Pointwise Order Dependency Discovery on Dynamic Data

Ai Ran [1,2] Zijing Tan [1,2] Shuai Ma [3,4] Sheng Qin [1,2]

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

***Abstract*— Pointwise order dependencies (PODs) are dependencies that specify ordering semantics on attributes of tuples. POD discovery refers to the process of identifying the set Σ of valid and minimal PODs on a given data set D. In practice D is typically large and dynamic. This urges the need for an incremental algorithm that computes changes $\triangle\Sigma$ to Σ such that $\Sigma \oplus \triangle\Sigma$ is the set of valid and minimal PODs on D with a set $\triangle D$ of tuple insertion updates, instead of computing all PODs from scratch. In this study, we make a first effort to study the incremental POD discovery problem. (1) We first propose a novel index technique for inputs Σ and D. We give algorithms to build and choose indexes for Σ and D, and to update indexes in response to $\triangle D$. We show that POD violations *w.r.t.* Σ incurred by $\triangle D$ can be efficiently identified by leveraging the proposed indexes, with a cost dependent on $log(|D|)$. (2) We then present an effective algorithm for computing $\triangle\Sigma$, based on Σ and identified violations caused by $\triangle D$. The PODs in Σ that become invalid on $D + \triangle D$ are efficiently detected with the proposed indexes, and further new valid PODs on $D + \triangle D$ are identified by refining those invalid PODs in Σ on $D + \triangle D$. (3) Finally, using both real-life and synthetic datasets, we experimentally show that our approach outperforms the batch approach that computes from scratch, up to orders of magnitude.**

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

I. Introduction

Data dependencies, *a.k.a.* integrity constraints, are valuable tools in specifying and enforcing data semantics. Various dependencies are proposed in literature, such as functional dependencies (FDs), conditional functional dependencies (CFDs) [6] and denial constraints (DCs) [4], among others. Pointwise order dependencies (PODs) are introduced in [7], [8], [17], as a dependency language to specify ordering semantics on attributes of tuples. Ordered attributes, *e.g.,* salary, date and time, are prevalent in data values, and sorting is one of the most important database operations. Therefore, PODs are not only of theoretical interest, but also find practical applications. We first present an example to illustrate features of PODs.

Example 1: Consider the sample tax data (relation instance D_1) in Table I. Each tuple denotes a person with the following attributes: Tuple ID (TID), first name (FName), last name (LName), tax date (Date), social security number (SSN), tax serial number (NUM), area code (AC), phone number (PH), state (ST), zip code (ZIP), salary (SAL), tax rate (RATE) and tax exemption amount (TXA). We can see that the following constraints hold on D_1:

σ_1: Persons with the same zip code live in the same state.
σ_2: The phone number can identify a person (SSN).
σ_3: A person with a lower salary and a higher tax exemption amount has a lower tax rate.
σ_4: The tax serial number increases with the tax date for a single person.

It can be seen that σ_1 and σ_2 are two functional dependencies: the *equality* (=) of zip code (resp. phone number) determines the *equality* (=) of state (resp. SSN) in σ_1 (resp. σ_2). In contrast, more comparison operators are required in the definition of σ_3 and σ_4, for conveying additional ordering semantics on tuple values. All these constraints can be expressed in the notation of PODs (to be formalized in Section III).

σ_1: $\{\mathsf{ZIP}^{=}\} \hookrightarrow \{\mathsf{ST}^{=}\}$
σ_2: $\{\mathsf{PH}^{=}\} \hookrightarrow \{\mathsf{SSN}^{=}\}$
σ_3: $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$
σ_4: $\{\mathsf{SSN}^{=}, \mathsf{Date}^{<}\} \hookrightarrow \{\mathsf{NUM}^{<}\}$

As an example, σ_4 states that for any two tuples t, s, if t has a same value on SSN as s, and a smaller value on Date than s, then t has a smaller value on NUM than s. □

To fully take advantage of the semantics specified by PODs, the set Σ of PODs that hold (are valid) on a relation D is expected to be known in advance. While designing PODs manually is typically too expensive: it requires domain experts, and is still time-consuming and subject to human errors. This highlights the quest for automatic discovery techniques, to find the set Σ of PODs that hold on D.

Worse, data in practice is typically *dynamic*, which means that a set $\triangle D$ of updates may be applied to D. In this paper, we make a first effort for the case of tuple insertions. While deletions may be prohibited, such as data warehouses and block chains, insertions are typically supported. For discovering the set of PODs on $D + \triangle D$, a naive way is to recompute all PODs from scratch on $D + \triangle D$, which is obviously computationally expensive.

Intuitively, when $\triangle D$ is small compared to D, a better approach is to compute changes $\triangle\Sigma$ to Σ, such that $\Sigma \oplus \triangle\Sigma$ is a set of PODs that hold on $D + \triangle D$. As will be seen shortly, we use the notation "$\oplus$" since some PODs in Σ no longer hold and are removed from Σ, while some new valid PODs may be discovered and added into Σ. This approach is known as incremental dependency discovery [1], [13], in contrast to batch (non-incremental) approaches that discover dependencies on

TABLE I
TAX DATA: D_1

TID	FName	LName	Date	SSN	NUM	AC	PH	ST	ZIP	SAL	RATE	TXA
t_0	Ali	Sadam	20140410	719975883	1448	719	6059466	CO	80612	32000	1.24	3000
t_1	Eser	Duparc	20140224	303975883	1401	303	5872027	CO	80612	50000	1.42	1500
t_2	Hennie	Hannen	20140413	701178073	1486	701	1638673	ND	58671	30000	1.21	3400
t_3	Rene	Beke	20130403	801350874	1386	801	6192334	UT	84308	55000	2.04	1300
t_4	Ali	Sadam	20140329	719975883	1427	719	6059466	CO	80612	7500	1.21	0
t_5	Hennie	Hannen	20130404	701178073	1386	701	1638673	ND	58671	6500	0.24	900
t_6	PerOlof	Motwani	20150324	970122634	1547	970	8484643	CO	80209	95000	2.85	1100

TABLE II
INCREMENTAL TAX DATA: $\triangle D_1$

TID	FName	LName	Date	SSN	NUM	AC	PH	ST	ZIP	SAL	RATE	TXA
t_7	Yuichiro	Uckun	20130322	435849162	1368	435	5872027	UT	84308	40000	1.42	3300
t_8	Hennie	Hannen	20140218	701178073	1478	701	1638673	ND	58671	33000	2.04	1400
t_9	Rene	Beke	20150213	801350874	1533	801	6192334	UT	84308	32000	1.23	3200

the whole data set. Incremental POD discovery is very intricate for tuple insertions, as illustrated in the following example.

Example 2: Consider the incremental $\triangle D_1$ applied to D_1 (Table II). Suppose $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$ in Example 1, is the set of PODs discovered on D_1. On $D_1 + \triangle D_1$, we can see that σ_1 and σ_4 still hold, while σ_2 and σ_3 are invalid now. Specifically, σ_2 does not hold, since t_1, t_7 violate σ_2. And σ_3 no longer holds, due to violations caused by t_1, t_7 and t_3, t_8.

Therefore, we should remove from Σ σ_2 and σ_3 that do not hold on $D_1 + \triangle D_1$. More importantly, we should discover a set of new PODs holding on $D_1 + \triangle D_1$ as additions to Σ. Intuitively, we can evolve new valid PODs from PODs in Σ that are invalid on $D_1 + \triangle D_1$.

Take σ_3 as an example. We may find two new PODs valid on $D_1 + \triangle D_1$ based on it. σ_3'=$\{\mathsf{ST}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ and σ_3'' = $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{\leq}\}$. σ_3' is obtained by introducing an additional attribute with a comparison operator to the left-hand side (LHS) of σ_3; the LHS conditions are strengthened. While σ_3'' relaxes σ_3 in its right-hand side (RHS) condition by changing '$<$' to '$\leq$'. Both σ_3' and σ_3'' hold on D_1; obviously any POD that holds on $D_1 + \triangle D_1$ holds on D_1. But neither σ_3' nor σ_3'' is in Σ. The reason is that discovery algorithms typically find only *minimal* constraints (formalized in Section III). σ_3' (resp. σ_3'') is not minimal on D_1 since σ_3 is valid on D_1 and *logically implies* σ_3' (resp. σ_3''), in the sense that any relation that satisfies σ_3 must satisfy σ_3' (resp. σ_3''). Since σ_3 does not hold on $D_1 + \triangle D_1$, σ_3' and σ_3'' are minimal and valid PODs that should be added into Σ. □

Contributions & organizations. We make a first effort to study incremental POD discovery, for Σ on D and a set $\triangle D$ of tuple insertions to D.

(1) Violation detection is a crucial step of incremental POD discovery. We present a novel index technique for two attributes with inequality operators. Leveraging the indexes, violations of Σ incurred by $\triangle D$ can be identified with a cost dependent on $log(|D|)$, where $|D|$ is the size of D. (Section IV).

(2) We develop an algorithm for building an *optimal* index for two attributes with inequality operators. We study techniques for choosing indexes for the set Σ of PODs, to balance the index space and efficiency. We also present an algorithm for updating indexes in response to $\triangle D$, when POD violations are identified simultaneously (Section V).

(3) We develop techniques for discovering $\triangle\Sigma$ based on Σ and violations incurred by $\triangle D$. After identifying in Σ invalid PODs on $D + \triangle D$, we present algorithms to evolve new valid and minimal PODs based on invalid PODs by refining LHS and (or) RHS attributes and operators (Section VI).

(4) Using a host of real-life and synthetic datasets, we conduct extensive experiments to verify the effectiveness and efficiency of our approaches (Section VII).

II. RELATED WORK

This section investigates works close to ours: the hierarchy of order dependency classes and denial constraints (DCs), discovery of order dependencies and DCs, incremental dependency discoveries, and techniques for inequality joins.

The hierarchy of order dependency classes and DCs. There is a different notion of order dependency in literature, referred to as lexicographical order dependency (LOD) [16], [17]. As opposed to PODs defined on sets of attributes, LODs are defined on lists of attributes. A lexicographical ordering with an attribute list AB is to sort tuples with A first, and then to break ties by sorting with B. PODs strictly generalize LODs, as proved in [17]. This means that any LOD can be expressed as a set of PODs, and any instance satisfying the LOD also satisfies all PODs in the set. The generalization of PODs against LODs is strict, since some PODs cannot be expressed in LODs. The concept of set-based canonical form of order dependencies is presented in [14], [15], as an alternative to LODs. PODs again strictly generalize this class of order dependencies.

Denial constraints (DCs) [3], [4] are given in a universally quantified first order logic formalism, as a negation of conjunction of predicates, where each predicate consists of two attribute values and one operator. DCs generalize PODs, and each POD can be encoded as a DC.

Discovery of order dependency. To our best knowledge, we are not aware of any existing works on POD discovery. Batch discovery methods are studied for LODs in [5], [11], and for

TABLE III
OPERATOR INVERSE AND IMPLICATION

op	$=$	$<$	$>$	$\leq$	$\geq$
$\overline{op}$	$<,>$	$\geq$	$\leq$	$>$	$<$
$im(op)$	$=,\geq,\leq$	$<,\leq$	$>,\geq$	$\leq$	$\geq$

set-based canonical order dependencies in [14], [15]. These techniques cannot be applied for discovering PODs. As noted earlier, PODs are more generalized dependencies, and adopt a set-based notion which is quite different from LODs. Batch DC discoveries are recently investigated in [2], [3]. Specifically, [2] is experimentally verified to outperform [3] by orders of magnitude. We implement a batch POD discovery algorithm by an adaption of [2], to compare against our incremental POD discovery technique in experimental evaluations (Section VII).

Different from batch discoveries, this work investigates the incremental discovery problem, aiming at computing $\triangle\Sigma$ in response to $\triangle D$, based on known Σ discovered on D.

Incremental Dependency Discovery. Compared to batch dependency discoveries, incremental discoveries on dynamic data have received less attention, despite their practical demand. Incremental discovery techniques are developed in [1] for unique column combinations (UCCs), *a.k.a.* candidate keys, and in [13] for FDs. Recently, incremental LOD discovery is also studied in [18]. Incremental POD discovery is far more complex, since PODs subsume FDs, which in turn subsume UCCs. As illustrated before, PODs also generalize LODs, and discovering PODs requires a completely different strategy compared to the list-based LODs. As will be illustrated in Section IV, the incremental POD violation detection problem already deserves an in-depth treatment and motivates us to develop a novel index technique. Also, a completely new refinement strategy for PODs is required for computing $\triangle\Sigma$, since we can not only add more attributes to the LHS attribute set, but also refine operators for both LHS and RHS attributes.

Inequality joins. Inequality joins are to join relations with inequality conditions, and violations of PODs on a relation D can be identified by a self-join on D with inequality conditions. [9] is the state-of-the-art technique for inequality joins, which leverages a sorted array on each attribute and a bit-array for encoding connections of two attributes. The technique is extended in [10] for incremental inequality joins in response to data updates. As opposed to [10] that has a cost dependent on $|D|$, we present a novel index technique that enables finding violating tuple pairs incurred by tuple insertions in a cost dependent on $log(|D|)$. Experimental evaluations show that our technique significantly outperforms [10] on tuple insertions.

III. PRELIMINARIES

In this section, we first review basic notations of PODs and present definitions of minimal PODs. We then formalize the incremental POD discovery problem.

Basic notations. $R(A,B,\ldots)$ denotes a relation schema with attributes A, B, ... We use D to denote a specific instance (relation). t, s denote tuples, and t_A denotes the value of attribute A in a tuple t. Each tuple t is associated with a distinct identifier (*id*), denoted by t_{id}.

Marked Attributes [7], [8]. For an attribute A, the marked attributes of A are of the form A^{op}, where operator $op \in \{<, \leq, >, \geq, =\}$. For two tuples t and s, we write $A^{op}(t, s)$, if $t_A \; op \; s_A$ holds.

Example 3: In Table I, we have $\mathsf{SAL}^{>}(t_1, t_0)$, since t_1 has a larger value on SAL than t_0. We also have $\mathsf{AC}^{=}(t_2, t_5)$. □

Pointwise Order Dependencies (PODs) [7], [8], [17]. A POD σ is of the from $\mathcal{X} \hookrightarrow \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are two sets of marked attributes. σ holds on a relation D, iff for any two tuples t, s in D, $B^{op}(t, s)$ for all $B^{op} \in \mathcal{Y}$ if $A^{op}(t, s)$ for all $A^{op} \in \mathcal{X}$. We say σ is valid on D if σ holds on D. We consider non-trivial PODs $\mathcal{X} \hookrightarrow \mathcal{Y}$, where each attribute A (neglecting operator) occurs at most once in $\mathcal{X} \cup \mathcal{Y}$.

Example 4: σ_4 = $\{\mathsf{SSN}^{=}, \mathsf{Date}^{<}\} \hookrightarrow \{\mathsf{NUM}^{<}\}$ holds on D_1 in Table I. It can be verified that for any two tuples s and t in D_1, if $\mathsf{SSN}^{=}(s, t)$ and $\mathsf{Date}^{<}(s, t)$, then $\mathsf{NUM}^{<}(s, t)$. □

Remark. Observe the following. (1) Functional dependencies (FDs) are special cases of PODs. Indeed, each FD $\mathcal{X} \rightarrow \mathcal{Y}$ is a POD when only operator "=" is used. (2) A POD $\mathcal{X} \hookrightarrow \mathcal{Y}$ can be expressed as a set of PODs $\mathcal{X} \hookrightarrow B_i^{op_i}$ for each $B_i^{op_i} \in \mathcal{Y}$. Obviously, $\mathcal{X} \hookrightarrow \mathcal{Y}$ is valid iff every $\mathcal{X} \hookrightarrow B_i^{op_i}$ is valid. We hence consider PODs with a single RHS marked attribute in the rest of paper. (3) Each POD σ has a *symmetry* POD σ_{sym} by reversing ">" and "<", *e.g.,* $\{\mathsf{A}_1^{=}, \mathsf{A}_2^{>}, \mathsf{A}_3^{\leq}\} \hookrightarrow \{\mathsf{B}^{>}\}$ and $\{\mathsf{A}_1^{=}, \mathsf{A}_2^{<}, \mathsf{A}_3^{\geq}\} \hookrightarrow \{\mathsf{B}^{<}\}$. It is easy to see that σ is valid iff σ_{sym} is valid. To avoid redundancy, we consider PODs that take $<, \leq, =$ for its RHS attribute, *e.g.,* $\{\mathsf{A}_1^{=}, \mathsf{A}_2^{<}, \mathsf{A}_3^{\geq}\} \hookrightarrow \{\mathsf{B}^{<}\}$. Obviously, this does not lose generality.

The number of PODs valid on an instance D can be very large, it is hence more instructive to find the set of *minimal* PODs, instead of all PODs. Before formalizing minimal PODs, we introduce some additional notations.

The inverse and implication of operator. We denote by $\overline{op}$ (resp. $im(op)$), the inverse (resp. implication) of an operator op, as summarized in Table III. It can be seen that, (1) either $A^{op}(t, s)$ or $A^{\overline{op}}(t, s)$, but never both; and (2) if $A^{op}(t, s)$ then $A^{im(op)}(t, s)$.

Containment of marked attribute sets. For two sets $\mathcal{X}$ and $\mathcal{X}'$ of marked attributes, we say $\mathcal{X}$ *contains* $\mathcal{X}'$, written as $\mathcal{X}' \subseteq \mathcal{X}$, if $\forall A^{op} \in \mathcal{X}$, $op' \in im(op)$ if $A^{op'} \in \mathcal{X}'$.

Example 5: $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \subseteq \{\mathsf{ST}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\}$, and $\{\mathsf{RATE}^{\leq}\} \subseteq \{\mathsf{RATE}^{<}\}$. Intuitively, $\mathcal{X}' \subseteq \mathcal{X}$ implies that $\mathcal{X}$ is "stricter" than $\mathcal{X}'$: $\mathcal{X} \hookrightarrow \mathcal{X}'$ is valid on any relation D. □

Formally, a POD σ is *not* minimal if σ is logically implied by another valid POD σ', which means that any relation D that satisfies σ' must satisfy σ. We are ready to formalize minimal PODs, based on our notion of containment of attribute sets.

Minimal PODs. A POD $\sigma = \mathcal{X} \hookrightarrow \mathcal{Y}$ is minimal if there does not exist another valid POD $\sigma' = \mathcal{X}' \hookrightarrow \mathcal{Y}'$, where $\mathcal{X}' \subseteq \mathcal{X}$ and $\mathcal{Y} \subseteq \mathcal{Y}'$.

Example 6: Neither $\{\mathsf{ST}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ nor $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{\leq}\}$ is minimal, if $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}$

$\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ is valid. Any relation that satisfies $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ must satisfy both $\{\mathsf{ST}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ and $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{\leq}\}$. □

Incremental POD discovery on dynamic data. Given a relation D of schema R, a complete set Σ of valid and minimal PODs on D, and a set $\triangle D$ of tuple insertions to D, incremental POD discovery is to find, changes $\triangle\Sigma$ to Σ that makes $\Sigma \oplus \triangle\Sigma$ a complete set of valid and minimal PODs on $D + \triangle D$.

Specifically, $\triangle\Sigma = \triangle\Sigma^{+} \cup \triangle\Sigma^{-}$, where $\triangle\Sigma^{+} \cap \triangle\Sigma^{-} = \emptyset$. (1) $\triangle\Sigma^{+} \cap \Sigma = \emptyset$: $\triangle\Sigma^{+}$ contains new valid minimal PODs on $D + \triangle D$ as additions to Σ. (2) $\triangle\Sigma^{-} \subseteq \Sigma$: $\triangle\Sigma^{-}$ contains PODs in Σ that are invalid on $D + \triangle D$ and are to be removed from Σ. That is, $\Sigma \oplus \triangle\Sigma$ is computed as $(\Sigma \cup \triangle\Sigma^{+}) \setminus \triangle\Sigma^{-}$.

As opposed to the incremental discovery problem discussed in this paper, we refer to the problem of discovering the set of minimal valid PODs on D as batch POD discovery. In practice, $\triangle D$ is typically (much) smaller than D. An incremental algorithm can greatly improve the efficiency if its computation cost is dependent on $|\triangle D|$ and $log(|D|)$, instead of $|D|$. We next present techniques that indeed achieve this goal.

IV. Indexes on Two Marked Attributes

In this section, we present a novel index structure for two marked attributes A^{op_1}, B^{op_2} with inequality operators. This indexing technique is an important building block for efficiently identifying POD violations in response to $\triangle D$.

It is important for any incremental dependency discovery algorithm to efficiently identify violations incurred by $\triangle D$. Existing works [1], [13], [18] leverage indexes that fetch required tuples via operator "=" only. As a POD may contain inequality operators, indexes supporting inequality operators become necessary. However, the introduction of inequality operators, *i.e.,* "$<, \leq, >, \geq$", significantly complicates POD violation detections on tuple insertions, as illustrated below.

Example 7: Consider the violation detection for $\sigma_3 = \{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ on relation D_1 in Table I, when tuple t_8 is inserted. As shown in Figure 1, (1) we first compute four sets. Specifically, $\{t|\mathsf{TXA}^{>}(t_8, t)\} = \{t_3, t_4, t_5, t_6\}$ and $\{t|\mathsf{SAL}^{<}(t_8, t)\} = \{t_1, t_3, t_6\}$, while $\{t|\mathsf{TXA}^{>}(t, t_8)\} = \{t_0, t_1, t_2\}$, and $\{t|\mathsf{SAL}^{<}(t, t_8)\} = \{t_0, t_2, t_4, t_5\}$. (2) We then compute two sets: $\{t|\mathsf{TXA}^{>}(t_8, t)\} \cap \{t|\mathsf{SAL}^{<}(t_8, t)\} = \{t_3, t_6\}$, and $\{t|\mathsf{TXA}^{>}(t, t_8)\} \cap \{t|\mathsf{SAL}^{<}(t, t_8)\} = \{t_0, t_2\}$. Intuitively, these two sets contain *candidates* of violating tuples *w.r.t.* t_8. On tuples in the two sets, (3) we finally check remaining conditions in σ_3 to see whether σ_3 is violated. For example, t_3, t_8 lead to a violation if $\mathsf{RATE}^{\geq}(t_8, t_3)$, since t_3 is in the set $\{t|\mathsf{TXA}^{>}(t_8, t)\} \cap \{t|\mathsf{SAL}^{<}(t_8, t)\}$.

All tuples are visited when computing the four sets in step (1). This is because for a tuple s in D_1, either $\mathsf{TXA}^{>}(s, t_8)$ or $\mathsf{TXA}^{>}(t_8, s)$; similarly for $\mathsf{SAL}^{<}$. Traditional indexes on operator "=" are obviously useless in this process. □

To this end, we introduce a novel index structure to encode two marked attributes with inequality operators.

Index Structure. Given a relation D and two marked attributes A^{op_1}, B^{op_2} $(op_1, op_2 \in \{<, \leq, >, \geq\})$, we present

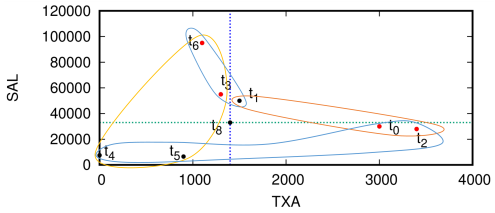


Fig. 1. Candidates of violating tuples *w.r.t.* t_8

an index structure for tuples in D *w.r.t.* A^{op_1}, B^{op_2}, denoted as $\mathsf{Index}(A^{op_1}, B^{op_2})$.

We conduct a pre-processing step. We cluster tuples in D based on their values on A and B, and keep an arbitrary tuple, say t, for each cluster (tuples with same values on both A and B). We remove all other tuples from D and put them into an additional hash map with t_{id} as the key, denoted as Equ^{t}_{AB}.

$\mathsf{Index}(A^{op_1}, B^{op_2}) = \{\mathsf{Sorted}_1, \ldots, \mathsf{Sorted}_k\}$, where each Sorted_i $(i \in [1, k])$ is a sorted data structure on tuples (*ids*) in D. We call k the *rank* of the index, and use $\mathsf{Sorted}_i[n]$ to denote the n-th element in Sorted_i, starting from 0.

(1) For each tuple t in D, there exists a single Sorted_i such that $t \in \mathsf{Sorted}_i$;

(2) For tuples t', t in each Sorted_i, $A^{op_1}(t', t)$ and $B^{op_2}(t', t)$ if t is after t.

Example 8: We can build $\mathsf{Index}(\mathsf{TXA}^{>}, \mathsf{SAL}^{<}) = \{[t_6, t_3, t_1, t_0, t_2], [t_4, t_5]\}$, as shown in Figure 2. Tuples are organized in two sorted structures Sorted_1 and Sorted_2. In each Sorted_i, $\mathsf{TXA}^{>}(t', t)$ and $\mathsf{SAL}^{<}(t', t)$ if t' is after t. □

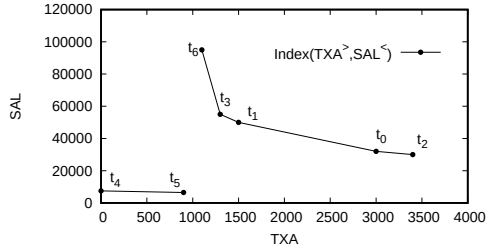


Fig. 2. $\mathsf{Index}(\mathsf{TXA}^{>}, \mathsf{SAL}^{<})$

The index has a size linear in $|D|$. Intuitively, it aims to divide D into several parts (Sorted_i), and tuples in the same Sorted_i are sorted on both A and B. The idea is enlightened by the conditional dependencies [6], which only hold on parts of the relation rather than the whole relation. We will study implementation details of our indexes in Section V-C. Before illustrating the benefit of indexes, we provide several notations.

Candidates of violating tuples. For a tuple s in $\triangle D$, we denote by $T^{s}_{A^{op_1}, B^{op_2}}$ the set of tuples in D, where $T^{s}_{A^{op_1}, B^{op_2}} = \{t|A^{op_1}(s, t), B^{op_2}(s, t)\}$, and by $\overline{T}^{s}_{A^{op_1}, B^{op_2}}$ the set of tuples in D, where $\overline{T}^{s}_{A^{op_1}, B^{op_2}} = \{t|A^{op_1}(t, s), B^{op_2}(t, s)\}$.

For example, $T^{t_8}_{\mathsf{TXA}^{>}\mathsf{SAL}^{<}} = \{t_3, t_6\}$ and $\overline{T}^{t_8}_{\mathsf{TXA}^{>}\mathsf{SAL}^{<}} = \{t_0, t_2\}$ for tuple $t_8 \in \triangle D_1$. They are partial results computed in Example 7, *i.e.,* candidates of violating tuples *w.r.t.* t_8.

We present Algorithm Fetch to efficiently compute both $T^{s}_{A^{op_1}, B^{op_2}}$ and $\overline{T}^{s}_{A^{op_1}, B^{op_2}}$, by leveraging $\mathsf{Index}(A^{op_1}, B^{op_2})$.

Algorithm. Fetch takes a tuple s and $\mathsf{Index}(A^{op_1}, B^{op_2})$ as inputs, and works on each Sorted_i of $\mathsf{Index}(A^{op_1}, B^{op_2})$.

Algorithm 1: Fetch

input : *Tuple* s, $\mathsf{Index}(A^{op_1}, B^{op_2})$
output: $T^s_{A^{op_1},B^{op_2}}$, $\overline{T}^s_{A^{op_1},B^{op_2}}$

1 $T^s_{A^{op_1},B^{op_2}} \leftarrow \emptyset$; $\overline{T}^s_{A^{op_1},B^{op_2}} \leftarrow \emptyset$;
2 **for** *each* $\mathsf{Sorted}_i \in \mathsf{Index}(A^{op_1}, B^{op_2})$ **do**
3 $\quad p \leftarrow find(s, A^{op_1}, \mathsf{Sorted}_i)$;
4 $\quad q \leftarrow find(s, B^{op_2}, \mathsf{Sorted}_i)$;
5 $\quad right \leftarrow max(p,q)+1$; $left \leftarrow min(p,q)$;
6 $\quad$ **while** *right* $<$ $\mathsf{Sorted}_i.size$ **do**
7 $\quad\quad$ add $\mathsf{Sorted}_i[right]$ into $\overline{T}^s_{A^{op_1},B^{op_2}}$;
8 $\quad\quad right \leftarrow right + 1$;
9 $\quad$ **while** *left* ≥ 0 **do**
10 $\quad\quad$ add $\mathsf{Sorted}_i[left]$ into $T^s_{A^{op_1},B^{op_2}}$;
11 $\quad\quad left \leftarrow left - 1$;

(1) It first finds position p (line 3). Specifically, (a) $p = -1$, if $A^{op_1}(\mathsf{Sorted}_i[0], s)$; or (b) $p = \mathsf{Sorted}_i.size-1$, if $A^{\overline{op_1}}(\mathsf{Sorted}_i[\mathsf{Sorted}_i.size-1], s)$; or (c) p is found such that $A^{\overline{op_1}}(\mathsf{Sorted}_i[p], s)$ and $A^{op_1}(\mathsf{Sorted}_i[p+1],s)$. It then finds position q similarly by replacing A^{op_1} with B^{op_2} (line 4);
(2) From position $max(p,q)+1$ to the right in Sorted_i, it collects all tuples in the set $\overline{T}^s_{A^{op_1},B^{op_2}}$ (lines 6-8);
(3) From position $min(p,q)$ to the left in Sorted_i, it collects all tuples in the set $T^s_{A^{op_1},B^{op_2}}$ (lines 9-11).

As a post-processing step, if a tuple $t \in T^s_{A^{op_1},B^{op_2}}$ (resp. $\overline{T}^s_{A^{op_1},B^{op_2}}$), then we also add all tuples in Equ^t_{AB} into $T^s_{A^{op_1},B^{op_2}}$ (resp. $\overline{T}^s_{A^{op_1},B^{op_2}}$). Recall that tuples in Equ^t_{AB} have same values on both A and B as t.

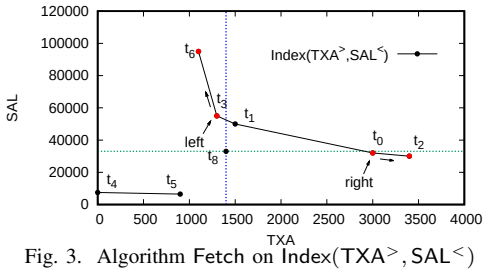


Fig. 3. Algorithm Fetch on $\mathsf{Index}(\mathsf{TXA}^{>}, \mathsf{SAL}^{<})$

Example 9: Consider $\mathsf{Index}(\mathsf{TXA}^{>}, \mathsf{SAL}^{<}) = \{[t_6, t_3, t_1, t_0, t_2], [t_4, t_5]\}$ and tuple $t_8 \in \triangle D_1$, as shown in Figure 3. We illustrate how to compute $T^{t_8}_{\mathsf{TXA}^{>}\mathsf{SAL}^{<}}$ and $\overline{T}^{t_8}_{\mathsf{TXA}^{>}\mathsf{SAL}^{<}}$.
(1) On $\mathsf{Sorted}_1 = [t_6, t_3, t_1, t_0, t_2]$, we find $\mathsf{TXA}^{\leq}(t_3, t_8)$ and $\mathsf{TXA}^{>}(t_1, t_8)$, and hence $p = 1$. Similarly, $\mathsf{SAL}^{\geq}(t_1, t_8)$ and $\mathsf{SAL}^{<}(t_0, t_8)$, and we find $q = 2$.

We have $right = max(p,q)+1 = 3$, and put t_0 into $\overline{T}^{t_8}_{\mathsf{TXA}^{>}\mathsf{SAL}^{<}}$. We then move to the right in Sorted_1 and put t_2 into $\overline{T}^{t_8}_{\mathsf{TXA}^{>}\mathsf{SAL}^{<}}$. We have $left = min(p,q) = 1$, and add t_3 into $T^{t_8}_{\mathsf{TXA}^{>}\mathsf{SAL}^{<}}$. We then move to the left in Sorted_1 and put t_6 into $T^{t_8}_{\mathsf{TXA}^{>}\mathsf{SAL}^{<}}$.
(2) On Sorted_2=$[t_4,t_5]$, $p = 1$ since $\mathsf{TXA}^{\leq}(t_5, t_8)$, and $q = -1$ since $\mathsf{SAL}^{<}(t_4, t_8)$. We set $right = \mathsf{Sorted}_2.size$ and $left = -1$, and find no new tuples for $T^{t_8}_{\mathsf{TXA}^{>}\mathsf{SAL}^{<}}$ or $\overline{T}^{t_8}_{\mathsf{TXA}^{>}\mathsf{SAL}^{<}}$.

To sum up, $T^{t_8}_{\mathsf{TXA}^{>}\mathsf{SAL}^{<}}$=$\{t_3, t_6\}$, $\overline{T}^{t_8}_{\mathsf{TXA}^{>}\mathsf{SAL}^{<}}$=$\{t_0, t_2\}$. □

Correctness & complexity. The correctness of Fetch can be easily inferred from the index specification. For s in $\triangle D$ and t in Sorted_i of $\mathsf{Index}(A^{op_1}, B^{op_2})$ on D, (1) if $A^{op_1}(t,s)$ (resp. $B^{op_2}(t,s)$), then $A^{op_1}(t',s)$ (resp. $B^{op_2}(t',s)$) for any t' after t in Sorted_i; and (2) if $A^{\overline{op_1}}(t,s)$ (resp. $B^{\overline{op_2}}(t,s)$), then $A^{\overline{op_1}}(t',s)$ (resp. $B^{\overline{op_2}}(t',s)$) for any t' before t in Sorted_i.

Given $\mathsf{Index}(A^{op_1}, B^{op_2}) = \{\mathsf{Sorted}_1, \ldots, \mathsf{Sorted}_k\}$, it takes $\sum_{i\in[1,k]} O(log(n_i)) \leq k \cdot O(log(|D|))$ to identify positions p, q on all Sorted_i, where n_i is the number of tuples in Sorted_i and $|D| = \sum_{i\in[1,k]} n_i$, is the number of tuples in D. This is because each Sorted_i is sorted on both A and B. It is linear in the size of $T^s_{A^{op_1},B^{op_2}}$ (resp. $\overline{T}^s_{A^{op_1},B^{op_2}}$) to collect tuples for $T^s_{A^{op_1},B^{op_2}}$ (resp. $\overline{T}^s_{A^{op_1},B^{op_2}}$) on all Sorted_i. This cost is linear in the result size, and is obviously necessary.

The complexity depends on $log(|D|)$ instead of $|D|$. It can be seen that the *rank* k of the index impacts the efficiency, and an index with a small rank is preferable. In Section V, we discuss how to choose marked attributes and build indexes, for indexes with small ranks. In Section VII, we experimentally study ranks of indexes on various datasets.

V. Algorithms for Indexes

In this section, we present a set of algorithms for index processing. Given inputs D, $\triangle D$ and Σ, we develop algorithms to (1) build an *optimal* index for two marked attributes with inequality operators (Section V-A), to (2) choose indexes for all PODs in Σ (Section V-B), and to (3) update indexes in response to $\triangle D$ (Section V-C).

A. Building Indexes

For a relation D and two marked attributes A^{op_1}, B^{op_2}, note that the possible indexes may not be unique. As an example, for $\mathsf{Date}^{<}$ and $\mathsf{NUM}^{<}$, besides index $I' = \{[t_6, t_3], [t_2, t_0, t_5], [t_4, t_1]\}$ (Figure 4), we have another index $I = \{[t_6, t_2, t_0, t_4, t_1, t_5], [t_3]\}$ (Figure 5). As illustrated in Section IV, they both satisfy the index specification, but we prefer I to I' since I has a smaller rank than I'.

We say $\mathsf{Index}(A^{op_1}, B^{op_2})$ is *optimal* if its rank is the minimum among all indexes for A^{op_1}, B^{op_2}. We develop Algorithm OptIndex for such index.

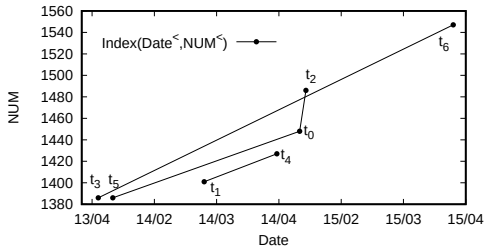


Fig. 4. Index I' for $\mathsf{Date}^{<}$ and $\mathsf{NUM}^{<}$

Algorithm. We present Algorithm OptIndex for building an optimal index with the minimum rank, for two marked attributes A^{op_1} and B^{op_2} on D ($op_1, op_2 \in \{<, \leq, >, \geq\}$).
(1) Tuples in D are sorted on A according to op_1, and then on B according to op_2 for breaking ties (line 1). Specifically, tuples are sorted in a descending order if op_1 (op_2) is in $\{<, \leq\}$, otherwise in an ascending order.
(2) OptIndex initializes the index with Sorted_1 containing the first tuple in D (line 2). It then enumerates all remaining tuples in D (lines 3-12). For each tuple s, if there exist

Algorithm 2: OptIndex

input : *relation* D *and marked attributes* A^{op_1}, B^{op_2}
output: *an optimal index for* A^{op_1} *and* B^{op_2} *on* D

1 Sort tuples in D on A according to op_1, and then on B according to op_2 for breaking ties;
2 $Index \leftarrow \{ [\ D[0]\ ] \}$;
3 **for** *each tuple* $s \in D \setminus D[0]$ **do**
4 $\quad pos \leftarrow NULL$; $min \leftarrow -1$;
5 $\quad$ **for** *each* $\mathsf{Sorted}_i \in Index$ **do**
6 $\quad\quad t \leftarrow$ the last element of Sorted_i;
7 $\quad\quad$ **if** $B^{op_2}(s,t)$ **then**
8 $\quad\quad\quad$ **if** $min = -1$ **or** $min > |(s_B - t_B)|$ **then**
9 $\quad\quad\quad\quad min \leftarrow |(s_B - t_B)|$;
10 $\quad\quad\quad\quad pos \leftarrow \mathsf{Sorted}_i$;
11 $\quad$ **if** $pos != NULL$ **then** append s to the end of pos;
12 $\quad$ **else** add $[s]$ into $Index$;

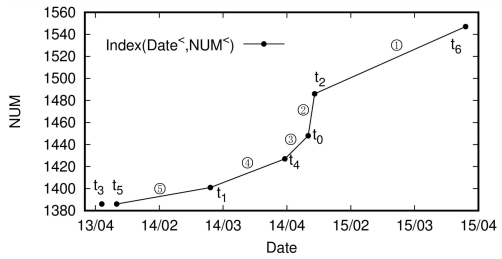


Fig. 5. Index I for $\mathsf{Date}^<$ and $\mathsf{NUM}^<$ with OptIndex

multiple Sorted_i that satisfy the index specification *w.r.t.* s, *i.e.,* $B^{op_2}(s,t)$ if t is the last element of Sorted_i, then OptIndex chooses the one to minimize the *difference* of attribute values on B incurred by s (lines 5-10), and adds s to the tail of it (line 11). Note that $A^{op_1}(s,t)$ is guaranteed since D is sorted on A according to op_1. If no such Sorted_i exists, then a new Sorted_j containing s is added into the index (line 12).

Example 10: We employ OptIndex to build an index on D_1 for $\mathsf{Date}^<$, $\mathsf{NUM}^<$, shown in Figure 5. (1) Tuples are sorted in descending order of Date, *i.e.,* $[t_6, t_2, t_0, t_4, t_1, t_5, t_3]$, and the index is initialized as $\{[t_6]\}$. (2) OptIndex then enumerates all other tuples in D_1. t_2 is appended to Sorted_1 containing t_6, since $NUM^<(t_2, t_6)$; similarly for t_0, t_4, t_1, t_5. (3) Since t_3 and t_5 have a same value on NUM, a new Sorted_2 is built for t_3. (4) Finally, we have $I = \{[t_6, t_2, t_0, t_4, t_1, t_5], [t_3]\}$. □

Intuitively, OptIndex aims to first minimize the value changes on A according to op_1, and then on B according to op_2, when adding a tuple into the index. It can be seen that the index rank does not change if we interchange A with B in OptIndex. We then show the optimality property of OptIndex.

Proposition 1: For two marked attributes A^{op_1} and B^{op_2} ($op_1, op_2 \in \{<, \leq, >, \geq\}$), OptIndex creates an index whose rank is the minimum among all indexes for A^{op_1}, B^{op_2}.

Proof. For two marked attributes A^{op_1}, B^{op_2} ($op_1, op_2 \in \{<, \leq, >, \geq\}$), we denote by I_{opt} the output index of OptIndex. For any index I' for A^{op_1}, B^{op_2}, we show that I' can be transformed into I_{opt} in a finite number of steps, and in each step, the index rank *never* increases. Therefore, I_{opt} has a rank that is the minimum among all indexes for A^{op_1}, B^{op_2}.

The transformation of I' to I_{opt} is conducted as follows. (1) We sort tuples in D on A according to op_1, and then on B according to op_2 for breaking ties; that is, we sort tuples in the same way as line 1 of OptIndex.

(a) before (b) after

Fig. 6. neither o nor p exists

(a) before (b) after

Fig. 7. p exists, but o does not exist

(a) before (b) after

Fig. 8. o exists, but p does not exist

(a) before (b) after

Fig. 9. both o and p exist

(2) We treat each tuple $s \in D$ one by one. For each tuple s, it remains unchanged in I' if (a) there is no tuple before s in I_{opt}, *i.e.,* s is the first element of some sorted structure, *i.e.,* Sorted_i in I_{opt}, or (b) s is after a tuple t both in I' and in I_{opt}. It is easy to see that there is no tuple before $D[0]$ in I_{opt} and hence $D[0]$ always remains unchanged in I'. Obviously, there is also no tuple before $D[0]$ in I'.

Now we suppose s is after t in some sorted structure of I_{opt}. If s is not after t in I', then we move s such that s is also after t in I'. In I', we denote the tuple before s as o, and denote the tuple after t as p.

(a) If neither o nor p exists, we move s as shown in Figure 6. Note that when we move s, all tuples after s in the same sorted structure are also moved, *i.e.,* still after s. It can be seen that we *reduce* the rank of I' in this way.

(b) If p exists but o does not exist, we move s as shown in Figure 7. p becomes the first element of a sorted structure in I'. This does not change the rank of I'.

(c) If o exists but p does not exist, we move s as shown in Figure 8. o becomes the last element of a sorted structure in I'. This does not affect the rank of I'.

(d) As shown in Figure 9, in I' we have tuple o before s and p after t. Recall that we treat tuples in the order *w.r.t.* A^{op_1}. Hence, we have $A^{op_1}(p, s)$; otherwise s cannot be after t in I_{opt}. Since s is after o in I', we know $A^{op_1}(s, o)$. Putting these together, we know $A^{op_1}(p, o)$. Obviously, o is processed before s. Recall that t incurs the minimum value change on attribute B against s, among all tuples that are processed before s. From this, we know $B^{op_2}(t, o)$. Since tuple p is after t in I', we know $B^{op_2}(p, t)$, Putting these together, we have $B^{op_2}(p, o)$. Since we know $A^{op_1}(p, o)$ and $B^{op_2}(p, o)$, we can move tuple p after o, when we move s after t, as shown in Figure 9. Again, this does not affect the rank of I'.

(3) After processing all tuples $s \in D$, we get index I_{opt} after the transformation. This is because, (a) a tuple s remains unchanged if s is the first element of some sorted structure in I_{opt}; and (b) s is after a tuple t if s is after t in I_{opt}. Moreover, recall that the transformation never increases the index rank, in either of (a), (b), (c), and (d) stated above.

Since I' is an arbitrary index for A^{op_1} and B^{op_2}, we conclude that I_{opt} has a minimum rank among all indexes for A^{op_1} and B^{op_2}. □

For two marked attributes A^{op_1} and B^{op_2} ($op_1, op_2 \in \{<, \leq, >, \geq\}$), we denote by I_{opt} the index built with OptIndex on A^{op_1}, B^{op_2}, and by I'_{opt} the index built with OptIndex on B^{op_2}, A^{op_1}. We denote by $|I_{opt}|$ (resp. $|I'_{opt}|$) the index rank of I_{opt} (resp. I'_{opt}), and show that $|I_{opt}| = |I'_{opt}|$.

Indeed, this can be easily inferred from Proposition 1. Obviously, I_{opt} is also an index for B^{op_2} and A^{op_1}, and I'_{opt} is also an index for A^{op_1} and B^{op_2}. We know $|I_{opt}| \leq |I'_{opt}|$ on A^{op_1} and B^{op_2}, while $|I_{opt}| \geq |I'_{opt}|$ on B^{op_2} and A^{op_1}, according to Proposition 1. Hence, $|I_{opt}| = |I_{opt'}|$.

Complexity. OptIndex has a complexity of $O(|D| \cdot log(|D|))$. Line 1 and lines 3-12 both have this complexity. We always check the tail elements of all Sorted_i to find the desired Sorted_i for the current tuple s. This is done in $O(log(k))$ by building an auxiliary sorted structure on the last elements of all Sorted_i, where $k \leq |D|$, is the rank of the index.

B. Choosing Indexes

We show that indexes help fetch violating tuples *w.r.t.* $\triangle D$, and that some indexes can be employed as alternatives to others. We then develop algorithms to choose indexes for Σ.

For Σ on D and $\triangle D$, we need to identify violations of Σ incurred by $\triangle D$. Specifically, it is to find all tuple pairs (s, t) such that s, t violate some PODs in Σ; obviously at least one of s, t is in $\triangle D$. We consider the case both s and t are in $\triangle D$ in Section V-C, and now suppose $s \in \triangle D$ and $t \in D$.

Fetching violating tuple pairs with indexes. If s, t violate $\sigma = \{A_1^{op_1}, \ldots, A_m^{op_m}\} \hookrightarrow B^{op'}$, then (a) $A_i^{op_i}(s, t)$ for all $i \in [1, m]$ and $B^{\overline{op'}}(s, t)$, or (b) $A_i^{op_i}(t, s)$ for all $i \in [1, m]$ and $B^{\overline{op'}}(t, s)$. If some op_i ($i \in [1,m]$) is "=" in $\{A_1^{op_1}, \ldots, A_m^{op_m}\} \hookrightarrow B^{op'}$, then we can adopt the traditional sorted structure on attribute A_i, denoted as $\mathsf{Index}(A_i^{=})$. We employ $\mathsf{Index}(A_i^{=})$ to find tuples t such that $A_i^{=}(t, s)$, and then check on t, s remaining conditions of σ for violating tuples *w.r.t.* s.

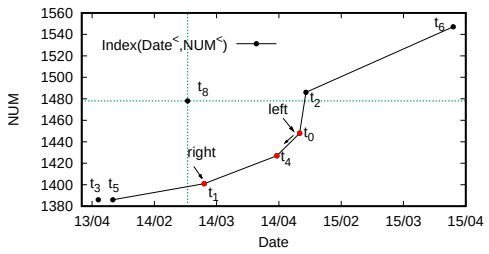


Fig. 10. leveraging an index reversely

Suppose $op_i, op_j \in \{<, \leq, >, \geq\}$ ($i, j \in [1,m]$). If s, t violate σ, then $t \in T^s_{A_i^{op_i}, A_j^{op_j}} \cup \overline{T}^s_{A_i^{op_i}, A_j^{op_j}}$. We can employ Algorithm Fetch (Section IV) on $\mathsf{Index}(A_i^{op_i}, A_j^{op_j})$ to compute $T^s_{A_i^{op_i}, A_j^{op_j}}$ and $\overline{T}^s_{A_i^{op_i}, A_j^{op_j}}$, in which t is further identified by considering remaining conditions of σ. Similarly, $\mathsf{Index}(A_i^{op_i}, B^{\overline{op'}})$ ($i \in [1,m]$) can be leveraged for this purpose as well. Note that we need two indexes $\mathsf{Index}(A_i^{op_i}, B^{>})$ and $\mathsf{Index}(A_i^{op_i}, B^{<})$, if op' is "=".

Alternative indexes. We show that some indexes can be employed as alternatives to others.

(1) $\mathsf{Index}(A^{\geq}, B^{op_2})$ can support $A^{>}$ and B^{op_2} as well, which requires only a slight modification in Algorithm Fetch. Specifically, we just neglect tuples t if $A^{=}(t, s)$, when collecting tuples for $T^s_{A^{>}, B^{op_2}}$ (resp. $\overline{T}^s_{A^{>}, B^{op_2}}$) on the index.

(2) By slight modifications, Algorithm Fetch can employ $\mathsf{Index}(A^{<}, B^{<})$ (resp. $\mathsf{Index}(A^{<}, B^{>})$) to compute $T^s_{A^{<}B^{>}}$ and $\overline{T}^s_{A^{<}B^{>}}$ (resp. $T^s_{A^{<}B^{<}}$ and $\overline{T}^s_{A^{<}B^{<}}$). This does not affect the computational complexity of Fetch. Without loss of generality, we illustrate this with the following example.

Example 11: As shown in Figure 10, we compute $T^{t_8}_{\mathsf{Date}^{<}\mathsf{NUM}^{>}}$ and $\overline{T}^{t_8}_{\mathsf{Date}^{<}\mathsf{NUM}^{>}}$ by employing $\mathsf{Index}(\mathsf{Date}^{<}, \mathsf{NUM}^{<}) = \{[t_6, t_2, t_0, t_4, t_1, t_5], [t_3]\}$. We find p=4 and q=1 in Sorted_1, such that $\mathsf{Date}^{<}(\mathsf{Sorted}_1[p{+}1], t_8)$, $\mathsf{Date}^{\geq}(\mathsf{Sorted}_1[p], t_8)$, and $\mathsf{NUM}^{<}(\mathsf{Sorted}_1[q+1], t_8)$, $\mathsf{NUM}^{\geq}(\mathsf{Sorted}_1[q], t_8)$.

Different from the original algorithm, (1) we let $left = min(p, q){+}1{=}2$, and $right{=}max(p, q){=}4$; and (2) check tuples from position $left$ to $right$ in Sorted_1 for $T^{t_8}_{\mathsf{Date}^{<}\mathsf{NUM}^{>}}$ and $\overline{T}^{t_8}_{\mathsf{Date}^{<}\mathsf{NUM}^{>}}$. All nodes from $left$ to $right$ belong to one of these two sets. We add t_0 into $T^{t_8}_{\mathsf{Date}^{<}\mathsf{NUM}^{>}}$, since $\mathsf{Date}^{<}(t_8, t_0)$ and $\mathsf{NUM}^{>}(t_8, t_0)$; t_4 and t_1 are also put into $T^{t_8}_{\mathsf{Date}^{<}\mathsf{NUM}^{>}}$. After that $T^{t_8}_{\mathsf{Date}^{<}\mathsf{NUM}^{>}} = \{t_0, t_1, t_4\}$, $\overline{T}^{t_8}_{\mathsf{Date}^{<}\mathsf{NUM}^{>}} = \emptyset$.

We find no new results on $\mathsf{Sorted}_2 = [t_3]$. Finally, we have $T^{t_8}_{\mathsf{Date}^{<}\mathsf{NUM}^{>}} = \{t_0, t_1, t_4\}$ and $\overline{T}^{t_8}_{\mathsf{Date}^{<}\mathsf{NUM}^{>}} = \emptyset$. □

Cover PODs in Σ by indexes. It is costly to build an index for each POD if Σ is large. We propose to build a set of indexes such that for each $\sigma \in \Sigma$ at least one index is usable; we say σ is *covered* in this case. According to our observations above, (1) we can use (a) $\mathsf{Index}(A_i^{=})$, or (b) $\mathsf{Index}(A_i^{op_i}, A_j^{op_j})$, or (c) $\mathsf{Index}(A_i^{op_i}, B^{\overline{op'}})$, to cover $\sigma = \{A_1^{op_1}, \ldots, A_m^{op_m}\} \hookrightarrow B^{op'}$; and (2) we can use $\mathsf{Index}(A^{\geq}, A'^{>})$ or $\mathsf{Index}(A^{>}, A'^{<})$, to cover PODs leveraging $\mathsf{Index}(A^{>}, A'^{>})$.

Indeed, it brings extra benefit to share indexes among PODs. In violation detections of several PODs for a tuple $s \in \triangle D$, it suffices to visit an index shared by these PODs only once.

Algorithm 3: ChooseIndex

input : *a set* Σ *of PODs*
output: *a set* $Ind(\Sigma)$ *of indexes for covering PODs in* Σ

1 **foreach** $\sigma = \mathcal{X} \hookrightarrow \mathcal{B}^{op'} \in \Sigma$ **do**
2 **if** *there exists* $A^{=} \in \mathcal{X}$ *and* $score(\mathsf{Index}(A^{=})) < l$ **then**
3 add $\mathsf{Index}(A^{=})$ into $Ind(\Sigma)$;
4 remove from Σ all PODs covered by $\mathsf{Index}(A^{=})$;
5 $Candidates \leftarrow \{\}$;
6 **foreach** $\sigma = \mathcal{X} \hookrightarrow \mathcal{B}^{op'} \in \Sigma$ **do**
7 **for** ${A_i}^{op_i}, {A_j}^{op_j} \in \mathcal{X} \cup \{\overline{B^{op'}}\}$ *(*$op_i, op_j \in \{<, \leq, >, \geq\}$*)* **do**
8 add $({A_i}^{op_i}, {A_j}^{op_j})$ into $Candidates$, in ascending order of $score({A_i}^{op_i}, {A_j}^{op_j})$;
9 **for** *each* $({A_i}^{op_i}, {A_j}^{op_j})$ *in* $Candidates$ **do**
10 **if** Σ *is empty* **then break** ;
11 **foreach** $\sigma = \mathcal{X} \hookrightarrow \mathcal{B}^{op'} \in \Sigma$ *such that* ${A_i}^{op_i}, {A_j}^{op_j}$ *cover* σ **do** remove σ from Σ;
12 $op_j \leftarrow$ reverse op_j if necessary according to $r(A_i, A_j)$ and parameter α;
13 add $\mathsf{Index}(A_i^{op_i}, A_j^{op_j})$ into $Ind(\Sigma)$;
14 **for** *each* $\sigma = \mathcal{X} \hookrightarrow \mathcal{B}^{op'} \in \Sigma$ **do**
15 choose $A_i^{=} \in \mathcal{X}$ with the smallest $score(\mathsf{Index}(A_i^{=}))$ among all $A^{=} \in \mathcal{X}$, and add $\mathsf{Index}(A_i^{=})$ into $Ind(\Sigma)$;

Experimental evaluations (Section VII) show that our approach is hence insensitive to the size of Σ. We then introduce score functions to show our preference for indexes.

Score for equality index. The operator "=" on attribute A may exhibit a good selectivity if the number of distinct values on A is large. We measure $\mathsf{Index}(A^{=})$ with the following scoring function, and prefer $\mathsf{Index}(A^{=})$ with a small score.

$$score(\mathsf{Index}(A^{=})) = 1 - \frac{the\ number\ of\ distinct\ values\ on\ A}{|D|}$$

Score for inequality index. It is more intricate to measure $\mathsf{Index}(A^{op_1}, B^{op_2})$. We adopt the following scoring function:

$$score(A^{op_1}, B^{op_2}) = \frac{1-|r(A,B)|}{coverage(A^{op_1}, B^{op_2})}$$

Herein, $coverage(A^{op_1}, B^{op_2})$ is the number of PODs covered by $\mathsf{Index}(A^{op_1}, B^{op_2})$, and $|r(A, B)|$ is the absolute value of correlation coefficient for attributes A and B. In a dataset D, $r(A, B)$ is computed based on all tuples $t \in D$, as follows:

$$r(A,B) = \frac{\sum(t_A * t_B) - \sum t_A \sum t_B}{\sqrt{\sum {t_A}^2 - (\sum t_A)^2}\sqrt{\sum {t_B}^2 - (\sum t_B)^2}}$$

We prefer indexes with small scores, *i.e.,* indexes that are on attributes with high correlation coefficient and that can cover more PODs. Moreover, we find that if A, B show a positive correlation, *i.e.,* $r(A, B)>0$, then indexes like $\mathsf{Index}(A^{>}, B^{>})$ or $\mathsf{Index}(A^{\geq}, B^{>})$ tend to have a small rank. In contrast, indexes like $\mathsf{Index}(A^{>}, B^{<})$ or $\mathsf{Index}(A^{\geq}, B^{<})$ are more likely to have a small rank if A, B show a negative correlation.

Example 12: On relation D_1 in Table I, instead of $\mathsf{Index}(\mathsf{Date}^{<}, \mathsf{NUM}^{\geq})$ that covers σ_4=$\{\mathsf{SSN}^{=}, \mathsf{Date}^{<}\} \hookrightarrow \{\mathsf{NUM}^{<}\}$, we build $\mathsf{Index}(\mathsf{Date}^{<}, \mathsf{NUM}^{<})$ that also covers σ_4 and has a small rank, since r(Date, NUM)=0.88. This explains why we compute $T^{t_8}_{\mathsf{Date}<\mathsf{NUM}>}$ and $\overline{T}^{t_8}_{\mathsf{Date}<\mathsf{NUM}>}$ with $\mathsf{Index}(\mathsf{Date}^{<}, \mathsf{NUM}^{<})$ in Example 11. □

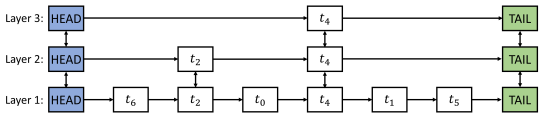


Fig. 11. Example index in SkipList

Algorithm. We present Algorithm ChooseIndex to find the set $Ind(\Sigma)$ of indexes for the given POD set Σ.

It first chooses indexes $\mathsf{Index}(A^{=})$ whose score values are below a predefined threshold l, if they can cover PODs in Σ (lines 1-4). We set l = 0.6 in our implementation. It then collects all marked attribute pairs for Σ in $Candidates$; these pairs are sorted in ascending order of their score values (lines 6-8). ChooseIndex then enumerates marked attribute pairs in $Candidates$, employs them to cover PODs if possible, and builds indexes (lines 9-13). We may adjust op_j for $\mathsf{Index}(A_i^{op_i}, A_j^{op_j})$ with a small rank, according to $r(A_i, A_j)$ (line 12). We use a parameter $\alpha > 0$ in our implementation. As an example, we reverse ">" as "<" on A_j and build $\mathsf{Index}(A_i^{<}, A_j^{<})$, if we have $A_i^{<}$ and $A_j^{>}$ but $r(A_i, A_j) > \alpha$. We will experimentally study our score functions and parameter α in Section VII. If there are still uncovered PODs, it can be seen that they must contain $A^{=}$ but $score(\mathsf{Index}(A^{=})) \geq l$. We choose $A_i^{=}$ with the smallest scores (lines 14-15).

Complexity. In its worst case, ChooseIndex takes $O(|R|^2 \cdot |\Sigma|)$ to cover all PODs in Σ, where $|R|$ is the number of attributes, and $|\Sigma|$ is the number of PODs in Σ. The upper bound for computing score values for $\mathsf{Index}(A^{=})$ and $\mathsf{Index}(A_i^{op_1}, A_j^{op_2})$ is $O(|R| \cdot |D| log(|D|))$ and $O(|R|^2 \cdot |D|)$ respectively. We use uniform random sampling on D, to estimate score values.

Remark. Since Σ on D is known, building indexes with ChooseIndex and OptIndex can be done in advance, as a preprocessing stage for the incremental POD discovery.

C. Updating Indexes & Implementation

We have shown that POD violations incurred by tuple $t \in D$ and $s \in \triangle D$ can be effectively identified by leveraging indexes. A remaining issue concerns how to check violations *w.r.t.* two tuples in $\triangle D$. In order to detect these violations by indexes as well, we propose to update the indexes in response to $\triangle D$.

We implement our indexes in a lightweight data structure, known as SkipList [12]. Each Sorted_i is implemented as a SkipList. We show an example in Figure 11 for Sorted_1 of $\mathsf{Index}(\mathsf{Date}^{<}, \mathsf{NUM}^{<})$ from Example 10. Each SkipList is built as a multi-layered sorted lists. The bottom layer contains all the nodes, while the number of nodes in a layer decreases as the layer number increases. The search of a node in a SkipList always starts from the uppermost layer, and walks down to lower layers gradually. Nodes at layer i + 1 serve as indexes for nodes at layer i, and help "skip" several nodes at layer i. [12] proves that it takes $O(log(n))$ to search for a node in a SkipList, where n is the number of nodes in the bottom layer.

We apply $\triangle D$ to each of the indexes. When dealing with an index I, (1) we sort tuples in $\triangle D$ in the same way as building I (line 1 of Algorithm OptIndex); and (2) for each tuple $s \in \triangle D$, check violations incurred by s leveraging I and update

I with s. In this way, violations incurred by $s, s' \in \triangle D$ can be detected with the updated indexes. We then show that updating indexes with s can be done along with identifying possible violations incurred by s (Algorithm Fetch). The additional cost of updating indexes is hence marginal.

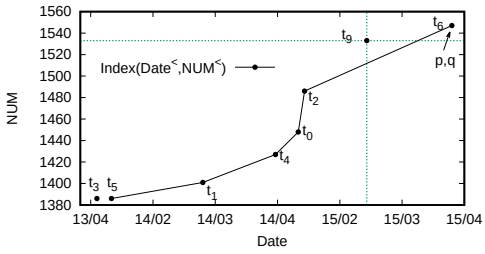


Fig. 12. Tuple t_9 and tuples in D_1

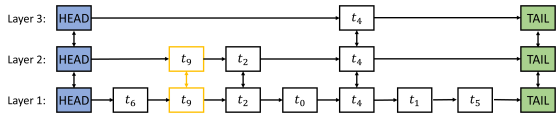


Fig. 13. Update index in SkipList

Example 13: Consider Sorted_1 of $\mathsf{Index}(\mathsf{Date}^<, \mathsf{NUM}^<)$ in Figure 11. We employ Algorithm Fetch on it to check possible violations incurred by tuple t_9. To help understanding, Figure 12 shows t_9 and all tuples in D_1 of Table I.

Just like Example 11, we first identify the position p, such that $Date^<(\mathsf{Sorted}_1[p+1], t_9)$ and $Date^\geq(\mathsf{Sorted}_1[p], t_9)$. We start from layer 3 of the SkipList, and go to the next layer since $Date^<(t_4, t_9)$. In layer 2, we find $Date^<(t_2, t_9)$ and then go to layer 1. Since $Date^<(t_2, t_9)$ and $Date^\geq(t_6, t_9)$, we find $p = 0$. Similarly, we have position $q = 0$ such that $NUM^<(\mathsf{Sorted}_1[q+1], t_9)$and $NUM^\geq(\mathsf{Sorted}_1[q], t_9)$.

For updating the index, t_9 can be inserted into this SkipList since $p = q$. According to Example 11, no violation is found. When updating a SkipList with a tuple s, s is always inserted into the bottom layer. Moreover, s may be inserted into higher layers as well, according to a randomization parameter [12]. This is the technique at the core of SkipList such that a node can be searched in $O(log(n))$. As an example, Figure 13 shows the SkipList after the insertion of t_9. □

Remark. A tuple $s \in \triangle D$ can be inserted into a SkipList if we find position $p = q$ on it. If there are multiple such SkipLists, then we heuristically pick the one with the maximum size. If none exists, then we create a new SkipList; this increase the index rank. We experimentally study this factor in Section VII.

VI. Incremental Algorithm for POD Discovery

In this section, we present our incremental POD discovery algorithm in response to $\triangle D$ of tuple insertions, by leveraging auxiliary indexes for the set Σ of PODs on D.

Algorithm. We develop Algorithm IncPOD for incrementally discovering PODs, with inputs Σ on D, $\triangle D$, attribute set R, and the set $Ind(\Sigma)$ of indexes for Σ on D. It finds a complete set Σ' of minimal valid PODs on $D + \triangle D$. This is done in an incremental way by computing $\triangle\Sigma$, such that $\Sigma' = \Sigma \oplus \triangle\Sigma$. Recall that $\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$, where $\triangle\Sigma^+$ contains new

Algorithm 4: IncPOD

Input: a complete set Σ of minimal and valid PODs on D, a set $\triangle D$ of tuple insertions, attribute set R, and the set $Ind(\Sigma)$ of indexes for covering PODs in Σ
Output: a complete set Σ' of minimal and valid PODs on $D + \triangle D$. $\Sigma' = \Sigma \oplus \triangle\Sigma$, where $\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$

1 find the set T of violating tuple pairs *w.r.t.* Σ on $D + \triangle D$, by leveraging $Ind(\Sigma)$;
2 $\Sigma' \leftarrow \Sigma$;
3 **for** *each tuple pair* $(t, s) \in T$ **do**
4 $\Sigma_{temp} \leftarrow \Sigma'$;
5 **for** *each* $\sigma \in \Sigma_{temp}$ **do**
6 **if** (t, s) *violate* σ **then**
7 $\Psi \leftarrow \mathsf{ExtendPOD}(\sigma, (t, s), R)$;
8 $\Sigma' \leftarrow \Sigma' \backslash \sigma \cup \Psi$;
9 $\Sigma' \leftarrow \mathsf{Minimize}\ (\Sigma', \Sigma' - \Sigma)$;
10 $\triangle\Sigma^+ \leftarrow \Sigma' \setminus \Sigma$; $\triangle\Sigma^- \leftarrow \Sigma \setminus \Sigma'$;

minimal valid PODs as additions to Σ, while $\triangle\Sigma^-$ contains PODs that are to be removed from Σ.

It first finds the set T of violating tuple pairs *w.r.t.* Σ on $D + \triangle D$, by leveraging $Ind(\Sigma)$ (line 1). This consists of index updates with $\triangle D$ (Section V-C) and index visits that help fetch violating tuple pairs (Section V-B). Σ' is initialized as Σ (line 2). For each violating tuple pair (t, s), it enumerates all $\sigma \in \Sigma'$ (lines 3-8). If (t, s) violates σ, then Algorithm ExtendPOD is called to evolve a set Ψ of new PODs based on σ for resolving the violation incurred by (t, s) (line 7). Σ' is then updated by removing σ and adding Ψ (line 8). That is, σ is put into $\triangle\Sigma^-$, while Ψ is added into $\triangle\Sigma^+$. After processing tuple pair (t, s), IncPOD moves to the next tuple pair with the updated Σ'. As will be illustrated shortly, new PODs added into Σ' in dealing with (t, s) may evolve again after handling subsequent tuple pairs, and once the violation incurred by (t, s) is resolved, it is guaranteed that further modifications of Σ' will never introduce new violations *w.r.t.* tuple pair (t, s).

Function Minimize is called to remove non-minimal PODs from $\Sigma' - \Sigma$ (line 9); it is to remove non-minimal PODs from the set of new valid PODs, *i.e.,* $\triangle\Sigma^+$. Note that PODs in Σ are still minimal on $D + \triangle D$ if they are valid on $D + \triangle D$. To check the minimality of a POD $\mathcal{X} \hookrightarrow \mathcal{Y}$ in $\Sigma' - \Sigma$ by definition (Section III), if suffices to only consider PODs $\mathcal{X}' \hookrightarrow \mathcal{Y}'$ in Σ', where $|\mathcal{X}'| \leq |\mathcal{X}|$ and $|\mathcal{Y}| \leq |\mathcal{Y}'|$. Σ' is then the complete set of minimal and valid PODs on $D + \triangle D$, and $\triangle\Sigma^+$ and $\triangle\Sigma^-$ are the differences between Σ and Σ' (line 10).

Algorithm ExtendPOD is at the core of IncPOD, which aims to generate new valid PODs in $\triangle\Sigma^+$ based on invalid PODs in $\triangle\Sigma^-$. ExtendPOD enumerates all possible ways to resolve the violation, for new PODs in Ψ. Specifically, ExtendPOD may (1) strengthen the LHS conditions when handling $\{\leq, \geq\}$ (lines 2-5), or (2) relax the RHS conditions when handling $\{=, <\}$ (lines 6-8), or (3) add more marked attributes on the LHS (lines 9-11). It suffices to only consider $=, <$ for the RHS attribute, due to symmetry (Section III). It is easy to see that σ logically implies any $\sigma' \in \Psi$: any relation that satisfies σ must satisfy σ'. Hence, it never introduces new violations when replacing σ by Ψ in Σ' (line 8 in IncPOD).

Example 14: For $\{\mathsf{TXA}^>, \mathsf{SAL}^< \} \hookrightarrow \{\mathsf{RATE}^< \}$, we find a

violating tuple pair (t_8, t_3): $TXA^{>}(t_8, t_3)$, $SAL^{<}(t_8, t_3)$ and $RATE^{=}(t_8, t_3)$. ExtendPOD may add more marked attributes (lines 9-11), *e.g.,* {ST$^{=}$, TXA$^{>}$, SAL$^{<}$ } $\hookrightarrow$ {RATE$^{<}$ } or {PH$^{=}$, TXA$^{>}$, SAL$^{<}$ } $\hookrightarrow$ {RATE$^{<}$ }, or modify its RHS operator (lines 6-8), *e.g.,* {TXA$^{>}$, SAL$^{<}$ } $\hookrightarrow$ {RATE$^{\leq}$ }.

Consider {PH$^{=}$, TXA$^{>}$, SAL$^{<}$ } $\hookrightarrow$ {RATE$^{<}$ } and another violating pair (t_7, t_1): $PH^{=}(t_7, t_1)$, $TXA^{>}(t_7, t_1)$, $SAL^{<}(t_7, t_1)$ and $RATE^{\geq}(t_7, t_1)$. We can see that (t_7, t_1) also violates {PH$^{=}$, TXA$^{>}$, SAL$^{<}$ } $\hookrightarrow$ {RATE$^{<}$ } and is hence in the set of violating tuple pairs identified on D. To resolve this violation, ExtendPOD may again add more marked attributes, *e.g.,* {ST$^{=}$, PH$^{=}$, TXA$^{>}$, SAL$^{<}$ } $\hookrightarrow$ {RATE$^{<}$ }. However, since {ST$^{=}$, TXA$^{>}$, SAL$^{<}$ } $\hookrightarrow$ {RATE$^{<}$ } is valid, {ST$^{=}$, PH$^{=}$, TXA$^{>}$, SAL$^{<}$ } $\hookrightarrow$ {RATE$^{<}$ } is not minimal and will be removed by Minimize (line 9 of IncPOD). □

Correctness. IncPOD finds a complete set Σ' of minimal valid PODs on $D + \triangle D$. (1) Validity. If a tuple pair (t, s) incurs a violation against σ, then ExtendPOD is called to generate a set Ψ of new PODs based on σ. ExtendPOD never introduces new violations beyond T, the set of violating tuple pairs detected on $D + \triangle D$ for Σ. Specifically, it can be seen that if any tuple pair (t', s') violates a POD in Ψ, then (t', s') also violates σ and is hence in T. Note that in this case (t', s') *cannot* be processed before (t, s) in IncPOD; otherwise, σ is removed from Σ' after processing (t', s'). Therefore, when no violation is detected on T, all PODs in Σ' are valid on $D + \triangle D$. (2) Minimality. The minimality of PODs is guaranteed due to Minimize in line 9 of IncPOD. (3) Completeness. Suppose a POD σ is valid on $D + \triangle D$ but is not in Σ'. We show σ is not minimal on $D + \triangle D$. (a) If σ is minimal on D, then σ is in Σ since σ is valid on D and Σ is the complete set of minimal valid PODs on D. This contradicts with the assumption that σ is not in Σ'=$\Sigma \oplus \triangle\Sigma$, since σ is obviously not in Σ^{-}. (b) If σ is not minimal on D then there exists σ' that is valid on D and logically implies σ. Obviously σ is not minimal if σ' is still valid on $D + \triangle D$. Now suppose σ' is invalid on $D + \triangle D$. Based on σ', ExtendPOD enumerates all possible ways for generating PODs valid on $D + \triangle D$. If σ is not in Σ', then there must be some POD in Σ' that logically implies σ, and hence makes σ not-minimal on $D + \triangle D$.

Complexity. IncPOD is far more efficient than its batch counterpart in practice, since new PODs are discovered based on PODs in Σ that become invalid on $D + \triangle D$. Specifically, (1) the set T of violating tuple pairs *w.r.t.* Σ on $D + \triangle D$ is efficiently computed leveraging $Ind(\Sigma)$, with a cost dependent on $|\triangle D|$ and $log(|D|)$. (2) It is linear in the size of T for computing $\triangle\Sigma$, since each tuple pair in T is treated only once. (3) ExtendPOD has a complexity linear in $|R|$, since it adds or modifies one marked attribute for each POD in Ψ.

Remark. It can be verified that the indexes built for Σ on D (ChooseIndex) also cover all PODs in $\Sigma \oplus \triangle\Sigma$. That is, the same set of indexes can be used for incremental POD discovery for $\Sigma \oplus \triangle\Sigma$, in response to further insertions to $D + \triangle D$.

Algorithm 5: ExtendPOD

Input: a violating tuple pair(t, s) *w.r.t.* $\sigma = \mathcal{X} \hookrightarrow B^{op'}$, attribute set R
Output: the set Ψ of PODs

1 $\Psi \leftarrow \{\}$;
2 **for** *each* $x^{op} \in \mathcal{X}$ **do**
3 **if** $op \in \{\leq, \geq\}$ **then**
4 $\mathcal{X}' \leftarrow$ replace $x^{\leq}$ (resp. $x^{\geq}$) by $x^{<}$ or $x^{=}$ (resp. $x^{>}$ or $x^{=}$) in $\mathcal{X}$, if the violation incurred by t, s is resolved;
5 $\Psi \leftarrow \Psi \cup \{\mathcal{X}' \hookrightarrow B^{op'}\}$;
6 **if** $op' \in \{=, <\}$ **then**
7 $op'' \leftarrow$ replace $op' =$ (or $<$) by $\leq$, if the violation incurred by t, s is resolved;
8 $\Psi \leftarrow \Psi \cup \{\mathcal{X} \hookrightarrow B^{op''}\}$;
9 **for** *attribute* $A \in R \backslash (X \cup \{B\})$ **do**
10 **for** *each* A^{op} *such that* $A^{\overline{op}}(t, s)$ **do**
11 $\Psi \leftarrow \Psi \cup \{\mathcal{X} \cup \{A^{op}\} \hookrightarrow B^{op'}\}$

VII. Experimental Evaluations

We first present the experimental settings, and then conduct experiments to (1) verify the effectiveness and efficiency of incremental POD discovery, to (2) demonstrate the effectiveness and efficiency of our indexing techniques, and to (3) analyze properties of indexes and algorithms in detail.

A. Experimental Settings

Datasets. We use a host of real-life and synthetic datasets that are employed in experimental studies on constraint discoveries [2], [11], [13]–[15] (see Section II). (1) Real-life data. SPS contains US stock records (http://pages.swcp.com/stocks/). FLI is about US flights information (www.transtats.bts.gov). LETTER is a dataset of character image features (https://archive.ics.uci.edu/ml/datasets). NCV contains data of voters from North Carolina (ncsbe.gov). STR is a dataset about 3D shapes of proteins, nucleic acids and complex assemblies (http://www.rcsb.org). (2) Synthetic data. FDR15 and FDR30 are two synthetic benchmark datasets (http://metanome.de).

We summarize details of all datasets in Table IV, and denote the number of attributes (resp. tuples) by $|R|$ (resp. $|D|$).

Implementation. We implement all the algorithms in Java.
(1) OptIndex and ChooseIndex (Section V), for building and choosing indexes, as a pre-processing stage for IncPOD.
(2) IncPOD, the algorithm for incremental POD discovery, which combines our techniques together. It employs Fetch (Section IV) on indexes to help identify POD violations (Section V-B), updates indexes (Section V-C) and computes new valid PODs by calling ExtendPOD (Section VI).
(3) BatchPOD, the algorithm for batch POD discovery. We implement BatchPOD based on Hydra [2], the state-of-the-art DC discovery technique. Each POD can always be encoded as a DC. For example, a POD {A$^{>}$,B$^{>}$ }$\hookrightarrow$ {C$^{>}$ } can be denoted as a DC: $\forall t, s, \neg(t_A > s_A \wedge t_B > s_B \wedge t_C \leqslant s_C)$. We modify the source code of Hydra (available online www.metanome.de), such that it only considers predicates of the form t_A *op* s_A where *op* $\in \{<, \leq, =, >, \geq\}$. In addition, we allow at most one predicate of the form $t_A \neq s_A$ in each DC, for encoding

TABLE IV
DATASETS AND EXECUTION STATISTICS

Data	$\|R\|$	$\|D\|$	$\|\triangle D\|$	IncPOD	BatchPOD	$\|Ind(\Sigma)\|$	$\|\Sigma\|$	$\|\triangle\Sigma^-\|$	$\|\triangle\Sigma^+\|$
SPS	7	90K	27K	**0.9s**	8.57s	5	20	4	6
FLI	17	300K	60K	**67s**	461s	22	1,481	48	84
STR	5	450K	125K	**9s**	109s	11	21	0	0
LETTER	12	15K	4.5K	**175s**	1,538s	23	8,172	661	58
NCV	18	300K	100K	**76s**	7,601s	13	1,134	17	19
FDR15	15	200K	50K	**14s**	65s	13	185	3	37
FDR30	30	200K	50K	**48s**	265s	31	1,152	6	695

PODs of the form $\{A_1^{op_1}, \ldots, A_m^{op_m}\} \hookrightarrow B^{=}$. In this way, BatchPOD traverses the entire space of PODs.

(4) IEJoin [10], the state-of-the-art algorithm for inequality joins with data updates. Violations of PODs on a relation D can be identified by a self-join on D with inequality conditions. For example, violating tuple pairs *w.r.t.* $\{\mathsf{A}^{>},\mathsf{B}^{>}\} \hookrightarrow \{\mathsf{C}^{>}\}$ can be computed as the result of a SQL query:

SELECT $r.id$, $s.id$
FROM D r, D s
WHERE $r.A > s.A$ AND $r.B > s.B$ AND $r.C \leq s.C$

Running environment. All experiments are run on a machine powered by an Intel Core(TM)2 Duo 3.00GHz CPU T7300 with 64GB of memory, using scientific Linux.

Parameter settings. We consider 3 parameters in our experiments: (1) $|D|$: the number of tuples; (2) $|\triangle D|$: the number of inserted tuples; and (3) $|R|$: the number of attributes. When required, we vary $|D|$, $|\triangle D|$ and $|R|$ by taking random sampling (or projections) of the data. We define the ratio of incremental data as $\frac{|\triangle D|}{|D|}$. We report the average of 5 runs.

Measurement. We compute Σ on D with BatchPOD, and build indexes for Σ on D. Leveraging the indexes, we incrementally find a complete POD set on $D + \triangle D$ with IncPOD. The correctness of IncPOD is verified by checking its result against that of BatchPOD on $D + \triangle D$. The time of IncPOD consists of the times for index visits and updates, for fetching violating tuples, and for computing $\triangle\Sigma$. The time of BatchPOD is the time for computing all PODs on $D + \triangle D$ from scratch.

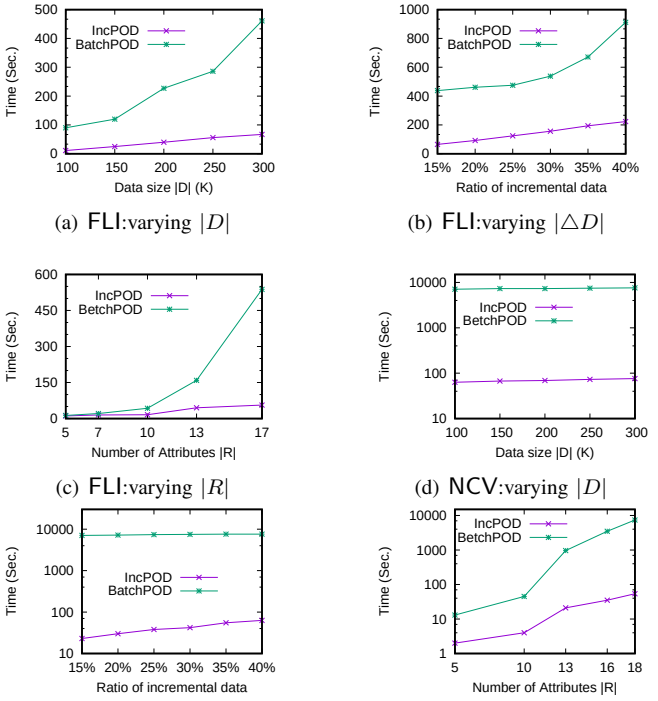
B. Experimental Findings

Exp-1: IncPOD against BatchPOD. Table IV reports running times (in seconds) of IncPOD against BatchPOD. We can see that IncPOD significantly outperforms BatchPOD on all datasets, up to two orders of magnitude. We also show in Table IV the number of PODs in Σ, $\triangle\Sigma^-$ and $\triangle\Sigma^+$, denoted by $|\Sigma|$, $|\triangle\Sigma^-|$ and $|\triangle\Sigma^+|$, respectively. We find even if $\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$ is large, *e.g.,* on datasets LETTER and FDR30, IncPOD still performs much better than BatchPOD.

We conduct more experiments by varying parameters.

(1) We set $|D| = 300K$, $\frac{|\triangle D|}{|D|} = 20\%$ and $|R| = 17$ by default on FLI, and vary one parameter in each experiment.

Varying $|D|$. Figure 14(a) shows the results by varying $|D|$ from 100K to 300K ($|\triangle D|$ from 20K to 60K). BatchPOD scales well, consistent with the results reported in [2]. IncPOD performs much better. It takes only 67 seconds for IncPOD but 461 seconds for BatchPOD, when $|D|$= 300K.



(a) FLI:varying $|D|$ (b) FLI:varying $|\triangle D|$
(c) FLI:varying $|R|$ (d) NCV:varying $|D|$
(e) NCV:varying $|\triangle D|$ (f) NCV: vary $|R|$

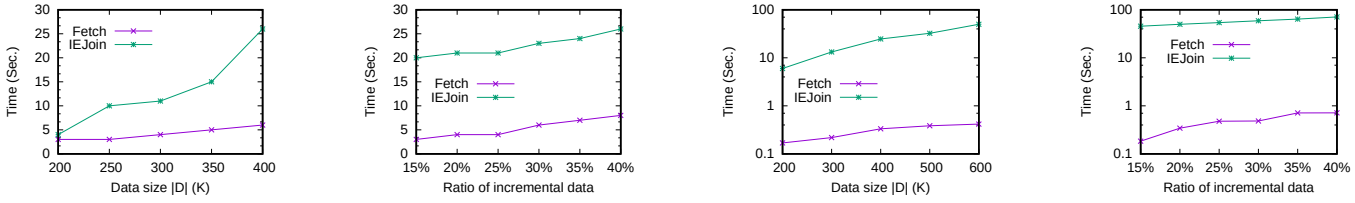
Fig. 14. IncPOD against BatchPOD

Varying $|\triangle D|$. Figure 14(b) shows results by varying $|\triangle D|$ from 45K to 120K (the ratio of $|\triangle D|$ to $|D|$ increases from 15% to 40%). We find IncPOD consistently outperforms BatchPOD, and scales very well with $|\triangle D|$: the time increases from 65 seconds to 223 seconds as $|\triangle D|$ increases.

Varying $|R|$. By varying $|R|$ from 5 to 17, we report results in Figure 14(c). It can be seen that $|R|$ significantly affects the efficiency of BatchPOD, consistent with [2]. This is because $|\Sigma|$ grows exponentially with $|R|$. IncPOD scales much better than BatchPOD: as $|R|$ increases from 5 to 17, the time of BatchPOD increases by more than 40 times, while the time of IncPOD only increases by 5 times. We find the time of IncPOD is affected by the number of indexes that cover all PODs in Σ ($Ind(\Sigma)$ of ChooseIndex in Section V-B). We show the results in Table IV, denoted as $|Ind(\Sigma)|$. We see $|Ind(\Sigma)|$ is typically much smaller than $|\Sigma|$; this explains why IncPOD is less sensitive to $|R|$ than BatchPOD. We will further analyze $|Ind(\Sigma)|$ in Exp-3.

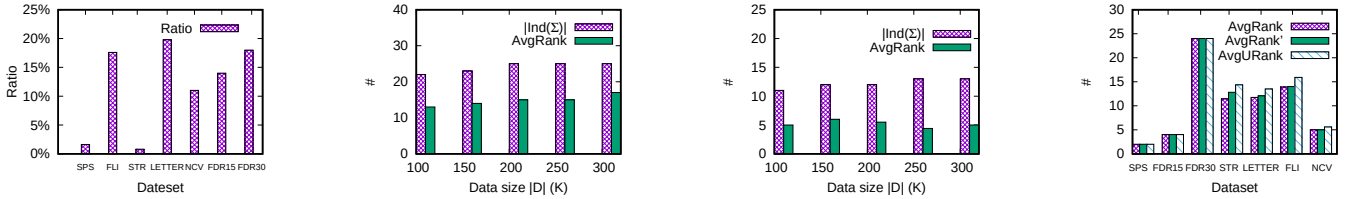
(2) We set $|D|$ = 300K, $|\triangle D| = 100K$ and $|R|$ = 18 by default on NCV. We vary $|D|$ from 100K to 300K in Figure 14(d), $\frac{|\triangle D|}{|D|}$ from 15% to 40% in Figure 14(e), and $|R|$ from 5 to 18 in Figure 14(f). We use log scale in these experiments, since IncPOD consistently outperforms BatchPOD by about two orders of magnitude. We find numbers of distinct values on most attributes of NCV are large. It hence takes BatchPOD far more time to handle inequality operators on NCV than FLI. In contrast, our incremental approach is less sensitive to this factor, and delivers great improvements in the efficiency.

Exp-2: Fetch against IEJoin. We experimentally verify the benefit of our index technique, against IEJoin [10]. We still

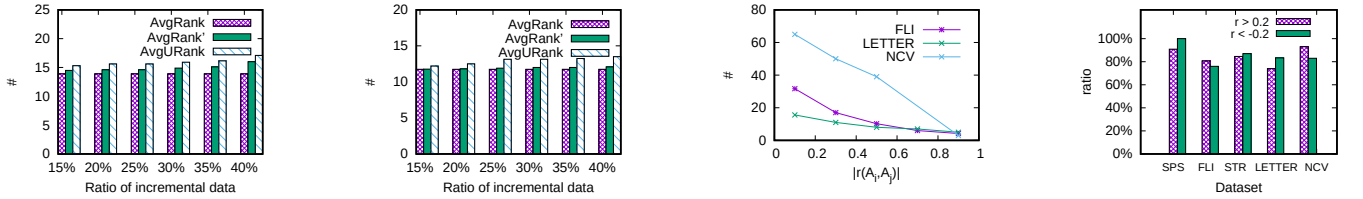


(a) FLI:varying $|D|$ (b) FLI:varying $|\triangle D|$ (c) NCV:varying $|D|$ (d) NCV:varying $|\triangle D|$

Fig. 15. Fetch against IEJoin



(a) Effectiveness of indexes (b) FLI: index number and rank (c) NCV: index number and rank (d) Rank changes on datasets



(e) FLI: rank changes (f) LETTER: rank changes (g) Rank *w.r.t.* $|r(A_i, A_j)|$ (h) Positive or negative correlation

Fig. 16. Effectiveness of indexes, index number and rank, and score function

report the average over 5 runs; each time we randomly choose an attribute pair A, B and enumerate all tuple $s \in \triangle D$. We compare fetch against IEJoin in the time for identifying $T^s_{A^{op_1}B^{op_2}}$ and $\overline{T}^s_{A^{op_1}B^{op_2}}$ (Section IV).

(1) We set $|D| = 300K$, $\frac{|\triangle D|}{|D|} = 20\%$ by default on FLI.

Varying $|D|$. Figure 15(a) shows results by varying $|D|$ from 200K to 400K. Fetch scales better than IEJoin: the time for Fetch increases from 3.3 seconds to 6.4 seconds, while the time for IEJoin increases from 4 seconds to 27 seconds. We find the time increase of Fetch is mainly due to more time required for collecting results.

Varying $|\triangle D|$. We vary ratio $\frac{|\triangle D|}{|D|}$ from 15% to 40% in Figure 15(b). Fetch scales well with $|\triangle D|$: the time increases from 3.4 seconds to 8 seconds; Fetch is almost linear in $|\triangle D|$. IEJoin does not enjoy this feature. It has a cost almost linear in $|D + \triangle D|$, consistent with the results in [10].

(2) We further compare Fetch against IEJoin using NCV with $|D|$ = 600K and $\frac{|\triangle D|}{|D|}$ = 20% by default. We vary $|D|$ from 200K to 600K in Figure 15(c), and $\frac{|\triangle D|}{|D|}$ from 15% to 40% in Figure 15(d). The result size on NCV is much smaller than that on FLI, and hence both Fetch and IEJoin take less time in collecting results. This significantly favors Fetch: the time for collecting results is a necessary cost of both algorithms, while Fetch takes far less time than IEJoin in other steps. We see Fetch scales better than IEJoin with $|D|$. As $|D|$ increases from 200K to 600K, the time for Fetch increase from 0.16 seconds to 0.4 seconds, while IEJoin increase from 6 seconds to 50 seconds. Fetch also significantly outperforms IEJoin when $\frac{|\triangle D|}{|D|}$ = 40%, by almost two orders of magnitude.

Exp-3: Analyses of indexes. We conduct experiments to study the effectiveness and properties of our indexes in detail.

(1) Indexes help find candidates of violating tuples *w.r.t.* $\triangle D$, on which remaining conditions of PODs are checked. We compute the ratio of the size of violating candidates to $|D|$ on all datasets of Table IV, shown in Figure 16(a). We see the ratios are below 20% on all datasets; indexes help skip more than 80% of the tuples. We find the combination of two marked attributes lead to a good selectivity in most cases.

(2) Recall that the efficiency of IncPOD is affected by $|Ind(\Sigma)|$ (Exp-1 (1)), and the ranks of indexes (Section IV). We denote by $AvgRank$ the average rank of indexes in $Ind(\Sigma)$, excluding equality indexes Index($A_i^=$). We set $|R|$ = 17 and vary $|D|$ from 100K to 300K on FLI, and report results in Figure 16(b). $|Ind(\Sigma)|$ only increases from 22 to 25. $AvgRank$ also varies in the range of [13, 17]. In Figure 16(c), we show results on NCV with $|R|$ = 18 and $|D|$ from 100K to 300K. $|Ind(\Sigma)|$ is in the range of [11, 13] and $AvgRank$ is in [4.4, 6]. We contend that both $|Ind(\Sigma)|$ and $AvgRank$ are insensitive to $|D|$.

(3) Recall that updating an index may increase its rank (Section V-C). We denote by $AvgURank$ the average index rank on D+$\triangle D$ after updates. For comparison, we also apply OptIndex to D+$\triangle D$ for indexes with minimum ranks, and denote the average rank by $AvgRank'$. Figure 16(d) reports results for all datasets in Table IV. We see $AvgURank$ always increases slightly, at most 115% of $AvgRank$. Moreover, the difference between $AvgURank$ and $AvgRank'$ is also small; $AvgRank'$ is at most larger than $AvgURank$ by 12%. We

contend that index ranks are not sensitive to updates with $\triangle D$.

We then vary the ratio of incremental data. We set $|R|$=17, $|D|$=300K, vary $\frac{|\triangle D|}{|D|}$ from 15% to 40% on FLI, and report results in Figure 16(e). $AvgURank$ increases slightly from 15.5 to 17.1, and is consistently within [105%, 107%] of $AvgRank'$. Figure 16(f) reports results on LETTER with $|R|$=12, $|D|$=15K. $AvgURank$ increases from 11.75 to 13.5 as $\frac{|\triangle D|}{|D|}$ increases from 15% to 40%, and is within [104%,112%] of $AvgRank'$. We see that the impact on index ranks incurred by updates is small even when $\frac{|\triangle D|}{|D|}$ is 40%.

Exp-4: Score function and parameter α. We experimentally study our score function and the parameter α employed in ChooseIndex (Section V-B).

(1) On real-life FLI, LETTER and NCV, we randomly choose 100 attribute pairs and build indexes on them. We compute the absolute value of correlation coefficient for each pair A_i, A_j, *i.e.,* $|r(A_i, A_j)|$. Based on $|r(A_i, A_j)|$, we cluster indexes into five ranges [0, 0.2], ..., [0.8, 1], and compute the average rank of indexes in each range. Figure 16(g) clearly shows that indexes on attribute pairs with a large $|r(A_i, A_j)|$ have small ranks. This justifies our score functions.

(2) Recall that we use a parameter $\alpha>0$ in ChooseIndex, and build Index$(A_i^<, A_j^<)$ (resp. Index$(A_i^<, A_j^>)$) instead of Index$(A_i^<, A_j^>)$ (resp. Index$(A_i^>, A_j^<)$) if $r(A_i, A_j)>\alpha$ (resp. $r(A_i, A_j)<-\alpha$). We experimentally test α on real-life datasets in Table IV. We find that the adjustment of operator according to $r(A_i, A_j)$, has on average 85% (at least 78%) chance of improving index ranks when α=0.2 (shown in Figure 16(h)), and it reaches almost 100% by setting α=0.3 (not shown).

Summary. From the tests, we see the followings.
(1) Incremental POD discovery is effective. IncPOD is on average 10 times faster than BatchPOD when $\frac{|\triangle D|}{|D|}$ is about 30%, excluding NCV. IncPOD consistently outperforms BatchPOD on NCV by about two orders of magnitude.
(2) Our index technique is efficient. Leveraging the indexes, Fetch is on average 4 times and 75 times faster than IEJoin on FLI and NCV, respectively.
(3) Our indexes are effective in reducing more than 80% of tuple comparisons in POD validations on tested datasets.
(4) Both $|Ind(\Sigma)|$ and $AvgRank$ are insensitive to $|D|$. As $|D|$ triples, $|Ind(\Sigma)|$ and $AvgRank$ increases by [13%, 18%] and [20%, 29%], respectively. The impact on index ranks incurred by updates is small. As $\frac{|\triangle D|}{|D|}$ increases from 15% to 40%, $AvgURank$ on $D+\triangle D$ is larger than $AvgRank$ on D by [4%,18%] on tested data.
(5) The correlation coefficient on attributes is a good measure for choosing attributes and operators for indexes.

VIII. Conclusions

This is a first effort towards POD discovery on dynamic data. We have presented an indexing technique for effectively identifying violations incurred by $\triangle D$, algorithms for choosing indexing attributes and for building and updating the indexes. We have also proposed and experimentally verified our methods to compute $\triangle\Sigma$.

A couple of topics need further investigation. We are to study techniques to handle updates with both tuple insertions and deletions, and methods in a distributed setting.

References

[1] Ziawasch Abedjan, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Detecting unique column combinations on dynamic data. In *ICDE*, 2014.
[2] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. Efficient denial constraint discovery with hydra. *PVLDB*, 11(3):311–323, 2017.
[3] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.
[4] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, 2013.
[5] Cristian Consonni, Paolo Sottovia, Alberto Montresor, and Yannis Velegrakis. Discovering order dependencies through order compatibility. In *EDBT*, 2019.
[6] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 2008.
[7] Seymour Ginsburg and Richard Hull. Order dependency in the relational model. *Theor. Comput. Sci.*, 26:149–195, 1983.
[8] Seymour Ginsburg and Richard Hull. Sort sets in the relational model. *J. ACM*, 33(3):465–488, 1986.
[9] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. Lightning fast and space efficient inequality joins. *PVLDB*, 8(13):2074–2085, 2015.
[10] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Panos Kalnis. Fast and scalable inequality joins. *VLDB J.*, 26(1):125–150, 2017.
[11] Philipp Langer and Felix Naumann. Efficient order dependency detection. *VLDB J.*, 25(2):223–241, 2016.
[12] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.
[13] Philipp Schirmer, Thorsten Papenbrock, Sebastian Kruse, Felix Naumann, Dennis Hempfing, Torben Mayer, and Daniel Neuschäfer-Rube. Dynfd: Functional dependency discovery in dynamic datasets. In *EDBT*, 2019.
[14] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. Effective and complete discovery of order dependencies via set-based axiomatization. *PVLDB*, 10(7):721–732, 2017.
[15] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. Effective and complete discovery of bidirectional order dependencies via set-based axioms. *VLDB J.*, 27(4):573–591, 2018.
[16] Jaroslaw Szlichta, Parke Godfrey, and Jarek Gryz. Fundamentals of order dependencies. *PVLDB*, 5(11):1220–1231, 2012.
[17] Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. Expressiveness and complexity of order dependencies. *PVLDB*, 6(14):1858–1869, 2013.
[18] Lin Zhu, Xu Sun, Zijing Tan, Kejia Yang, Weidong Yang, Xiangdong Zhou, and Yingjie Tian. Incremental discovery of order dependencies on tuple insertions. In *DASFAA*, 2019.