental) counterpart in terms of traditional complexity, we show that it has good data locality. It is linear in the size of $\triangle D$ to validate on $D+\triangle D$ any OD φ that is valid on D. (2) We present effective incremental OD discovery techniques, leveraging an intelligent traversal strategy for finding $\triangle\Sigma$ and chosen indexes to minimize access to D. Our approach computes $\triangle\Sigma$ based on ODs in Σ, and is independent of the size of D. (3) Using real-life data, we experimentally verify that our approach substantially outperforms its batch counterpart by orders of magnitude.

1 Introduction

Data dependencies, *a.k.a.* integrity constraints, specify data semantics and inherent attribute relationships. They are widely employed in schema design, query optimization [15] and data cleaning [3,5,10,11], among other things. Recently, order dependencies (ODs) [15,18] are proposed to describe the relationship between two lexicographical ordering specifications on lists of attributes. ODs

G. Li et al. (Eds.): DASFAA 2019, LNCS 11446, pp. 157–174, 2019.
https://doi.org/10.1007/978-3-030-18576-3_10

properly subsume functional dependencies (FDs), and can define lexicographic orders used in the SQL order-by clause. Hence, ODs are proved to be useful in query optimizations concerning sorting [15,18]. Compared to traditional dependencies based on sets, *e.g.,* FDs, denial constraints (DCs) [5] and differential dependencies (DDs) [14], ODs are defined on lists, and are hence quite different. We will review formal definitions of OD in Sect. 2, and first provide an illustrative example to highlight features of OD.

		A	B	C	D	E	F	G
D	t_1:	1	2	3	4	1	1	1
	t_2:	1	2	3	4	2	1	2
	t_3:	2	1	4	2	4	1	3
	t_4:	2	4	5	1	4	1	4
$\triangle D$	t' :	1	2	3	5	4	2	5
	t'':	1	2	4	3	5	6	7

Fig. 1. An instance D, and $\triangle D$ of tuple insertions to D.

Example 1: Figure 1 shows an instance D with four tuples $\{t_1, t_2, t_3, t_4\}$. If we sort tuples by attribute A, and then break ties by attribute B, these tuples are also sorted by attribute C first and then by attribute D. This sorting specification is in accordance with the SQL order by clauses. With the notation of OD, this is written as AB $\mapsto$ CD, *i.e.,* AB *orders* CD. Here AB and CD are lists of attributes. Leveraging this example, we illustrate several unique features of ODs.

(1) OD AB $\mapsto$ CD states that values on CD are monotonically non-decreasing with respect to values on AB [15]. Specifically, (a) AB $\mapsto$ CD implies an FD $\mathcal{AB} \rightarrow \mathcal{CD}$. Here set $\mathcal{AB}$ (resp. $\mathcal{CD}$) denotes the set of elements in list AB (resp. CD). To guarantee that when tuples are sorted by AB, they are also sorted by CD, tuples with a same value on AB must have a same value on CD, *e.g.,* t_1 and t_2. (b) ODs also impose order semantics on tuples with different values on AB. For example, t_3 has a larger AB's value than t_2, the CD's value of t_3 cannot be less than that of t_2, to satisfy AB $\mapsto$ CD.

(2) Unlike other constraints, *e.g.,* FDs and DCs, ODs are specified on *lists* of attributes, and the order of attributes on the left-hand side (LHS) and right-hand side (RHS) matters. For example, neither BA $\mapsto$ CD nor AB $\mapsto$ DC holds.

(3) FDs can be always converted into the form with a single RHS attribute. For example, $\mathcal{AB} \rightarrow \mathcal{CD}$ can be expressed as $\mathcal{AB} \rightarrow \mathcal{C}$ and $\mathcal{AB} \rightarrow \mathcal{D}$. This simplifies FD discovery [9,13]. In contrast, RHS attributes of an OD are taken as a whole and *may not* be splitted. As an example, AB $\mapsto$ D does not hold. □

No matter how desirable, it is prohibitively time-consuming to design ODs manually, even by domain experts. Automatic discovery techniques for ODs [12, 16,17] are hence studied, just like those for FDs [9,13], DCs [2,4] and DDs [14], among others. The need for automatic OD discovery is even more evident, since the number of possible list-based ODs is of a factorial complexity in the number of attributes [12], much larger than that of FDs.

Discovering ODs is already shown to be a hard problem. Worse, Data in practice is typically dynamic, *i.e.,* frequently updated. Even if tuple deletions are not allowed, tuple insertions are generally supported on most data. It is too expensive to recompute all ODs, especially when data grow with tuple insertions. This highlights the quest for incremental OD discovery techniques, to update the set Σ of discovered (*valid* and *minimal*) ODs on an instance D as a set $\triangle D$ of tuple insertions is applied to D. Intuitively, when $\triangle D$ is small compared to D, it is more efficient to find update $\triangle\Sigma$ to Σ than the entire set of ODs on $D + \triangle D$ from scratch. However, the incremental OD discovery problem is very intricate, as illustrated by the example below.

Example 2: We illustrate two ways to find ODs in $\triangle\Sigma$. In Fig. 1, suppose $\triangle D$ with a single tuple t' is applied to D (neglecting t'' at this time).

(1) In instance D, the algorithm for OD discovery, *e.g.,* [12], finds $\mathsf{AB} \mapsto \mathsf{CD}$ and adds it into Σ. It can be verified that $\mathsf{AB} \mapsto \mathsf{CD}$ is no longer valid (holds) on $D + \triangle D$. As a violation, tuples t_1, t' agree on their AB's value, but have different CD's values. An incremental OD discovery algorithm then has to compute updates $\triangle\Sigma$ to Σ. It finds that $\mathsf{ABE} \mapsto \mathsf{CD}$, $\mathsf{ABF} \mapsto \mathsf{CD}$, $\mathsf{ABG} \mapsto \mathsf{CD}$ and $\mathsf{AB} \mapsto \mathsf{C}$ are all valid on $D+\triangle D$. Therefore, it adds all these ODs into $\triangle\Sigma^+$, the set of ODs to be added into Σ, and adds $\mathsf{AB} \mapsto \mathsf{CD}$ into $\triangle\Sigma^-$, the set of ODs to be removed from Σ. Note that none of $\mathsf{ABE} \mapsto \mathsf{CD}$, $\mathsf{ABF} \mapsto \mathsf{CD}$, $\mathsf{ABG} \mapsto \mathsf{CD}$ or $\mathsf{AB} \mapsto \mathsf{C}$ is in Σ, since they are not *minimal* (formalized in Sect. 2). Theoretically, they are not in Σ because $\mathsf{AB} \mapsto \mathsf{CD}$ is in Σ and $\mathsf{AB} \mapsto \mathsf{CD}$ logically implies them [12,15]: any instance that satisfies $\mathsf{AB} \mapsto \mathsf{CD}$ also satisfies them. However, when $\mathsf{AB} \mapsto \mathsf{CD}$ no longer holds, our incremental method has to discover them, since they are valid on $D + \triangle D$, and are *minimal* now.

(2) Both $\mathsf{E} \mapsto \mathsf{F}$ and $\mathsf{EB} \mapsto \mathsf{G}$ are in Σ. $\mathsf{EFB} \mapsto \mathsf{G}$ is not in Σ; it is valid but not minimal. Theoretically, $\mathsf{EFB} \mapsto \mathsf{G}$ is an "embedded" OD *w.r.t.* $\mathsf{EB} \mapsto \mathsf{G}$ according to $\mathsf{E} \mapsto \mathsf{F}$ [12,15]. Since the order of E determines that of F, adding F after E in a list does not impose any new order restrictions and is regarded as redundancy (formalized in Sect. 2). However, $\mathsf{E} \mapsto \mathsf{F}$ is not valid after t' is inserted. Our incremental algorithm has to check the validity of $\mathsf{EFB} \mapsto \mathsf{G}$. Indeed, although $\mathsf{EB} \mapsto \mathsf{G}$ is not valid on $D + \triangle D$, $\mathsf{EFB} \mapsto \mathsf{G}$ is valid. $\mathsf{EFB} \mapsto \mathsf{G}$ is hence put into $\triangle\Sigma^+$: it is both valid and minimal now. Note that even when $\mathsf{EB} \mapsto \mathsf{G}$ is valid on $D + \triangle D$ and is not removed from Σ, $\mathsf{EFB} \mapsto \mathsf{G}$ is still minimal. This is because EB is not a *prefix* of EFB. Also note that $\mathsf{EFB} \mapsto \mathsf{G}$ *cannot* be discovered following the strategy in (1), *i.e.,* adding attributes to the tail of LHS attribute list, or removing attributes from the tail of RHS attribute list. □

Contributions. We make a first effort to investigate incremental OD discovery.

(1) We formalize the incremental OD discovery problem (Sect. 3). We show that this problem has a same complexity as its batch (non-incremental) counterpart, in terms of the traditional complexity analysis. Nevertheless, we prove that the incremental problem has a good data locality. Specifically, given an OD φ valid on D and a set $\triangle D$ of tuple insertions to D, it is linear in the size of $\triangle D$ to check the validity of φ on $D + \triangle D$.
(2) We present efficient methods for incremental OD discovery (Sect. 4). We present an intelligent traversal strategy for finding new valid and minimal ODs on $D + \triangle D$, based on those already discovered ODs in Σ. We study techniques for choosing indexes, such that the access to D is minimized, and hence the required *local* data can be effectively fetched. Our approach has a desirable property that it is independent of the size of D.
(3) Using real-life data, we experimentally verify that our incremental algorithm outperforms the batch counterpart by orders of magnitude (Sect. 5).

Related Work. Dependency discovery is one of the most important aspects of data profiling. To alleviate the burden of users, automatic dependency discoveries are conducted for a host of different constraints; see *e.g.,* FDs [9,13], conditional FDs (CFDs) [6,8], DCs [2,4] and DDs [14].

Order dependencies (ODs) [15,18] state a relationship of order between lists of attributes. Theoretical foundations of ODs are well discussed in [15,18]. ODs properly subsume FDs, and are well employed in query optimizations concerning order. [12] presents the first approach for discovering ODs, with a level-wise bottom-up traversal of the lattice of permutations of attributes, an efficient OD validation method and some pruning rules to reduce the search space. Since ODs are defined on lists, the search space (the number of possible ODs) is factorial in the number of attributes. [16,17] present a polynomial mapping form ODs defined in lists to a canonical form of ODs in sets; this canonical form of ODs in sets has an advantage that the search space is exponential in the number of attributes. [16,17] then present discovery techniques for ODs via set-based axioms. In this paper, we follow the notation of list-based ODs. This is because (1) ODs defined in lists are preferable, since they naturally model the lexicographic orders employed in the SQL order-by clause; (2) each list-based OD of the form $\mathsf{X} \mapsto \mathsf{Y}$ has to be expressed in $|\mathsf{X}|\cdot|\mathsf{Y}|$ set-based ODs, where $|\mathsf{X}|$ (resp. $|\mathsf{Y}|$) is the number of attributes in list X (resp. Y). Therefore, the discovery of set-based ODs does not lead to the discovery of list-based ODs; and (3) we address the incremental OD discovery problem, to improve the efficiency from another perspective.

Incremental techniques are developed in different aspects of data quality. [3] discusses incremental data repairing for CFDs. Taking a clean relation D *w.r.t.* a set Σ of CFDs and a set $\triangle D$ of tuple insertions, [3] presents methods to repair tuples in $\triangle D$ such that Σ is satisfied. [7] investigates incremental detection of CFD violations in distributed data. Given a set V of violations *w.r.t.* a set Σ of CFDs on a distributed database D and updates $\triangle D$ to D, [7] aims to find changes $\triangle V$ to V, with minimum data shipment among sites. Our work considers incremental constraint discovery for ODs, and significantly differs from [3,7].

To our best knowledge, the only incremental algorithm for constraint discovery on dynamic data is [1], concerning unique column combinations (Uccs), *a.k.a.* candidate keys. [1] employs indexes on attributes to reduce access to old data, and expands attributes in old Uccs for new Uccs when old Uccs no longer hold. Note that OD subsumes FD, and FD subsumes Ucc. We consider ODs with multiple LHS and RHS attributes in lists. This makes the traversal for OD candidates, OD validations and index choice far more complicated, compared to [1].

2 Preliminaries

We review basic notations and formal definitions of ODs [12,15–18].

Relation. $R(A_1, \dots, A_m)$ denotes a relation schema, where each $A_j (j \in [1, m])$ denotes a single attribute. D denotes a specific instance, and t, s denote tuples. For an attribute A_j, t_{A_j} denotes the value of attribute A_j in a tuple t.

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

For an attribute list $\mathsf{X} = [A_1, \dots, A_k]$, we say X *contains* another list Y, when there exists some $1 \leq i \leq j \leq k$ such that $\mathsf{Y} = [A_i, \dots, A_j]$. We use $prefixes(\mathsf{X})$ to denote the set of all possible prefixes of X, *i.e.,* $[A_1, \dots, A_i]$ for any $i \leq k$.

Order on Lists. For a tuple t and an attribute list $\mathsf{X} = [A_1, \dots, A_k]$, we use t_{X} to denote the projection of tuple t on X, *i.e.,* $[t_{A_1}, \dots, t_{A_k}]$. For two tuples t, s,

(1) $t \prec_{\mathsf{X}} s$ if there exists some $i \leq k$ such that $t_{A_i} < s_{A_i}$ and for all $j < i$, $t_{A_i} = s_{A_i}$.
(2) $t =_{\mathsf{X}} s$ when $t_{A_i} = s_{A_i}$ for all $i \in [1, k]$.
(3) $t \preceq_{\mathsf{X}} s$ if $t \prec_{\mathsf{X}} s$ or $t =_{\mathsf{X}} s$.

Order Dependency [12,15–18]**.** For attributes lists X, Y on schema R, $\mathsf{X} \mapsto \mathsf{Y}$ denotes an *order dependency*. An instance D of R satisfies an OD $\varphi = \mathsf{X} \mapsto \mathsf{Y}$, if for any two tuples $t, s \in D$, when $t \preceq_{\mathsf{X}} s$, $t \preceq_{\mathsf{Y}} s$. We say φ is valid on D and φ holds on D interchangeably. If $\mathsf{X} \mapsto \mathsf{Y}$ is not valid on D, we write $\mathsf{X} \not\mapsto \mathsf{Y}$.

Remark. As stated in [15,18], ODs *strictly generalize* FDs. Specifically, each OD $\mathsf{X} \mapsto \mathsf{Y}$ has an "embedded FD" $\mathcal{X} \rightarrow \mathcal{Y}$; if $\mathsf{X} \mapsto \mathsf{Y}$ holds, $\mathcal{X} \rightarrow \mathcal{Y}$ holds.

Violations of Order Dependency [15,18]**.** For an OD $\varphi = \mathsf{X} \mapsto \mathsf{Y}$, two sources of OD violations exist:

(1) A split *w.r.t.* φ is a pair of tuples t and s such that $t =_{\mathsf{X}} s$, but $t \neq_{\mathsf{Y}} s$.
(2) A swap *w.r.t.* φ is a pair of tuples t and s such that $t \prec_{\mathsf{X}} s$ but $s \prec_{\mathsf{Y}} t$.

Example 3: (1) A split is actually a violation of the "embedded" FD. In Fig. 1, t_1 and t_2 lead to a split *w.r.t.* $\mathsf{F} \mapsto \mathsf{G}$. $t_1 =_{\mathsf{F}} t_2$, but $t_1 \neq_{\mathsf{G}} t_2$. (2) t_1 and t_3 cause a swap *w.r.t.* $\mathsf{A} \mapsto \mathsf{B}$. $t_1 \prec_{\mathsf{A}} t_3$, but $t_3 \prec_{\mathsf{B}} t_1$. □

The number of ODs valid on an instance can be very large. Similar to discovery techniques for other constraints, it is more instructive to find *minimal* valid ODs than to find all valid ODs. List-based ODs lead to an intricate definition of minimality. We follow similar criteria as [12,15], formalized as follows.

Minimality of an Attribute List. An attribute list X is minimal, iff for any disjoint sub-lists Y and W in X: if W follows (maybe not directly) Y,$\mathsf{Y} \not\mapsto \mathsf{W}$.

Intuitively, when the order of Y determines that of W, adding W after Y in a list does not impose any new order restrictions. It is easy to see that an attribute A_i occurs at most once in any minimal attribute list X.

Minimality of ODs. An OD $\mathsf{X} \mapsto \mathsf{Y}$ is minimal, iff

(1) $\nexists$ $\mathsf{X}' \mapsto \mathsf{Y}\mathsf{Y}'$, such that $\mathsf{X}' \in prefixes(\mathsf{X})$ and $\mathsf{X}' \mapsto \mathsf{Y}\mathsf{Y}'$ is valid (if $\mathsf{X}' = \mathsf{X}$, Y' is not empty; otherwise Y' can be empty); and
(2) both X and Y are minimal.

Example 4: Recall the instance D presented in Fig. 1. (1) $\mathsf{AB} \mapsto \mathsf{CD}$ is minimal, but $\mathsf{ABE} \mapsto \mathsf{CD}$ is not minimal. (2) Because $\mathsf{E} \mapsto \mathsf{F}$, EFB is not a minimal attribute list and hence $\mathsf{EFB} \mapsto \mathsf{G}$ is not a minimal OD. □

3 Data Locality for Incremental OD Discovery

We formalize the incremental OD discovery problem and show its complexity. We then justify that the incremental OD discovery problem has good data locality.

Incremental OD Discovery. Given a relation D of schema R, a set Σ of valid and minimal ODs on D, and a set $\triangle D$ of tuple insertions to D, incremental OD discovery is to find, changes $\triangle\Sigma$ to Σ that makes $\Sigma \oplus \triangle\Sigma$ a set of valid and minimal ODs on $D + \triangle D$; $\triangle\Sigma$ contains both new ODs to be added to Σ and outdated ODs to be removed from Σ.

Note that each batch (non-incremental) OD discovery problem on D can be directly modeled as an incremental OD discovery problem with inputs D', $\triangle D'$ and Σ, by setting $D' = \phi$, $\triangle D' = D$ and $\Sigma = \phi$. Since the incremental OD discovery includes the batch counterpart as a special case, the incremental problem at least has a same complexity as the batch one in terms of traditional complexity. In practice, $\triangle D$ is typically (much) smaller than D. In contrast to batch algorithms that recompute the output from scratch, an incremental algorithm can greatly improve efficiency if its cost is independent of D.

In light of this, we classify *local* data for an inserted tuple t' and an OD φ that is already valid on D, followed by computations concerning only local data.

Local Data of a Single Tuple Insertion. Given an OD $\mathsf{X} \mapsto \mathsf{Y}$ valid on D and a tuple t' inserted into D, we can find the following three sets of tuples on D, as *local data* of t' *w.r.t.* $\mathsf{X} \mapsto \mathsf{Y}$:

$\underline{equ(\mathsf{X}, t')}$: tuple $s \in equ(\mathsf{X}, t')$, if $s =_{\mathsf{X}} t'$;

$\underline{low(\mathsf{X}, t')}$: tuple $s \in low(\mathsf{X}, t')$, if (1) $s \prec_{\mathsf{X}} t'$; and (2) there exists no s' such that $s \prec_{\mathsf{X}} s' \prec_{\mathsf{X}} t'$;

$\underline{high(\mathsf{X}, t')}$: tuple $s \in high(\mathsf{X}, t')$, if (1) $t' \prec_{\mathsf{X}} s$; and (2) there exists no s' such that $t' \prec_{\mathsf{X}} s' \prec_{\mathsf{X}} s$;

Note that $equ(\mathsf{X}, t')$ (resp. $low(\mathsf{X}, t')$, $high(\mathsf{X}, t')$) may be empty. Incremental OD discovery takes as inputs D and the set Σ of ODs valid on D. Hence, some auxiliary data structure can be built to effectively obtain required local data.

Example 5: In Fig. 2(a), we show a simplified B+ tree built on D (Fig. 1) with AB as the key. For each key value in a leaf node, we store the set of tuple *ids*. In addition, we build a doubly linked list between successive leaf nodes. For t', $equ(\mathsf{AB}, t') = \{t_1, t_2\}$, $high(\mathsf{AB}, t') = \{t_3\}$ and $low(\mathsf{AB}, t') = \{\ \}$. With the B+ tree, it takes $O(\log |D|)$ to fetch $equ(\mathsf{AB}, t')$, and then $O(1)$ to fetch (non-empty) $low(\mathsf{AB}, t')$ (resp. $high(\mathsf{AB}, t')$), where $|D|$ is the number of tuples in D. □

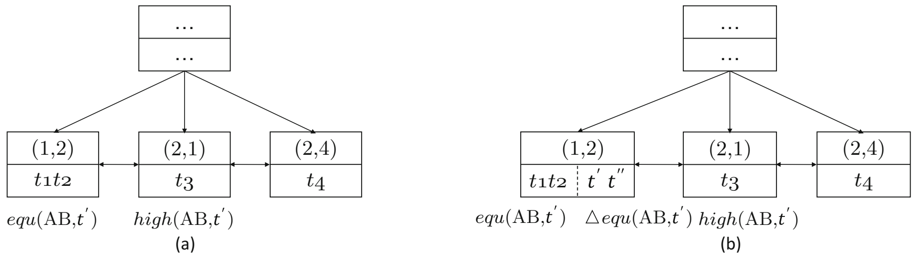


Fig. 2. Example local data

Note that $\forall s, t \in equ(\mathsf{X}, t')$ (resp. $low(\mathsf{X}, t')$, $high(\mathsf{X}, t')$), $t =_{\mathsf{Y}} s$ on instance D, since $\mathsf{X} \mapsto \mathsf{Y}$ is valid on D. We choose an arbitrary tuple from $equ(\mathsf{X}, t')$ (resp. $low(\mathsf{X}, t')$, $high(\mathsf{X}, t')$), denoted as $t_e^{'\mathsf{X}}$ (resp. $t_l^{'\mathsf{X}}$, $t_h^{'\mathsf{X}}$), when the set is not empty. The following theorem states that whether $\mathsf{X} \mapsto \mathsf{Y}$ is valid on $D + \{t'\}$ concerns computations only on t' and $t_e^{'\mathsf{X}}$, $t_l^{'\mathsf{X}}$, $t_h^{'\mathsf{X}}$.

Theorem 1: *$\varphi = \mathsf{X} \mapsto \mathsf{Y}$ is valid on $D + \{t'\}$, when (1) there is no* split *between t' and $t_e^{'\mathsf{X}}$; and (2) there is no* swap *between t' and $t_l^{'\mathsf{X}}$ (resp. $t_h^{'\mathsf{X}}$).* □

Local Data of $\triangle D$. We then consider $\triangle D$ with multiple tuple insertions. For a tuple t' in $\triangle D$, we still denote by $equ(\mathsf{X}, t')$, $low(\mathsf{X}, t')$ and $high(\mathsf{X}, t')$ local data on D, while denote by $equ'(\mathsf{X}, t')$, $low'(\mathsf{X}, t')$ and $high'(\mathsf{X}, t')$ local data on $D + \triangle D$. $equ'(\mathsf{X}, t') = equ(\mathsf{X}, t') \cup \triangle equ(\mathsf{X}, t')$, where $\triangle equ(\mathsf{X}, t')$ is the set of tuples $t \in \triangle D$ such that $t =_{\mathsf{X}} t'$. Obviously, $t' \in \triangle equ(\mathsf{X}, t')$; $equ'(\mathsf{X}, t') = equ'(\mathsf{X}, t'')$ if $t'' =_{\mathsf{X}} t'$. Similarly for $low'(\mathsf{X}, t')$ and $high'(\mathsf{X}, t')$.

Leveraging the auxiliary structure to effectively obtain required data, Theorem 2 shows the good data locality of incremental OD validation, as an important building block of incremental OD discovery.

Theorem 2: *For any OD φ that is valid on D, it is linear in $|\triangle D|$ to check the validity of φ on $D + \triangle D$, where $|\triangle D|$ is the number of tuples in $\triangle D$.* □

Algorithm. We prove Theorem 2 by providing algorithm $\triangle Check(\varphi)$ with the required property, for checking the validity of $\varphi = \mathsf{X} \mapsto \mathsf{Y}$ on $D + \triangle D$. $\triangle Check(\varphi)$ divides $\triangle D$ into k disjoint sets $= \{\triangle D_1, \ldots, \triangle D_k\}$ with hashing, such that $\forall t', t'' \in \triangle D_i$ $(i \in [1, k])$, $t' =_{\mathsf{X}} t''$. It then selects an arbitrary t'_i in each $\triangle D_i$.

(1) To check swap *w.r.t.* t'_i, it finds $t_e^{'max} = argmax_t(t_{\mathsf{Y}})$ and $t_e^{'min} = argmin_t(t_{\mathsf{Y}})$ in all $t \in equ'(\mathsf{X}, t'_i)$ (ties broken by an arbitrary one). It then finds $t_l^{'max}$ (resp. $t_h^{'min}$) in $low'(\mathsf{X}, t'_i)$ (resp. $high'(\mathsf{X}, t'_i)$) similarly. There is no swap iff $t_e^{'max} \preceq_{\mathsf{Y}} t_h^{'min}$ and $t_l^{'max} \preceq_{\mathsf{Y}} t_e^{'min}$.

(2) To check split *w.r.t.* t'_i, it suffices to check whether $t_e^{'max} =_{\mathsf{Y}} t_e^{'min}$.

Example 6: Consider $\triangle D$ with two tuples t', t'' (Fig. 1) and $\varphi = \mathsf{AB} \mapsto \mathsf{CD}$. As shown in Fig. 2(b), $equ'(\mathsf{AB}, t') = equ'(\mathsf{AB}, t'') = \{t_1, t_2, t', t''\}$: $equ(\mathsf{AB}, t') = \{t_1, t_2\}$, $\triangle equ(\mathsf{AB}, t') = \{t', t''\}$. $high'(\mathsf{AB}, t') = high(\mathsf{AB}, t') = \{t_3\}$.

$t_e^{'max} = t''$, $t_e^{'min} = t_1$ and $t_h^{'min} = t_3$. split exists since $t_1 \neq_{\mathsf{CD}} t''$, and swap exists since $t_3 \prec_{\mathsf{CD}} t''$. □

Complexity. It is easy to see the correctness of $\triangle Check(\varphi)$. We then prove $\triangle Check(\varphi)$ is linear in $|\triangle D|$. Observe that the total number of required $equ'(\mathsf{X}, t'_i)$, $low'(\mathsf{X}, t'_i)$ and $high'(\mathsf{X}, t'_i)$ for all t'_i is at most $3 \times |\triangle D|$. In $equ'(\mathsf{X}, t'_i)$, it takes $O(1 + |\triangle equ(\mathsf{X}, t'_i)|)$ to find $t_e^{'max}$ and $t_e^{'min}$, where $|\triangle equ(\mathsf{X}, t'_i)|$ is the number of tuples in $\triangle equ(\mathsf{X}, t'_i)$. This is because all tuples in $equ(\mathsf{X}, t'_i)$ agree on values of Y. Note that the sum of $|\triangle equ(\mathsf{X}, t'_i)|$ for all t'_i equals $|\triangle D|$.

If $\triangle low(\mathsf{X}, t'_i)$ is not empty for some t'_i, $low'(\mathsf{X}, t'_i) = equ'(\mathsf{X}, s')$ for any $s' \in \triangle low(\mathsf{X}, t'_i)$. In this case, no additional computation on $low'(\mathsf{X}, t'_i)$ is required. If $\triangle low(\mathsf{X}, t'_i)$ is empty, it takes $O(1)$ to find $t_l^{'max}$ in $low'(\mathsf{X}, t'_i)$ because all tuples in $low(\mathsf{X}, t'_i)$ have a same value on Y. Similarly for $high'(\mathsf{X}, t'_i)$.

To sum up, $\triangle Check(\varphi)$ is linear in $|\triangle D|$.

4 Incremental OD Discovery

We first discuss methods for finding $\triangle\Sigma$ on $D + \triangle D$, based on Σ. We then study techniques for choosing indexes, to minimize access to the original data.

4.1 Finding ODs in $\triangle\Sigma$

Note that $\triangle\Sigma$ consists of two disjoint sets $\triangle\Sigma^+$ and $\triangle\Sigma^-$; $\triangle\Sigma^+$ contains new valid and minimal ODs as additions to Σ, while $\triangle\Sigma^-$ contains non-valid ODs that should be removed from Σ. Taking as input the set Σ of minimal and valid ODs on D, it is relatively easy to compute $\triangle\Sigma^-$. As stated in Theorem 2, we can effectively check the validity of any OD φ in Σ on $D + \triangle D$ by $\triangle Check(\varphi)$. If φ no longer holds on $D + \triangle D$, we add it to $\triangle\Sigma^-$.

It is, however, much more intricate to compute $\triangle\Sigma^+$ as shown in Example 2. To fully take advantage of incremental computations, we should always leverage $\triangle Check(\varphi)$ in the validation of any OD φ; a prerequisite is that φ must hold on D. This is possible since any OD valid on $D + \triangle D$ must be valid on D, and hence

ODs in $\triangle\Sigma^+$ can be computed based on ODs in Σ. We present two ways to find ODs (candidates) in $\triangle\Sigma^+$, referred to as $\mathsf{enrichment}$ and $\mathsf{expansion}$, respectively.

Enrichment of an Attribute List. Given an OD $\varphi = \mathsf{X} \mapsto \mathsf{Y} \in \Sigma$, but invalid on $D + \triangle D$, and an attribute list Z, when (1) Z contains X, and (2) Z and Y are disjoint, we "enrich" Z by φ, to generate a set of attribute lists, denoted by $enrich(\mathsf{Z}, \varphi)$. W.l.o.g., let $\mathsf{Z} = \mathsf{X'XA_{1'}} \ldots \mathsf{A_{k'}}$. $enrich(\mathsf{Z}, \varphi) = \{ \mathsf{Z}, \mathsf{X'XYA'_1} \ldots \mathsf{A'_k}, \ldots, \mathsf{X'XA'_1} \ldots \mathsf{A'_kY} \}$. If either condition (1) or (2) is false, $enrich(\mathsf{Z}, \varphi) = \{ \mathsf{Z} \}$.

Intuitively, $enrich(\mathsf{Z}, \varphi)$ (excluding Z) is a set of *non-minimal* attribute lists due to the validity of $\mathsf{X} \mapsto \mathsf{Y}$ on D. When $\mathsf{X} \mapsto \mathsf{Y}$ no longer holds on $D + \triangle D$, these attribute lists become minimal. We then present our first way to generate candidates in $\triangle\Sigma^+$, referred to as $\mathsf{enrichment}$.

Enrichment of an OD. Given an OD $\varphi = \mathsf{X} \mapsto \mathsf{Y}$ in Σ and a set Υ of ODs $\{\varphi_1, \ldots, \varphi_m\}$, where each φ_i ($i \in [1, m]$) is in Σ, but is invalid on $D + \triangle D$, $\mathsf{enrichment}$ of φ by Υ, denoted by $Enrich(\varphi, \Upsilon)$ is to generate a set of ODs = $\{\mathsf{U} \mapsto \mathsf{V}\}$, where $\mathsf{U} \in enrich(\mathsf{X}, \varphi_i)$, $\mathsf{V} \in enrich(\mathsf{Y}, \varphi_j)$, $\forall \varphi_i, \varphi_j \in \Upsilon$.

Example 7: Suppose $\varphi = \mathsf{AB} \mapsto \mathsf{CD}$ and $\Upsilon = \{\mathsf{B} \mapsto \mathsf{E}, \mathsf{C} \mapsto \mathsf{F}\}$, $\mathsf{enrichment}$ of φ by Υ is $\{\mathsf{AB} \mapsto \mathsf{CD}, \mathsf{AB} \mapsto \mathsf{CFD}, \mathsf{AB} \mapsto \mathsf{CDF}, \mathsf{ABE} \mapsto \mathsf{CD}, \mathsf{ABE} \mapsto \mathsf{CFD}, \mathsf{ABE} \mapsto \mathsf{CDF}\}$. □

Complexity. Observe that $\mathsf{enrichment}$ is conducted based on attributes in ODs. Its complexity is irrelevant of $|R|$, the number of attributes of the schema R, and it requires no data access to D.

Intuitively, $\mathsf{enrichment}$ of φ by Υ is to enrich the LHS and RHS attribute lists of φ by ODs in Υ respectively. It is easy to prove the following results. (1) Any OD in $Enrich(\varphi, \Upsilon)$ is valid on D; and (2) none of ODs in $Enrich(\varphi, \Upsilon)$ is minimal on D (excluding $\varphi = \mathsf{X} \mapsto \mathsf{Y}$). ODs generated by $\mathsf{enrichment}$ are only candidates in $\triangle\Sigma^+$, since some of them are invalid on $D + \triangle D$. Based on those invalid candidates and ODs in $\triangle\Sigma^-$, we provide another approach to computing ODs in $\triangle\Sigma^+$, referred to as $\mathsf{expansion}$.

Algorithm. Algorithm Expand is presented to apply $\mathsf{expansion}$ to an OD $\varphi = \mathsf{X} \mapsto \mathsf{Y}$ valid on D but invalid on $D + \triangle D$. It produces a set Υ of ODs valid on $D + \triangle D$. Each OD $\in \Upsilon$ is of the form $\mathsf{XZ} \mapsto \mathsf{Y'}$, where $\mathsf{Y'} \in prefixes(\mathsf{Y})$. It is easy to see that these ODs are valid but not minimal on D.

(1) Expand first eliminates possible swap by removing attributes from the tail of Y one by one, until no swap exists or Y is empty (lines 2–4). Note that adding attributes to the tail of X does not help remove swap. If swap cannot be removed, Expand returns an empty set; no valid ODs can be generated based on $\mathsf{X} \mapsto \mathsf{Y}$.
(2) Expand then turns to eliminate split. In its loop (lines 6–11), Expand tries $\mathsf{X} \mapsto \mathsf{Y'}$ for each prefix $\mathsf{Y'}$ of Y (line 11). (a) If no split exists, Expand returns current results (line 8). This is because adding more attributes to the tail of X and (or) removing attributes from the tail of Y cannot further produce *minimal* ODs. (b) Otherwise, function *ExpandL* is called to remove split by

Algorithm 1: Expand

```
input  : a relation $D$ of schema $R$, an OD $\varphi = \mathsf{X} \mapsto \mathsf{Y}$ valid on $D$, and a set
         $\triangle D$ of tuple insertions to $D$.
output: a set $\Upsilon$ of ODs valid on $D + \triangle D$, where each OD $\in \Upsilon$ is of the form
         $\mathsf{XZ} \mapsto \mathsf{Y}'$, where $\mathsf{Y}' \in prefixes(\mathsf{Y})$.
1  $\Upsilon := \{\}$; $LHS_{set} := \{\}$; $V_{set} := \{\}$;
2  while $\mathsf{Y}$ is not empty do
3      if there is no swap detected by $\triangle Check(\varphi)$ then break;
4      Remove the last attribute from $\mathsf{Y}$;
5  if $\mathsf{Y}$ is empty then return { };
6  while $\mathsf{Y}$ is not empty do
7      Check split by $\triangle Check(\varphi)$, and put violations into $V_{set}$;
8      if $V_{set}$ is empty then return $\Upsilon \cup \{\mathsf{X} \mapsto \mathsf{Y}\}$;
9      foreach $\mathsf{X}' \in ExpandL(V_{set}, \mathsf{X}, \mathsf{Y}, LHS_{set})$ do
10         Add $\mathsf{X}' \mapsto \mathsf{Y}$ to $\Upsilon$; Add $\mathsf{X}'$ to $LHS_{set}$;
11     Remove the last attribute from $\mathsf{Y}$;
12 return $\Upsilon$;
13 Function $ExpandL(V_{set}, \mathsf{U}, \mathsf{W}, LHS_{set})$
   input  : a set $V_{set}$ of split violations; each violation is a set of tuples with a
            same $\mathsf{U}$'s value but different $\mathsf{W}$'s values. $LHS_{set}$ is a set of attribute
            lists: $\forall$ $\mathsf{Z} \in LHS_{set}$, $\mathsf{Z} \mapsto \mathsf{WW}'$ is valid on $D + \triangle D$ for some list $\mathsf{W}'$.
   output: a set $\Omega$ of attribute lists, $\forall U' \in \Omega$, $\mathsf{U}' \mapsto \mathsf{W}$ is valid on $D + \triangle D$.
14 $\Omega := \{\}$;
15 foreach attribute $A \in R \backslash \mathcal{U}$ such that $swap\_free(V_{set}, A)$, and there is no $\mathsf{Z}$ in
   $LHS_{set}$, where $\mathsf{Z} \in prefixes(\mathsf{U}A)$ do
16     $V'_{set} := update(V_{set}, A)$;
17     if $V'_{set}$ is empty then $\Omega := \Omega \cup \{ \mathsf{U}A \}$;
18     else $\Omega := \Omega \cup ExpandL(V'_{set}$, $\mathsf{U}A$, $\mathsf{W}$, $LHS_{set})$;
19 return $\Omega$;
```

adding attributes to the tail of X, and its results are kept (lines 9–10). Recall that $\triangle Check(\varphi)$ divides tuples in $\triangle D$ into $\{\triangle D_1, \ldots, \triangle D_k\}$ based on their X's values. It then detects split on $equ'(\mathsf{X}, t'_i)$ for a tuple t'_i in each $\triangle D_i$. All $equ'(\mathsf{X}, t'_i)$ with different Y's values are collected in V_{set} (line 7), as a parameter of *ExpandL* (line 9).

(3) Function *ExpandL* takes a set V_{set} of split violations, where each violation is a set of tuples that have a same value on U but different values on W. *ExpandL* returns a set Ω of attribute lists. Each $\mathsf{U}' \in \Omega$ is obtained from U by adding attributes to its tail, and $\mathsf{U}' \mapsto \mathsf{W}$ is valid on $D + \triangle D$. Instead of simply trying permutation of all attributes in $R \backslash \mathcal{U}$, *ExpandL* employs both instance-based and schema-based strategies to effectively prune the search space (line 15). (a) *ExpandL* only chooses attribute A that does not cause swap among tuples in a same set in V_{set} (checked by $swap\_free(V_{set}, A)$). Recall that in V_{set}, each set (violation) $vio = \{t_1, \ldots, t_m\}$, contains tuples that have a same value on U but different values on W. Adding A to the tail of U does not cause swap if for any two tuples t_i, t_j in vio, when $t_i \prec_{\mathsf{A}} t_j$,

$t_i \preceq_{\mathsf{W}} t_j$. (b) *ExpandL* avoids UA when $\mathsf{Z} \in \mathit{prefixes}(\mathsf{UA})$ for some Z in LHS_{set}; if $\mathsf{Z} \mapsto \mathsf{WW}'$ is valid, $\mathsf{UA} \mapsto \mathsf{W}$ is not minimal. Recall that LHS_{set} is maintained when new ODs are found (line 10).

(4) V_{set} is updated after A is added to the tail of U ($\mathit{update}(V_{set}, A)$) (line 16). Specifically, (a) it further divides sets in V_{set} based on values on A; tuples t', t'' are in a same set in V'_{set} when $t' =_{\mathsf{UA}} t''$; and (b) it discards set vio in V'_{set} when vio contains only one tuple, or $\forall t', t'' \in \mathit{vio}$, $t' =_{\mathsf{W}} t''$. If V'_{set} is empty, no further attribute additions are required (line 17). Otherwise, *ExpandL* is recursively called with updated violations and a lengthened LHS attribute list (line 18).

Example 8: Recall D and $\triangle D$ with two tuples t', t'' (Fig. 1) and $\varphi = \mathsf{AB} \mapsto \mathsf{CD}$. (1) Expand first tries to eliminate swap by removing attributes from the RHS of φ; this is done after removing D. (2) Expand detects split on $\mathit{equ}'(\mathsf{AB}, t')$, and hence the set $\{t_1, t_2, t', t''\}$ is put into V_{set}. (3) *ExpandL* tries to eliminate split by adding attributes to the end of AB. For example, adding E does not cause a swap. (4) After that, the only violation $\{t_1, t_2, t', t''\}$ in V_{set} is divided into four singleton sets $\{t_1\}$,$\{t_2\}$,$\{t'\}$,$\{t''\}$. Therefore, no split exists now. ABE is collected in Ω. There is no need for more attributes at the end of ABE, and step (3) is repeated by trying other attributes at the end of AB. (5) After *ExpandL* returns, all ODs of the form $\mathsf{ABS} \mapsto \mathsf{C}$ (resp. ABS) are collected in Υ (resp. LHS_{set}). Since no more attributes can be removed from the RHS of φ, Expand terminates. □

Complexity. (1) In terms of data complexity, recall that $\triangle \mathit{Check}(\varphi)$ is linear in $|\triangle D|$. On V_{set}, function $\mathit{update}(V_{set}, A)$ is linear in m on a set vio with m tuples (line 16). The most expensive part is function $\mathit{swap\_free}(V_{set}, A)$ (line 15). To check whether adding attribute A causes swap *w.r.t.* W, it takes $O(m \cdot \mathit{log} m)$ to sort tuples in vio based on values on A, followed by a linear scan to check values on W between successive tuples in $O(m)$. V_{set} is initialized with $\mathit{equ}'(\mathsf{X}, t'_i)$ with different Y's values (line 7). Hence, Expand is irrelevant of $|D|$. (2) Removing attributes from the tail of Y is linear in the size of Y, while adding attributes to the tail of X has a worst-case factorial complexity in the number of attributes in $R \backslash \mathcal{X}$. However, *ExpandL* is also bounded by the number of violations in V_{set}; the size of each violation monotonously decreases and all violations are eventually eliminated. Moreover, effective pruning rules are applied in lines 8, 15 and 17.

We are now ready to present the algorithm to compute $\triangle \Sigma$, by combining enrichment and expansion together.

Algorithm. Algorithm IncOD takes as inputs a relation D of schema R, a set Σ of valid and minimal ODs on D, and a set $\triangle D$ of tuple insertions to D. It computes $\triangle \Sigma$ such that $\Sigma \oplus \triangle \Sigma$ is a set of valid and minimal ODs on $D + \triangle D$.

(1) It initializes three empty sets Σ_{cand}, Σ_{valid} and Σ_{pre}, for OD candidates, new valid ODs in $D + \triangle D$, and ODs in Σ that are also valid on $D + \triangle D$, respectively. It validates every $\varphi \in \Sigma$ on $D + \triangle D$ by $\triangle \mathit{Check}(\varphi)$, and puts invalid (resp. valid) φ into $\triangle \Sigma^-$ (resp. Σ_{pre}) (lines 2–4).

(2) It applies enrichment to every $\varphi \in \Sigma$ by $\triangle \Sigma^-$, and collects results in Σ_{cand} (line 5). ODs in Σ_{cand} are then validated on $D + \triangle D$. Those valid ones

Algorithm 2: IncOD

```
input  : a relation $D$ of schema $R$, a set $\Sigma$ of valid and minimal ODs on $D$,
         and a set $\triangle D$ of tuples insertions to $D$.
output : $\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$. $\triangle\Sigma^+$ contains new valid and minimal ODs as
         additions to $\Sigma$, $\triangle\Sigma^-$ contains non-valid ODs to be removed from $\Sigma$.
1 $\Sigma_{cand} := \{\}$; $\Sigma_{valid} := \{\}$; $\Sigma_{pre} := \{\}$;
2 foreach $\varphi \in \Sigma$ do
3     if $\varphi$ is invalid by $\triangle Check(\varphi)$ then add $\varphi$ into $\triangle\Sigma^-$;
4     else add $\varphi$ into $\Sigma_{pre}$;
5 foreach $\varphi \in \Sigma$ do $\Sigma_{cand} := \Sigma_{cand} \cup Enrich(\varphi, \triangle\Sigma^-)$;
6 foreach $\varphi \in \Sigma_{cand}$ do
7     if $\varphi$ is valid by $\triangle Check(\varphi)$ then move $\varphi$ from $\Sigma_{cand}$ to $\Sigma_{valid}$;
8 foreach $\varphi \in \Sigma_{cand} \cup \triangle\Sigma^-$ do $\Sigma_{valid} := \Sigma_{valid} \cup Expand(D, \varphi, \triangle D)$;
9 $\triangle\Sigma^+ := Prune(\Sigma_{valid}, \Sigma_{pre})$;
```

are moved from Σ_{cand} to Σ_{valid} (lines 6–7). It applies expansion to ODs in $\Sigma_{cand} \cup \triangle\Sigma^-$, *i.e.,* ODs valid on D but invalid on $D + \triangle D$, and adds results to Σ_{valid} (line 8).

(3) It finally prunes non-minimal ODs in Σ_{valid} to get $\triangle\Sigma^+$ (line 9); Σ_{pre} is required in this step. (a) For each OD $\mathsf{X} \mapsto \mathsf{Y} \in \Sigma_{valid}$, it requires to check whether there exists some OD $\mathsf{U} \mapsto \mathsf{V}$ in $\Sigma_{valid} \cup \Sigma_{pre}$, such that $\mathsf{U} \in prefixes(\mathsf{X})$ and $\mathsf{Y} \in prefixes(\mathsf{V})$. It suffices to consider only those ODs $\mathsf{U} \mapsto \mathsf{V}$, whose $|\mathsf{U}| \leq |\mathsf{X}|$, and whose $|\mathsf{V}| \geq |\mathsf{Y}|$. (b) To verify whether X (resp. Y) is minimal, it requires to check whether there exists some OD $\mathsf{U} \mapsto \mathsf{V}$ in $\Sigma_{valid} \cup \Sigma_{pre}$, such that U is before V, both contained in X (resp. Y). It suffices to consider only those ODs $\mathsf{U} \mapsto \mathsf{V}$, whose $|\mathsf{U}| + |\mathsf{V}| \leq |\mathsf{X}|$ (resp. $|\mathsf{Y}|$).

Complexity. IncOD employs $\triangle Check(\cdot)$, $Enrich(\cdot)$ and $Expand(\cdot)$ in OD validations and computations of Σ_{cand}, Σ_{valid} and Σ_{pre}. Pruning of non-minimal ODs in Σ_{valid} concerns attributes of ODs in $\Sigma_{valid} \cup \Sigma_{pre}$, and requires no visits to D. To conclude, IncOD is irrelevant of $|D|$, and $\triangle\Sigma$ is computed based on ODs in Σ via enrichment and expansion only.

We provide insights into the interaction between enrichment and expansion, for developing optimization techniques.

Theorem 3: *On $D + \triangle D$, if* $\mathsf{W} \mapsto \mathsf{V}$ *does not cause a* split*, (1) when* $\mathsf{UWA_1} \ldots \mathsf{A_k} \mapsto \mathsf{Y}$ *is valid (resp. invalid),* $\mathsf{UWA_1} \ldots \mathsf{A_i V A_{i+1}} \ldots \mathsf{A_k} \mapsto \mathsf{Y}$ *is valid (resp. invalid); and (2) when* $\mathsf{UWA_1} \ldots \mathsf{A_k Z} \mapsto \mathsf{Y'}$ *is valid for some* Z*, and some* $\mathsf{Y'} \in prefixes(\mathsf{Y})$*,* $\mathsf{UWA_1} \ldots \mathsf{A_i V A_{i+1}} \ldots \mathsf{A_k Z} \mapsto \mathsf{Y'}$ *is valid.* □

Theorem 3 states that when an OD φ is invalid only due to swap (no split), (1) the enrichment of any valid OD ξ by φ also generates valid ODs; and (2) the enrichment of any invalid OD ξ by φ also generates invalid ODs, and any expansion of ξ that results in valid ODs also works for those ODs. We leverage these observations to avoid unnecessary expansion in our implementation. This optimization is proved to be very effective in our experimental studies, since expansion is the most expensive part of IncOD.

Algorithm 3: CoverIndex

```
input  : a set $\Sigma$ of ODs
output: a set of attribute lists on which indexes to be built
1 $U := \phi$; $output := \phi$;
2 foreach $\mathsf{X} \mapsto \mathsf{Y} \in \Sigma$ do
3     foreach $\mathsf{X}' \in prefixes(\mathsf{X})$ do
4         if $\mathsf{X}' \notin U$ then
5             add $\mathsf{X}'$ to $U$; $\mathsf{X}'.price := 0$; $\mathsf{X}'.weight := assignweight(\mathsf{X}')$;
6 while there exists $\mathsf{X} \mapsto \mathsf{Y}$ such that $\forall\, \mathsf{X}' \in prefixes(\mathsf{X})$, $\mathsf{X}'.price < \mathsf{X}'.weight$ do
7     $\mathsf{Z} := \underset{\mathsf{X}' \in prefixes(\mathsf{X})}{\text{argmin}} (\mathsf{X}'.weight - \mathsf{X}'.price)$;
8     foreach $\mathsf{X}' \in prefixes(\mathsf{X})$ do $\mathsf{X}'.price := \mathsf{X}'.price + \mathsf{Z}.weight - \mathsf{Z}.price$;
9 foreach $\mathsf{X}' \in U$ do if $\mathsf{X}'.weight = \mathsf{X}'.price$ then put $\mathsf{X}'$ into $output$;
```

4.2 Building Indexes

Only local data are required in IncOD. Our incremental OD discovery problem takes as inputs D and the set Σ of ODs valid on D, and hence some auxiliary structures can be built to help fetch those required data more efficiently.

We employ composite indexes (indexes on multiple attributes) as our auxiliary structure. In a composite index, tuples are sorted by concatenating values of the indexed attributes (see Example 5). Note that a composite index on attributes [ABC...] can be used when values of A, or AB or ABC are provided. We use memory-based B+ tree to implement composite indexes in this paper. Since B+ tree is well adopted in most commercial DBMS, our approach can be easily extended to handle data stored in DBMS as well.

To speed up data visits concerning $\mathsf{X} \mapsto \mathsf{Y}$, a straightforward way is to build a composite index ind_{X} on X. In practice when the number of ODs in Σ is large, building composite indexes on all distinct LHS attribute lists for ODs may become costly in terms of both computation and storage. We present another strategy that aims to build a *minimal* set of composite indexes and guarantees that for any OD at least one index is usable. We tackle this by relating the problem of building indexes to techniques for weighted vertex cover problems [19].

More specifically, for all $\mathsf{X} \mapsto \mathsf{Y}$ in Σ, (1) for any $\mathsf{X}' \in prefixes(\mathsf{X})$, we treat X' as a *vertex*, to build a set of vertices; and (2) we treat X as a *hyperedge*, with all $\mathsf{X}' \in prefixes(\mathsf{X})$ as its vertices. Then, our goal is to index at least one $\mathsf{X}' \in prefixes(\mathsf{X})$ for any $\mathsf{X} \mapsto \mathsf{Y}$ in Σ, the same as the goal of vertex cover, to pick at least one vertex for any hyperedge. We also assign a weight to each prefix $\mathsf{X}' = [A_1, \ldots, A_k]$, based on its *selectivity.* The weight of X' is computed as $(1 - \frac{dist(A_1)}{|D|}) \cdot \ldots \cdot (1 - \frac{dist(A_k)}{|D|})$, where $dist(A_i)$ is the number of distinct values of A_i. We use uniform random sampling to estimate $dist(A_i)$ in our implementation. If the weight of some X' is zero, we assign a small number α as its weight.

Algorithm. Algorithm CoverIndex is to find a set of attribute lists on which we build indexes. It is an adaption of the "pricing" method for weighted vertex cover. It first initializes the set U of vertices (lines 2–5). It then continues to pick

X (hyperedge) when neither of its prefix X' (vertex) is *tight* (line 6); a prefix X' is tight when X'.price = X'.weight. It then increases the price of all X' as much as possible, but guarantees that $\mathsf{X}'.price \leq \mathsf{X}'.weight$ (lines 7–8). Finally, all *tight* prefixes are collected as the output (line 9).

Complexity. CoverIndex terminates when at least one prefix is tight for each X, and all tight prefixes form a cover. CoverIndex is linear in the size of U, *i.e.,* the number of prefixes of all $\mathsf{X} \mapsto \mathsf{Y} \in \Sigma$. CoverIndex is a d-approximation algorithm where $d = \max(|X|)$ for all $\mathsf{X} \mapsto \mathsf{Y} \in \Sigma$ in our setting, following [19].

Remark. The index built on a prefix X' of X can be used for new ODs based on $\mathsf{X} \mapsto \mathsf{Y}$ by both expansion and enrichment. We denote by $local'(\mathsf{X}, t') = equ'(\mathsf{X}, t') \cup low'(\mathsf{X}, t') \cup high'(\mathsf{X}, t')$. Observe that (1) in expansion, we generate ODs of the form $\mathsf{XZ} \mapsto prefixes(\mathsf{Y})$, $local'(\mathsf{XZ}, t') \subseteq local'(\mathsf{X}', t')$; and (2) in enrichment, we generate ODs of the form $\mathsf{X}''\mathsf{Z} \mapsto \mathsf{Y}'$, where $\mathsf{X}'' \in prefixes(\mathsf{X})$. (a) If X' is a prefix of X'', $local'(\mathsf{X}''\mathsf{Z}, t') \subseteq local'(\mathsf{X}', t')$. (b) If X'' is a prefix of X', $local'(\mathsf{X}''\mathsf{Z}, t') \subseteq local'(\mathsf{X}'', t')$, and the index on X' can be used when X'' value is available.

5 Experimental Study

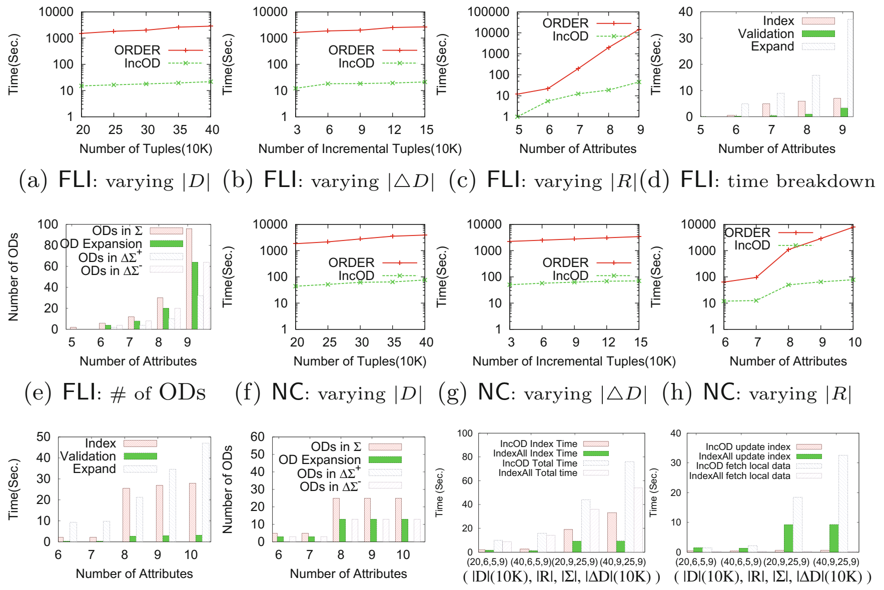
Experimental Setting. We used one machine with Intel Xeon CPU E5-2640 and 32GB RAM, ran each experiment 3 times and report the average here.

Data. We used two real datasets that have been used to evaluate OD discovery algorithms [12,16,17]. FLI is about US flights information, with 500K tuples and 20 attributes (www.transtats.bts.gov). NC contains data of registered voters from North Carolina, with 1M tuples and 22 attributes (ncsbe.gov). To improve efficiency and avoid uninteresting ODs, we replaced attribute values with integers in a way that the ordering is preserved, and removed tuples with NULL values, similar to [12,16,17].

Algorithms. We implemented our algorithms in Java: IncOD for incremental OD discovery (with $\triangle Check(\cdot)$, $Enrich(\cdot)$ and $Expand(\cdot)$) and CoverIndex for choosing attributes on which to build minimal indexes. For comparison, we obtained a batch OD discovery implementation ORDER [12] from www.metanome.de. To our best knowledge, this is the only algorithm for list-based OD discovery.

All experiments are controlled by 3 parameters: (1) $|D|$: the number of original tuples; (2) $|\triangle D|$: the number of tuples inserted into D; and (3) $|R|$: the number of attributes. We vary $|R|$ by taking random projections of the dataset. We employ ORDER to compute Σ on D, as inputs of CoverIndex for index building. IncOD then computes $\triangle\Sigma$ with inputs D, $\triangle D$ and Σ, leveraging indexes. The correctness of IncOD is verified by checking whether $\Sigma \oplus \triangle\Sigma$ equals the results of ORDER on $D + \triangle D$. We report the time of ORDER on $D + \triangle D$, against the time of IncOD for updating indexes and computing $\triangle\Sigma$ on tuple insertions.

Exp-1. We compare IncOD against ORDER using FLI. We set $|D| = 300\text{K}$,$|\triangle D| = 90\text{K}$ and $|R| = 8$ by default, and vary one parameter in each of the experiments.



(a) FLI: varying $|D|$ (b) FLI: varying $|\triangle D|$ (c) FLI: varying $|R|$ (d) FLI: time breakdown

(e) FLI: # of ODs (f) NC: varying $|D|$ (g) NC: varying $|\triangle D|$ (h) NC: varying $|R|$

(i) NC: time breakdown (j) NC: # of ODs (k) total and index (l) index time breakdown

Fig. 3. Experimental results

Varying $|D|$. Fig. 3(a) shows results by varying $|D|$ from 200K to 400K. ORDER scales well with $|D|$, consistent with results in [12]. Times of IncOD increase slightly, due to more *local* data *w.r.t.* $|\triangle D|$ as $|D|$ increases. IncOD outperforms ORDER by two orders of magnitude on all sizes of D. As an example, ORDER takes more than 45 min when $|D|$ is 400K, while IncOD takes only 22 s.

Varying $|\triangle D|$. Fig. 3(b) shows results by varying $|\triangle D|$ from 30K to 150K. We find IncOD scales very well with $|\triangle D|$: the time increases from 12 s to 21 s, when the ratio of $|\triangle D|$ to $|D|$ increases from 10% to 50%. IncOD outperforms ORDER by two orders of magnitude even when $|\triangle D|$ is half of $|D|$.

Varying $|R|$. We vary $|R|$ from 5 to 9 in Fig. 3(c). $|R|$ has the most effect on the time of list-based OD discovery, since the number of possible list-based ODs is of a factorial complexity in $|R|$. ORDER does not scale well with $|R|$, consistent with results in [12]. The scalability of IncOD is far more better. As $|R|$ increases from 8 to 9, the time for ORDER increases from 33 min to more than 4 h, while the time for IncOD only increases from 19 s to 46 s.

In Fig. 3(d) we decompose the overall time into times for (i) updating indexes and obtaining local data via indexes for $\triangle D$, (ii) OD validations by $\triangle Check(\cdot)$, and (iii) OD expansion; other times are marginal. The times for (i) and (ii) are related to ODs in Σ, while time (iii) is related to ODs for expansion, whose numbers are shown in Fig. 3(e). We also report in Fig. 3(e) the number of ODs in $\triangle\Sigma$. We find time (i) is short, due to the fact that almost all of ODs on FLI contain a single LHS attribute, and hence local data *w.r.t.* $\triangle D$ can be directly fetched via indexes built by CoverIndex. Time (ii) is also short; $\triangle Check(\cdot)$ requires only local data of $\triangle D$ and is linear in $|\triangle D|$ (Theorem 2). The time for expansion (Time (iii)) governs the overall time. The search space of our approach is much smaller than its batch counterpart since $\triangle \Sigma$ is computed based on Σ, fully leveraging incremental computations. Moreover, instance-based pruning rules in expansion and optimizations by Theorem 3 are proved to be quite effective.

Exp-2. We then compare IncOD against ORDER using NC, with $|D| = 300$K, $|\triangle D| = 90$K and $|R| = 9$ by default. We vary $|D|$ from 200K to 400K in Fig. 3(f), vary $|\triangle D|$ from 30K to 150K in Fig. 3(g), and vary $|R|$ from 6 to 10 in Fig. 3(h). In the same setting as Fig. 3(h), we report the time breakdown and number of related ODs in Figs. 3(i) and 3(j). The results confirm our observations on FLI. (1) IncOD significantly outperforms ORDER: IncOD is on average 48 and 51 times faster in Figs. 3(f) and 3(g), respectively. (2) IncOD scales much better with $|R|$. As $|R|$ increases from 6 to 10 in Fig. 3(h), the time for ORDER increases by more than two orders of magnitude, while the time for IncOD increases by less than 7 times. (3) Fig. 3(i) shows that more time is required in the index processing phase of NC. Most of ODs found on NC have multiple LHS attributes. Since CoverIndex may choose to build indexes on prefixes of LHS attributes, some post-processing after index visits is required to fetch local data of $\triangle D$. Specifically, to fetch local data $local'(\mathsf{X}, t')$ with an index $ind_{\mathsf{X}'}$ where X' is a prefix of X, we need to sort tuples in $local'(\mathsf{X}', t')$ on $\mathsf{X} \setminus \mathsf{X}'$; this incurs additional costs. Note that as $|R|$ increases from 8 to 10, the same number of ODs are found on D (Fig. 3(j)).

Exp-3. We evaluate different index strategies on NC. We denote by IndexAll when indexes are built on all distinct LHS attribute lists of ODs in Σ, and compare it against IncOD with CoverIndex. We denote by ($|D|$,$|R|$,$|\Sigma|$,$|\triangle D|$) indexes with different settings: index building depends on D and Σ; Σ is determined by D and R; running times concern $\triangle D$. We report in Fig. 3(k) total running time and index processing time; index time is part of the running time and IndexAll differs from IncOD only in this time. We also show in Fig. 3(l) index time breakdown. We find IndexAll takes less time compared to IncOD, as expected. The total time of IndexAll is about [68%, 88%] of that of IncOD in Fig. 3(k). The efficiency of IndexAll comes at the cost of more indexes. For the case that $|\Sigma| = 25$ in Fig. 3(k), IndexAll has to build 25 indexes since each OD has a distinct LHS attribute list, while CoverIndex suffices to cover all ODs with 5 indexes. Hence, IndexAll takes more time to update indexes, shown in Fig. 3(l). IncOD takes more time for fetching local data of $\triangle D$ due to required post-processing, as illustrated before. We contend that CoverIndex is a better choice when index space is a major concern, *e.g.,* for large $|\Sigma|$ or $|D|$. IncOD already achieves very good performance. In practice if we can afford more space, we can combine some extra indexes with the indexes built by CoverIndex, to further improve the efficiency.

6 Conclusions

We have formalized the problem of incremental OD discovery, studied its computational complexity, discussed its data locality property, presented algorithms and optimizations, and experimentally demonstrated our approaches.

We are developing distributed techniques for incremental OD discovery to further enhance the scalability, and studying incremental discoveries for other constraints.

Acknowledgements. This work is supported by NSFC 61572135, NSFC 61370157, The Shanghai Innovation Action Project 17DZ1203600, China Grid (Shanghai) 52094017001x, Shanghai Science and Technology Project (No. 16DZ1110102), Airplane Research Project, Industry Internet Innovation Development Project.

References

1. Abedjan, Z., Quian-Ruiz, J., Naumann, F.: Detecting unique column combinations on dynamic data. In: ICDE (2014)
2. Bleifub, T., Kruse, S., Naumann, F.: Efficient denial constraint discovery with hydra. PVLDB **11**(3), 311–323 (2017)
3. Cong, G., Fan, W., Geerts, F., Jia, X., Ma, S.: Improving data quality: consistency and accuracy. In: VLDB (2007)
4. Chu, X., Ilyas, I., Papotti, P.: Discovering denial constraints. PVLDB **6**(13), 1498–1509 (2013)
5. Chu, X., Ilyas, I., Papotti, P.: Holistic data cleaning: putting violations into context. In: ICDE (2013)
6. Fan, W., Geerts, F., Li, J., Xiong, M.: Discovering conditional functional dependencies. TKDE **23**(5), 683–698 (2011)
7. Fan, W., Li, J., Tang, N., Yu, W.: Incremental detection of inconsistencies in distributed data. TKDE **26**(6), 1367–1383 (2014)
8. Golab, L., Karloff, H., Korn, F., Srivastava, D., Yu, B.: On generating near-optimal tableaux for conditional functional dependencies. PVLDB **1**(1), 376–390 (2008)
9. Huhtala, Y., Karkkainen, J., Porkka, P., Toivonen, H.: TANE: an efficient algorithm for discovering functional and approximate dependencies. Comput. J. **42**(2), 100–111 (1999)
10. Hao, S., Tang, N., Li, G., He, J., Ta, N., Feng, J.: A novel cost-based model for data repairing. TKDE **29**(4), 727–742 (2017)
11. Khayyat, Z., et al.: BigDansing: a system for big data cleansing. In: SIGMOD (2015)
12. Langer, P., Naumann, F.: Efficient order dependency detection. VLDB J. **25**(2), 223–241 (2016)
13. Papenbrock, T., Naumann, F.: A hybrid approach to functional dependency discovery. In: SIGMOD (2016)
14. Song, S., Chen, L.: Differential dependencies: reasoning and discovery. TODS **36**(3), 16:1–16:41 (2011)
15. Szlichta, J., Godfrey, P., Gryz, J.: Fundamentals of order dependencies. PVLDB **5**(11), 1220–1231 (2012)
16. Szlichta, J., Godfrey, P., Golab, L., Kargar, M., Srivastava, D.: Effective and complete discovery of order dependencies via set-based axiomatization. PVLDB **10**(7), 721–732 (2017)

17. Szlichta, J., Godfrey, P., Golab, L., Kargar, M., Srivastava, D.: Effective and complete discovery of bidirectional order dependencies via set-based axioms. VLDB J. **27**(4), 573–591 (2018)
18. Szlichta, J., Godfrey, P., Gryz, J., Zuzarte, C.: Expressiveness and complexity of order dependencies. PVLDB **6**(14), 1858–1869 (2013)
19. Vazirani, V.: Approximation Algorithms. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-662-04565-7