# Pointwise Order Dependency Discovery on Dynamic Data

Zijing Tan[1] Ai Ran[1] Shuai Ma[2] Sheng Qin[1]
[1]School of Computer Science, Fudan University, Shanghai, China
[2]SKLSDE Lab, Beihang University, Beijing, China
{zjtan, aran17, stan17}@fudan.edu.cn mashuai@buaa.edu.cn

## 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 $\Sigma$ of valid and minimal PODs on a given data set $D$. In practice $D$ is typically large and keeps changing, and it is prohibitively expensive to compute $\Sigma$ from scratch every time. In this paper, we make a first effort to study the incremental POD discovery problem, aiming at computing changes $\triangle\Sigma$ to $\Sigma$ 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. (1) We first propose a novel indexing technique for inputs $\Sigma$ and $D$. We give algorithms to build and choose indexes for $\Sigma$ and $D$, and to update indexes in response to $\triangle D$. We show that POD violations *w.r.t.* $\Sigma$ 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 $\Sigma$ and identified violations caused by $\triangle D$. The PODs in $\Sigma$ 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 $\Sigma$ 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.

**PVLDB Reference Format:**
Zijing Tan, Ai Ran, Shuai Ma, Sheng Qin. Pointwise Order Dependency Discovery on Dynamic Data. *PVLDB*, xx(xxx): xxxx-yyyy, xx.
DOI: https://doi.org/xxxxx/xxxxxxx.xxxxxxx

## 1. INTRODUCTION

Data dependencies are valuable in, *e.g.,* schema design [4], query optimization [6,30,35] and data quality management [10, 20], among others. A host of dependencies are proposed in literature; see *e.g.,* functional dependencies (FDs), conditional functional dependencies [11], denial constraints (DCs) [8], sequential dependencies [17] and differential dependencies [31]. Recently, pointwise order dependencies (PODs) are introduced in [15, 16, 36], as a dependency language to specify ordering semantics on attributes of tuples.

**Example 1:** Consider the sample tax data (relation instance $D_1$) in Table 1. 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 present some constraints that hold on $D_1$.
$\sigma_1$: Persons with the same zip code live in the same state.
$\sigma_2$: The SSN of a person determines his/her name.
$\sigma_3$: A person with a lower salary and a higher tax exemption amount has a lower tax rate.
$\sigma_4$: The tax serial number increases with the tax date for a single person.

We see that $\sigma_1$ and $\sigma_2$ are two FDs that only require the *equality* operator (=). In contrast, more comparison operators are needed for $\sigma_3$ and $\sigma_4$, to express ordering semantics on tuple values. Specifically, all these constraints can be expressed in the notation of PODs (formalized in Section 3).
$\sigma_1$: $\{\mathsf{ZIP}^{=}\} \hookrightarrow \{\mathsf{ST}^{=}\}$
$\sigma_2$: $\{\mathsf{SSN}^{=}\} \hookrightarrow \{\mathsf{FName}^{=}, \mathsf{LName}^{=}\}$
$\sigma_3$: $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$
$\sigma_4$: $\{\mathsf{SSN}^{=}, \mathsf{Date}^{<}\} \hookrightarrow \{\mathsf{NUM}^{<}\}$ □

PODs subsume FDs as a special case, and can additionally specify ordering semantics. This is desirable since ordered attributes are very common in data values. Hence, PODs can be employed to define data quality rules [10]. PODs can also be leveraged to optimize queries with inequality joins [21]. For example, an inequality join on "$t.TXA > s.TXA$ and $t.SAL < s.SAL$" can be rewritten as "$t.RATE < s.RATE$" by the known valid $\sigma_3$ and hence more efficiently computed.

To take advantage of PODs, the set $\Sigma$ of PODs that hold (are valid) on a relation $D$ is expected to be known. It is typically too expensive to design dependencies manually [1, 2], which highlights the quest for automatic discovery techniques, to find the set $\Sigma$ of PODs that hold on $D$.

Worse, data in practice typically keeps changing, 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. We must update dependencies for tuple insertions because tuple insertions (but not deletions) may cause valid dependencies to be invalid and using invalid dependencies leads to errors.


*Proceedings of the VLDB Endowment,* Vol. xx, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/xxxxx/xxxxxxx.xxxxxxx

**Table 1: 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 2: 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 |

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 $\Sigma$, such that $\Sigma \oplus \triangle\Sigma$ is a set of PODs that hold on $D + \triangle D$. We use the notation "$\oplus$" since some PODs in $\Sigma$ no longer hold and are removed from $\Sigma$, while some new valid PODs may be discovered and added into $\Sigma$. This approach is known as incremental dependency discovery [3, 29, 39], in contrast to batch (non-incremental) approaches that discover dependencies on the whole data set. Incremental POD discovery is very intricate for tuple insertions, as illustrated in the following example.

**Example 2:** Consider the incremental $\triangle D_1$ (Table 2). 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 $\sigma_1$, $\sigma_2$ and $\sigma_4$ still hold, but $\sigma_3$ is invalid now. Specifically, $\sigma_3$ is invalid due to violations caused by tuples $t_1$, $t_7$ and $t_3$, $t_8$.

Therefore, from $\Sigma$ we should remove $\sigma_3$ that does 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 $\Sigma$. Intuitively, we can evolve new valid PODs based on PODs (in $\Sigma$) that are invalid on $D_1 + \triangle D_1$.

Based on $\sigma_3$, we may find two new PODs valid on $D_1 + \triangle D_1$. $\sigma'_3$=$\{\mathsf{ST}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ and $\sigma''_3 = \{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{\leq}\}$. $\sigma'_3$ introduces an additional attribute with a comparison operator to the left-hand side (LHS) of $\sigma_3$; the LHS conditions are strengthened. In contrast, $\sigma''_3$ relaxes $\sigma_3$ in its right-hand side (RHS) condition by changing '$<$' to '$\leq$'. Both $\sigma'_3$ and $\sigma''_3$ hold on $D_1$; obviously any POD that holds on $D_1 + \triangle D_1$ holds on $D_1$. But neither $\sigma'_3$ nor $\sigma''_3$ is in $\Sigma$. The reason is that discovery algorithms typically find only *minimal* constraints (formalized in Section 3). $\sigma'_3$ (resp. $\sigma''_3$) is not minimal on $D_1$ since $\sigma_3$ is valid on $D_1$ and *logically implies* $\sigma'_3$ (resp. $\sigma''_3$), in the sense that any relation that satisfies $\sigma_3$ must satisfy $\sigma'_3$ (resp. $\sigma''_3$). Since $\sigma_3$ does not hold on $D_1 + \triangle D_1$, $\sigma'_3$ and $\sigma''_3$ are minimal and valid PODs that should be added into $\Sigma$. □

**Contributions & organizations.** In this work, we make a first effort to study incremental POD discovery, for $\Sigma$ 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 indexing technique for two attributes with inequality operators. Leveraging the indexes, we show that violations of $\Sigma$ incurred by $\triangle D$ can be identified with a cost dependent on $log(|D|)$, where $|D|$ is the size of $D$. (Section 4).

(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 $\Sigma$ 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 5).

(3) We develop techniques for discovering $\triangle\Sigma$ based on $\Sigma$ and violations incurred by $\triangle D$. After identifying in $\Sigma$ 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 6).

(4) Using a host of real-life and synthetic datasets, we conduct extensive experiments to verify the effectiveness and efficiency of our approaches (Section 7).

## 2. RELATED WORK

Dependency discoveries are known as one of the most important aspects of data profiling [1, 2], and are extensively studied for a host of dependencies; see *e.g.,* [12–14,17–19,23, 27,28,38]. 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 dependencies and DCs.** There is a different notion of order dependency in literature, referred to as lexicographical order dependency (LOD) [34,36]. As opposed to PODs on sets of attributes, LODs are defined on lists of attributes. PODs *strictly generalize* LODs [36], in the sense that each LOD can be mapped into a set of PODs, and any relation satisfies the LOD iff it satisfies all PODs in the set. The generalization is *strict*, since some PODs cannot be expressed in LODs. Set-based canonical order dependencies are presented in [32, 33], which generalize LODs. PODs again strictly generalize canonical order dependencies.

Denial constraints (DCs) [7, 8] are given in a universally quantified first order logic formalism. 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 [9, 24], and for set-based canonical ODs in [32, 33]. They cannot be applied for discovering PODs, since PODs are more general.

As will be studied in Section 3, the batch discovery of PODs has a complexity *exponential* in the number of attributes, similar to the set-based canonical OD discovery [32,33], and is better than the LOD discovery [9,24], which is of a *factorial* complexity in the number of attributes.

Batch DC discoveries are investigated in [5, 7, 25]. We implement batch POD discovery algorithms by adapting DC discovery methods, to compare against our incremental POD discovery technique in experimental evaluations (Section 7).

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

**Incremental Dependency Discovery.** Incremental discoveries have received an increasing attention, due to their practical demands. Incremental discovery techniques are developed in [3] for unique column combinations (UCCs), *a.k.a.* candidate keys, and in [29] for FDs. Recently, incremental LOD discovery is also studied in [39].

Given a set $\Sigma$ of discovered constraints on $D$ and a set $\triangle D$ of updates, incremental discoveries mainly address two problems. The first one is to efficiently identify violations of $\Sigma$ incurred by $\triangle D$, and the second one is to refine $\Sigma$ for $\Sigma \oplus \triangle\Sigma$ that is valid on the updated data set. The solutions of the two problems, however, differ significantly from one constraint type to another. Incremental POD discovery is necessarily more complex; PODs subsume FDs (UCCs) and generalize LODs. As will be studied in Section 4, the incremental POD violation detection problem already deserves an in-depth treatment and motivates us to develop a novel indexing 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, 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. [21] is the state-of-the-art technique for inequality joins, which is further extended in [22] for incremental inequality joins in response to data updates. As opposed to [22] that has a cost dependent on $|D|$, we present a novel indexing technique that enables finding violating tuple pairs incurred by tuple insertions in a cost dependent on $log(|D|)$. Experimental evaluations demonstrate that our technique significantly outperforms [22] on tuple insertions.

## 3. PRELIMINARIES

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

**Basic notations.** $R(A,B,\dots)$ 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 [15, 16].** 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 1, we have $\mathsf{SAL}^{>}(t_1, t_0)$, since $t_1$ has a larger value on SAL than $t_0$. We also have $\mathsf{AC}^{=}(t_2, t_5)$. □

**Table 3: Operator inverse and Implication**

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

**Pointwise Order Dependencies (PODs) [15,16,36].** A POD $\sigma$ is of the form $\mathcal{X} \hookrightarrow \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are two sets of marked attributes. $\sigma$ 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 $\sigma$ is valid on $D$ if $\sigma$ holds on $D$.

Formal theoretical foundations of PODs, *e.g.,* a sound and complete set of inference rules and logical implication, are studied in [15]. Below we present some observations and additional definitions, inspired by the inference rules.

In the sequel, we consider PODs in which each attribute (neglecting operator) occurs at most once in the LHS and RHS. It can be verified that PODs with multiple occurrences of a same attribute can be safely skipped, *e.g.,* $\{\mathsf{A}^{<}\} \hookrightarrow \{\mathsf{A}^{\leq}\}$ and $\{\mathsf{A}^{<}, \mathsf{A}^{>}\} \hookrightarrow \{\mathsf{B}^{\leq}\}$ trivially hold; $\{\mathsf{A}^{<}, \mathsf{A}^{\leq}\} \hookrightarrow \{\mathsf{B}^{\leq}\}$ is equivalent to $\{\mathsf{A}^{<}\} \hookrightarrow \{\mathsf{B}^{\leq}\}$.

**Example 4:** $\sigma_4 = \{\mathsf{SSN}^{=}, \mathsf{Date}^{<}\} \hookrightarrow \{\mathsf{NUM}^{<}\}$ holds on $D_1$ (Table 1): 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 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 sequel. (3) Each POD $\sigma$ has a *symmetry* POD $\sigma_{sym}$ 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 $\sigma$ is valid iff $\sigma_{sym}$ is valid. To avoid redundancy, we consider PODs that take $op \in \{<, \leq, =\}$ on its RHS, *e.g.,* $\{\mathsf{A}_1^{=}, \mathsf{A}_2^{<}, \mathsf{A}_3^{\geq}\} \hookrightarrow \{\mathsf{B}^{<}\}$.

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 3. 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:** It can be seen that $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \subseteq \{\mathsf{ST}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\}$, 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 $\sigma$ is *not* minimal if $\sigma$ is logically implied by another valid POD $\sigma'$, which means that any relation $D$ that satisfies $\sigma'$ must satisfy $\sigma$.

**Minimal PODs.** A POD $\sigma = \mathcal{X} \hookrightarrow B^{op}$ is minimal if there does not exist another valid POD $\sigma' = \mathcal{X}' \hookrightarrow B^{op'}$, where $\mathcal{X}' \subseteq \mathcal{X}$ and $op \in im(op')$.

**Example 6:** It can be verified by definition that 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. Obviously, any relation that satisfies $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ must satisfy both $\{\mathsf{ST}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow$

$\{\mathsf{RATE}^{<}\}$ and $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{\leq}\}$. □

**Batch POD discovery.** Given a relation $D$ of schema $R$, batch POD discovery is to find the complete set $\Sigma$ of minimal and valid PODs on $D$.

PROPOSITION 1. *Batch POD discovery has a theoretical worst-case complexity of* $O(3 \cdot |R| \cdot 6^{(|R|-1)})$ *in the number* $|R|$ *of attributes.*

PROOF. The complexity is measured by the total number of candidate PODs for a given schema $R$ (the search space for batch POD discovery). Observe that for any POD, (1) on the RHS, an attribute $A \in R$ is present with one operator among the three possible ones ($\{<, \leq, =\}$ due to symmetry); and (2) on the LHS, any attribute in $R \setminus A$ is either (a) not present, or (b) present with one operator among the five possible ones ($\{<, \leq, >, \geq, =\}$). □

**Remark.** (1) POD discovery has a complexity exponential in $|R|$, better than the *factorial* complexity $O(|R|!)$ of LOD discovery. LOD discoveries [9, 24] have this complexity, although LODs with a common prefix on the LHS and RHS are skipped[1]. Observe that LODs *no longer* generalize FDs in this setting [36]. (2) Set-based canonical ODs [32,33] generalize LODs, and its discovery has a complexity exponential in $|R|$. This is because different LODs may be mapped to the same canonical ODs. PODs further generalize canonical ODs, still with an exponential complexity.

We find the increased expressive power of PODs is necessary. For example, $\{\mathsf{Mid}^{>}, \mathsf{Final}^{>}\} \hookrightarrow \{\mathsf{Grade}^{>}\}$ states that a student has a better grade if he/she has a better score in both the midterm and the final exam [15]. This ordering specification common in practice *cannot* be expressed in either LODs or canonical ODs.

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

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

The incremental discovery problem has a worst-case complexity as its batch counterpart: each batch discovery on $D'$ is equivalent to the incremental discovery if $D = \phi$, $\Sigma = \phi$ and $\triangle D = D'$. 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 present techniques that indeed achieve this goal in the following sections.

# 4. INDEXES FOR INEQUALITY OPERATORS

In this section, we present a novel index structure, which is an important building block for efficiently identifying POD violations in response to $\triangle D$.

[1] [24] considers LODs whose LHS and RHS attribute lists are disjoint. A recent errata note by authors of [32, 33] states that, [9] neglects LODs with a common prefix on the LHS and RHS. https://arxiv.org/pdf/1905.02010.pdf

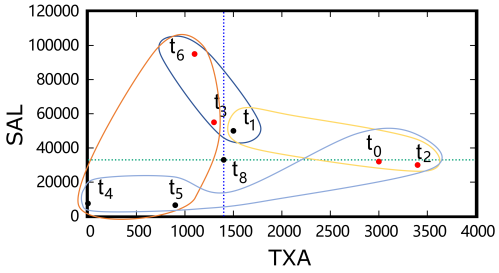



**Figure 1: Candidates of violating tuples *w.r.t.* $t_8$**

It is important for any incremental dependency discovery algorithm to efficiently identify violations incurred by $\triangle D$. Based on the known $D$ and $\Sigma$, all existing incremental discoveries [3, 29, 39] leverage indexes to speed up violation detections. However, the introduction of inequality operators ($<, \leq, >, \geq$) in PODs, significantly complicates the problem and hinders common indexing techniques.

**Example 7:** Consider the violation detection for $\sigma_3 = \{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ on relation $D_1$ (Table 1), when tuple $t_8$ is inserted. As shown in Figure 1, (1) we first compute four sets: $\{t|\mathsf{TXA}^{>}(t_8, t)\} = \{t_3, t_4, t_5, t_6\}$ 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 $\sigma_3$ to see whether $\sigma_3$ 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 for the computing in step (1). The reason is that for a tuple $s$ in $D_1$, either $\mathsf{TXA}^{>}(s, t_8)$ or $\mathsf{TXA}^{>}(t_8, s)$; similarly for $\mathsf{SAL}^{<}$. Intuitively, inequality operators typically have a *low selectivity* [21, 22]. □

Incremental POD discovery calls for novel indexing techniques. In contrast, incremental FD (UCC) discoveries [3, 29] only consider operator "=", and incremental LOD discovery [39] is facilitated by $B^{+}$-trees on list of attributes.

Intuitively, indexes can be built by combining together several marked attributes with inequality operators to remedy the low selectivity of a single one. There is a trade-off concerning the number of marked attributes: more attributes would lead to more *selective* indexes, that are however more specific, *i.e.,* only apply to PODs with all these attributes (Section 5.2), and that lead to indexes with large *ranks* (to be explained shortly). In the sequel, we develop a set of novel techniques for indexes on two marked attributes with inequality operators, and they are experimentally verified to perform very well.

**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 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. 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 $\mathsf{Sorted}_i$ ($i \in [1, k]$) 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

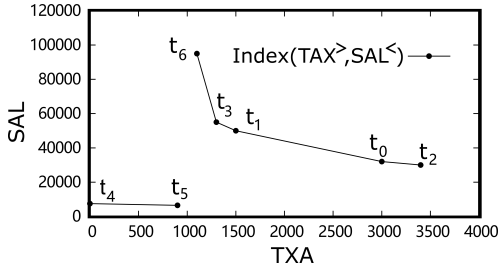



**Figure 2:** Index($\mathsf{TXA}^{>}, \mathsf{SAL}^{<}$)

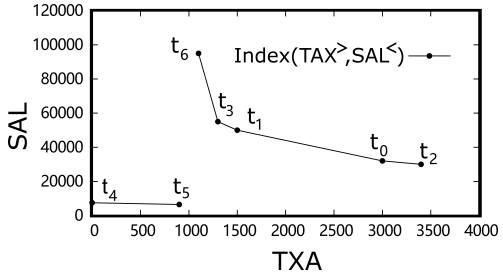



**Figure 3:** Index($\mathsf{TXA}^{>}, \mathsf{SAL}^{<}$)

denote the $n$-th element in $\mathsf{Sorted}_i$, starting from 0.

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

(2) For tuples $t'$, $t$ in each $\mathsf{Sorted}_i$, $A^{op_1}(t', t)$ 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 $\mathsf{Sorted}_1$ and $\mathsf{Sorted}_2$. In each $\mathsf{Sorted}_i$, $\mathsf{TXA}^{>}(t', t)$ and $\mathsf{SAL}^{<}(t', t)$ if $t'$ is after $t$. □

Intuitively, we aim to divide $D$ into several parts ($\mathsf{Sorted}_i$), and tuples in the same $\mathsf{Sorted}_i$ are sorted on both $A$ and $B$. The idea is enlightened by the conditional dependencies [11], which only hold on parts of the relation rather than the whole relation. Implementation details of the indexes will be studied in Section 5.3. 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, $T^{t_8}_{\mathsf{TXA}>\mathsf{SAL}<} = \{t_3, t_6\}$, $\overline{T}^{t_8}_{\mathsf{TXA}>\mathsf{SAL}<} = \{t_0, t_2\}$ 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 $\overline{T}^s_{A^{op_1}, B^{op_2}}$, 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 $\mathsf{Sorted}_i$ of $\mathsf{Index}(A^{op_1}, B^{op_2})$.

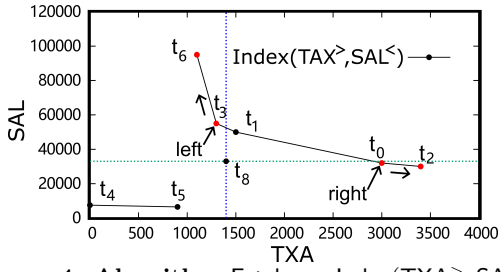



**Figure 4: Algorithm** Fetch **on** $\mathsf{Index}(\mathsf{TXA}^{>}, \mathsf{SAL}^{<})$

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

(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 $\mathsf{Sorted}_i$, 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 $\mathsf{Sorted}_i$, 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 $t$.

**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 4. 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 $\mathsf{Sorted}_1$ 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 $\mathsf{Sorted}_1$ and put $t_6$ into $T^{t_8}_{\mathsf{TXA}>\mathsf{SAL}<}$.

(2) On $\mathsf{Sorted}_2 = [t_4, t_5]$, $p = 1$ 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.

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 $\mathsf{Sorted}_i$, where $n_i$ is the number of tuples in $\mathsf{Sorted}_i$ and $|D| = \sum_{i \in [1,k]} n_i$, is the number of tuples in $D$. This is because each $\mathsf{Sorted}_i$ 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 $\mathsf{Sorted}_i$. This cost is linear in the result size, and is obviously necessary.

**Remark.** 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 5, we study how to choose marked attributes and build indexes for small ranks. In Section 7, we experimentally study ranks of indexes on various datasets.

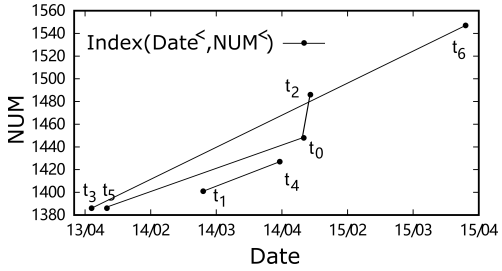



**Figure 5: Index $I'$ for Date$^<$ and NUM$^<$**

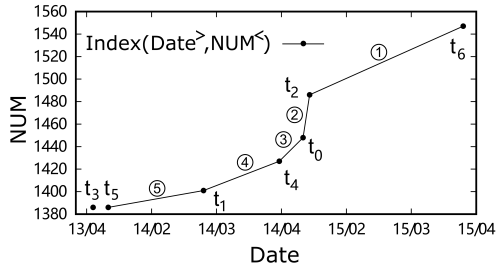



**Figure 6: Index $I$ for Date$^<$ and NUM$^<$ with OptIndex**

# 5. ALGORITHMS FOR INDEXES

In this section, we present a set of algorithms for index processing. Given inputs $D$, $\triangle D$ and $\Sigma$, we develop algorithms to (1) build an *optimal* index for two marked attributes with inequality operators (Section 5.1), to (2) choose indexes for all PODs in $\Sigma$ (Section 5.2), and to (3) update indexes in response to $\triangle D$ (Section 5.3).

## 5.1 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 Date$^<$ and NUM$^<$, besides index $I' = \{[t_6, t_3], [t_2, t_0, t_5], [t_4, t_1]\}$ (Figure 5), we have another index $I = \{[t_6, t_2, t_0, t_4, t_1, t_5], [t_3]\}$ (Figure 6). As illustrated in Section 4, they both satisfy the index specification, but we prefer $I$ to $I'$ since $I$ has a smaller rank than $I'$.

We say 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.

**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 (line 2), and then enumerates remaining tuples (lines 3-12). For each tuple $s$, if there exist 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:** Index is built on $D_1$ for Date$^<$, NUM$^<$ with OptIndex, shown in Figure 6. (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

**Algorithm 2:** OptIndex

```
input : relation D and marked attributes A^op1, B^op2
output: an optimal index for A^op1 and B^op2 on D
1  Sort tuples in D on A according to op1, and then on B
     according to op2 for breaking ties;
2  Index ← { [ D[0] ] };
3  for each tuple s ∈ D \ D[0] do
4      pos ← NULL; min ← −1;
5      for each Sorted_i ∈ Index do
6          t ← the last element of Sorted_i;
7          if B^op2(s, t) then
8              if min = −1 or min > |(s_B − t_B)| then
9                  min ← |(s_B − t_B)|;
10                 pos ← Sorted_i;
11     if pos! = NULL then append s to the end of pos;
12     ;
13     else add [s] into Index;
14     ;
```

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 $I = \{[t_6, t_2, t_0, t_4, t_1, t_5], [t_3]\}$. □

We then show the optimality property of OptIndex.

THEOREM 2. *For two marked attributes $A^{op_1}$ and $B^{op_2}$ ($op_1, op_2 \in \{<, \leq, >, \geq\}$), OptIndex creates an index with the minimum rank among all indexes for $A^{op_1}$, $B^{op_2}$.*

*Proof sketch:* We denote by $I_{opt}$ the output of OptIndex. Given any index $I'$ for $A^{op_1}$ and $B^{op_2}$, we show that $I'$ can be transformed into $I_{opt}$ in a finite number of steps, and after each step, (a) the index specification is still satisfied, and (b) the index rank *never* increases. Specifically,

(1) As a pre-processing, we determine an order $\mu$ for tuples in $D$, following the same way as line 1 of OptIndex.

(2) We treat tuples $s$ in $I'$ one by one following the order $\mu$, by considering $s$ in $I_{opt}$. Tuple $s$ remains unchanged in $I'$, if (a) $s$ is the first element of some Sorted$_i$ in $I_{opt}$, or (b) $s$ is right behind a tuple $t$ both in $I'$ and in $I_{opt}$.

Now suppose $s$ is right behind $t$ in $I_{opt}$, but not in $I'$. We should move $s$ such that it is also right behind $t$ in $I'$ (all tuples behind $s$ in the same Sorted$_j$ of $I'$ are also moved). In $I'$, we denote by $o$ the tuple right in front of $s$, and by $p$ the tuple right behind $t$.

case Figure 7a: If neither $o$ nor $p$ exists, then we move $s$ to be right behind $t$ in $I'$. This *reduces* the rank of $I'$ by 1.

case Figure 7b: If $p$ exists but $o$ does not, then $p$ becomes the first element of the new Sorted$_j$ in $I'$ after moving $s$. This does not change the rank of $I'$.

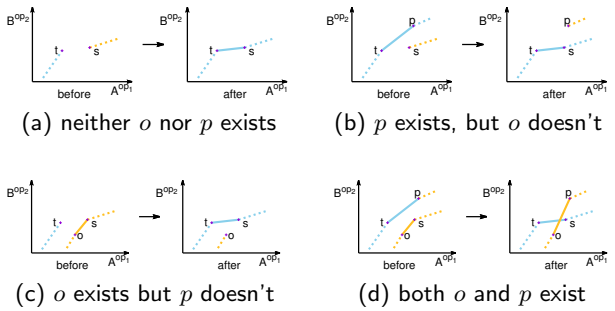



**Figure 7: Proof for Theorem 2**

case Figure 7c: If $o$ exists but $p$ does not, then $o$ becomes the last element of $\mathsf{Sorted}_j$ in $I'$ after we move $s$. This does not affect the rank of $I'$.

case Figure 7d: Now both $o$ and $p$ exist. We know that $A^{op_1}(p,s)$ since $s$ is right behind $t$ in $I_{opt}$, and that $A^{op_1}(s,o)$ in $I'$. Hence, we have $A^{op_1}(p,o)$. 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 $B^{op_2}(t,o)$. We also know $B^{op_2}(p,t)$ in $I'$, and hence have $B^{op_2}(p,o)$. Since we know $A^{op_1}(p,o)$ and $B^{op_2}(p,o)$, we can put $p$ right behind $o$, and simultaneously put $s$ right behind $t$. Again, this does not affect the rank of $I'$.

(3) It can be verified that we get index $I_{opt}$ after (2). Recall that the index rank never increases, in either of (a), (b), (c), or (d) stated above. Since $I'$ is an arbitrary index for $A^{op_1}$ and $B^{op_2}$, we conclude that $I_{opt}$ has the minimum rank among all indexes for $A^{op_1}$ and $B^{op_2}$. □

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

## 5.2 Choosing Indexes

In this subsection, we first show that indexes can be employed for efficiently identifying violating tuples incurred by $\triangle D$, and that some indexes can be used as alternatives to others. We then develop algorithms to choose indexes for $\Sigma$.

Given $\Sigma$ valid on $D$, we need to identify violations of $\Sigma$ incurred by $\triangle D$. Specifically, it is to find all tuple pairs $(s, t)$ such that $s, t$ violate some PODs in $\Sigma$; 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 5.3, 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 we know (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 $B^{\overline{op'}}(t,s)$. We do the following.

(1) Leveraging indexes on attributes of $\sigma$, we can identify a (small) set of candidates of $t \in D$ for $s \in \triangle D$. Specifically,

(a) if some $op_i$ is "=", then we can adopt a simple index that sorts $D$ on attribute $A_i$, denoted as $\mathsf{Index}(A_i^{=})$. $\mathsf{Index}(A_i^{=})$ helps efficiently find all tuples $t$ such that $A_i^{=}(t,s)$.

(b) If some $op_i, op_j{\in}\{<,\leq,>,\geq\}$, then $t \in T^s_{A_i^{op_i},A_j^{op_j}} \cup \overline{T}^s_{A_i^{op_i},A_j^{op_j}}$. Leveraging indexes for $A_i^{op_i}$, $A_j^{op_j}$, we can employ Algorithm Fetch to find such $t$ (Section 4). Note that $\mathsf{Index}(A_i^{op_i}, B^{\overline{op'}})$ can be used as well ($\overline{op'} \in \{<,\leq,>,\geq\}$, as summarized in Table 3); we need two indexes $\mathsf{Index}(A_i^{op_i}, B^{>})$ and $\mathsf{Index}(A_i^{op_i}, B^{<})$, if $op'$ is "=".

(2) On $s$ and each candidate in the set, we then check remaining conditions of $\sigma$ for the set of genuine $t$ such that $s$, $t$ violate $\sigma$ (recall Example 7).

**Alternative indexes.** We then show that some indexes can be used 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

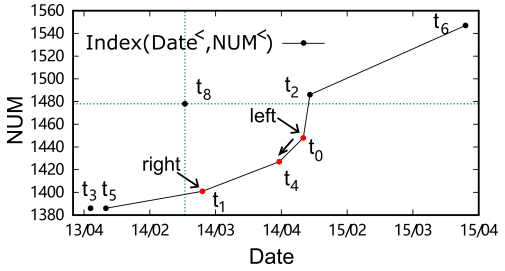



**Figure 8: leveraging an index reversely**

$\mathsf{Index}(A^{<}, B^{<})$ (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 8, 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 $\mathsf{Sorted}_1$, such that $\mathsf{Date}^{<}(\mathsf{Sorted}_1[p+1], t_8)$, $\mathsf{Date}^{\geq}(\mathsf{Sorted}_1[p], t_8)$, $\mathsf{NUM}^{<}(\mathsf{Sorted}_1[q+1], t_8)$, and $\mathsf{NUM}^{\geq}(\mathsf{Sorted}_1[q], t_8)$.

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 $\mathsf{Sorted}_1$ 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 $\Sigma$ by indexes.** It is costly to build an index for each POD if $\Sigma$ 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 $\sigma$ is *covered* in this case. According to our discussions before, (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 $\mathsf{Index}(A^{>}, A'^{<})$, in place of $\mathsf{Index}(A^{>}, A'^{>})$.

We then introduce score functions to show the 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 a small score.

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

**Score for inequality index.** We adopt the following scoring function for measuring $\mathsf{Index}(A^{op_1}, B^{op_2})$.

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

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 $D$, $r(A,B)$ is computed based on all tuples $t \in D$ as follows:

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

Note that $\mathsf{Index}(A^{>}, B^{>})$ (resp. $\mathsf{Index}(A^{>}, B^{<})$) has a rank of 1, if $r(A,B) = 1$ (resp. $-1$). We prefer indexes with small scores, *i.e.,* indexes that are on attributes with

**Algorithm 3:** ChooseIndex

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

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 $\Sigma$ 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 $Candidates$;
9 **while** $Candidates$ *is not empty* **do**
10  **if** $\Sigma$ *is empty* **then break** ;
11  ;
12  pick $({A_i}^{op_i}, {A_j}^{op_j})$ from $Candidates$ with the minimum $score({A_i}^{op_i}, {A_j}^{op_j})$;
13  **foreach** $\sigma = \mathcal{X} \hookrightarrow \mathcal{B}^{op'} \in \Sigma$ *such that* ${A_i}^{op_i}, {A_j}^{op_j}$ *cover* $\sigma$ **do** remove $\sigma$ from $\Sigma$;
14  ;
15  $op_j \leftarrow$ reverse $op_j$ if necessary according to $r(A_i, A_j)$ and a parameter $\alpha$;
16  add $\mathsf{Index}(A_i^{op_i}, A_j^{op_j})$ into $Ind(\Sigma)$;
17 **for** *each* $\sigma = \mathcal{X} \hookrightarrow \mathcal{B}^{op'} \in \Sigma$ **do**
18  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 $Ind(\Sigma)$;

high correlation coefficient and cover more PODs. We also find that if $A, B$ show a positive correlation, *i.e.,* $r(A, B)>0$, then $\mathsf{Index}(A^{>}, B^{>})$ and $\mathsf{Index}(A^{\geq}, B^{>})$ tend to have a small rank, while $\mathsf{Index}(A^{>}, B^{<})$ and $\mathsf{Index}(A^{\geq}, B^{<})$ are likely to have a small rank if $A, B$ show a negative correlation. We experimentally justify our score functions in Section 7.

**Example 12:** On relation $D_1$ in Table 1, instead of building $\mathsf{Index}(\mathsf{Date}^{<}, \mathsf{NUM}^{\geq})$ that covers $\sigma_4$=$\{\mathsf{SSN}^{=}, \mathsf{Date}^{<}\} \hookrightarrow \{\mathsf{NUM}^{<}\}$, we build $\mathsf{Index}(\mathsf{Date}^{<}, \mathsf{NUM}^{<})$ that also covers $\sigma_4$ and has a small rank, since $r(\mathsf{Date}, \mathsf{NUM})$=0.88 on $D_1$. 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. □

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

(1) It first employs indexes $\mathsf{Index}(A^{=})$ whose score values are below a predefined threshold $l$ (we set $l = 0.6$ in the implementation), to serve $\Sigma$ (lines 1-4).

(2) After collecting all possible marked attribute pairs in a set $Candidates$ (lines 5-8), it continues to pick $({A_i}^{op_i}, {A_j}^{op_j})$ with the minimum $score$ values from $Candidates$, for covering PODs and building indexes, until all pairs are used up or $\Sigma$ is empty (lines 9-14). This is similar to the heuristic approach for vertex cover [37]. We may adjust $op_j$ for $\mathsf{Index}(A_i^{op_i}, A_j^{op_j})$ of a small rank, according to $r(A_i, A_j)$ and a parameter $\alpha > 0$ (line 13). For 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 set $\alpha = 0.3$ in our implementation.

(3) If there are still uncovered PODs, then it can be seen that they must contain $A^{=}$ but $score(\mathsf{Index}(A^{=})) \geq l$. We heuristically build index on $A_i^{=}$ with the smallest scores for the PODs (lines 15-16).

**Complexity.** ChooseIndex takes $O(|R|^2 \cdot |\Sigma|)$ in the worst case, where $|\Sigma|$ is the number of PODs in $\Sigma$. Note that (1) the upper bounds for computing score values of $\mathsf{Index}(A^{=})$ and $\mathsf{Index}(A_i^{op_1}, A_j^{op_2})$ are $O(|R| \cdot |D| log(|D|))$ and $O(|R|^2 \cdot |D|)$ respectively. We use uniform random sampling to estimate score values in our implementation; and (2) only $coverage(A^{op_1}, B^{op_2})$ needs to be recomputed in line 11 as $\Sigma$ changes, with a cost independent of $|D|$.

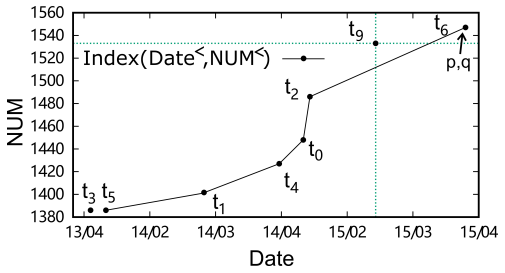



**Figure 9: Tuple $t_9$ and tuples in $D_1$**

**Remark.** Since $\Sigma$ on $D$ is known, building indexes with ChooseIndex and OptIndex is conducted as a *pre-processing* stage for the incremental POD discovery.

**Application of indexes.** For each $s \in \triangle D$, $Ind(\Sigma)$ is applied for finding violating tuples *w.r.t.* it. Specifically, indexes in $Ind(\Sigma)$ are treated one by one in the order they are added into $Ind(\Sigma)$. (1) By an index $I$, we first identify a (small) set $T$ of candidate violating tuples for $s$. (2) For each POD $\sigma$ covered by $I$ in ChooseIndex (line 4 or 12), we then check remaining conditions of $\sigma$ to find those genuine violating tuples $t$ from $T$ such that $s, t$ violate $\sigma$. Note that PODs covered by $I$ have different requests in this step.

## 5.3 Updating Indexes & Implementation

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

**Applying $\triangle D$ to indexes.** For each index $I$, we first sort tuples in $\triangle D$ in the same way as building $I$ (line 1 of Algorithm OptIndex). For each tuple $s \in \triangle D$, we then (1) run Algorithm Fetch on $I$ for $s$, and (2) 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 running Fetch for $s$. The additional cost of updating indexes is hence marginal.

**Example 13:** Consider $\mathsf{Sorted}_1$ of $\mathsf{Index}(\mathsf{Date}^{<}, \mathsf{NUM}^{<})$ in Example 10. We employ Algorithm Fetch on it to check candidate violations incurred by tuple $t_9$. To help understanding, Figure 9 shows $t_9$ and all tuples in $D_1$ of Table 1.

Just like Example 11, we first identify position $p = 0$, such that $Date^{<}(\mathsf{Sorted}_1[p+1], t_9)$ and $Date^{\geq}(\mathsf{Sorted}_1[p], t_9)$. Similarly, we have position $q = 0$ such that $NUM^{<}(\mathsf{Sorted}_1[q+1], t_9)$and $NUM^{\geq}(\mathsf{Sorted}_1[q], t_9)$. $t_9$ can be inserted into $\mathsf{Sorted}_1$ since we have $p = q$ on it. According to Example 11, we know no violation is found. □

**Remark.** Note the following. (1) A tuple $s \in \triangle D$ can be inserted into a $\mathsf{Sorted}_i$ if we find position $p = q$ on it. If there are multiple such $\mathsf{Sorted}_i$, then we heuristically pick the one with the maximum size in the implementation. If none exists, then we create a new $\mathsf{Sorted}_j$; this increases the index rank by 1. (2) Updating an index may violate the *optimality* of the index rank. That is, the updated index with $\triangle D$ may have a larger rank than the index on $D + \triangle D$ built with OptIndex. However, the difference between the ranks is small, as experimentally verified in Section 7.

(3) Each index is visited and updated once for each tuple in $\triangle D$. This further justifies our idea of sharing indexes among PODs, for a small number of indexes. Moreover, experimental evaluations (Section 7) show that our approach is hence insensitive to the size of $\Sigma$.

**Implementation.** We implement each $\mathsf{Sorted}_i$ of the indexes in a lightweight data structure, known as SkipList [26]. Recall that a SkipList is a multi-layered sorted list, where the bottom layer contains all the nodes, and nodes at layer $i$+1 serve as indexes for nodes at layer $i$, helping "skip" some nodes at layer $i$ in the search. As proved in [26], it takes $O(log(n))$ to search and update a SkipList, where $n$ is the number of nodes in the bottom layer.

# 6. INCREMENTAL DISCOVERY

In this section, we present our incremental POD discovery algorithm in response to $\triangle D$ of tuple insertions.

**Algorithm.** We develop Algorithm IncPOD for incrementally discovering PODs, with inputs $\Sigma$ on $D$, $\triangle D$, attribute set $R$, and the set $Ind(\Sigma)$ of indexes for $\Sigma$ on $D$. It finds the complete set $\Sigma'$ 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 minimal valid PODs as additions to $\Sigma$, while $\triangle\Sigma^-$ contains PODs that are to be removed from $\Sigma$.

(1) It first finds the set $T$ of violating tuple pairs, leveraging $Ind(\Sigma)$ (line 1). This step consists of index visits with Fetch (Section 4) that help identify violating tuple pairs (Section 5.2) and index updates with $\triangle D$ (Section 5.3).

(2) $\Sigma'$ is initialized as $\Sigma$ (line 2). For each violating tuple pair ($t$, $s$), IncPOD enumerates all $\sigma \in \Sigma'$ (lines 3-8). If ($t$, $s$) violates $\sigma$, then Algorithm ExtendPOD is called to evolve a set $\Psi$ of new PODs based on $\sigma$ for resolving the violation incurred by ($t$, $s$) (line 7). $\Sigma'$ is then updated by removing $\sigma$ and adding $\Psi$ (line 8). That is, $\sigma$ is put into $\triangle\Sigma^-$, while $\Psi$ contains candidates for $\triangle\Sigma^+$. After processing tuple pair ($t$, $s$), IncPOD moves to the next tuple pair with the updated $\Sigma'$. As will be seen shortly, new PODs added into $\Sigma'$ 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 $\Sigma'$ will never introduce new violations *w.r.t.* ($t$, $s$). Note that in line 6 it suffices to only check $\sigma$ that is newly introduced to $\Sigma'$ (line 8). This is because all tuple pairs violating $\sigma \in \Sigma$ are already known (line 1). Moreover, we only check $\sigma$ with ($t$, $s$) if $\sigma$ is evolved from $\sigma' \in \Sigma$ and ($t$, $s$) violates $\sigma'$. As will be seen shortly, ExtendPOD guarantees that ($t$, $s$) *cannot* violate $\sigma$ if it does not violate $\sigma'$.

(3) Function Minimize is to remove non-minimal PODs from $\Sigma' \setminus \Sigma$ (line 9), *i.e.,* the set of new valid PODs. PODs in $\Sigma$ are still minimal if they are valid on $D + \triangle D$. To check the minimality of $\sigma = \mathcal{X} \hookrightarrow B^{op'}$ by definition (Section 3), it suffices to only consider PODs $\mathcal{X}' \hookrightarrow B^{op''}$ in $\Sigma'$, where $|\mathcal{X}'| \leq |\mathcal{X}|$ and $op' \in im(op'')$. $\Sigma'$ is then the complete set of minimal and valid PODs on $D + \triangle D$, and $\triangle\Sigma^+$ and $\triangle\Sigma^-$ are the differences between $\Sigma$ and $\Sigma'$ (line 10).

**Algorithm.** ExtendPOD is employed to generate new valid PODs based on invalid PODs in $\triangle\Sigma^-$. ExtendPOD enumerates all possible ways to resolve the violation, for new PODs in $\Psi$. Specifically, ExtendPOD may (1) strengthen the LHS conditions when handling $\{\leq, \geq\}$ (lines 2-5), or (2) relax the

**Algorithm 4:** IncPOD

**Input:** the complete set $\Sigma$ 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 $\Sigma$
**Output:** the complete set $\Sigma'$ of minimal valid PODs on $D + \triangle D$. $\Sigma' = \Sigma \oplus \triangle\Sigma$, $\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$

1 find the set $T$ of violating tuple pairs *w.r.t.* $\Sigma$ 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* $\sigma$ **then**
7  $\Psi \leftarrow$ ExtendPOD$(\sigma, (t, s), R)$;
8  $\Sigma' \leftarrow \Sigma' \backslash \sigma \cup \Psi$;
9 $\Sigma' \leftarrow$ Minimize $(\Sigma', \Sigma' \backslash \Sigma)$;
10 $\triangle\Sigma^+ \leftarrow \Sigma' \setminus \Sigma$; $\triangle\Sigma^- \leftarrow \Sigma \setminus \Sigma'$;

**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 $\Psi$ 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 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  $\Psi \leftarrow \Psi \cup \{\mathcal{X} \cup \{A^{op}\} \hookrightarrow B^{op'}\}$

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 3). By inference rules of PODs [15], it can be seen that $\sigma$ logically implies any $\sigma' \in \Psi$: any relation that satisfies $\sigma$ must satisfy $\sigma'$. Hence, it never introduces new violations when replacing $\sigma$ by $\Psi$ in $\Sigma'$ (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.,* $\{\mathsf{ST}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ or $\{\mathsf{PH}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$, or modify its RHS operator (lines 6-8), *e.g.,* $\{\mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{\leq}\}$.

Consider $\{\mathsf{PH}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ and another violating pair $(t_7, t_1)$, where $PH^{=}(t_7, t_1)$, $TXA^{>}(t_7, t_1)$, $SAL^{<}(t_7, t_1)$, $RATE^{\geq}(t_7, t_1)$. We can see that $(t_7, t_1)$ also violates $\{\mathsf{PH}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ 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.,* $\{\mathsf{ST}^{=}, \mathsf{PH}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$. However, since $\{\mathsf{ST}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ is valid, $\{\mathsf{ST}^{=}, \mathsf{PH}^{=}, \mathsf{TXA}^{>}, \mathsf{SAL}^{<}\} \hookrightarrow \{\mathsf{RATE}^{<}\}$ is not minimal and will be removed by Minimize (line 9 of IncPOD). □

THEOREM 3. IncPOD *finds the complete set* $\Sigma'$ *of minimal valid PODs on* $D + \triangle D$.

PROOF. (1) *Validity.* If a tuple pair $(t, s)$ incurs a violation against $\sigma$, then ExtendPOD is called to generate a set

$\Psi$ of new PODs based on $\sigma$. ExtendPOD never introduces new violations beyond $T$, the set of violating tuple pairs detected on $D + \triangle D$ for $\Sigma$. Specifically, it can be seen that if any tuple pair $(t', s')$ violates a POD in $\Psi$, then $(t', s')$ also violates $\sigma$ and is hence in $T$. Therefore, when no violation is detected on $T$, all PODs in $\Sigma'$ 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 $\sigma$ is valid on $D + \triangle D$ but is not in $\Sigma'$. We show $\sigma$ is not minimal on $D + \triangle D$. (a) If $\sigma$ is minimal on $D$, then $\sigma$ is in $\Sigma$ since $\sigma$ is valid on $D$ and $\Sigma$ is the complete set of minimal valid PODs on $D$. This contradicts the assumption that $\sigma$ is not in $\Sigma' = \Sigma \oplus \triangle\Sigma$, since $\sigma$ is obviously not in $\Sigma^-$. (b) If $\sigma$ is not minimal on $D$ then there exists $\sigma'$ that is valid on $D$ and logically implies $\sigma$. Obviously $\sigma$ is not minimal if $\sigma'$ is still valid on $D + \triangle D$. Now suppose $\sigma'$ is invalid on $D + \triangle D$. Based on $\sigma'$, ExtendPOD enumerates all possible ways for generating PODs valid on $D + \triangle D$. If $\sigma$ is not in $\Sigma'$, then there must be some POD in $\Sigma'$ that logically implies $\sigma$, and hence makes $\sigma$ not-minimal on $D + \triangle D$. □

**Complexity.** IncPOD is far more efficient than the batch counterpart in practice, since new PODs are discovered by refining PODs in $\Sigma$ that become invalid on $D + \triangle D$. Specifically, (1) the set $T$ of violating tuple pairs 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.

**Remark.** If $\sigma \in \Sigma$ becomes invalid on $D + \triangle D$, then $\sigma$ is replaced by a set $\Psi$ of more "specified" PODs in $\Sigma \oplus \triangle\Sigma$. PODs in $\Psi$ are more "specified": if a tuple pair $(t,s)$ violates any POD in $\Psi$, then it must violate $\sigma$. Hence, the index that serves $\sigma$ also serves all PODs in $\Psi$, for identifying possible violations. This feature is desirable: indexes are built in an *off-line* process with a *one-time* cost and not required to be rebuilt for a series of sets of tuple insertions $\triangle D_1, \ldots, \triangle D_k$.

# 7. EXPERIMENTAL STUDY

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.

## 7.1 Experimental Settings

*Datasets.* We use a host of real-life and synthetic datasets that are employed in experimental studies on constraint discoveries [5, 24, 29, 32, 33] (see Section 2). (1) Real-life data. SPS contains 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). CLAIM contains the airport baggage claims data (https://www.dhs.gov/tsa-claims-data). (2) Synthetic data. FDR15 and FDR30 are two synthetic datasets (http://metanome.de).

We summarize details of all datasets in Table 4, denote the number of attributes (resp. tuples) by $|R|$ (resp. $|D|$).

*Implementation.* We implement all the algorithms in Java. (1) ChooseIndex and OptIndex (Section 5), for choosing and building indexes, as a pre-processing stage for IncPOD.

**Table 4: DataSets and execution statistics**

| Data | $\|R\|$ | $\|D\|$ | $\|\triangle D\|$ | Inc | Hydra* | Finder* | $\|Ind(\Sigma)\|$ | $\|\Sigma\|$ | $\|\triangle\Sigma\|$ |
|---|---|---|---|---|---|---|---|---|---|
| SPS | 7 | 90K | 27K | **0.9s** | 8.57s | 104.5s | 5 | 20 | 10 |
| FLI | 17 | 300K | 60K | **67s** | 461s | 1927s | 25 | 1,481 | 132 |
| STR | 5 | 450K | 125K | **9s** | 109s | 2376s | 11 | 21 | 0 |
| LETTER | 12 | 15K | 4.5K | **175s** | 1538s | 4912s | 23 | 8,172 | 719 |
| NCV | 18 | 300K | 100K | **76s** | 7601s | 3188s | 13 | 1,134 | 36 |
| CLAIM | 11 | 97K | 50K | **1.54s** | 17.1s | 49.8s | 4 | 12 | 29 |
| FDR15 | 15 | 200K | 50K | **14s** | 65s | 1521s | 13 | 185 | 40 |
| FDR30 | 30 | 200K | 50K | **48s** | 265s | 1791s | 31 | 1,152 | 701 |

(2) IncPOD (Section 6), the incremental POD discovery algorithm that combines our techniques together.
(3) Hydra* and Finder*, algorithms for batch POD discovery. We implement Hydra* (resp. Finder*) by adapting [5] (resp. [25]), state-of-the-art DC discovery techniques. This is possible since each POD can 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 obtained source code (available online www.metanome.de) such that only predicates of the form $t_A$ *op* $s_A$ are considered, where $op \in \{<, \leq, =, >, \geq\}$. In addition, we allow at most one predicate of the form $t_B \neq s_B$ in each DC, for encoding PODs of the form $\{A_1^{op_1}, \ldots, A_m^{op_m}\} \hookrightarrow B^=$.
(4) IEJoin [22], 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.* $\{A^>, B^>\} \hookrightarrow \{C^>\}$ 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: (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 $\frac{|\triangle D|}{|D|}$. The average of 5 runs is reported here.

*Measurement.* We first compute $\Sigma$ on $D$ with the batch methods (Hydra* and Finder*), and then build indexes for $\Sigma$. Leveraging the indexes, we incrementally find the complete set of minimal valid PODs on $D + \triangle D$ with IncPOD. The correctness of IncPOD is verified by checking its result against that of the batch method on $D + \triangle D$. Note that the complete set of minimal valid PODs on a dataset is deterministic. 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 the batch method is for computing all PODs on $D + \triangle D$ from scratch.

## 7.2 Experimental Findings

**Exp-1: IncPOD against Batch methods.** We report running times (in seconds) of incremental and batch methods in Table 4. We see IncPOD performs far better on all datasets, up to two orders of magnitude. We also show the number of PODs in $\Sigma$ and $\triangle\Sigma$, denoted by $|\Sigma|$ and $|\triangle\Sigma|$, respectively. We find even when $\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$ is large, *e.g.,* on LETTER and FDR30, IncPOD still performs much better.

We conduct more experiments by varying parameters.

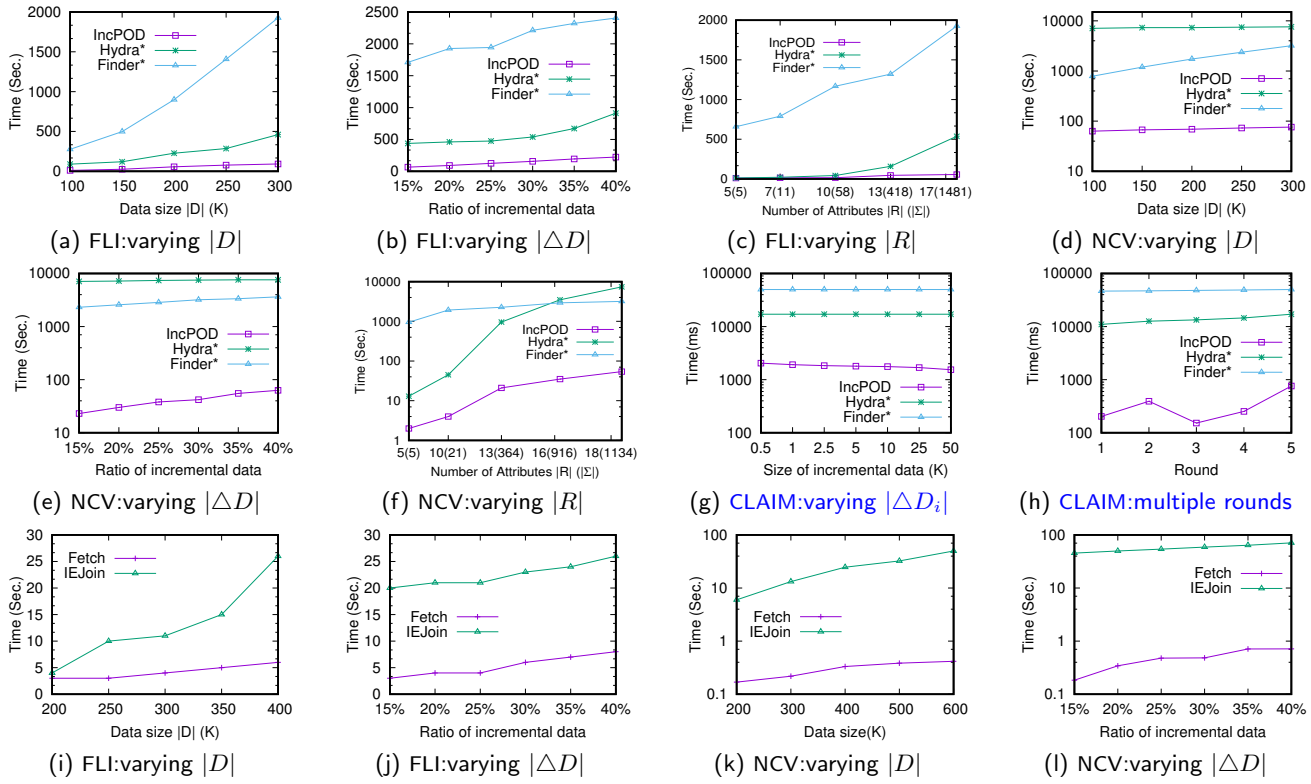



**Figure 10:** IncPOD **against Batch methods, and** Fetch **against** IEJoin

(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 10a shows results by varying $|D|$ from 100K to 300K ($|\triangle D|$ from 20K to 60K). Finder* is more sensitive to $|D|$ than Hydra*, consistent with the results in [25]. IncPOD performs much better. It takes only 67 seconds for IncPOD but 461 seconds for Hydra*, when $|D|$= 300K.

Varying $|\triangle D|$. Figure 10b shows results by varying $|\triangle D|$ from 45K to 120K (the ratio of $|\triangle D|$ to $|D|$ increases from 15% to 40%). IncPOD consistently outperforms batch methods, 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 10c. $|R|$ significantly affects the efficiency, since $|\Sigma|$ grows exponentially with $|R|$. For reference, we show $|\Sigma|$ (in the bracket) together with $|R|$ on the x-axis. Hydra* does not scale well with $|R|$, consistent with the results in [25]. IncPOD scales much better: as $|R|$ increases from 5 to 17, the time of Hydra* 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 $\Sigma$ ($Ind(\Sigma)$ of ChooseIndex in Section 5.2). We show the results in Table 4, denoted as $|Ind(\Sigma)|$. It can be seen that $|Ind(\Sigma)|$ is typically much smaller than $|\Sigma|$; this explains why IncPOD is less sensitive to $|R|$. 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 10d, $\frac{|\triangle D|}{|D|}$ from 15% to 40% in Figure 10e, and $|R|$ from 5 to 18 in Figure 10f. Note that we use log scale in these experiments. We find numbers of distinct values on most attributes of NCV are large. It hence takes both batch methods 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.

(3) We conduct experiments on CLAIM with $|\triangle D| = 50K$. Tuple insertions are processed in the order of their timestamps, so as to follow the real change history. We handle $\triangle D$ as a series of sets of tuple insertions ($\triangle D = \triangle D_1 \cup \cdots \cup \triangle D_k$), where $\triangle D_i$ ($i \in [1, k]$) is of the same size. Recall that indexes are not required to be rebuilt in the whole process.

In Figure 10g, we vary $|\triangle D_i|$ from 0.5K to 50K, and report the times for the whole change history. For example, a single set of tuple insertions is applied to $D$ if $|\triangle D_i| = 50K$, while 100 sets of tuple insertions are applied one by one if $|\triangle D_i| = 0.5K$. We see IncPOD always significantly outperforms batch methods. The times for batch methods are computed on $D + \triangle D$ and hence not affected by $|\triangle D_i|$. The time for IncPOD decreases by nearly 25% as $|\triangle D_i|$ increases from 0.5K to 50K, as expected. Although incremental violation detections are not affected by varying $|\triangle D_i|$, the costs of computing $\triangle\Sigma$ decrease for less rounds of requests.

We fix $|\triangle D_i| = 10K$ in Figure 10h, and show the times for applying $\triangle D_1$ to $D$, $\triangle D_2$ to $D + \triangle D_1$, etc. The times for batch methods monotonically increase as data sizes grow, but in IncPOD times for $\triangle D_i$ may vary since different $\triangle D_i$ causes different violations and changes in the POD set.

**Exp-2:** Fetch **against** IEJoin**.** We experimentally verify the benefit of our indexing technique against IEJoin [22]. We still report the average over 5 runs; each time we ran-

**Table 5: Indexes for DataSets**

| Data | $\|R\|$ | $\|D\|$ | $\|Ind(\Sigma)\|$ | Data Memory | Index Memory | Time |
|---|---|---|---|---|---|---|
| SPS | 7 | 90K | 5 | 28MB | 42MB | 1.44s |
| FLI | 17 | 300K | 25 | 198MB | 501MB | 68s |
| STR | 5 | 450K | 11 | 124MB | 650MB | 21s |
| LETTER | 12 | 15K | 23 | 2MB | 8MB | 2.68s |
| NCV | 18 | 300K | 13 | 355MB | 635MB | 18s |
| CLAIM | 11 | 97K | 12 | 6MB | 26MB | 1.4s |
| FDR15 | 15 | 200K | 13 | 167MB | 263MB | 9.6s |
| FDR30 | 30 | 200K | 31 | 345MB | 580MB | 22s |

domly 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 4).

(1) We set $|D| = 300K$, $\frac{|\triangle D|}{|D|} = 20\%$ by default on FLI. Figure 10i shows results by varying $|D|$ from 200K to 400K. Fetch scales better than IEJoin: the time for Fetch (resp. IEJoin) increases from 3.3 (resp. 4) seconds to 6.4 (resp. 27) seconds. We find the time increase of Fetch is mainly due to more time required for collecting results. We vary $\frac{|\triangle D|}{|D|}$ from 15% to 40% in Figure 10j. Fetch scales well with $|\triangle D|$: the time increases from 3.4 seconds to 8 seconds, which is almost linear in $|\triangle D|$. IEJoin does not enjoy this feature. It has a cost almost linear in $|D + \triangle D|$, consistent with [22].

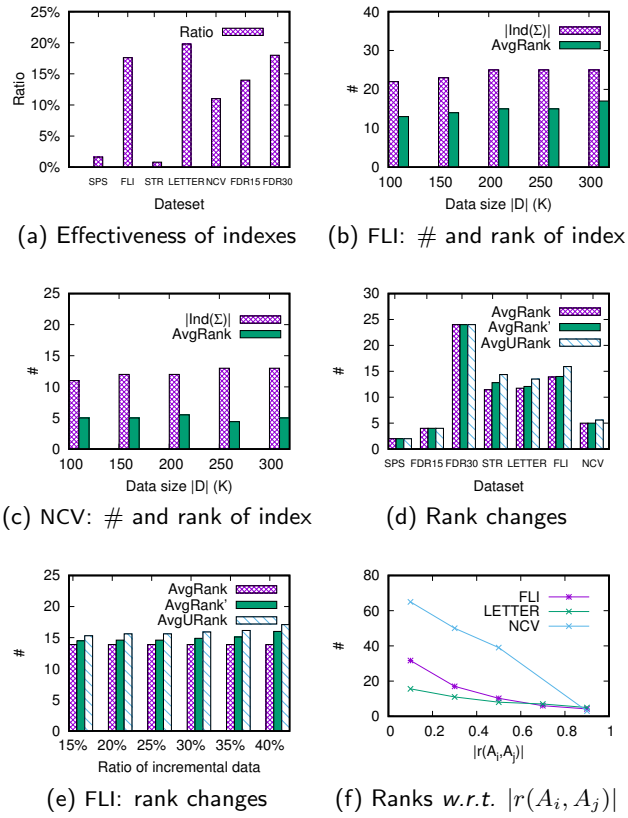
(2) We further compare Fetch against IEJoin using NCV with $|D| = 600$K and $\frac{|\triangle D|}{|D|} = 20\%$ by default. We vary $|D|$ from 200K to 600K in Figure 10k, and $\frac{|\triangle D|}{|D|}$ from 15% to 40% in Figure 10l. The result size on NCV is much smaller than that on FLI, and hence both Fetch and IEJoin take less time in collecting results. This 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 properties of our indexes in detail.

(1) The costs (memory footprint and building time) of indexes are shown in Table 5. For reference, we also show the memory usage and the number $|Ind(\Sigma)|$ of indexes for each dataset. Note that the costs of indexes are also affected by data types and numbers of distinct values of the indexing attributes. Recall that building indexes is a pre-processing step, and its time is not included in that of IncPOD.

(2) Indexes are employed to find candidate violating tuples for $\triangle D$, on which remaining conditions of PODs are checked. In Figure 11a we compute the ratio of the size of candidates to $|D|$ on datasets of Table 4. We see the ratios are always below 20%; indexes help skip more than 80% of the tuples. The combinations of two marked attributes lead to good selectivity on the tested datasets.

(3) The efficiency of IncPOD is affected by $|Ind(\Sigma)|$ and index ranks (Section 4). We denote by $AvgRank$ the average rank of indexes in $Ind(\Sigma)$, excluding equality indexes. We set $|R| = 17$, vary $|D|$ from 100K to 300K on FLI, and report results in Figure 11b. We see that $|Ind(\Sigma)|$ only increases from 22 to 25, and that $AvgRank$ also varies in the range of [13, 17]. In Figure 11c, 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 $|D|$.




(a) Effectiveness of indexes (b) FLI: # and rank of index

(c) NCV: # and rank of index (d) Rank changes

(e) FLI: rank changes (f) Ranks *w.r.t.* $|r(A_i, A_j)|$

**Figure 11: Effectiveness of indexes, index numbers and ranks, and score functions**

(4) Updating an index may increase its rank (Section 5.3). 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 guaranteed minimum ranks, and denote the average rank by $AvgRank'$. Figure 11d shows results for the datasets in Table 4. We see $AvgURank$ always increases slightly, at most 115% of $AvgRank$ on $D$. The difference between $AvgURank$ and the optimum $AvgRank'$ is also small; $AvgURank$ is at most larger than $AvgRank'$ by 12%. The results show 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 11e. $AvgURank$ increases slightly from 15.5 to 17.1, and is consistently within [105%, 107%] 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 functions.** We experimentally study the score functions employed in ChooseIndex (Section 5.2).

(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 $[0, 0.2], \ldots, [0.8, 1]$, and compute the average rank of indexes in each range. Figure 11f clearly shows that indexes on attribute pairs with a large $|r(A_i, A_j)|$ have small ranks. This justifies our score functions.

# 8. CONCLUSIONS

This is a first effort towards incremental POD discovery. 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 currently studying methods to handle updates with both tuple insertions and deletions; the need for this is evident in practice. We are also to extend our indexing techniques for DCs. Different from PODs, DCs allow comparisons across attributes and the inequality operator "$\neq$". Although it is relatively easy to support comparisons across attributes of a same tuple, and the inequality operator by taking it as the union of ">" and "<", more efforts are required to cope with comparisons across attributes of different tuples, *e.g.,* $t_A < s_B$, and still guarantee a cost independent of $|D|$.

# 9. REFERENCES


[1] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: a survey. *VLDB J.*, 24(4):557–581, 2015.

[2] Z. Abedjan, L. Golab, and F. Naumann. Data profiling: A tutorial. In *SIGMOD 2017*, pages 1747–1751, 2017.

[3] Z. Abedjan, J. Quiané-Ruiz, and F. Naumann. Detecting unique column combinations on dynamic data. In *ICDE*, pages 1036–1047, 2014.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] T. Bleifuß, S. Kruse, and F. Naumann. Efficient denial constraint discovery with hydra. *PVLDB*, 11(3):311–323, 2017.

[6] Q. Cheng, J. Gryz, F. Koo, T. Y. C. Leung, L. Liu, X. Qian, and K. B. Schiefer. Implementation of two semantic query optimization techniques in DB2 universal database. In *VLDB 1999*, pages 687–698, 1999.

[7] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.

[8] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, pages 458–469, 2013.

[9] C. Consonni, P. Sottovia, A. Montresor, and Y. Velegrakis. Discovering order dependencies through order compatibility. In *EDBT*, pages 409–420, 2019.

[10] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[11] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2):6:1–6:48, 2008.

[12] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE Trans. Knowl. Data Eng.*, 23(5):683–698, 2011.

[13] W. Fan, C. Hu, X. Liu, and P. Lu. Discovering graph functional dependencies. In *SIGMOD 2018*, pages 427–439, 2018.

[14] C. Ge, I. F. Ilyas, and F. Kerschbaum. Secure multi-party functional dependency discovery. *PVLDB*, 13(2):184–196, 2019.

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

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

[17] L. Golab, H. J. Karloff, F. Korn, A. Saha, and D. Srivastava. Sequential dependencies. *PVLDB*, 2(1):574–585, 2009.

[18] L. Golab, H. J. Karloff, F. Korn, D. Srivastava, and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. *PVLDB*, 1(1):376–390, 2008.

[19] A. Heise, J. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. *PVLDB*, 7(4):301–312, 2013.

[20] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2015.

[21] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and P. Kalnis. Lightning fast and space efficient inequality joins. *PVLDB*, 8(13):2074–2085, 2015.

[22] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J. Quiané-Ruiz, N. Tang, and P. Kalnis. Fast and scalable inequality joins. *VLDB J.*, 26(1):125–150, 2017.

[23] S. Kruse and F. Naumann. Efficient discovery of approximate dependencies. *PVLDB*, 11(7):759–772, 2018.

[24] P. Langer and F. Naumann. Efficient order dependency detection. *VLDB J.*, 25(2):223–241, 2016.

[25] E. H. M. Pena, E. C. de Almeida, and F. Naumann. Discovery of approximate (and exact) denial constraints. *PVLDB*, 13(3):266–278, 2019.

[26] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

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

[28] H. Saxena, L. Golab, and I. F. Ilyas. Distributed implementations of dependency discovery algorithms. *PVLDB*, 12(11):1624–1636, 2019.

[29] P. Schirmer, T. Papenbrock, S. Kruse, F. Naumann, D. Hempfing, T. Mayer, and D. Neuschäfer-Rube. Dynfd: Functional dependency discovery in dynamic datasets. In *EDBT*, pages 253–264, 2019.

[30] D. E. Simmen, E. J. Shekita, and T. Malkemus. Fundamental techniques for order optimization. In *SIGMOD 1996*, pages 57–67, 1996.

[31] S. Song and L. Chen. Differential dependencies: Reasoning and discovery. *ACM Trans. Database Syst.*, 36(3):16:1–16:41, 2011.

[32] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. Effective and complete discovery of order dependencies via set-based axiomatization. *PVLDB*, 10(7):721–732, 2017.

[33] J. Szlichta, P. Godfrey, L. Golab, M. Kargar, and D. Srivastava. Effective and complete discovery of bidirectional order dependencies via set-based axioms. *VLDB J.*, 27(4):573–591, 2018.

[34] J. Szlichta, P. Godfrey, and J. Gryz. Fundamentals of order dependencies. *PVLDB*, 5(11):1220–1231, 2012.

[35] J. Szlichta, P. Godfrey, J. Gryz, W. Ma, W. Qiu, and C. Zuzarte. Business-intelligence queries with order dependencies in DB2. In *EDBT 2014*, pages 750–761, 2014.

[36] J. Szlichta, P. Godfrey, J. Gryz, and C. Zuzarte. Expressiveness and complexity of order dependencies. *PVLDB*, 6(14):1858–1869, 2013.

[37] V. V. Vazirani. *Approximation algorithms*. Springer, 2001.

[38] Y. Wang, S. Song, L. Chen, J. X. Yu, and H. Cheng. Discovering conditional matching rules. *TKDD*, 11(4):46:1–46:38, 2017.

[39] L. Zhu, X. Sun, Z. Tan, K. Yang, W. Yang, X. Zhou, and Y. Tian. Incremental discovery of order dependencies on tuple insertions. In *DASFAA*, pages 157–174, 2019.