The VLDB Journal (2023) 32:1289–1313
https://doi.org/10.1007/s00778-023-00788-y

REGULAR PAPER

Incremental discovery of denial constraints

Chaoqin Qian[1] · Menglu Li[1] · Zijing Tan[1] · Ai Ran[1] · Shuai Ma[2]

Received: 24 December 2021 / Revised: 25 December 2022 / Accepted: 17 February 2023 / Published online: 17 March 2023

Abstract
We investigate the problem of incremental denial constraint (DC) discovery, aiming at discovering DCs in response to a set $\triangle r$ of tuple insertions to a given relational instance r and the known set Σ of DCs holding on r. The need for the study is evident since real-life data are often frequently updated, and it is often prohibitively expensive to perform DC discovery from scratch for every update. We tackle this problem with two steps. We first employ indexing techniques to efficiently identify the *incremental evidences* caused by $\triangle r$. We present algorithms to build indexes for Σ and r in the pre-processing step, and to visit and update indexes in response to $\triangle$r. In particular, we propose a novel indexing technique for two inequality comparisons possibly across the attributes of r. By leveraging the indexes, we can identify all the tuple pairs incurred by $\triangle$r that simultaneously satisfy the two comparisons, with a cost dependent on log(|r|). We then compute the changes $\triangle\Sigma$ to Σ based on the incremental evidences, such that $\Sigma \oplus \triangle\Sigma$ is the set of DCs holding on $r + \triangle r$. $\triangle\Sigma$ may contain new DCs that are added into Σ and obsolete DCs that are removed from Σ. Our experimental evaluations show that our incremental approach is faster than the two state-of-the-art batch DC discovery approaches that compute from scratch on $r + \triangle r$ by orders of magnitude, even when $\triangle r$ is up to 30% of r.

Keywords Data profiling · Denial constraint · Dependency discovery

1 Introduction

Data dependencies, *a.k.a.* integrity constraints, are valuable tools in many data management tasks, such as schema design, query optimization [11,33,54,61] and data quality management [14,16,26]. Denial constraints (DCs) [13] are expressive enough to subsume many other dependencies as special cases, *e.g.,* unique column combinations (UCCs), functional dependencies (FDs), and pointwise order dependencies (PODs) [23,24]. DCs also generalize the recently proposed lexicographical order dependencies (LODs) [60,62].

Briefly, a DC defines a set of predicates that *cannot* hold simultaneously, and a relational instance satisfies the DC if at least one predicate is violated by a combination of attribute values from the instance. The formal definition of DCs will be reviewed in Sect. 3. Here, we first start with an illustrative example.

Example 1 Consider the relational instance r_1 in Table 1, which is adapted from the example of [12]. Each tuple denotes a person with the following attributes: Tuple ID (TID), name (Name), tax date (Date), social security number (SSN), tax serial number (NUM), marital status (MS), has children (CH), phone number (PH), state (ST), zip code (ZIP), salary (SAL), tax rate (RATE), tax exemption amount (TXA), tax exemption amount if single (STX) and having children (CTX).

The following constraints hold on r_1.

(1) φ_1: The SSN of a person determines the name.
(2) φ_2: Persons that have the same zip code are in the same state.

✉ Zijing Tan
zjtan@fudan.edu.cn

Chaoqin Qian
cqqian20@fudan.edu.cn

Menglu Li
liml20@fudan.edu.cn

Ai Ran
aran17@fudan.edu.cn

Shuai Ma
mashuai@buaa.edu.cn

1 School of Computer Science, Fudan University, Shanghai, China

2 SKLSDE Lab, Beihang University, Beijing, China

Table 1 Tax data: r_1

TID	Name	Date	SSN	NUM	MS	CH	PH	ST	ZIP	SAL	RATE	TXA	STX	CTX
t_0	Sadam	20140410	719975883	1448	S	Y	6059466	CO	80612	32000	1.24	3000	400	2000
t_1	Duparc	20140224	303975883	1401	M	N	5872027	CO	80612	50000	1.42	1500	0	0
t_2	Hannen	20140413	701178073	1486	M	N	1638673	ND	58671	30000	1.21	3400	0	0
t_3	Beke	20130403	801350874	1386	M	Y	6192334	UT	84308	55000	2.04	1300	0	40
t_4	Sadam	20140329	719975883	1427	S	N	6059466	CO	80612	7500	1.21	0	50	0
t_5	Hannen	20130404	701178073	1386	S	Y	1638673	ND	58671	6500	0.24	900	0	1000
t_6	Motwani	20150324	970122634	1547	M	N	8484643	CO	80209	95000	2.85	1100	0	0

(3) φ_3: For any two persons, the person with a lower salary and a higher tax exemption amount has a lower tax rate than the other one.
(4) φ_4: The tax serial number increases with the tax date for a person.
(5) φ_5: There cannot exist two persons such that they have the same STX but different MS, and the STX of a person is larger than the CTX of the other one. This DC [12] states that a person must be single (MS = "S") if his/her STX > 0.

We can see φ_1, φ_2 are FDs, φ_3, φ_4 are PODs, and φ_5 concerns inequality comparisons across (on two different) attributes. All these constraints can be expressed by DCs as follows, where t and s denote two distinct tuples, respectively.

(1) φ_1: $\neg\,(t.SSN = s.SSN \wedge t.Name \neq s.Name)$
(2) φ_2: $\neg\,(t.ZIP = s.ZIP \wedge t.ST \neq s.ST)$
(3) φ_3: $\neg\,(t.TXA > s.TXA \wedge t.SAL < s.SAL \wedge t.RATE > s.RATE)$
(4) φ_4: $\neg\,(t.SSN = s.SSN \wedge t.Date < s.Date \wedge t.NUM \geq s.NUM)$
(5) φ_5: $\neg\,(t.STX = s.STX \wedge t.STX > s.CTX \wedge t.MS \neq s.MS)$

To fully take advantage of the benefits of DCs, the set Σ of valid DCs is expected to be known in advance. However, it is typically expensive to design dependencies manually, which requires domain experts and is time-consuming and subject to human errors [1–3,57]. This motivates the studies on automatic DC discovery techniques [6,12,37,42,43], for finding the set Σ of valid DCs on a given instance r. The solutions are often very costly, partially due to the high expressiveness of DCs.

Data in practice keep changing, which means a set $\triangle r$ of updates is applied to r. This motivates us to study the incremental DC discovery problem in response to the updates rather than start from scratch every time. In this study, we make a first effort for the case of tuple insertions. The discovered valid dependencies can be employed to rewrite user queries in query optimization [11,33,54,61], and in such scenarios, we must update dependencies after tuple insertions, but we may tolerate deletions and still use former dependencies, since deletions never change valid dependencies to invalid ones. In addition, the costs of DC discovery methods [6,12,37,42,43,69] often increase as the size of the instance increases, which makes the computation from scratch more expensive for tuple insertions. In short, we find the necessity to handle tuple insertions first for incremental DC discovery.

For discovering the set of DCs on $r + \triangle r$, a naive way is to recompute all DCs from scratch on $r + \triangle r$, which is obviously computationally expensive. Intuitively, when $\triangle r$ is small compared with r, a better approach is to compute the changes $\triangle\Sigma$ to Σ such that $\Sigma \oplus \triangle\Sigma$ is the set of DCs that hold on $r + \triangle r$. We use the notation "$\oplus$" since some DCs in Σ no longer hold and are removed from Σ, while some new valid DCs may be discovered and added into Σ. This approach is known as incremental (dynamic) dependency discovery [4,8,51,53,63,70,71], in contrast to batch (static) dependency discovery that discovers dependencies on the entire data set. Incremental DC discovery is, however, intricate for tuple insertions, as illustrated by the following example.

Example 2 Consider the incremental data $\triangle r_1$ (Table 2). Suppose that $\Sigma = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5\}$ is the set of DCs discovered on r_1. On $r_1 + \triangle r_1$, we can see that $\varphi_1, \varphi_2, \varphi_4$ and φ_5 still hold, but φ_3 is invalid due to the violations of tuple pairs (t_7, t_1) and (t_8, t_3). Therefore, we should remove φ_3 from Σ. More importantly, we should discover the set of new DCs holding on $r_1 + \triangle r_1$ as additions to Σ. Indeed, new valid DCs can be found based on the DCs in Σ that become invalid on $r_1 + \triangle r_1$.

For example, we have two new DCs valid on $r_1 + \triangle r_1$ based on φ_3. φ_3': $\neg\,(t.TXA > s.TXA \wedge t.SAL < s.SAL \wedge t.RATE > s.RATE \wedge t.ST = s.ST)$ and φ_3'': $\neg\,(t.TXA > s.TXA \wedge t.SAL < s.SAL \wedge t.RATE > s.RATE \wedge t.Date = s.Date)$. An additional value comparison is introduced to φ_3 to form φ_3' (resp. φ_3'').

Table 2 Incremental tax data: $\triangle r_1$

TID	Name	Date	SSN	NUM	MS	CH	PH	ST	ZIP	SAL	RATE	TXA	STX	CTX
t_7	Uckun	20130322	435849162	1368	M	Y	5872027	UT	84308	40000	1.43	3300	0	0
t_8	Hannen	20140218	701178073	1478	S	N	1638673	ND	58671	33000	2.05	1400	40	40
t_9	Beke	20150213	801350874	1533	S	Y	6192334	UT	84308	32000	1.23	3200	0	1000

Note that both φ'_3 and φ''_3 are valid on r_1. Obviously, any DC valid on $r_1 + \triangle r_1$ is also valid on r_1. Neither φ'_3 nor φ''_3 is in Σ, because discovery algorithms typically find only *minimal* constraints (formalized in Sect. 3). φ'_3 and φ''_3 are not minimal on r_1 since φ_3 is valid on r_1 and *logically implies* φ'_3 and φ''_3, which means that any instance satisfying φ_3 must satisfy φ'_3 and φ''_3. Note that φ_3 does not hold on $r_1 + \triangle r_1$, and φ'_3 and φ''_3 are minimal and valid DCs on $r_1 + \triangle r_1$, which should be added into Σ.

The incremental discovery approach is very relevant to data profiling in practice. When dependencies valid on data are employed to rewrite SQL queries for better efficiency [11, 33,54,61], our approach will make it feasible to maintain such dependencies with minimal overheads. Our approach is also an effective solution for a very large instance r. As will be seen shortly, we build indexes for r in the pre-processing step in a one-time manner. For all the tuple insertions to r, visiting the whole set of tuples in r is avoided with the indexes in our incremental DC discovery, and the indexes are not required to be rebuilt.

This work extends our conference paper on incremental POD discovery [63]. DCs subsume PODs as a special case are recognized as one of the most important dependencies and are used in many data management tasks, *e.g.,* [15,20–22,30,33,47]. We have made significant new contributions in this work. We have added (a) the framework of incremental DC discovery in the context of *evidence set* (Sect. 4), (b) a new algorithm to identify tuple pairs that satisfy two inequality comparisons across attributes, *e.g.,* $t.A > s.B \wedge t.C > s.D$, with a cost dependent on $log(|r|)$ in response to $\triangle r$, by leveraging our indexes on r (Sect. 5.3), (c) implementation details of our indexes (Sect. 6.4), (d) a novel incremental method for computing $\triangle\Sigma$ based on Σ and the incremental *evidences* incurred by $\triangle r$ (Sect. 8), and (e) completely new sets of experiments that compare our approach against the state-of-the-art batch DC discovery methods (Sect. 9). We have also extended (a) the method to update indexes (Sect. 6.3) and (b) the method to select indexes for Σ on r (Sect. 7), to deal with the full set of comparison operators of DCs.

Contributions We make a first effort to study incremental DC discovery, given a set Σ of DCs on r and a set $\triangle r$ of tuple insertions to r.

(1) Incremental violation detection is a crucial step of incremental DC discovery. We first propose an indexing technique to handle two inequality comparisons where each comparison can be performed across attributes. Leveraging the indexes on r, we show that all tuple pairs incurred by $\triangle r$ that satisfy the two comparisons can be identified with a cost dependent on $log(|r|)$, where $|r|$ is the size of r. We further present algorithms to build the proposed indexes on r and update the indexes in response to $\triangle r$.

(2) We then study techniques to select indexes for Σ on r, to cope with the full set of comparison operations supported in DCs. Our aim is to balance the index space and efficiency, and to share computations among the validations of DCs.

(3) We then present an efficient method to compute $\triangle\Sigma$, based on Σ and the incremental *evidences* incurred by $\triangle r$. After identifying invalid DCs on $r + \triangle r$ from Σ, we evolve new minimal and valid DCs by refining the invalid DCs.

(4) We finally conduct extensive experiments to verify our approaches, using a host of real-life and synthetic datasets. The results show that our incremental DC discovery method significantly outperforms the state-of-the-art batch counterparts up to orders of magnitude, even when $\triangle r$ is up to 30% of r.

Organizations We discuss the related work in Sect. 2. The basic notations of DCs are reviewed in Sect. 3. We give an overview of our approach in Sect. 4. We propose our indexing technique, present methods to build and update the proposed indexes, and study techniques to select indexes for the given set Σ of DCs in Sect. 5, Sect. 6, and Sect. 7, respectively. We present an efficient method to compute $\triangle\Sigma$ in Sect. 8. We conduct an experimental study to verify our approach in Sect. 9, followed by conclusions and future works in Sect. 10.

2 Related work

Dependency discovery methods are one of the most important aspects of data profiling [1–3] and studied for many kinds of dependencies, *e.g.,* UCCs [5,25], (approximate) FDs [18,19,35,41,66,67], CFDs [17], (approximate) order dependencies [27–29,36,58,59], matching dependencies [50,55,56], inclusion dependencies [64], relaxed FDs [9,10] and conditional inclusion dependencies [38]. In this section, we mainly investigate works close to ours: (a) DC discovery methods, (b) DC violation detection methods, and (c) incremental dependency discovery approaches.

Discovery of DCs Batch DC discovery methods are investigated in [6,12,37,42,43,69]. Specifically, [6,42] investigate the discovery of exact DCs holding on data, [37,69] consider discovering approximate DCs that hold with some exceptions, and [12,43] study methods for discovering both exact and approximate DCs.

We consider exact DCs in this study. Hydra [6] and DCFinder [43] are known as the state-of-the-art techniques for discovering exact DCs. Hydra adopts a hybrid strategy that combines DC discovery on small sample data with DC refinement based on DC violations on the full instance. DCFinder is a row-based approach. It develops efficient techniques for building *evidences* of all tuple pairs of the instance and then, discovers DCs from the evidences along the same lines as [12]. Multi-threaded parallelism techniques are also exploited to further improve the efficiency. According to the evaluations of [43], Hydra is more suitable to datasets with a relatively small number of *predicates*, while DCFinder performs better for datasets with a relatively small number of tuples and a large number of *predicates*.

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

Violation detection techniques of DCs Violation detections are employed in dependency discovery processes, for checking whether dependencies hold on a given instance and returning violations when necessary. Violation detection techniques of DCs are developed in [6,44,45]. The main difficulties in the violation detection of DCs arise from the inequality comparisons, *i.e.,* $\{<, \leq, >, \geq, \neq\}$. An inequality comparison on r can be implemented as a self *inequality-join* on r. In particular, the technique of inequality joins, called IEJoin [31], is adopted in Hydra [6]. Since traditional database systems usually cannot efficiently process inequality-joins, some distributed systems are developed [34,40,68]. In this study, we focus on the centralized setting.

Specifically, in VioFinder [45] every inequality comparison is treated separately. For each attribute, a sorted list of clusters is built and tuples with the same value in the attribute are in the same cluster. For a predicate on two different columns, VioFinder adopts a *sort-merge* approach to scan the two sorted lists for finding tuple pairs that satisfy the predicate. In contrast, IEJoin [31] aims to combine two inequality comparisons together, by building sorted arrays for different attributes and maintaining the connection between different sorted arrays with a *permutation* array. It has recently been observed that the performance of IEJoin and VioFinder is affected by the cardinalities of the columns (attributes) involved in the inequality comparisons [44]. The FACET system [44] organizes the inequalities of DCs based on the column cardinalities and adopts a flexible strategy that chooses algorithm IEJoin when all the related column cardinalities are above a predefined threshold, or the algorithm VioFinder otherwise.

Our goal is to perform *incremental* violation detections to facilitate incremental DC discovery. Since all DCs from Σ hold on r, we aim to identify DC violations incurred by $\triangle r$ without visiting the whole set of tuples in r, by leveraging some auxiliary structures built on r in the pre-processing step. Our method is a departure from the common (batch) violation detection methods.

We find performing efficient incremental inequality-joins is the main challenge. To our best knowledge, the only work on incremental inequality-joins is proposed in [32], by extending IEJoin [31] to cope with data updates. As opposed to [32] that has a cost dependent on $|r|$, we present a novel indexing technique that enables finding all the violating tuple pairs incurred by tuple insertions with a cost dependent on $log(|r|)$. Experimental evaluations demonstrate that our technique significantly outperforms [32] for tuple insertions.

Incremental (dynamic) dependency discovery Incremental (dynamic) dependency discovery techniques have recently received an increasing attention. Incremental discovery methods consider only tuple insertions, while dynamic discovery ones cope with both insertions and deletions. More specifically, dynamic discovery techniques are developed for UCCs [4], FDs [51,70] and inclusion dependencies [53], while incremental discovery methods are studied for PODs [63], LODs [71] and FDs [7,8].

Given the set Σ of constraints on r and a set $\triangle r$ of updates, the incremental discovery methods typically address two problems. The first one is to efficiently identify violations of Σ incurred by $\triangle r$, and the second one is to compute the changes $\triangle\Sigma$ to Σ based on Σ and the identified violations. The methods necessarily depend heavily on the dependency types, and hence, different incremental violation detection techniques and search space traversal methods are developed for different kinds of dependencies.

The technique for incrementally discovering DCs is more complicated than those other dependencies. This is because DCs are (far) more expressive than other dependencies (DCs subsume UCCs, FDs and PODs, and generalize LODs).

3 Preliminaries

In this section, we review basic notations of DCs [6,12,13,43]. For ease of reading, we summarize the notations to be introduced in Table 3.

We use R to denote a relational schema (an attribute set), r to denote a specific instance (relation) of R, t and s to denote tuples in r, and $t.A$ to denote the value of attribute A in a tuple t. Each tuple t has a distinct identifier, denoted by $t.id$. DCs support a rich set of operators on attribute values, *i.e.,* $\{<, \leq, >, \geq, =, \neq\}$. We denote by $\overline{op}$ (resp. $im(op)$, $sym(op)$), the inverse (resp. implication, symmetry) of an operator op, as summarized in Table 4.

Along the same lines as [6,43], we focus on *variable* DCs on *two* tuples in this study.

Denial constraints (DCs) DCs are defined based on *predicates*, where each predicate p is of the form $t.A_i\ op\ s.A_j$, $t, s \in r$, $t \neq s$ (different identifiers), $A_i, A_j \in R$, and the operator $op \in \{<, \leq, >, \geq, =, \neq\}$. Note that the comparison is on the same attribute if $i = j$ and is performed across two attributes, otherwise.

Table 3 Summary of notations (Sect. 3)

Notation	Semantics
R	A relational schema
r	An instance of schema R
A, B	A single attribute of R
t, s	Tuples in r
$t.A$	The value of attribute A in t
$t.id$	The distinct identifier of t
$p = t.A_i\ op\ s.A_j$	A predicate
$\mathcal{P}$	The predicate space
$\varphi' \subseteq \varphi$	The set of predicates of φ' is
	A subset of that of φ
$\varphi' \subset \varphi$	$\varphi' \subseteq \varphi$ and $\varphi \nsubseteq \varphi'$
$Evi(t, s)$	The *evidences* of a tuple pair (t, s)

Table 4 Operator inverse, implication and symmetry

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

A DC is defined as the negation of the conjunction of predicates. Specifically, a DC φ on a relational instance r is defined as: $\forall t, s \in r, \neg(p_1 \wedge \cdots \wedge p_m)$, where $p_1, \ldots, p_m$ are predicates concerning t, s. To simplify the notation, we omit the tuple quantifiers t and s when they are clear from the context.

Along the same lines as [6,12,43], we consider *non-trivial* DCs. A DC is trivial if it has two contradicting predicates, *e.g.,* $t.A > s.A$ and $t.A = s.A$. Moreover, the *symmetric* DC of a DC φ is the DC obtained by substituting t with s, and s with t in φ [12]. It suffices to consider either φ or its symmetric DC in DC discovery.

We further review the definitions of minimal and valid DCs [6,12,43].

Valid DCs A DC $\varphi = \neg(p_1 \wedge \cdots \wedge p_m)$ is valid (holds) on r, iff for any tuple pair (t, s) where t, s are from r, at least one of $p_1, \ldots, p_m$ is unsatisfied. A tuple pair (t, s) is referred to as a DC violation of φ, *i.e.,* (t, s) violates φ, if it satisfies all the predicates of φ.

Minimal DCs A DC φ is minimal iff there does not exist a valid DC φ' such that the set of the predicates of φ' is a proper subset of that of φ.

For two DCs φ and φ', we denote $\varphi' \subseteq \varphi$ if the set of predicates of φ' is a subset of that of φ, and denote $\varphi' \subset \varphi$ if $\varphi' \subseteq \varphi$ but $\varphi \nsubseteq \varphi'$.

Exact DC discovery methods aim to discover the set Σ of all *minimal* and *valid* DCs.

Batch DC discovery Given a relation r of schema R, batch DC discovery is to find the complete set Σ of minimal and valid DCs on r.

The search space of DC discovery is measured by $|\mathcal{P}|$, where $|\mathcal{P}|$ is the size of the *predicate space* and the predicate space is the set of all the predicates considered in the DC discovery on r [12]. Recall that DCs can use a set of six operators $\{<, \leq, >, \geq, =, \neq\}$ to compare attribute values, and comparisons can be conducted on two different attributes. It is meaningless and too costly to consider all operators on all combinations of attribute pairs. Some rules for determining the predicate space are proposed in [6,12,43]. Roughly speaking, (1) all the six operators can be used on numerical values, *e.g.,* salary and tax rate, while only "=" and "≠" can be used on categorical values, *e.g.,* state and phone number. (2) Comparisons across two attributes are only considered if the two attributes have the same type and at least 30% of common values. The predicate space can be determined in a pre-processing step of DC discovery. With a given predicate space $\mathcal{P}$, DC discovery has the complexity of $2^{|\mathcal{P}|}$ [12]. This complexity is much larger than those of UCCs, FDs, and PODs.

Evidence set [6,12,43] With a given predicate space $\mathcal{P}$, the *evidence* $Evi(t, s)$ of a tuple pair (t, s) is the set of predicates from $\mathcal{P}$ satisfied by (t, s), and the *evidence set* of an instance r is the set of evidences for all (t, s) from r^2. The number of evidences in the evidence set is typically much smaller than the number of tuple pairs, since different pairs may produce the same evidence.

The evidence set of r immediately determines all valid DCs on r. Recall that a DC φ is violated by a tuple pair (t, s), iff (t, s) satisfies all the predicates of φ. Hence, φ is valid on r, iff φ contains at least one predicate not in $Evi(t, s)$ for every (t, s) from r^2. For a predicate p: $t.A_i\ op\ s.A_j$, p is not in $Evi(t, s)$ iff its inverse $\overline{p}$: $t.A_i\ \overline{op}\ s.A_j$ is in $Evi(t, s)$. Based on this observation, the discovery of DCs on r can be recast as an enumeration problem on the evidence set of r. Different methods are presented to address the problem [6,12].

Example 3 The full predicate space is large for the instance r_1 in Table 1. For simplicity, we select four attributes SAL, TXA, RATE and ST. We show in Table 5 the predicate space $\mathcal{P}$ for the predicates concerning the four attributes.

Consider the tuple pair (t_7, t_1) on $r_1 + \triangle r_1$. The evidence is the set of predicates from $\mathcal{P}$ that are satisfied by the pair, and hence, $Evi(t_7, t_1) = \{P_2, P_5, P_6, P_8, P_9, P_{10}, P_{14}, P_{15}, P_{16}, P_{20}\}$. For $\varphi_3 = \neg(t.TXA > s.TXA \wedge t.SAL < s.SAL \wedge t.RATE > s.RATE)$, we see all the predicates are in $Evi(t_7, t_1)$. Therefore, φ_3 is violated by (t_7, t_1) and invalid on $r_1 + \triangle r_1$. A predicate not in $Evi(t_7, t_1)$, *e.g.,* P_{19}, can be added to φ_3 for resolving the violation of (t_7, t_1).

4 Overview of our approach

In this section, we first formalize the problem of incremental DC discovery and then, establish our framework for solving the problem in the context of evidence set.

Incremental DC discovery Given the complete set Σ of minimal valid DCs on relation r, and a set $\triangle r$ of tuple insertions to r, incremental DC discovery is to find, the change set $\triangle\Sigma$ of DCs to Σ that makes $\Sigma \oplus \triangle\Sigma$ the complete set of minimal and valid DCs on $r + \triangle r$. Specifically, $\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$, where $\triangle\Sigma^+$ and $\triangle\Sigma^-$ are disjoint. (1) $\triangle\Sigma^+ \cap \Sigma = \emptyset$, and $\triangle\Sigma^+$ contains the new minimal valid DCs on $r + \triangle r$ as additions to Σ. (2) $\triangle\Sigma^- \subseteq \Sigma$, and $\triangle\Sigma^-$ contains the DCs that should be removed from Σ. That is, $\Sigma \oplus \triangle\Sigma$ is computed as $(\Sigma \cup \triangle\Sigma^+) \setminus \triangle\Sigma^-$. We summarize these notations in Table 6.

Every batch DC discovery on r can be modeled as an incremental one with inputs r', $\triangle r'$ and Σ, where $r' = \emptyset$, $\triangle r' = r$ and $\Sigma = \emptyset$. The incremental discovery problem has the same worst-case complexity as its batch counterpart. In practice, $\triangle r$ is typically (much) smaller than r. An incremental algorithm can greatly improve the efficiency if its computation cost is dependent on $|\triangle r|$ but not $|r|$.

Avoiding the evidence set of r We aim to avoid storing the evidence set of r to reduce memory footprint. This is possible since we have the complete set Σ of minimal valid DCs on r. The following result guides our incremental DC discovery method.

Proposition 1 *(1) A DC $\varphi \in \Sigma$ belongs to $\triangle\Sigma^-$, iff φ is invalid on $r + \triangle r$. (2) If $\varphi \in \triangle\Sigma^+$, then there exists $\varphi' \in \triangle\Sigma^-$ such that $\varphi' \subset \varphi$.*

Proof (1) $\triangle\Sigma^-$ contains DCs that should be removed from Σ, *i.e.,* DCs that are not minimal or invalid on $r + \triangle r$. For any $\varphi \in \Sigma$, if φ is valid on $r + \triangle r$, then it must be minimal as well. Otherwise, we can find some φ' valid on $r + \triangle r$ such that $\varphi' \subset \varphi$. This contradicts our assumption that φ is minimal on r, since φ' is also valid on r. Therefore, $\varphi \in \triangle\Sigma^-$ iff φ is invalid on $r + \triangle r$.
(2) $\varphi \in \triangle\Sigma^+$, iff (a) φ is minimal and valid on $r + \triangle r$, and (b) $\varphi \notin \Sigma$. φ is valid on r since it is valid on $r + \triangle r$. Hence, $\varphi \notin \Sigma$ only when φ is not minimal on r. In this case, we have a $\varphi' \subset \varphi$ that is valid and minimal on r, by removing some predicates from φ. Such φ' is in Σ, but cannot be valid on $r + \triangle r$. Otherwise, φ is not minimal on $r + \triangle r$. Hence, we have $\varphi' \in \triangle\Sigma^-$. □

Proposition 1 states that there is no need to build $Evi(t, s)$ when both t and s are from r. All the DCs in $\triangle\Sigma^-$ contain

Table 5 The predicate space $\mathcal{P}$ for Example 3

$P_1 : t.SAL = s.SAL$	$P_2 : t.SAL \neq s.SAL$
$P_3 : t.SAL > s.SAL$	$P_4 : t.SAL \geq s.SAL$
$P_5 : t.SAL < s.SAL$	$P_6 : t.SAL \leq s.SAL$
$P_7 : t.TXA = s.TXA$	$P_8 : t.TXA \neq s.TXA$
$P_9 : t.TXA > s.TXA$	$P_{10} : t.TXA \geq s.TXA$
$P_{11} : t.TXA < s.TXA$	$P_{12} : t.TXA \leq s.TXA$
$P_{13} : t.RATE = s.RATE$	$P_{14} : t.RATE \neq s.RATE$
$P_{15} : t.RATE > s.RATE$	$P_{16} : t.RATE \geq s.RATE$
$P_{17} : t.RATE < s.RATE$	$P_{18} : t.RATE \leq s.RATE$
$P_{19} : t.ST = s.ST$	$P_{20} : t.ST \neq s.ST$

Table 6 Summary of notations (Sect. 4)

Notation	Semantics
Σ	The set of minimal valid DCs on r
$\triangle r$	The set of tuple insertions to r
$\triangle\Sigma=\triangle\Sigma^+\cup\triangle\Sigma^-$	The change to Σ for $\triangle r$
$\triangle\Sigma^+$	New minimal valid DCs on $r + \triangle r$
$\triangle\Sigma^-$	DCs that are removed from Σ

at least one predicate not in $Evi(t, s)$, and hence, $Evi(t, s)$ is not helpful for identifying DCs in $\triangle\Sigma^-$ or $\triangle\Sigma^+$.

Framework of incremental DC discovery We outline our incremental DC discovery approach.

(1) We first employ the indexing techniques to help identify the *incremental* evidences caused by $\triangle r$. As noted earlier, it suffices to consider $Evi(t, s)$ where at least one of t, s is from $\triangle r$. The required evidences can be further significantly reduced, since $Evi(t, s)$ is related to $\triangle\Sigma$ iff (t, s) violates some $\varphi \in \Sigma$. We know (t, s) violates φ iff (t, s) satisfies all the predicates of φ. By selecting some predicates with *high selectivity* and validating the selected predicates against the tuple pairs incurred by $\triangle r$, we can expect a small number of tuple pairs that satisfy the predicates, and hence, a small number of evidences to be computed.

We employ indexes to speed up the validations of the selected predicates, and the indexes are built on r in the pre-processing step, since both Σ and r are known. By validating predicates with indexes and updating indexes in response to $\triangle r$, for every $s \in \triangle r$, all $t \in r + \triangle r$ such that (s, t) or (t, s) satisfies the predicates, can be obtained without visiting the whole set of tuples in r. By sharing indexes among DCs from Σ that have the same predicates, the index space is greatly reduced and validations of predicates are shared among DCs, which is important for a large Σ. The required indexing techniques are highly non-trivial. We will address the related issues in Sects. 5, 6 and 7.

(2) We then compute $\triangle\Sigma$ ($\triangle\Sigma^-$ and $\triangle\Sigma^+$), based on the incremental evidences incurred by $\triangle r$ and the known Σ on r. According to Proposition 1, $\triangle\Sigma^-$ is a subset of Σ and contains all DCs that become invalid on $r + \triangle r$, and all DCs in $\triangle\Sigma^+$ can be obtained by adding more predicates to DCs in $\triangle\Sigma^-$. This guides our strategy to explore the search space in incremental DC discovery (Sect. 8): DCs in $\triangle\Sigma^+$ are found based on DCs from $\triangle\Sigma^-$, instead of starting from scratch.

Example 4 Recall Example 1 and Example 2. Suppose that $\Sigma = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4, \varphi_5\}$ is the set of DCs discovered on r_1. In the pre-processing step, we build indexes for Σ and r_1. In response to $\triangle r_1$, we perform incremental DC discovery to compute $\triangle\Sigma$ as follows.

(1) We identify the set of candidate violating tuple pairs incurred by $\triangle r_1$, by searching and updating the indexes (details in Sects. 5, 6 and 7). Suppose that two tuple pairs (t_7, t_1) and (t_8, t_3) are identified after visiting the indexes. We then compute the evidences of the two tuple pairs to form the *incremental* evidences.
(2) We compute $\triangle\Sigma$ based on the incremental evidences and Σ. We find $\varphi_3 = \neg(t.TXA > s.TXA \wedge t.SAL < s.SAL \wedge t.RATE > s.RATE)$ from Σ is no longer valid and put it into $\triangle\Sigma^-$. We then explore the search space for new minimal and valid DCs by adding predicates to φ_3, and collect them in $\triangle\Sigma^+$. For example, $\neg(t.TXA > s.TXA \wedge t.SAL < s.SAL \wedge t.RATE > s.RATE \wedge t.ST = s.ST)$ and $\neg(t.TXA > s.TXA \wedge t.SAL < s.SAL \wedge t.RATE > s.RATE \wedge t.Date = s.Date)$ are two DCs in $\triangle\Sigma^+$.

5 Index for inequality comparisons

In this section, we present a novel indexing technique to efficiently cope with inequality comparisons (possibly) across attributes. For reference, Table 7 shows the notations to be introduced in this section.

5.1 Motivation

It is important for incremental discovery algorithms to efficiently identify the incremental violations incurred by $\triangle r$. However, the introduction of inequality operators significantly complicates the problem, as illustrated by the following example.

Example 5 Consider φ_3 in Example 1: $\neg(t.TXA > s.TXA \wedge t.SAL < s.SAL \wedge t.RATE > s.RATE)$. For tuple t_8 from $\triangle r_1$ that is inserted into r_1, we aim to identify DC violations of φ_3, *i.e.,* (t_8, t) and (t, t_8) violating φ_3 where t is from r_1. As shown in Fig. 1, this can be conducted as follows.

(1) We compute two sets: $\{t \mid t_8.TXA > t.TXA\} = \{t_3, t_4, t_5, t_6\}$ and $\{t | t_8.SAL < t.SAL\} = \{t_1, t_3, t_6\}$. We then compute the intersection of the two sets: $\{t \mid t_8.TXA > t.TXA\} \cap \{t \mid t_8.SAL < t.SAL\} = \{t_3, t_6\}$. This set contains *candidates* of t such that (t_8, t) violates φ_3. Further, we have $t_8.RATE > t_3.RATE$ and hence, all the predicates of φ_3 are satisfied by (t_8, t_3). Therefore, φ_3 is violated by (t_8, t_3).

(2) Similarly, we have $\{t \mid t.TXA > t_8.TXA\} = \{t_0, t_1, t_2\}$, $\{t \mid t.SAL < t_8.SAL\} = \{t_0, t_2, t_4, t_5\}$, and $\{t$

Table 7 Summary of notations (Sect. 5)

Notation	Semantics
A^{op}	A shorthand for $t.A\ op\ s.A$
$A^{op}(t, s)$	$t.A\ op\ s.A$
Index(A^{op_1}, B^{op_2})	The index for A^{op_1} and B^{op_2}
Sorted$_i$	Sorted structure in Index(A^{op_1}, B^{op_2})
$T^s(A^{op_1}, B^{op_2})$	$\{t \mid A^{op_1}(s, t) \wedge B^{op_2}(s, t)\}$
$\overline{T}^s(A^{op_1}, B^{op_2})$	$\{t \mid A^{op_1}(t, s) \wedge B^{op_2}(t, s)\}$
$T^s(p_1, p_2)$	$s' \in T^s(p_1, p_2)$ iff (s, s') satisfies p_1, p_2
$\overline{T}^s(p_1, p_2)$	$s' \in \overline{T}^s(p_1, p_2)$ iff (s', s) satisfies p_1, p_2

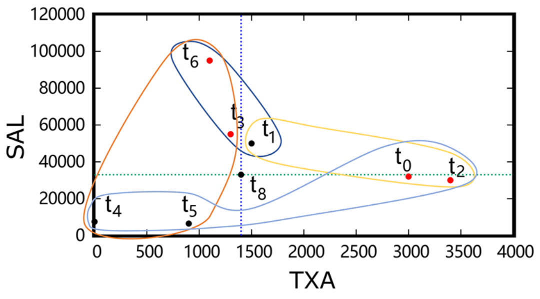


Fig. 1 Candidates of violating tuples *w.r.t.* t_8

$| t.TXA > t_8.TXA\} \cap \{t \mid t.SAL < t_8.SAL\} = \{t_0, t_2\}$. This set contains the *candidates* of t such that (t, t_8) violates φ_3. After checking the comparison on $RATE$, no violations are found from this set.

Observe that all the tuples in r_1 are visited in the computation of $\{t \mid t_8.TXA > t.TXA\}$ and $\{t \mid t.TXA > t_8.TXA\}$. For a tuple t from r_1, either $t.TXA > t_8.TXA$ or $t_8.TXA > t.TXA$, and it is similar for SAL. The reason is that inequality operators typically have a *low selectivity*, as illustrated in [31,32,44,45,68].

The above example highlights the quest for novel indexing techniques to facilitate the incremental DC discovery processes. Intuitively, indexes can be built by combining together several inequality comparisons to remedy the low selectivity of a single one. That is, we identify tuple pairs that simultaneously satisfy several inequality comparisons rather than one. There is a trade-off concerning the number of inequality comparisons: more comparisons would lead to more *selective* indexes, but are more specific, *i.e.,* only apply to DCs with all the comparisons and have large *ranks* (to be explained shortly). We will develop novel indexing techniques for two inequality comparisons, where each comparison concerns one attribute (Sect. 5.2). We will further show the proposed indexes can support comparisons across attributes (Sect. 5.3).

5.2 Index structure and applications

We propose a novel index structure for two operators $op_1, op_2 \in \{<, \leq, >, \geq\}$, and each operator concerns value comparison on the same attribute of two tuples, *e.g., $t.A\ op_1\ s.A$*. Since tuple quantifiers t and s are irrelevant of the index definition, we use the notation of *marked attributes* [23,24]. A marked attribute A^{op} where $op \in \{<, \leq, >, \geq\}$ is a shorthand for $t.A\ op\ s.A$. We write $A^{op}(t, s)$ if $t.A\ op\ s.A$. As an example, we have $\mathsf{SAL}^{>}(t_1, t_0)$ in Table 1.

Index structure Given a relation r and two marked attributes A^{op_1}, B^{op_2} $(op_1, op_2 \in \{<, \leq, >, \geq\})$, we propose an index structure, denoted as $\mathsf{Index}(A^{op_1}, B^{op_2})$. We conduct a preprocessing step, by clustering tuples in r based on their values in A and B (tuples with the same values in both A and B are in the same cluster). We keep an arbitrary tuple, say t, for each cluster, and move all the other tuples to 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 r. 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. Specifically,
(1) For each tuple t in r, there exists a single Sorted_i such that $t \in \mathsf{Sorted}_i$.
(2) For tuples t', t in each Sorted_i, we have $A^{op_1}(t', t)$ and $B^{op_2}(t', t)$ if t' is after t.

Example 6 We build $\mathsf{Index}(TXA^{>}, SAL^{<}) = \{[t_6, t_3, t_1, t_0, t_2], [t_4, t_5]\}$, as shown in Fig. 2. All tuples are organized by two sorted structures Sorted_1 and Sorted_2. In each Sorted_i, we have $\mathsf{TXA}^{>}(t', t)$ and $\mathsf{SAL}^{<}(t', t)$ if t' is after t.

Intuitively, the goal of the index is to divide r 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 [17] holding on parts of the relation rather than the whole relation.

We will present the method for building our index (the construction of Sorted_i) in Sect. 6. In what follows, we illustrate the usage (benefit) of the index and start with some notations.

Tuple pairs satisfying marked attributes For tuple s from Δr, we denote by $T^s(A^{op_1}, B^{op_2})$ the set of tuples from r, where $T^s(A^{op_1}, B^{op_2}) = \{t | A^{op_1}(s, t) \wedge B^{op_2}(s, t)\}$, and by $\overline{T}^s(A^{op_1}, B^{op_2})$ the set of tuples from r, where $\overline{T}^s(A^{op_1}, B^{op_2}) = \{t | A^{op_1}(t, s) \wedge B^{op_2}(t, s)\}$.

For example, we have $T^{t_8}(TXA^{>}, SAL^{<}) = \{t \mid t_8.TXA > t.TXA \wedge t_8.SAL < t.SAL\} = \{t_3, t_6\}$. That is, for any $t \in T^{t_8}(TXA^{>}, SAL^{<})$, (t_8, t) simultaneously satisfies the two predicates $t_8.TXA > t.TXA$ and $t_8.SAL < t.SAL$. Similarly, $\overline{T}^{t_8}(TXA^{>} SAL^{<}) = \{t | t.TXA > t_8.TXA \wedge t.SAL < t_8.SAL\} = \{t_0, t_2\}$.

Leveraging $\mathsf{Index}(A^{op_1}, B^{op_2})$, we present an algorithm for computing $T^s(A^{op_1}, B^{op_2})$ and $\overline{T}^s(A^{op_1}, B^{op_2})$. *Algorithm* Fetch (Algorithm 1) takes as inputs a tuple s from Δr and $\mathsf{Index}(A^{op_1}, B^{op_2})$ and works on every Sorted_i of $\mathsf{Index}(A^{op_1}, B^{op_2})$ as follows.

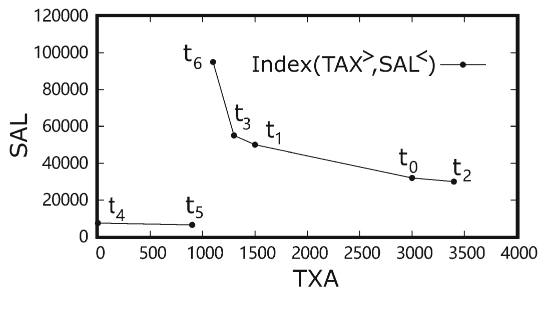


Fig. 2 $\mathsf{Index}(TXA^{>}, SAL^{<})$

Algorithm 1: Fetch

```
input : a tuple $s$, Index($A^{op_1}$, $B^{op_2}$)
output: $T^s(A^{op_1}, B^{op_2})$, $\overline{T}^s(A^{op_1}, B^{op_2})$
$T^s(A^{op_1}, B^{op_2}) \leftarrow \emptyset$; $\overline{T}^s(A^{op_1}, B^{op_2}) \leftarrow \emptyset$;
foreach Sorted$_i$ $\in$ Index($A^{op_1}$, $B^{op_2}$) do
    $p \leftarrow find(s, A^{op_1}, \text{Sorted}_i)$;
    $q \leftarrow find(s, B^{op_2}, \text{Sorted}_i)$;
    $right \leftarrow max(p, q) + 1$; $left \leftarrow min(p, q)$;
    while $right < \text{Sorted}_i.size$ do
        add Sorted$_i[right]$ to $\overline{T}^s(A^{op_1}, B^{op_2})$;
        $right \leftarrow right + 1$;
    while $left \geq 0$ do
        add Sorted$_i[left]$ to $T^s(A^{op_1}, B^{op_2})$;
        $left \leftarrow left - 1$;
```

(1) It first finds position p (line 3). Specifically, (a) p=−1 if A^{op_1}(Sorted$_i$[0], s), or (b) p = Sorted$_i.size$−1 if $A^{\overline{op_1}}$ (Sorted$_i$[Sorted$_i.size-1$], s), or (c) p is found such that $A^{\overline{op_1}}$(Sorted$_i$[p], s) and A^{op_1}(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, it adds tuples to the set $\overline{T}^s(A^{op_1}, B^{op_2})$ (lines 6-8).
(3) From position $min(p, q)$ to the left, it adds tuples to the set $T^s(A^{op_1}, B^{op_2})$ (lines 9-11).

As a post-processing step, if $t \in T^s(A^{op_1}, B^{op_2})$ (resp. $\overline{T}^s(A^{op_1}, B^{op_2})$), then all tuples from Equ^t_{AB} are also added to $T^s(A^{op_1}, B^{op_2})$ (resp. $\overline{T}^s(A^{op_1}, B^{op_2})$). Recall that tuples in Equ^t_{AB} have the same values in both A and B as t.

Example 7 Consider Index($TXA^>$, $SAL^<$) = {[t_6, t_3, t_1, t_0, t_2], [t_4, t_5]} and $t_8 \in \triangle r_1$ (shown in Fig. 3). We illustrate the computation of $T^{t_8}(TXA^>, SAL^<)$ and $\overline{T}^{t_8}(TXA^>, SAL^<)$.

(1) On Sorted$_1$ = [t_6, t_3, t_1, t_0, t_2], we have p = 1, because TXA$^\leq$(t_3, t_8) and TXA$^>$(t_1, t_8). Similarly, q = 2 because SAL$^\geq$(t_1, t_8) and SAL$^<$(t_0, t_8).

We have $right$ = $max(p, q)$ + 1 = 3, and add t_0 into $\overline{T}^{t_8}(TXA^>, SAL^<)$. We then move to the right in Sorted$_1$ and add t_2 to $\overline{T}^{t_8}(TXA^>, SAL^<)$. We have $left = min(p, q)$ = 1, and add t_3 to $T^{t_8}(TXA^>, SAL^<)$. We then move to the left in Sorted$_1$ and add t_6 to $T^{t_8}(TXA^>, SAL^<)$.

(2) On Sorted$_2$ = [t_4, t_5], p = 1 because TXA$^\leq$(t_5, t_8), and q = −1 because SAL$^<$(t_4, t_8). We have $right$ = Sorted$_2.size$ and $left$ = -1, and there are no new tuples for $T^{t_8}(TXA^>, SAL^<)$ or $\overline{T}^{t_8}(TXA^>, SAL^<)$.

Finally, we have $T^{t_8}(TXA^>, SAL^<)$ = {t_3, t_6}, and $\overline{T}^{t_8}(TXA^>, SAL^<)$ = {t_0, t_2}.

Proposition 2 *Algorithm* Fetch *correctly computes* $T^s(A^{op_1}, B^{op_2})$ *and* $\overline{T}^s(A^{op_1}, B^{op_2})$.

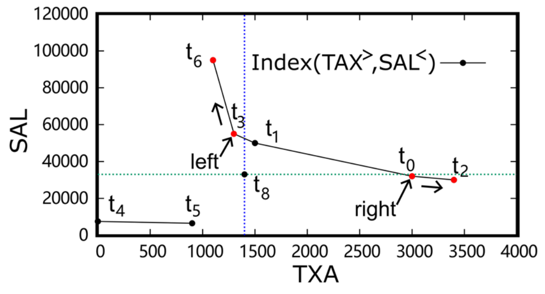


Fig. 3 Algorithm Fetch on Index($TXA^>$, $SAL^<$)

Proof The correctness of Fetch can be inferred from the index specification. For s from $\triangle r$ and t in Sorted$_i$, (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 we have $A^{\overline{op_1}}(t',s)$ (resp. $B^{\overline{op_2}}(t',s)$) for any t' before t in Sorted$_i$. □

Complexity On Index(A^{op_1}, B^{op_2}) = { Sorted$_1$, . . . , Sorted$_k$ }, it takes $\sum_{i\in[1,k]} O(log(n_i)) \leq k \cdot O(log(|r|))$ time to identify positions p, q on all Sorted$_i$, where n_i is the number of tuples in Sorted$_i$ and $|r| = \sum_{i\in[1,k]} n_i$, is the number of tuples in r. We use binary search to find p, q in Fetch, since 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, which is obviously necessary.
Remarks The complexity depends on $log(|r|)$ rather than $|r|$. We see a small rank k favors the efficiency of the index. We will present a method to build the index with the *minimum* rank in Sect. 6, and experimentally study index ranks in Sect. 9.

5.3 Indexes for comparisons across attributes

Algorithm Fetch is given to cope with two predicates $t.A$ op_1 $s.A$ and $t.B$ op_2 $s.B$. However, a predicate in DC can concern value comparison across attributes on two tuples, *e.g.,* $t.A$ op $s.B$. Hence, two predicates can involve at most 4 attributes if comparisons are performed across attributes in both predicates.

We further present an algorithm to handle two predicates of the most general form: $t.A$ op_1 $t'.B$ and $t.C$ op_2 $t'.D$, where $op_1, op_2 \in \{<, \leq, >, \geq\}$. In case, the second predicate is in the form of $t'.C$ op_2 $t.D$, note that it is equivalent to $t.D$ $sym(op_2)$ $t'.C$ (recall $sym(op_2)$ is the symmetry of op_2, as shown in Table 4). We start with some notations.
Tuple pairs satisfying two predicates For a tuple s from $\triangle r$, predicates $p_1 = t.A$ op_1 $t'.B$ and $p_2 = t.C$ op_2 $t'.D$, we denote by $T^s(p_1, p_2)$ the set of tuples from r, where $T^s(p_1, p_2)$ = {s' | $s.A$ op_1 $s'.B$ ∧ $s.C$ op_2 $s'.D$}, and by $\overline{T}^s(p_1, p_2)$ the set of tuples from r, where $\overline{T}^s(p_1, p_2)$ = {s' | $s'.A$ op_1 $s.B$ ∧ $s'.C$ op_2 $s.D$}. That is, (s, s') satisfies p_1 and p_2 for any

Algorithm 2: Fetch$^+$

```
input : a tuple $s$, Index($A^{op_1}$, $C^{op_2}$) and
        Index($B^{sym(op_1)}$, $D^{sym(op_2)}$)
output: $T^s(p_1, p_2)$ and $\overline{T}^s(p_1, p_2)$, where $p_1 = t.A\ op_1\ t'.B$
        and $p_2 = t.C\ op_2\ t'.D$
1  $T^s(p_1, p_2) \leftarrow \emptyset$; $\overline{T}^s(p_1, p_2) \leftarrow \emptyset$;
2  foreach Sorted$_i$ $\in$ Index($A^{op_1}$, $C^{op_2}$) (resp.
   Index($B^{sym(op_1)}$, $D^{sym(op_2)}$)) do
3      $p \leftarrow find(s.B, A^{op_1}$, Sorted$_i$);
4      (resp. $p \leftarrow find(s.A, B^{sym(op_1)}$, Sorted$_i$);)
5      $q \leftarrow find(s.D, C^{op_2}$, Sorted$_i$);
6      (resp. $q \leftarrow find(s.C, D^{sym(op_2)}$, Sorted$_i$);)
7      $cur \leftarrow max(p, q) + 1$;
8      while $cur <$ Sorted$_i$.$size$ do
9          add Sorted$_i[cur]$ into $\overline{T}^s(p_1, p_2)$;
10         (resp. add Sorted$_i[cur]$ into $T^s(p_1, p_2)$;)
11         $cur \leftarrow cur + 1$;
```

Table 8 Sample data: r_2

TID	A	B	C	D
t_0	7	5	7	9
t_1	8	6	3	25
t_2	30	8	9	11
t_3	40	0	2	15
t_4	15	7	17	22
t_5	25	2	15	30

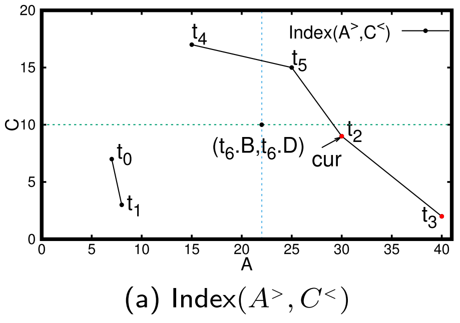
$s' \in T^s(p_1, p_2)$, and (s', s) satisfies p_1 and p_2 for any $s' \in \overline{T}^s(p_1, p_2)$.

Algorithm For two predicates $p_1 = t.A\ op_1\ t'.B$ and $p_2 = t.C\ op_2\ t'.D$, Fetch$^+$ (Algorithm 2) takes as inputs a tuple s and a pair of indexes Index(A^{op_1}, C^{op_2}), Index($B^{sym(op_1)}$, $D^{sym(op_2)}$), and outputs $\overline{T}^s(p_1, p_2)$ and $T^s(p_1, p_2)$.

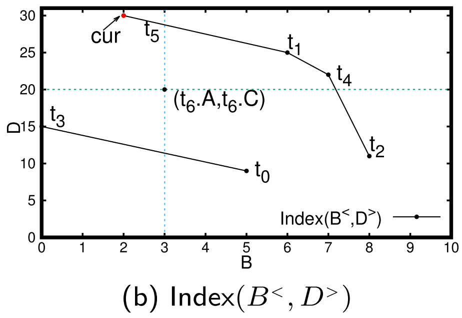
Fetch$^+$ works along the similar lines as Fetch to find positions p and q, on every Sorted$_i$ of Index(A^{op_1}, C^{op_2}) (resp. Index($B^{sym(op_1)}$, $D^{sym(op_2)}$)). The differences are that (a) the values of attributes B, D in s are used to visit Index(A^{op_1}, C^{op_2}), while the values of attributes A, C in s are used to visit Index($B^{sym(op_1)}$, $D^{sym(op_2)}$), and (b) the results that are obtained from Index(A^{op_1}, C^{op_2}) (resp. Index($B^{sym(op_1)}$, $D^{sym(op_2)}$)) only contribute to $\overline{T}^s(p_1, p_2)$ (resp. $T^s(p_1, p_2)$).

Table 9 Sample incremental data: $\triangle r_2$

TID	A	B	C	D
t_6	3	22	20	10



(a) Index($A^>$, $C^<$)



(b) Index($B^<$, $D^>$)

Fig. 4 Indexes for Example 8

Example 8 Consider two predicates: $p_1 = t.A > t'.B$ and $p_2 = t.C < t'.D$. We build a pair of indexes on r_2 (Table 8) for the two predicates: Index($A^>$, $C^<$) = $\{[t_4, t_5, t_2, t_3], [t_0, t_1]\}$, and Index($B^<$, $D^>$) = $\{[t_2, t_4, t_1, t_5], [t_0, t_3]\}$, as shown in Fig. 4. In response to tuple t_6 from $\triangle r_2$ (Table 9), we illustrate the computation of $T^{t_6}(p_1, p_2)$ and $\overline{T}^{t_6}(p_1, p_2)$ with Fetch$^+$.

(1) We first visit Index($A^>$, $C^<$), as shown in Fig. 4a. On Sorted$_1$ = $[t_4, t_5, t_2, t_3]$, we have $p = 0$, because $t_4.A \le t_6.B$ and $t_5.A > t_6.B$. Similarly, $q = 1$ because $t_5.C \ge t_6.D$ and $t_2.C < t_6.D$. We have $cur = max(p, q) + 1 = 2$. We add t_2 and then t_3 into $\overline{T}^{t_6}(p_1, p_2)$. On Sorted$_2$ = $[t_0, t_1]$, we have $p = 1$ because $t_1.A \le t_6.B$, and $q = -1$ because $t_0.C < t_6.D$. We set $cur = max(p, q) + 1 = 2$ and find no new tuples for $\overline{T}^{t_6}(p_1, p_2)$.
(2) We then visit Index($B^<$, $D^>$), as shown in Fig. 4b. We add t_5 into $T^{t_6}(p_1, p_2)$.
(3) Finally, we have $T^{t_6}(p_1, p_2) = \{t_5\}$, and $\overline{T}^{t_6}(p_1, p_2) = \{t_2, t_3\}$.

Complexity Algorithm Fetch$^+$ employs a pair of indexes, in contrast to Fetch that leverages one index. On each index, Fetch$^+$ uses the values of attributes different from the attributes on which the index is built, which obviously does not affect the complexity. That is, Fetch$^+$ has the same complexity as Fetch on each index.

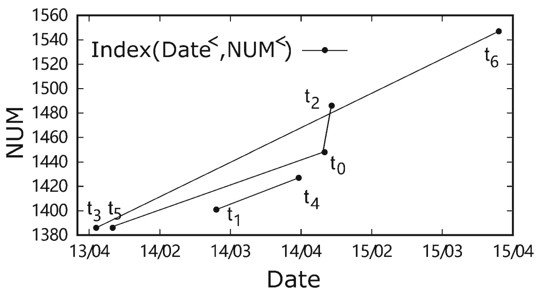


Fig. 5 Index I' for $Date^<$ and $NUM^<$

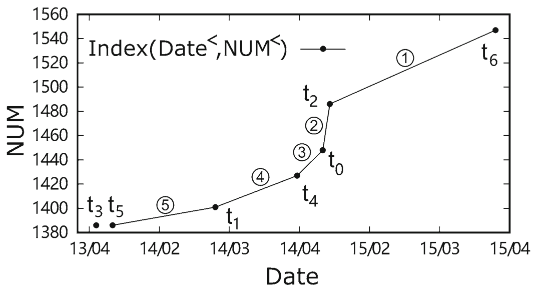


Fig. 6 Index I for $Date^<$ and $NUM^<$ with OptIndex

6 Techniques for our proposed index

In this section, we present techniques for the index type proposed in Sect. 5. Specifically, we provide an algorithm to build an *optimal* Index(A^{op_1}, B^{op_2}) in terms of *rank* (Sect. 6.1), show that some indexes can be used interchangeably (Sect. 6.2), study the updates of indexes (Sect. 6.3), and present implementation details of the indexes (Sect. 6.4).

6.1 Building indexes

For a relation r and two marked attributes A^{op_1}, B^{op_2}, the possible indexes may not be unique. As an example, for $Date^<$ and $NUM^<$, besides index $I' = \{[t_6, t_3],[t_2, t_0, t_5], [t_4, t_1]\}$ (Fig. 5), we have another index $I = \{[t_6, t_2, t_0, t_4, t_1, t_5], [t_3]\}$ (Fig. 6). They both satisfy the index specification, but I is preferable 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 OptIndex (Algorithm 3) builds an optimal index on r, for two marked attributes A^{op_1} and B^{op_2} ($op_1, op_2 \in \{<, \leq, >, \geq\}$).

(1) Tuples in r 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 the remaining tuples (lines 3-14). For each tuple s, if there exist multiple Sorted$_i$ that satisfy the index specification, *i.e.,* $B^{op_2}(s,t)$ where t is the last element of Sorted$_i$, then OptIndex selects the one to minimize the *difference* of attribute values in B incurred by s (lines 5-10), and adds s to the tail of the selected one (line 12). Observe that $A^{op_1}(s,t)$ is always guaranteed since tuples in r are 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 14).

Example 9 We build an index on r_1 with OptIndex, for $Date^<$ and $NUM^<$, as shown in Fig. 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 all the other tuples in r_1. t_2 is appended to Sorted$_1$ containing t_6, since $NUM^<(t_2, t_6)$, and it is similar for t_0, t_4, t_1, t_5. (3) Since t_3 and t_5 have the same value in NUM, Sorted$_2$ is built with t_3. (4) Finally, we have $I = \{[t_6, t_2, t_0, t_4, t_1, t_5], [t_3]\}$.

Algorithm 3: OptIndex

```
input  : a relation r and two marked attributes $A^{op_1}$, $B^{op_2}$
output: an optimal index on r, for $A^{op_1}$ and $B^{op_2}$
1  sort tuples in r on A according to $op_1$, and then on B according
   to $op_2$ for breaking ties;
2  $Index \leftarrow \{ [ r[0] ] \}$;
3  foreach tuple $s \in r \setminus \{r[0]\}$ do
4      $pos \leftarrow NULL$; $min \leftarrow -1$;
5      for each Sorted$_i \in Index$ do
6          $t \leftarrow$ the last element of Sorted$_i$;
7          if $B^{op_2}(s,t)$ then
8              if $min = -1$ or $min > |s.B - t.B|$ then
9                  $min \leftarrow |s.B - t.B|$;
10                 $pos \leftarrow$ Sorted$_i$;
11     if $pos != NULL$ then
12         append s to the end of pos;
13     else
14         add [s] into Index;
```

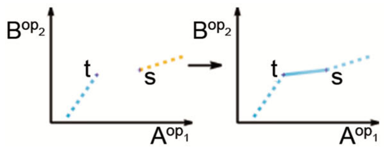
We then prove the optimality of OptIndex.

Proposition 3 *For 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}.*

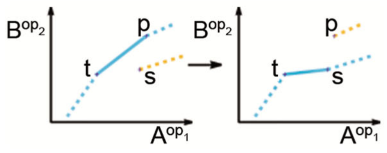
Proof We denote by I_{opt} the output of OptIndex. For an arbitrary index I' for A^{op_1} and B^{op_2}, we show that I' can be transformed into I_{opt} in a finite number of steps, and after each step, (a) the index specification is still satisfied, and (b) the index rank *never* increases.

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

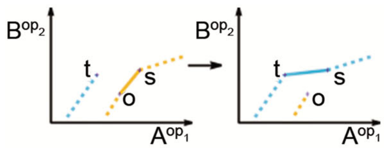
(2) We enumerate tuples s from I' one by one following the order μ, and treat s by considering the same s (tuple identifier) in I_{opt}. Tuple s remains unchanged in I', if (a) s is the first element of some Sorted$_i$ in I_{opt}, or (b) s is right behind a tuple t both in I' and in I_{opt}.



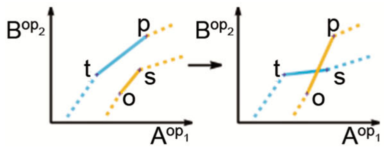
(a) neither o nor p exists



(b) p exists but o doesn't



(c) o exists but p doesn't



(d) both o and p exist

Fig. 7 Different cases in the Proof of Proposition 3

Now, suppose that s is right behind t in I_{opt}, but not in I'. We will move s to make s 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. Specifically, there are 4 cases.

Case Figure 7a: If neither o nor p exists, then we move s to make s 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. The rank of I' is not affected.

Case Figure 7c: If o exists but p does not, then o becomes the last element of 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 in 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 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 the complexity of $O(|r| \cdot log(|r|))$. Line 1 and lines 3-14 both have this complexity. We check the last element of every Sorted$_i$ to find the desired 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 Sorted$_i$, where k is the rank of the index and $k << |r|$ in practice.

6.2 Alternative indexes

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

(1) 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 tuple t if $A^{=}(t, s)$, when collecting tuples from the index for $T^s(A^{>}, B^{op_2})$ (resp. $\overline{T}^s(A^{>}, B^{op_2})$).

(2) With slight modifications, Algorithm Fetch can employ Index$(A^{<}, B^{<})$ (resp. 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 10 In Fig. 8, we show the computation of $T^{t_8}(Date^{<}, NUM^{>})$ and $\overline{T}^{t_8}(Date^{<}, NUM^{>})$, by leveraging Index$(Date^{<}, NUM^{<})$ = $\{[t_6, t_2, t_0, t_4, t_1, t_5],[t_3]\}$. We have $p = 4$, $q = 1$ in Sorted$_1$: $Date^{\geq}$(Sorted$_1[p], t_8$), $Date^{<}$(Sorted$_1[p+1], t_8$), $NUM^{<}$(Sorted$_1[q+1], t_8$), and $NUM^{\geq}$(Sorted$_1[q], t_8$).

Different from the original algorithm, we (1) let *left* = $min(p, q) + 1 = 2$, and *right* = $max(p, q) = 4$, and (2) check tuples from *left* to *right* in Sorted$_1$ to compute $T^{t_8}(Date^{<}, NUM^{>})$ and $\overline{T}^{t_8}(Date^{<}, NUM^{>})$. All nodes from *left* to *right* belong to one of these two sets. We add t_0 into $T^{t_8}(Date^{<}, NUM^{>})$ since $Date^{<}(t_8, t_0)$ and $NUM^{>}(t_8, t_0)$, and so are t_4 and t_1. After that we have $T^{t_8}(Date^{<}, NUM^{>}) = \{t_0, t_1, t_4\}$, and have $\overline{T}^{t_8}(Date^{<}, NUM^{>}) = \emptyset$. We find no new results on Sorted$_2$ = $[t_3]$.

Finally, we have $T^{t_8}(Date^{<}, NUM^{>}) = \{t_0, t_1, t_4\}$ and $\overline{T}^{t_8}(Date^{<}, NUM^{>}) = \emptyset$.

Remarks (1) The above-mentioned observations also apply to indexes employed in Fetch^{+}. (2) As will be illustrated in Sect. 7, alternative indexes can help reduce the number of indexes and can lead to indexes with small rank.

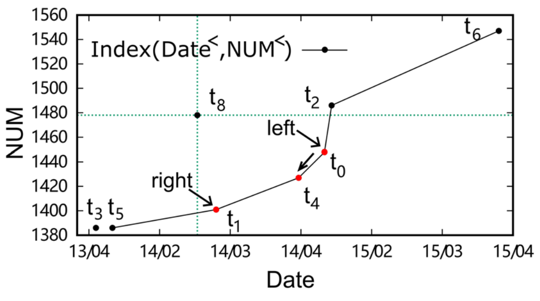


Fig. 8 Leveraging an index reversely

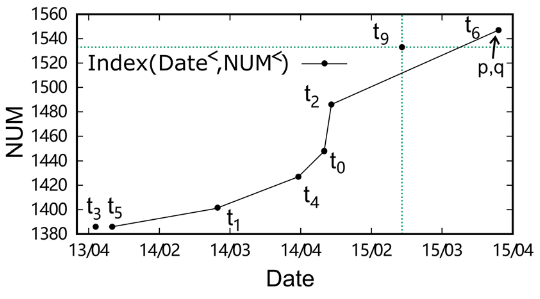


Fig. 9 Tuple t_9 and tuples in r_1

6.3 Updating indexes

We study updating the indexes for Δr, so that the indexes can help efficiently fetch tuple pairs (t, s) that simultaneously satisfy two predicates p_1 and p_2, where t, s are both from Δr.

(1) We first consider the case of $p_1 = A^{op_1}$ (*i.e.*, $t.A$ op_1 $s.A$), $p_2 = B^{op_2}$ (*i.e.*, $t.B$ op_2 $s.B$), and one index Index(A^{op_1}, B^{op_2}).

Applying Δr to indexes Before visiting an index I, we first sort tuples in Δr in the same way as line 1 of OptIndex. Tuples $s \in \Delta r$ are then applied to I one by one: (1) run Algorithm Fetch on I for s, and (2) update I with s. This enables us to deal with $s, s' \in \Delta r$ with the updated indexes. Indeed, updating indexes with s can be done along with running Fetch with s. The additional cost of updating indexes is hence marginal.

Example 11 Recall Sorted$_1$ of Index$(Date^<, NUM^<)$ from Example 9. We run Fetch on it for tuple t_9. To help understanding, Fig. 9 shows t_9 and all tuples of Table 1. Just like Example 10, we have $p = 0$ such that $Date^<$(Sorted$_1[p+1], t_9$) and $Date^{\geq}$(Sorted$_1[p], t_9$), and we have $q = 0$ such that $NUM^<$(Sorted$_1[q+1], t_9$) and $NUM^{\geq}$(Sorted$_1[q], t_9$). We find t_9 can be inserted into Sorted$_1$ since $p = q$ on Sorted$_1$.

Remarks (1) When updating an index with $s \in \Delta r$, s can be inserted into some Sorted$_i$ of the index if we find position $p = q$ on Sorted$_i$. If there are multiple such Sorted$_i$, then we heuristically pick the one with the maximum size.

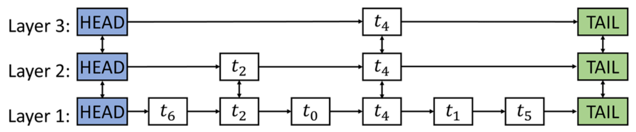


Fig. 10 Example index in SkipList

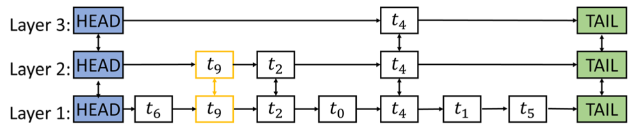


Fig. 11 Update index in SkipList

If none exists, then we create a new Sorted$_j$ with s, which increases the index rank by 1. (2) Updating an index may violate the *optimality* of the rank. That is, the updated index with Δr may have a larger rank than the index on $r + \Delta r$ built with OptIndex. However, the difference between the ranks is usually small, as experimentally verified in Sect. 9.

(2) We then consider the general case of $p_1 = t.A$ op_1 $t'.B$ and $p_2 = t.C$ op_2 $t'.D$. Recall that we have two indexes Index(A^{op_1}, C^{op_2}) and Index$(B^{sym(op_1)}, D^{sym(op_2)})$ in this case. When updating the indexes for $s \in \Delta r$, we visit the indexes in the same way as Fetch$^+$ but with different attribute values. More specifically, recall that in Fetch$^+$, $s.B$ and $s.D$ are used on Index(A^{op_1}, C^{op_2}), and $s.A$ and $s.C$ are used on Index$(B^{sym(op_1)}, D^{sym(op_2)})$. In contrast, we update the indexes by using $s.A$ and $s.C$ on Index(A^{op_1}, C^{op_2}), and using $s.B$ and $s.D$ on Index$(B^{sym(op_1)}, D^{sym(op_2)})$. Therefore, updating indexes *cannot* be done along with running Fetch$^+$, different from the case of Fetch.

6.4 Implementation details

We implement our indexes in a lightweight data structure, referred to as SkipList [46]. Each Sorted$_i$ is implemented as a SkipList. In Fig. 10, we show an example for Sorted$_1$ of Index$(Date^<, NUM^<)$. Each SkipList is built as a multi-layered sorted lists. The bottom layer contains all the nodes, and the number of nodes in a layer decreases as the layer number increases. The search of a node always starts from the uppermost layer and walks down to lower layers gradually. Intuitively, nodes at layer $i+1$ serve as indexes for nodes at layer i and can help "skip" several nodes at layer i. It is proved in [46] that it takes $O(log(n))$ time to find a node in SkipList, where n is the number of nodes in the bottom layer.

Example 12 Recall Example 11. For the SkipList given in Fig. 10, we illustrate the processing of t_9 in Fig. 11. To identify p, such that $Date^<$(Sorted$_1[p+1], t_9$) and $Date^{\geq}$(Sorted$_1[p], t_9$), the search starts from the layer 3 of the SkipList and goes to the next layer since $Date^<(t_4, t_9)$. In the layer 2, it finds $Date^{<}(t_2, t_9)$ and then, goes to the layer 1. It finds $p = 0$, since $Date^{<}(t_2, t_9)$ and $Date^{\geq}(t_6, t_9)$.

When updating SkipList with a tuple, the tuple is always inserted into the bottom layer. The tuple may also be inserted into higher layers, according to a randomization parameter [46]. This is the key of SkipList to enable finding a node in $O(log(n))$. For example, Fig. 11 shows the SkipList after the insertion of t_9.

7 Building indexes for DCs

In this section, we present our methods to build a set of indexes for Σ. Since Σ on r is known, building indexes is conducted as a *pre-processing* stage for the incremental DC discovery.

Our aim is to select predicates of DCs from Σ and build indexes for the predicates, such that all tuple pairs satisfying the predicates can be efficiently obtained by leveraging the indexes, where in every tuple pair at least one tuple is from $\triangle r$. As stated in Sect. 4, it suffices to consider these tuple pairs for computing the incremental evidences incurred by $\triangle r$. Several issues need to be considered: (a) different index types (analogous to the access patterns discussed in [39]), which is necessary to cope with the rich set of comparison operators of DCs, (b) the *selectivity* of predicates, since predicates with high selectivity are expected to exclude most tuple pairs, (c) the *rank* of the inequality indexes proposed in this study, which is closely related to the efficiency, and (d) the *coverage* of indexes, as an index should serve all DCs with the predicates for which the index is built.

Index types

(a) We use the common equality indexes for predicates with operator "=". Specifically, for a predicate of the form $t.A = s.A$, we build an equality index that clusters tuples from r on A, denoted as Index($A^{=}$). It takes $O(1)$ time to find the cluster for tuple $s \in \triangle r$ with hashing. If the predicate concerns comparison across attributes, *e.g., t.A = s.B*, then we use a pair of indexes Index($A^{=}$) and Index($B^{=}$). For $s \in \triangle r$, we use $s.A$ (resp. $s.B$) to search Index($B^{=}$) (resp. Index($A^{=}$)) for tuple pairs (s, t) (resp. (t, s)) satisfying $s.A = t.B$ (resp. $s.B = t.A$), while use $s.B$ (resp. $s.A$) to update Index($B^{=}$) (resp. Index($A^{=}$)).

(b) For two predicates $t.A\ op_1\ s.A$ and $t.B\ op_2\ s.B$, where $op_1, op_2 \in \{<, \leq, >, \geq\}$, we use Index($A^{op_1}, B^{op_2}$). For two predicates $t.A\ op_1\ t'.B$ and $t.C\ op_2\ t'.D$, where $op_1, op_2 \in \{<, \leq, >, \geq\}$, we use Index($A^{op_1}, C^{op_2}$) and Index($B^{sym(op_1)}, D^{sym(op_2)}$). As stated in Sect. 6.2, some inequality indexes can be used interchangeably. For example, Index($A^{\geq}, B^{>}$) and Index($A^{>}, B^{<}$) can be used in place of Index($A^{>}, B^{>}$).

(c) For a predicate $t.A_i \neq s.A_j$, we take it as the union of two predicates $t.A_i > s.A_j$ and $t.A_i < s.A_j$. Note that $t.A_i > s.A_j$ (resp. $t.A_i < s.A_j$) can then be combined with other inequality predicate of the same DC, to enable Fetch or Fetch^{+}.

(d) The above-mentioned indexes suffice to serve almost all DCs. Very occasionally, there are DCs of the form $\neg(t.A\ op_1\ s.B)$, where $op_1 \in \{<, \leq, >, \geq, \neq\}$. If op_1 is not "$\neq$", then we build an index on A (resp. B) that sorts tuples from r on A (resp. B). For $s \in \triangle r$, we use $s.A$ (resp. $s.B$) to search the index on B (resp. A) for tuple pairs (s, t) (resp. (t, s)) satisfying $s.A\ op_1\ t.B$ (resp. $t.A\ op_1\ s.B$), while use $s.B$ (resp. $s.A$) to update the index on B (resp. A). If op_1 is "$\neq$", then we take the DC as the union of $\neg(t.A > s.B)$ and $\neg\ (t.A < s.B)$ and build indexes accordingly.

We then introduce score functions to measure the preference of indexes.

Score for equality index Index($A^{=}$) can provide good selectivity when there are many distinct values in A. We measure Index($A^{=}$) with the following scoring function, and small values are preferred.

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

Score for inequality index We use the following function to measure Index(A^{op_1}, B^{op_2}).

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

In the function, $coverage(A^{op_1}, B^{op_2})$ is the number of DCs that can be served by Index(A^{op_1}, B^{op_2}) (DCs with predicates concerning comparisons across attributes are also taken into account, if Index(A^{op_1}, B^{op_2}) applies to the DCs), and $|r(A, B)|$ is the absolute value of correlation coefficient for attributes A and B. $r(A, B)$ is computed based on all tuples from the instance:

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

Note that it can be verified Index($A^{>}, B^{>}$) (resp. Index($A^{>}, B^{<}$)) has the rank of 1, if $r(A, B) = 1$ (resp. -1). We prefer indexes with small scores, *i.e.,* indexes that are on attributes with high correlation coefficient and serve more DCs. We also find that if A, B show a positive correlation, *i.e., r(A, B)*>0, then Index($A^{>}, B^{>}$) and Index($A^{\geq}, B^{>}$) are more likely to have small ranks, while Index($A^{>}, B^{<}$) and Index($A^{\geq}, B^{<}$) are likely to have small ranks if A, B show a negative correlation. We will further experimentally justify our score functions in Sect. 9.

Example 13 We use Index($Date^{<}, NUM^{<}$) rather than Index($Date^{<}, NUM^{\geq}$) for $\varphi_4 = \neg(t.SSN = s.SSN\ \wedge$

Algorithm 4: SelectIndex

```
   input  : a set $\Sigma$ of DCs
   output: a set $Ind(\Sigma)$ of indexes for covering DCs in $\Sigma$
1  foreach $\varphi = \neg(\mathcal{X}) \in \Sigma$ do
2      if there exists a predicate $t.A = s.A \in \mathcal{X}$ such that
       $score(\mathsf{Index}(A^{=})) < l$ then
3          add $\mathsf{Index}(A^{=})$ into $Ind(\Sigma)$;
4          remove from $\Sigma$ all DCs that are covered by $\mathsf{Index}(A^{=})$;
5  foreach $\varphi = \neg(\mathcal{X}) \in \Sigma$ do
6      if there exists a predicate $t.A = s.B \in \mathcal{X}$ such that
       $score(\mathsf{Index}(A^{=})) < l$ and $score(\mathsf{Index}(B^{=})) < l$ then
7          add $\mathsf{Index}(A^{=})$, $\mathsf{Index}(B^{=})$ into $Ind(\Sigma)$;
8          remove from $\Sigma$ all DCs covered by $\mathsf{Index}(A^{=})$ and
           $\mathsf{Index}(B^{=})$;
9  $Cand \leftarrow \emptyset$;
10 collect in $Cand$ all pairs of marked attributes $(A^{op_1}, B^{op_2})$ if
   $\mathsf{Index}(A^{op_1}, B^{op_2})$ can serve DCs from $\Sigma$ ;
11 foreach $(A^{op_1}, B^{op_2}) \in Cand$ do
12     if the ratio of the number of distinct values in A (or B) to $|r|$
       is smaller than $l'$ then
13         remove $(A^{op_1}, B^{op_2})$ from $Cand$;
14 while $Cand$ is not empty do
15     if $\Sigma$ is empty then break
16     pick $(A^{op_1}, B^{op_2})$ from $Cand$ with the minimum
       $score(A^{op_1}, B^{op_2})$;
17     $op_2 \leftarrow$ reverse $op_2$ if necessary according to $r(A, B)$ and a
       parameter $\alpha$;
18     add $\mathsf{Index}(A^{op_1}, B^{op_2})$ (and the related index if required by
       $\mathsf{Fetch}^{+}$) into $Ind(\Sigma)$ ;
19     remove all DCs from $\Sigma$ that are served by the index(es);
20 continue picking equality indexes in ascending order of the score
   values to cover DCs in $\Sigma$;
21 continue picking inequality indexes in ascending order of the
   score values to cover DCs in $\Sigma$;
```

$t.Date < s.Date \wedge t.NUM \geq s.NUM$), because r(Date, NUM) = 0.88. This explains why $T^{t_8}(Date^{<}, NUM^{>})$ and $\overline{T}^{t_8}(Date^{<}, NUM^{>})$ are computed by leveraging $\mathsf{Index}(Date^{<}, NUM^{<})$, as shown in Example 10.

Cover DCs *in* Σ *by indexes* It is necessarily costly to build an index for each DC if Σ is large. We propose to build a set of indexes such that for each $\varphi \in \Sigma$, at least one index is built for some predicates of φ, and we say φ is *covered* in this case.

Algorithm SelectIndex (Algorithm 4) is given to find the set $Ind(\Sigma)$ of indexes for covering all DCs from the given DC set Σ. As a pre-processing, every DC with predicate $t.A_i \neq s.A_j$ is replaced by two DCs with $t.A_i > s.A_j$ and $t.A_i < s.A_j$, respectively.

SelectIndex builds indexes to cover DCs with equality indexes followed by inequality indexes, since operations on equality indexes enjoy small theoretical complexity. SelectIndex prefers attributes with many distinct values when building indexes, which are expected to deliver better selectivity. To achieve this goal, SelectIndex uses predefined thresholds l and l' in lines 2, 6 and 12. We set l = 0.6 and l' = 0.1 in the implementation. When using inequality indexes to cover DCs, SelectIndex continues picking (A^{op_1}, B^{op_2}) with the minimum $score$ value from the set $Cand$ (line 16), until all pairs are used up or Σ is empty. This idea is similar to the heuristic approach to vertex cover [65]. It may also adjust op_2 for $\mathsf{Index}(A^{op_1}, B^{op_2})$ with a small rank. This is done according to $r(A, B)$ and a parameter α>0 (line 17). For example, it will reverse ">" as "<" on B and build $\mathsf{Index}(A^{<}, B^{<})$, if we have $(A^{<}, B^{>})$ but $r(A, B) > \alpha$. We set α = 0.3 in the implementation. When there are still uncovered DCs after the first round (lines 1-19), SelectIndex continues picking equality indexes and then inequality indexes to cover them, but relaxes the restrictions on the number of distinct values in the used attributes. It terminates when all DCs are covered.

Complexity In the worst case, SelectIndex has the complexity of $O(|R|^2 \cdot |\Sigma|)$, where $|\Sigma|$ is the number of DCs. Here, (1) the upper bounds for computing score values of $\mathsf{Index}(A^{=})$ and $\mathsf{Index}(A^{op_1}, B^{op_2})$ are $O(|R| \cdot |r|)$ and $O(|R|^2 \cdot |r|)$, respectively, and we use the uniform random sampling to estimate score values in our implementation, and (2) only $coverage(A^{op_1}, B^{op_2})$ is recomputed in SelectIndex, with a cost independent of $|r|$.

Remark We build indexes to efficiently fetch all the tuple pairs satisfying the predicates implemented by the indexes, where in every tuple pair at least one tuple is from $\triangle r$. Note that for every tuple $s \in \triangle r$, we not only visit the indexes for fetching such tuple pairs with s, but also update the indexes for s (updates of indexes are discussed in Sect. 6.3 and Sect. 7). Therefore, all such tuple pairs with both $s, t \in \triangle r$, are also fetched by leveraging the indexes.

8 Incremental discovery of DCs

In this section, we present our incremental DC discovery algorithm to compute $\triangle\Sigma$.

Algorithm IncDC (Algorithm 5) takes as inputs Σ on r, $\triangle r$, predicate space $\mathcal{P}$ and the set $Ind(\Sigma)$ of indexes, and computes $\triangle\Sigma$, such that $\Sigma \oplus \triangle\Sigma$ is the complete set of minimal valid DCs on $r + \triangle r$. Recall $\triangle\Sigma = \triangle\Sigma^{+} \cup \triangle\Sigma^{-}$, where $\triangle\Sigma^{+}$ contains the new minimal valid DCs added to Σ, while $\triangle\Sigma^{-}$ contains the DCs removed from Σ.

(1) IncDC finds the set T of candidate violating tuple pairs in response to $\triangle r$, by leveraging $Ind(\Sigma)$ (line 1). Specifically, for each tuple t from $\triangle r$, it works on each index *index* from $Ind(\Sigma)$ as follows. (a) Visit *index* with t to obtain tuple pairs satisfying the predicates implemented by *index*, and (b) update *index* with t. Our experimental evaluations (Sect. 9) show that the tuple pairs obtained with the indexes are often a small proportion of the new tuple pairs incurred by $\triangle r$.

(2) IncDC employs BuildEvidence (Algorithm 6) to compute the set E of *incremental* evidences resulting from T (line 2). The set T obtained by visiting indexes contains *candidate* violating tuple pairs. A tuple pair violates a DC when the

Algorithm 5: IncDC

```
Input: the complete set $\Sigma$ of minimal valid DCs on $r$, a set $\triangle r$
       of tuple insertions, predicate space $\mathcal{P}$ and the set $Ind(\Sigma)$
       of indexes for $\Sigma$
Output: $\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$, such that $\Sigma \oplus \triangle\Sigma$ is the
        complete set of minimal valid DCs on $r + \triangle r$
1  find the set $T$ of candidate violating tuple pairs in response to
   $\triangle r$, by leveraging $Ind(\Sigma)$;
2  $E \leftarrow$ BuildEvidence($T$) ;
3  $\Sigma' \leftarrow \Sigma$; $\triangle\Sigma^+ \leftarrow \emptyset$; $\triangle\Sigma^- \leftarrow \emptyset$;
4  foreach evidence $e \in E$ do
5      foreach $\varphi \in \Sigma'$ do
6          if $\varphi$ has a predicate not in $e$ then
7              continue;
8          else
9              $\Sigma' \leftarrow \Sigma' \backslash \{\varphi\}$;
10             if $\varphi \in \triangle\Sigma^+$ then
11                 $\triangle\Sigma^+ \leftarrow \triangle\Sigma^+ \backslash \{\varphi\}$;
12             else
13                 $\triangle\Sigma^- \leftarrow \triangle\Sigma^- \cup \{\varphi\}$;
14             foreach $p \in \mathcal{P} \backslash e$ do
15                 $\varphi' \leftarrow \varphi \cup \{p\}$;
16                 if there does not exist $\varphi'' \in \Sigma' \cup \triangle\Sigma^+$ such that the
                   set of predicates of $\varphi''$ is a subset of that of $\varphi'$ then
17                     $\triangle\Sigma^+ \leftarrow \triangle\Sigma^+ \cup \{\varphi'\}$;
18     $\Sigma' \leftarrow \Sigma' \cup \triangle\Sigma^+$;
19 $\triangle\Sigma^+ \leftarrow$ Minimize($\triangle\Sigma^+$);
20 $\triangle\Sigma \leftarrow \triangle\Sigma^+ \cup \triangle\Sigma^-$;
```

Algorithm 6: BuildEvidence

```
Input: the set $T$ of tuple pairs obtained with $Ind(\Sigma)$
Output: the set $E$ of incremental evidences
1  $E \leftarrow \emptyset$;
2  foreach tuple pair $(t, s) \in T$ do
3      $e \leftarrow \emptyset$;
4      for attributes $A_i, A_j \in R$ do
5          if $A_i$ and $A_j$ are categorical attributes then
6              if $t.A_i = s.A_j$ then
7                  $e \leftarrow e \cup \{t.A_i = s.A_j\}$;
8              else
9                  $e \leftarrow e \cup \{t.A_i \neq s.A_j\}$;
10         if $A_i$ and $A_j$ are numerical attributes then
11             $value = t.A_i - s.A_j$;
12             if $value < 0$ then
13                 $e \leftarrow e \cup \{t.A_i < s.A_j\} \cup \{t.A_i \leq s.A_j\} \cup \{t.A_i \neq s.A_j\}$ ;
14             if $value > 0$ then
15                 $e \leftarrow e \cup \{t.A_i > s.A_j\} \cup \{t.A_i \geq s.A_j\} \cup \{t.A_i \neq s.A_j\}$ ;
16             if $value = 0$ then
17                 $e \leftarrow e \cup \{t.A_i \leq s.A_j\} \cup \{t.A_i \geq s.A_j\} \cup \{t.A_i = s.A_j\}$ ;
18     $E \leftarrow E \cup e$;
```

pair satisfies all predicates of the DC, but indexes typically concern some (not all) predicates of every DC. *True* violating pairs can be obtained by further checking the remaining predicates of every DC. When Σ is large, DCs from Σ concern almost all predicates from the predicate space. Hence, checking the remaining predicates of DCs against tuple pairs from T does not reduce computation cost, compared with computing the evidences of the pairs from T.

For every pair (t, s) from T, BuildEvidence computes the evidence e for every attribute pair A_i, A_j. Note that A_i, A_j may refer to the same attribute, *i.e.,* $i = j$, or two comparable attributes (as noted in Sect. 3, two attributes are comparable if they have the same type and at least 30% of common values). All the predicates concerning A_i, A_j that are satisfied by (t, s) are efficiently identified with a single value comparison of $t.A_i$ and $s.A_j$, and are then collected in e. The evidence e of (t, s) is put into E. Note that different tuple pairs may produce the same evidence.

(3) IncDC computes $\triangle\Sigma$ based on E and Σ (lines 3-20). Σ' is initialized as Σ, and $\triangle\Sigma^+$ and $\triangle\Sigma^-$ are initialized as empty sets (line 3). For each evidence e, IncDC enumerates all $\varphi \in \Sigma'$ (lines 5-17). If all predicates of φ are in e, then φ is violated by the tuple pairs that produce e and is removed from Σ' (line 9). φ may be a DC generated by IncDC according to the evidences processed before e, in which case φ is also removed from $\triangle\Sigma^+$ (lines 10-11). Otherwise, φ is originally in Σ and hence put into $\triangle\Sigma^-$ (lines 12-13).

IncDC enumerates all possible ways to resolve the violation of φ *w.r.t.* the evidence e, by adding a predicate p not in e to φ every time (lines 14-15). The new candidate DC $\varphi' = \varphi \cup \{p\}$ is added into $\triangle\Sigma^+$ (line 17), if φ' is minimal by the definition (Sect. 3). Σ' is updated after processing all DCs for the evidence e (line 18). Note that updates of Σ' according to evidences after e will not introduce new violations *w.r.t.* e, since DCs are always extended with more predicates in IncDC.

After all the evidences from E are processed, Function Minimize is called to remove trivial and symmetric DCs from $\triangle\Sigma^+$ (recall Section 3). We have $\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$ as the final result.

Example 14 Recall Table 1 and Table 2. Tuple pairs (t_8, t_3) and (t_7, t_1) can be identified, by leveraging the index built for $\varphi_3 = \neg\,(t.TXA > s.TXA \wedge t.SAL < s.SAL \wedge t.RATE > s.RATE)$.

Suppose (t_8, t_3) is processed in IncDC first. (t_8, t_3) is found to violate φ_3, since all predicates of φ_3 belong to the evidence of (t_8, t_3). φ_3 is removed from Σ' and put into $\triangle\Sigma^-$. To resolve the violation, IncDC generates new DCs by adding predicates not in the evidence of (t_8, t_3) to φ_3, *e.g.,* $\varphi' = \neg\,(t.TXA > s.TXA \wedge t.SAL < s.SAL \wedge t.RATE > s.RATE \wedge t.ST = s.ST)$, and $\varphi'' = \neg\,(t.TXA > s.TXA \wedge t.SAL < s.SAL \wedge t.RATE > s.RATE \wedge t.PH = s.PH)$. φ' and φ'' are put into $\triangle\Sigma^+$ and then Σ'.

Now, (t_7, t_1) is processed. It can be verified all the predicates of φ'' belong to the evidence of this pair. Hence, φ'' is removed from Σ' and $\triangle\Sigma^+$. To resolve the violation, predicates outside the evidence of (t_7, t_1) can be used to form new DCs based on φ'', *e.g.,* $\neg(t.TXA > s.TXA \wedge t.SAL$

$< s.SAL \wedge t.RATE > s.RATE \wedge t.PH = s.PH \wedge t.ST = s.ST$). However, this DC is not minimal and not added into $\triangle\Sigma^+$, because it contains all predicates of φ' and φ' is already in Σ'.

Proposition 4 IncDC *computes* $\triangle\Sigma = \triangle\Sigma^+ \cup \triangle\Sigma^-$, *such that* $\Sigma \oplus \triangle\Sigma$ *is the complete set of minimal valid* DCs *on* $r + \triangle r$.

Proof The correctness of IncDC follows from Proposition 1. A DC from Σ is put into $\triangle\Sigma^-$ if all predicates of the DC belong to an evidence e in the set E, and all DCs in $\triangle\Sigma^+$ are generated by adding predicates to DCs from $\triangle\Sigma^-$. Specifically, (1) every DC in $\triangle\Sigma^+$ is *valid* after processing all evidences from E, since every DC guarantees to contain at least one predicate not in evidence e for every e from E. (2) Every DC φ' in $\triangle\Sigma^+$ is *minimal*, since the minimality is checked against all DCs in $\Sigma' \cup \triangle\Sigma^+$ before φ' is added into $\triangle\Sigma^+$ (line 16). This is because DCs in Σ' are always replaced by DCs with more predicates. Moreover, if φ'' and φ' are obtained by adding two different predicates to the same DC φ, then it is easy to see φ'' does not affect the minimality of φ'. (3) $\Sigma \oplus \triangle\Sigma$ is complete, since for a DC φ violated by an evidence e, all predicates not in e are enumerated and added to φ, for generating candidate DCs based on φ. □

Complexity IncDC can be regarded as an *incremental* version of the *evidence inversion* method from [6]. In contrast to the original method that computes Σ on r from scratch, IncDC computes $\triangle\Sigma$, based on Σ and the *incremental* evidences incurred by $\triangle r$. For every φ in $\triangle\Sigma^-$, in the worst case $2^{|\mathcal{P}|-|\varphi|}$ DCs can be generated based on φ after processing all evidences, where $|\varphi|$ is the number of predicates of φ. In practice, IncDC is far more efficient than the batch counterpart.
Remarks If $\varphi \in \Sigma$ becomes invalid on $r + \triangle r$, then φ is replaced by a set Ψ of more "specified" DCs in $\triangle\Sigma^+$ that are obtained by adding more predicates to φ. Hence, the index that serves φ also serves all DCs of Ψ. This feature is desirable: our indexes are built in an *off-line* process with a *one-time* cost and are not required to be rebuilt. The indexes are visited and updated in each run of IncDC, to facilitate the next run of IncDC.

9 Experimental study

In this section, we conduct an experimental study to verify the effectiveness and efficiency of our incremental DC discovery approach and indexing techniques.

All the tested datasets and code are available at https://github.com/snailtort/IncDC.

9.1 Experimental settings

Datasets. We use a host of real-life and synthetic datasets that are used in previous studies on dependency discovery methods [6,36,43,51,58,59,63]. We summarize the properties of all the datasets in Table 10, where $|r|$ (resp. $|R|$, $|\mathcal{P}|$) denotes the number of tuples (resp. attributes, predicates). These datasets have different data distributions and are employed for comprehensive analyses of our methods. Recall that the search space of DC discovery is measured by $|\mathcal{P}|$. In addition to the number $|\mathcal{P}|$ of all the predicates, the number of the predicates concerning comparisons across attributes is given in the bracket following $|\mathcal{P}|$.
Implementation. We implement all the algorithms in Java. (1) OptIndex (Algorithm 3) and SelectIndex (Algorithm 4), for building indexes in the pre-processing stage before IncDC. (2) IncDC (Algorithm 5), our incremental DC discovery algorithm. Multi-threaded parallelism is leveraged to visit indexes and build incremental evidences in parallel in the implementation of IncDC.
Compared Algorithms. (1) Hydra [6] and DCFinder [43]: the two state-of-the-art batch DC discovery methods are available online.[1] We use the default multi-thread setting of the two algorithms. (2) The incremental version of IEJoin [32]: the state-of-the-art algorithm for inequality joins with data updates. We obtain the implementation from the authors. (3) FACET [44]: the state-of-the-art batch (non-incremental) DC validation technique that combines IEJoin [31] and VioFinder [45]. We develop a best-effort implementation of it.
Parameter settings. Besides $|r|$ and $|\mathcal{P}|$, we use another parameter $|\triangle r|$: the number of inserted tuples. We define the ratio of incremental data as $|\triangle r|/|r|$. When varying $|r|$, $|\triangle r|$ or $|\mathcal{P}|$ is required, we take random sampling of data or predicates from the predicate space.
Running environment. All the experiments are run on a machine powered by an Intel Xeon Bronze 3204 1.90 G CPU (6 physical cores), with 128GB of memory and CentOS Linux release 7.9.2009 (Core). The average of 5 runs is reported as the experimental results.
Evaluations. We first discover Σ on r with the batch method and then, build indexes for Σ and r in the pre-processing step. Note that discovering Σ on r with the batch method is to produce the input of IncDC, and building indexes in the pre-processing step incurs a *one-time* cost and is not part of IncDC.

Leveraging the indexes, we perform incremental DC discovery to compute $\triangle\Sigma$ with IncDC. The correctness of IncDC is verified by checking $\Sigma \oplus \triangle\Sigma$ against the result of the batch method on $r + \triangle r$.

[1] https://github.com/HPI-Information-Systems/metanome-algorithms/tree/hydra. https://github.com/HPI-Information-Systems/metanome-algorithms/tree/master/dcfinder.

Table 10 Datasets and execution statistics (time in seconds)

Dataset properties					$\lvert\triangle r\rvert = 10\%\lvert r\rvert$				$\lvert\triangle r\rvert = 20\%\lvert r\rvert$				$\lvert\triangle r\rvert = 30\%\lvert r\rvert$			
Dataset	$\lvert r\rvert$	$\lvert R\rvert$	$\lvert\mathcal{P}\rvert$	$\lvert\Sigma\rvert$	$\lvert\triangle\Sigma\rvert$	DCFinder	Hydra	IncDC	$\lvert\triangle\Sigma\rvert$	DCFinder	Hydra	IncDC	$\lvert\triangle\Sigma\rvert$	DCFinder	Hydra	IncDC
Atom	105k	13	60(6)	544	78	523.18	218.20	**28.10**	78	630.85	276.69	**60.30**	74	865.87	321.33	**75.51**
FDR25	187k	25	130(60)	2,651	218	1,908	175.41	**17.37**	362	2,284	192.36	**23.57**	423	2,676	209.03	**35.14**
FDR15	187k	15	44(6)	412	1	761.75	53.51	**7.71**	1	913.18	57.77	**11.14**	1	1,126	64.36	**16.46**
Claim	112k	11	36(2)	33	9	484.83	27.71	**3.73**	55	574.47	38.07	**4.02**	33	686.47	38.51	**5.49**
FLI	367k	14	60(14)	62	0	7,380	102.39	**19.27**	0	8,987	111.23	**41.90**	0	10,856	116.34	**55.93**
NCV	675k	15	38(0)	727	286	36,047	240.93	**14.36**	915	60,320	277.38	**21.74**	987	56,250	285.43	**23.07**
DIT	780k	8	52(12)	50	24	55,121	359.06	**7.29**	30	71,620	1,145	**8.57**	38	85,600	1,330	**12.60**
AirPort	30k	11	32(6)	31	10	30.79	4.76	**0.70**	10	37.89	5.22	**0.87**	17	39.94	5.66	**1.03**
INSP	150k	13	40(2)	4,076	2,281	916.78	109.62	**16.90**	3,081	1,068	122.02	**29.09**	4,488	1,232	135.83	**43.08**
Adult	24k	15	60(6)	59,291	37,627	1,018	3,665	**467.03**	63,218	1,251	3,867	**644.20**	79,350	1,470	4,288	**849.26**
UCE	10k	11	60(6)	27,813	21,655	316.59	1,428	**160.39**	29,652	376.61	1,604	**206.67**	33,530	444.15	1,657	**274.04**

The results in bold are the best among the results of the compared algorithms

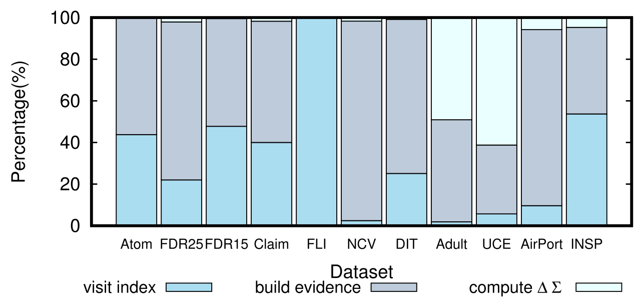


Fig. 12 Percentage of the time of different parts of IncDC

9.2 Experimental findings

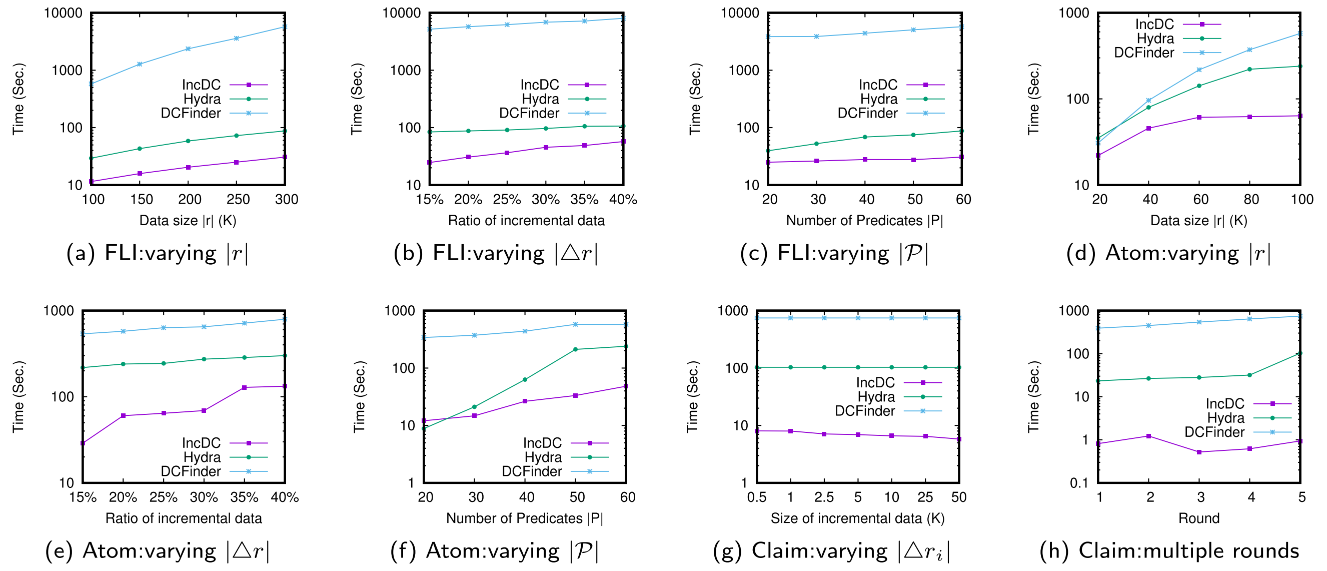
Exp-1: IncDC *against Batch methods* We compare IncDC against the batch methods discovering DCs on $r + \triangle r$ from scratch, in terms of the running time.

(1) We report the running time (in seconds) of incremental and batch methods on all datasets in Table 10, by setting $|\triangle r|/|r| = 10\%, 20\%$ and 30%, respectively. The results show that IncDC significantly outperforms batch methods on all the tested datasets up to orders of magnitude, even when $|\triangle r|/|r| = 30\%$. For in-depth analyses of the results, we also show in Table 10 the number $|\Sigma|$ of DCs discovered on r, and the number $|\triangle\Sigma|$ of DCs discovered by IncDC in response to $\triangle r$. Intuitively, $|\triangle\Sigma|$ measures the changes to Σ, and hence, the inherent difficulties of incremental DC discovery. We see on some datasets, *e.g.,* Adult and UCE, both $|\Sigma|$ and $|\triangle\Sigma|$ are very large. Indeed, when $|\triangle r|/|r| = 30\%$ on Adult and UCE, $|\triangle\Sigma|$ is even larger than $|\Sigma|$, which implies dramatic changes to Σ. Nevertheless, IncDC is still much faster than the batch methods on these datasets, showing the benefit of our incremental approach.

It is worth mentioning that the number of DCs is much larger than those of PODs and FDs. Take Adult as an example. There are only 2,766 PODs from the 59,291 discovered DCs, and further 63 FDs from the PODs.

(2) We divide the running time of IncDC into three parts: index visit/update, building evidence, and computing $\triangle\Sigma$, and report the results with $|\triangle r|/|r| = 30\%$ in Fig. 12. We see the first two parts take most of the time on most datasets, which implies that the computation of $\triangle\Sigma$ is very efficient based on the incremental evidences. However, computing $\triangle\Sigma$ becomes costly on Adult and UCE. This is reasonable, since both $|\Sigma|$ and $|\triangle\Sigma|$ are very large on Adult and UCE, as shown in Table 10. Index visits and updates take almost all the time on FLI. On this dataset, the number of candidate violating tuple pairs obtained with the indexes is very small, and hence, the time of computing the incremental evidences is negligible. Moreover, as shown in Table 10, $\triangle\Sigma$ is empty on FLI, *i.e.,* there are no changes to Σ.

(3) We vary the parameters to study the scalability of different methods. We set $|r| = 300\text{K}$, $|\triangle r|/|r| = 20\%$ and $|\mathcal{P}| =$



(a) FLI:varying $|r|$ (b) FLI:varying $|\triangle r|$ (c) FLI:varying $|\mathcal{P}|$ (d) Atom:varying $|r|$

(e) Atom:varying $|\triangle r|$ (f) Atom:varying $|\mathcal{P}|$ (g) Claim:varying $|\triangle r_i|$ (h) Claim:multiple rounds

Fig. 13 IncDC against Hydra and DCFinder

60 by default on FLI and vary one parameter in each of the experiments.

Varying $|r|$. Fig. 13a shows results by varying $|r|$ from 100 to 300K ($|\triangle r|$ from 20 to 60K). DCFinder is more sensitive to $|r|$ than Hydra, consistent with the results reported in [43]. IncDC performs much better than the batch methods, *e.g.,* IncDC takes only 30 s but Hydra takes 87 s, when $|r|$= 300K.

Varying $|\triangle r|$. Fig. 13b shows results by varying $|\triangle r|$ from 45 to 120K (the ratio of $|\triangle r|$ to $|r|$ increases from 15 to 40%). Intuitively, as $|\triangle r|/|r|$ increases, it becomes more challenging for our incremental algorithm to beat the batch algorithms. IncDC is consistently faster than the batch methods, and scales very well with $|\triangle r|$. The time of IncDC increases from 25 to 57 s as $|\triangle r|$ increases from 45 to 120K.

Varying $|\mathcal{P}|$. By varying $|\mathcal{P}|$ from 20 to 60, we report experimental results in Fig. 13c. IncDC is less sensitive to $|\mathcal{P}|$ than the batch methods. As $|\mathcal{P}|$ varies from 20 to 60, IncDC takes 25 to 30 s, as opposed to 39 to 87 s by Hydra. This is desirable since $|\Sigma|$ can grow exponentially with $|\mathcal{P}|$, which typically significantly affects the efficiency of DC discovery. We find the efficiency of IncDC is more affected by the number of indexes that *cover* all DCs in Σ, *i.e., $Ind(\Sigma)$* generated in SelectIndex (Sect. 7). For reference, we show the number of indexes in Table 11, denoted as $|Ind(\Sigma)|$. It can be seen that $|Ind(\Sigma)|$ is typically much smaller than $|\Sigma|$, which explains why IncDC is less sensitive to $|\mathcal{P}|$. We will further analyze $|Ind(\Sigma)|$ in Exp-2.

(4) We set $|r| = 100$K, $|\triangle r| = 20K$ and $|\mathcal{P}| = 60$ by default on Atom. We vary $|r|$ from 20 to 100K in Fig. 13d, $|\triangle r|/|r|$ from 15 to 40% in Fig. 13e, and $|\mathcal{P}|$ from 20 to 60 in Fig. 13f. The results verify our observations on FLI. IncDC can be faster than the batch methods up to orders of magnitude. Although IncDC is slightly slower than Hydra when $|\mathcal{P}|$ is 20, the advantage of IncDC becomes very evident as $|\mathcal{P}|$ increases, and IncDC is 5 times faster than Hydra when $|\mathcal{P}| = 60$.

(5) We conduct experiments on Claim with $|\triangle r| = 50$K. On this real-life dataset, tuple insertions are processed in the order of their timestamps, so as to follow the real change history. We handle $\triangle r$ as a series of sets of tuple insertions ($\triangle r = \triangle r_1 \cup \cdots \cup \triangle r_k$), where $\triangle r_i$ ($i \in [1, k]$) is of the same size. Recall that our indexes are not required to be rebuilt in the whole process.

In Fig. 13g, we vary $|\triangle r_i|$ from 0.5 to 50K and report the time for the whole change history. For example, a single set of tuple insertions is applied to r if $|\triangle r_i| = 50$K, while 100 sets of tuple insertions are applied one by one if $|\triangle r_i| = 0.5$K. IncDC is always much faster than the batch methods. The time of batch methods is computed on $r + \triangle r$ and hence, not affected by $|\triangle r_i|$. The time of IncDC decreases by nearly 28% as $|\triangle r_i|$ increases from 0.5 to 50K, as expected. Although the total time of index visits is almost not affected by varying $|\triangle r_i|$, we find the total time of computing $\triangle\Sigma$ decreases due to fewer rounds of requests, as $|\triangle r_i|$ increases.

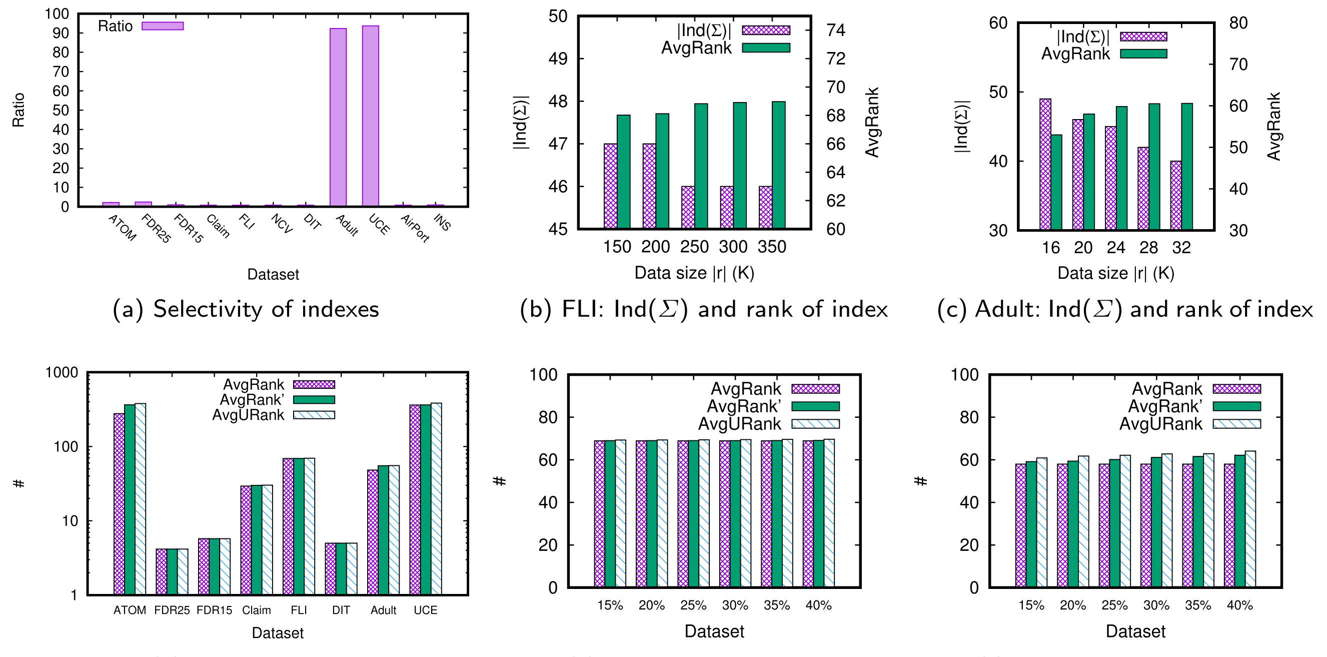
We fix $|\triangle r_i| = 10$K in Fig. 13h and show the time for applying $\triangle r_1$ to r, $\triangle r_2$ to $r + \triangle r_1$, etc. The time of batch methods always increases as data sizes grow. However, the time of IncDC for a specific $\triangle r_i$ may vary since different $\triangle r_i$ causes different violations and changes to the DC set.

Exp-2: Benefits of indexes We experimentally study the benefits of our indexing techniques.

(1) For all the datasets in Table 10, we show the memory footprint and creation time of indexes in Table 11. For reference, we also show the number $|\Sigma|$ of DCs, the number $|Ind(\Sigma)|$ of indexes, and the memory footprint of each dataset. The memory of indexes is also affected by data types and numbers of

Table 11 Indexes for datasets

Dataset	$\lvert R\rvert$	$\lvert r\rvert$	$\lvert\Sigma\rvert$	$\lvert Ind(\Sigma)\rvert$	Data	Index	Time
Atom	13	105K	544	52	87MB	300MB	52.62 s
FDR25	25	187K	2,651	24	512MB	774MB	7.99 s
FDR15	15	187K	412	16	321MB	530MB	3.25 s
Claim	11	112K	33	5	138MB	154MB	2.16 s
FLI	14	367K	62	46	281MB	582MB	70.99 s
NCV	15	675K	727	4	1,300MB	1,505MB	4.96 s
DIT	8	780K	50	6	1,502MB	1,708MB	16.16 s
AirPort	11	30K	31	9	52MB	135MB	0.59 s
INSP	13	150K	4,076	7	220MB	241MB	1.69 s
Adult	15	24K	59,291	45	29MB	99MB	9.51 s
UCE	11	10K	27,813	55	5MB	137MB	50.37 s



(a) Selectivity of indexes (b) FLI: Ind(Σ) and rank of index (c) Adult: Ind(Σ) and rank of index

(d) Rank changes (e) FLI: rank changes (f) Adult: rank changes

Fig. 14 Selectivity of indexes, and the numbers and ranks of indexes

distinct values of the indexing attributes. We see $|Ind(\Sigma)|$ is much smaller than $|\Sigma|$. Since each index in $Ind(\Sigma)$ is visited and updated for each tuple from $\triangle r$, a small $|Ind(\Sigma)|$ usually leads to high efficiency in obtaining the set of candidate violating tuple pairs in response to $\triangle r$. We also see the memory footprint of indexes is usually comparable to that of data and indexes can be efficiently created on all the tested datasets. Recall building indexes is conducted as a pre-processing step with a one-time cost, and the time is not included in that of IncDC.

(2) In Fig. 14a, we show the ratio of the number of tuple pairs obtained with the indexes to the total number of tuple pairs incurred by $\triangle r$, *i.e.*, $|\triangle r|(|r + \triangle r|)$. This ratio measures the selectivity of indexes. We see the ratio is very low (below 5%) on most datasets, which implies that indexes help prune more than 95% of tuple pairs and explain why IncDC can be faster than the batch methods by orders of magnitude on

Table 12 Different types of indexes (time in milliseconds)

Dataset	Index Type	# of indexes	# of DCs	Creation Time (AVG)	Visit/update Time (AVG)
Atom	Equality	1	32	211	207
	Inequality	51	512	1,027	2,375
UCE	Equality	6	24,001	48	72
	Inequality	49	3,812	1,022	1,499

some datasets. The ratios on Adult and UCE, however, are very high. As shown in Table 10, $|\triangle\Sigma|$ is extremely large on Adult and UCE, which states that $\triangle r$ incurs a huge number of violating tuple pairs. The inherent difficulties necessarily hinder the selectivity of indexes, since indexes can only help prune non-violating tuple pairs. Even in these extreme settings, the results in Table 10 show that IncDC still outperforms the batch methods by about 60% on Adult and UCE, due to incremental computations of evidences and $\triangle\Sigma$.

(3) In Table 12, we study the behaviors of equality and inequality indexes, respectively, using datasets Atom and UCE. Specifically, equality indexes are used for predicates with "=", while inequality indexes are used for a pair of inequality comparisons. These two types of indexes can serve (cover) all DCs from Σ on Atom and UCE. Recall "$\neq$" may be treated as the union of "$>$" and "$<$" and combined with other inequality comparison of the same DC, to enable indexes for a pair of inequality comparisons, as illustrated in Sect. 7.

For each index type, we show the number of indexes of this type (# of indexes), and the number of DCs that are served by this type of indexes (# of DCs). For example, there is only one equality index built on Atom, but the index is used to fetch candidate violating tuple pairs for 32 DCs. The number of inequality indexes is typically larger than that of equality indexes, since the number of different combinations of inequality comparisons in DCs can be large. Nevertheless, a relatively small number of indexes suffice to serve a very large number of DCs, *e.g.,* on UCE. For each index type, we also show the average (AVG) creation time and visit/update time of indexes of this type. As expected, it takes less time to create and leverage equality indexes than the inequality indexes, which is consistent with the theoretical complexity analyses. The results demonstrate our indexes can effectively address the new challenges introduced by the inequality comparisons in DCs.

Summary. Putting the results together, we summarize the benefits of our indexing techniques. (a) The number of indexes is much smaller than that of DCs. The validations of many DCs can be conducted simultaneously with the indexes, which is crucial for the efficiency when the number of DCs is very large. (b) The number of candidate violating tuple pairs obtained with the indexes is usually much smaller than that of the new tuple pairs incurred by $\triangle r$, and the evidences of most new tuple pairs are not required to be computed. (c) The visits to the whole set of tuples in r are avoided with the use of indexes.

Exp-3: Analyses of indexes We experimentally study the properties of our indexes in detail.

(1) We study $|Ind(\Sigma)|$ and the ranks of indexes built on r, which affect the efficiency of index visits. We denote by *AvgRank* the average rank of indexes in $Ind(\Sigma)$, excluding equality indexes. We set $|\mathcal{P}|$ = 60, vary $|r|$ from 150 to 350K on FLI, and report results in Fig. 14b. Both $|Ind(\Sigma)|$ and *AvgRank* almost remain unchanged as $|r|$ increases. In Fig. 14c, we show results on Adult by fixing $|\mathcal{P}|$ = 60 and varying $|r|$ from 16 to 32K. $|Ind(\Sigma)|$ is in [40, 49], and *AvgRank* is in [53, 61]. Both $|Ind(\Sigma)|$ and *AvgRank* are not sensitive to the increase of $|r|$, which is desirable. Note that on Adult, $|\Sigma|$ decreases as $|r|$ increases, and hence, $|Ind(\Sigma)|$ decreases.

(2) Updating an inequality index may increase its rank (Sect. 6.3). We denote by *AvgURank* the average index rank on r+$\triangle r$ after updates. For comparison, we also apply OptIndex to r+$\triangle r$ for indexes with guaranteed minimum ranks, and denote the average rank by *AvgRank'*. Figure 14d shows results on the datasets from Table 10 (excluding those with only equality indexes), by setting $|\triangle r|$ = 30%$|r|$. *AvgURank* in response to $\triangle r$ is at most 136% of *AvgRank* on r. The difference between *AvgURank* and the optimum *AvgRank'* is also small, and *AvgURank* is at most larger than *AvgRank'* by 6%. The results show index ranks are not sensitive to updates with $\triangle r$.

We then vary the ratio of incremental data. We set $|\mathcal{P}|$=60, $|r|$=300K, vary $|\triangle r|/|r|$ from 15 to 40% on FLI, and report the results in Fig. 14e. *AvgURank* increases slightly from 69.28 to 69.67, and is consistently within 102% of *AvgRank'*. Figure 14f reports results on Adult with $|\mathcal{P}|$=60, $|r|$ = 20K. *AvgURank* increases from 61.33 to 64.13 as $|\triangle r|/|r|$ increases from 15 to 40%, and is within [102%,104%] of *AvgRank'*. The impact on index ranks incurred by updates is small even when $|\triangle r|/|r|$ is 40%.

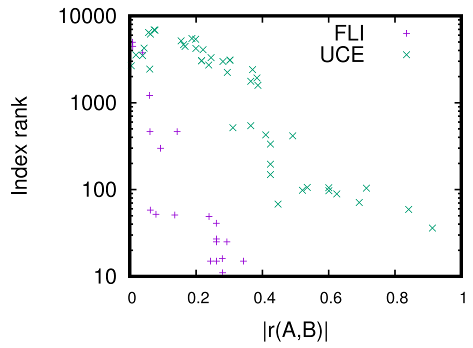


Fig. 15 Index rank *w.r.t.* $|r(A, B)|$

(3) We experimentally study the score function employed in SelectIndex (Sect. 7). On FLI and UCE, we compute the absolute value of correlation coefficient of every pair A, B from R, *i.e.,* $|r(A, B)|$. In Fig. 15, we show $|r(A, B)|$ on the x-axis, and the rank of the inequality index built on A, B on the y-axis. The results show indexes on attribute pairs with large $|r(A, B)|$ have small ranks, which justify our score function.

Exp-4: Fetch *and* Fetch$^+$ *against incremental* IEJoin We compare Fetch against IEJoin in the time for computing $T^s(A^{op_1}, B^{op_2})$ and $\overline{T}^s(A^{op_1}, B^{op_2})$, and compare Fetch$^+$ against IEJoin in the time for computing $T^s(p_1, p_2)$ and $\overline{T}^s(p_1, p_2)$ where p_1, p_2 are inequality comparisons across attributes, for all tuples s from $\triangle r$.

(1) We set $|r| = 300K$, $|\triangle r|/|r| = 20\%$ by default on FLI. We compare Fetch against IEJoin with the predicate pair (*t.DayOfWeek* < *s.DayOfWeek* ∧ *t.AirlineID* > *s.AirlineID*). Figure 16a shows results by varying $|r|$ from 200 to 400K. Fetch scales better than IEJoin, and the time of Fetch (resp. IEJoin) increases from 3.2 (resp. 88) seconds to 6.8 (resp. 397) seconds. We find the time increase in Fetch is mainly due to more time for collecting results. We vary $|\triangle r|/|r|$ from 15 to 40% in Fig. 16b. Fetch scales well with $|\triangle r|$, and the time increases from 3.4 to 12.8 s, which is almost linear in $|\triangle r|$. IEJoin does not enjoy this feature and takes much longer time.

We then compare Fetch$^+$ against IEJoin by using (*t.OriginStateFips* ≥ *s.OriginStateFips* ∧ *t.AirlineID* < *s.OriginAirportID*). Figure 16c shows Fetch$^+$ is much faster than IEJoin. When $|r| = 400K$, IEJoin takes 658 s while Fetch$^+$ only takes 36 s. We further vary $|\triangle r|$ in Fig. 16d, and see Fetch$^+$ performs much better. As an example, Fetch$^+$ takes 46 s while IEJoin takes 629 s when $|\triangle r|/|r| = 40\%$.

(2) We use Atom with $|r|$ = 100K and $|\triangle r|/|r| = 20\%$ by default. We first compare Fetch against IEJoin with the pair (*t.COL2* < *s.COL2* ∧ *t.COL5* ≥ *s.COL5*). We vary $|r|$ from 80 to 120K in Fig. 16e, and $|\triangle r|/|r|$ from 15 to 40% in Fig. 16f. Fetch scales better than IEJoin with $|r|$. As $|r|$ varies from 80 to 120K, Fetch takes 2.7 to 5.2 s, as opposed to 20 to 50 s by IEJoin. Fetch is about 5 times faster than IEJoin when $|\triangle r|/|r| = 40\%$.

We then compare Fetch$^+$ against IEJoin with the pair (*t.COL2* < *s.COL7* ∧ *t.COL5* ≥ *s.COL12*). The results in Figs. 16g, h show that Fetch$^+$ performs much better than IEJoin.

(3) We set $|r| = 25K$ (resp. $10K$), $|\triangle r|/|r| = 20\%$ on Adult (resp. UCE). In Fig. 16i, we compare Fetch against IEJoin on Adult with 5 predicate pairs, while in Fig. 16j we compare Fetch$^+$ against IEJoin on Adult with 5 predicate pairs concerning inequality comparisons across attributes.[2] Similarly, we report the results on UCE in Figs. 16k, l, respectively.

Recall that Adult and UCE have small $|r|$, and that index visits with $\triangle r$ incur very large result sets on the datasets. This significantly favors IEJoin, since (a) the time of IEJoin depends on $|r|$, and (b) the step of collecting results takes more time, while Fetch (Fetch$^+$) is much faster than IEJoin in the other steps. Nevertheless, we see Fetch (resp. Fetch$^+$) is on average 29% (resp. 12%) and up to 49% (resp. 39%) faster than IEJoin on Adult, and on average 16% (resp. 23%) and up to 42% (resp. 35%) faster than IEJoin on UCE, for the tested predicate pairs.

Exp-5: Fetch *and* Fetch$^+$ *against* FACET We compare Fetch against FACET in the time for computing $T^s(A^{op_1}, B^{op_2})$ and $\overline{T}^s(A^{op_1}, B^{op_2})$, and compare Fetch$^+$ against FACET in the time for computing $T^s(p_1, p_2)$ and $\overline{T}^s(p_1, p_2)$, for all tuples s from $\triangle r$. Recall FACET is designed as a batch (non-incremental) validation method, without considering data updates. Hence, FACET needs to rebuild its auxiliary structures in response to $\triangle r$ and is not designed to only compute results *w.r.t.* $\triangle r$.

(1) We set $|r| = 300K$, $|\triangle r|/|r| = 20\%$ by default on FLI. We compare Fetch against FACET with the predicate pair (*t.DayOfMonth* < *s.DayOfMonth* ∧ *t.AirlineID* > *s.AirlineID*). Figure 17a shows results by varying $|r|$ from 200 to 400K. Fetch scales better than FACET, and the time of Fetch (resp. FACET) increases from 3.7 (resp. 15.4) seconds to 12.1 (resp. 81.3) seconds. We also vary $|\triangle r|/|r|$ from 15 to 40% in Fig. 17b. Fetch scales well with $|\triangle r|$, and the time increases from 5.6 to 15.2 s, which is almost linear in $|\triangle r|$.

We compare Fetch$^+$ against FACET with the pair (*t.OriginStateFips* > *s.OriginStateFips* ∧ *t.AirlineID* < *s.OriginAirportID*). Figure 17c shows Fetch$^+$ is faster than FACET. When $|r| = 400K$, FACET takes 96.3 s while Fetch$^+$ only takes 38 s. We further vary $|\triangle r|$ in Fig. 17d, and see Fetch$^+$ still performs better. As an example, Fetch$^+$ takes 45.8 s while FACET takes 77.5 s when $|\triangle r|/|r| = 40\%$.

(2) We use Atom with $|r|$ = 100K and $|\triangle r|/|r| = 20\%$ by default. We first compare Fetch against FACET with the pair (*t.COL2* < *s.COL2* ∧ *t.COL12* ≥ *s.COL12*). We vary $|r|$

[2] The predicate pairs are not shown because attribute names of Adult and UCE do not have semantic meaning.

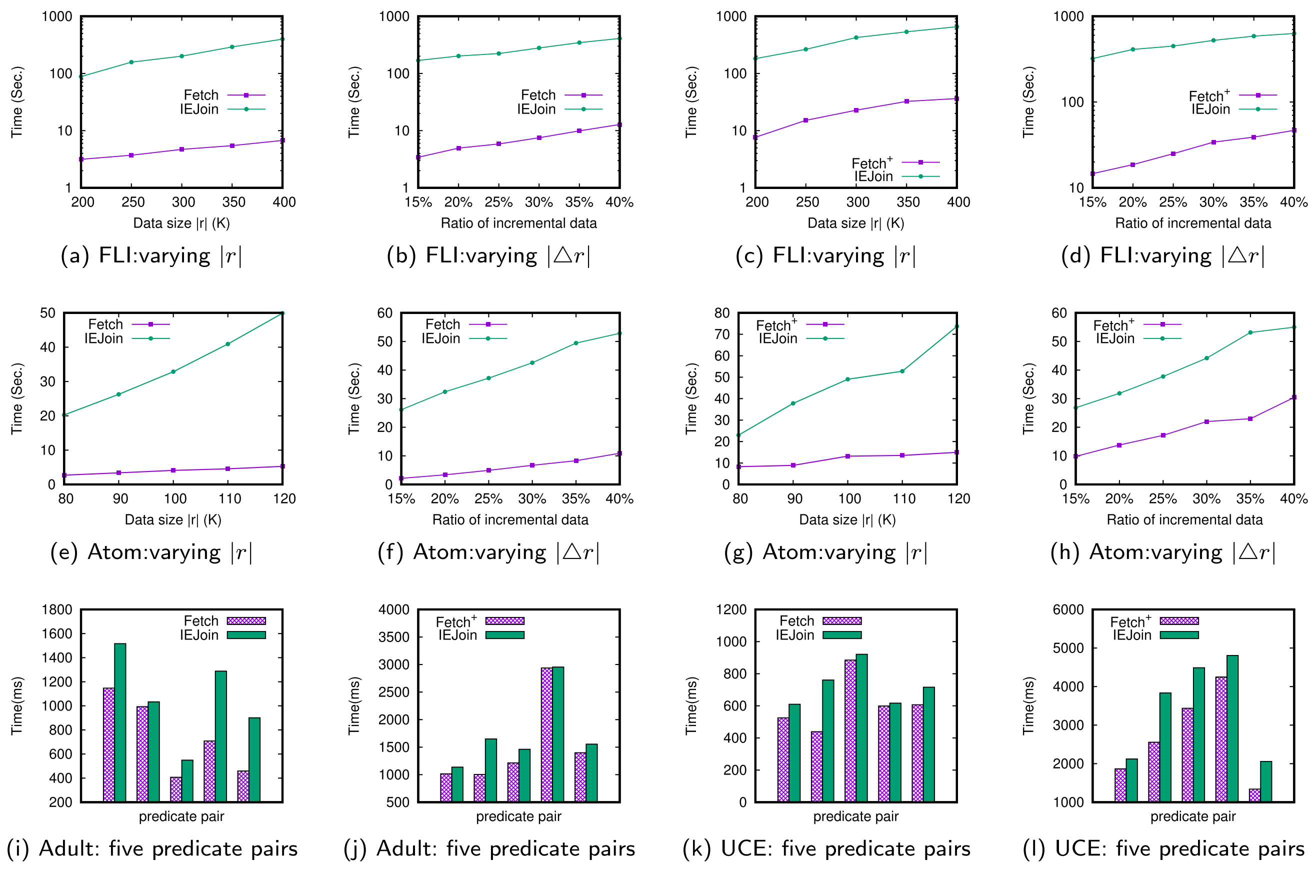


Fig. 16 Fetch and Fetch$^+$ against incremental IEJoin

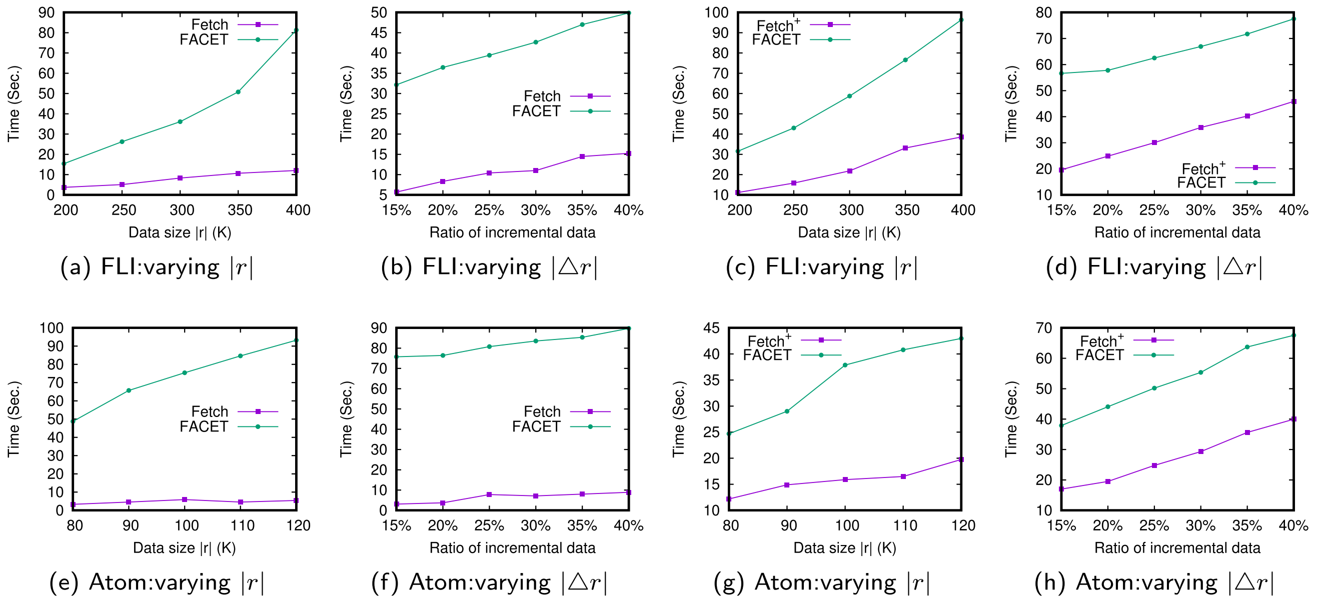


Fig. 17 Fetch and Fetch$^+$ against FACET

from 80 to 120K in Fig. 17e, and $|\triangle r|/|r|$ from 15 to 40% in Fig. 17f. Fetch scales better than FACET with $|r|$. As $|r|$ varies from 80 to 120K, Fetch takes 3.3 to 5.4 s. Fetch is about 9 times faster than FACET when $|\triangle r|/|r| = 40\%$.

We then compare Fetch$^+$ against FACET with the pair ($t.COL2 < s.COL7 \wedge t.COL12 \geq s.COL14$). The results shown in Figs. 17g, h confirm the advantage of Fetch$^+$ over FACET.

10 Conclusions

This is a first effort toward incremental DC discovery aiming at discovering DCs in response to a set of tuple insertions. We have presented an indexing technique for efficiently handling inequality comparisons (possibly) across attributes in response to tuple insertions $\triangle r$, developed algorithms for selecting, building and updating indexes, and given methods to compute $\triangle\Sigma$ based on the incremental evidences and Σ. We have also conducted extensive experiments to verify our approaches.

A couple of topics need further investigation. We are currently studying methods to handle updates with both tuple insertions and deletions, where the need is evident in practice. Another topic is to study incremental discovery techniques for approximate DCs [37,43,69], for dirty and dynamic data. Our methods are designed to run with a centralized setting, and they do not scale to very large databases in production that fail to fit into the main memory of a single machine. To address the scalability issue, we also intend to combine the techniques of incremental dependency discovery with those of distributed dependency discovery [48,49,52] to handle very large dynamic datasets.

Acknowledgements This work is supported by National Natural Science Foundation of China 62172102 and 61925203. We thank authors of [6,32,43] for sharing their codes for our experimental evaluation.

References

1. Abedjan, Z., Golab, L., Naumann, F.: Profiling relational data: a survey. VLDB J. **24**(4), 557–581 (2015)
2. Abedjan, Z., Golab, L., Naumann, F.: Data profiling: a tutorial. In SIGMOD, pp. 1747–1751 (2017)
3. Abedjan, Z., Golab, L., Naumann, F., Papenbrock, T.: Data Profiling. In: Synthesis lectures on data management. Morgan and Claypool Publishers, San Rafael (2018)
4. Abedjan, Z., Quiané-Ruiz, J. A., Naumann, F.: Detecting unique column combinations on dynamic data. In ICDE, pp. 1036–1047 (2014)
5. Birnick, J., Bläsius, T., Friedrich, T., Naumann, F., Papenbrock, T., Schirneck, M.: Hitting set enumeration with partial information for unique column combination discovery. Proc. VLDB Endow. **13**(11), 2270–2283 (2020)
6. Bleifuß, T., Kruse, S., Naumann, F.: Efficient denial constraint discovery with hydra. PVLDB **11**(3), 311–323 (2017)
7. Caruccio, Loredana: Cirillo, Stefano: incremental discovery of imprecise functional dependencies. ACM J. Data Inf. Qual. **12**(4), 19:1-19:25 (2020)
8. Caruccio, L., Cirillo, S., Deufemia, V., Polese, G.: Incremental discovery of functional dependencies with a bit-vector algorithm. In SEBD (2019)
9. Caruccio, L., Deufemia, V., Naumann, F., Polese, G.: Discovering relaxed functional dependencies based on multi-attribute dominance. IEEE Trans. Knowl. Data Eng. **33**(9), 3212–3228 (2021)
10. Caruccio, L., Deufemia, V., Polese, G.: Mining relaxed functional dependencies from data. Data Min. Knowl. Discov. **34**(2), 443–477 (2020)
11. Qi C. Jarek G., Fred K., Cliff Leung, T. T., Linqi Liu, X. Q., and Bernhard Schiefer, K.: Implementation of two semantic query optimization techniques in DB2 universal database. In VLDB, pp. 687–698, (1999)
12. Chu, X., Ilyas, I.F., Papotti, P.: Discovering denial constraints. PVLDB **6**(13), 1498–1509 (2013)
13. Chu, X., Ilyas, I. F., Papotti, P.: Holistic data cleaning: Putting violations into context. In ICDE, 458–469 (2013)
14. Gao C., Wenfei F., Floris G., Xibei J., and Shuai M.: Improving data quality: consistency and accuracy. In VLDB, pp. 315–326, 2007
15. Dallachiesa, Michele, E., Amr, E., Ahmed, E., Ahmed, K., Ilyas, I. F., Ouzzani, M., Tang, N.: Nadeef: a commodity data cleaning system. In SIGMOD, 541–552 (2013)
16. Fan, W., Geerts, F.: Foundations of Data Quality Management. In Synthesis lectures on data management. Morgan and Claypool Publishers, San Rafael (2012)
17. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Conditional functional dependencies for capturing data inconsistencies. ACM Trans. Database Syst. **33**(2), 6:1-6:48 (2008)
18. Fan, W., Chunming, H., Liu, X., Ping, L.: Discovering graph functional dependencies. ACM Trans. Database Syst. **45**(3), 151–1542 (2020)
19. Ge, C., Ilyas, I.F., Kerschbaum, F.: Secure multi-party functional dependency discovery. PVLDB **13**(2), 184–196 (2019)
20. Geerts, F., Mecca, G., Papotti, P., Santoro, D.: Cleaning data with llunatic. VLDB J. **29**(4), 867–892 (2020)
21. Giannakopoulou, S., Karpathiotakis, M., Ailamaki, A.: Cleaning denial constraint violations through relaxation. In SIGMOD, pp. 805–815 (2020)
22. Gilad, A., Deutch, D., Roy, S.: On multiple semantics for declarative database repairs. In SIGMOD, pp. 817–831 (2020)
23. Ginsburg, S., Hull, R.: Order dependency in the relational model. Theor. Comput. Sci. **26**, 149–195 (1983)
24. Ginsburg, S., Hull, R.: Sort sets in the relational model. J. ACM **33**(3), 465–488 (1986)
25. Heise, A., Quiané-Ruiz, J.-A., Abedjan, Z., Jentzsch, A., Naumann, F.: Scalable discovery of unique column combinations. PVLDB **7**(4), 301–312 (2013)
26. Ihab, F.I., Xu, C.: Data Cleaning. ACM, New York City (2019)
27. Jin, Y., Tan, Z., Zeng, W., Ma, S.: Approximate order dependency discovery. In ICDE, pp. 25–36 (2021)
28. Jin, Y., Zhu, L., Tan, Z.: Efficient bidirectional order dependency discovery. In ICDE, pp. 61–72 (2020)
29. Karegar, R., Godfrey, P., Golab, L., Kargar, M., Srivastava, D., Szlichta, J.: Efficient discovery of approximate order dependencies. In EDBT, pp. 427–432 (2021)
30. Khayyat, Z., Ilyas, I. F., Jindal, A., Madden, S., Ouzzani, M., Papotti, P., Quiané-Ruiz, J. A., Tang, N., Yin, S.: Bigdansing: a system for big data cleansing. In SIGMOD, pp. 1215–1230 (2015)
31. Khayyat, Z., Lucia, W., Singh, M., Ouzzani, M., Papotti, P., Quiané-Ruiz, J.-A., Tang, N., Kalnis, P.: Lightning fast and space efficient inequality joins. PVLDB **8**(13), 2074–2085 (2015)
32. Khayyat, Z., Lucia, W., Singh, M., Ouzzani, M., Papotti, P., Quiané-Ruiz, J.-A., Tang, N., Kalnis, P.: Fast and scalable inequality joins. VLDB J. **26**(1), 125–150 (2017)

33. Kossmann, J., Papenbrock, T., Naumann, F.: Data dependencies for query optimization: a survey. VLDB J. **31**(1), 1–22 (2022)
34. Koumarelas, I.K., Naskos, A., Gounaris, A.: Flexible partitioning for selective binary theta-joins in a massively parallel setting. Distributed Parallel Databases **36**(2), 301–337 (2018)
35. Kruse, S., Naumann, F.: Efficient discovery of approximate dependencies. PVLDB **11**(7), 759–772 (2018)
36. Langer, P., Naumann, F.: Efficient order dependency detection. VLDB J. **25**(2), 223–241 (2016)
37. Livshits, E., Heidari, A., Ilyas, I.F., Kimelfeld, B.: Approximate denial constraints. PVLDB **13**(10), 1682–1695 (2020)
38. Ma, S., Fan, W., Bravo, L.: Extending inclusion dependencies with conditions. Theort. Comput. Sci. **515**, 64–95 (2014)
39. Nerone, M. A., Holanda, P., de Almeida, E. C., and Manegold, S.: Multidimensional adaptive and progressive indexes. In ICDE, pp. 624–635, 2021
40. Okcan, A., Riedewald, M.: Processing theta-joins using map reduce. SIGMOD **1**(1), 949–960 (2011)
41. Papenbrock, T., Naumann, F.: A hybrid approach to functional dependency discovery. In SIGMOD, pp. 821–833 (2016)
42. Pena, E. H. M., and de Almeida, E. C. D.: BFASTDC: A bitwise algorithm for mining denial constraints. In DEXA, pp. 53–68, 2018
43. Pena, E.H.M., de Almeida, E.C.D., Felix, N.: Discovery of approximate (and exact) denial constraints. PVLDB **13**(3), 266–278 (2019)
44. Pena, E.H.M., de Almeida, E.C., Felix, N.: Fast detection of denial constraint violations. Proc VLDB Endow **15**(4), 859–871 (2021)
45. Pena, E. H. M., Filho, E. R. L., de Almeida, E. C., and Felix N.: Efficient detection of data dependency violations. In CIKM, pp. 1235–1244, (2020)
46. Pugh, W.: Skip lists: a probabilistic alternative to balanced trees. Commun. ACM **33**(6), 668–676 (1990)
47. Theodoros, R., Xu, C., Ihab, F., Christopher Ré, I.: Holoclean: holistic data repairs with probabilistic inference. Proc VLDB Endow **10**(11), 1190–1201 (2017)
48. Saxena, H., Golab, L., Ilyas, I. F.: Distributed discovery of functional dependencies. In ICDE, pp. 1590–1593 (2019)
49. Saxena, H., Golab, L., Ilyas, I.F.: Distributed implementations of dependency discovery algorithms. Proc. VLDB Endow **12**(11), 1624–1636 (2019)
50. Schirmer, P., Papenbrock, T., Koumarelas, I.K., Naumann, F.: Efficient discovery of matching dependencies. ACM Trans. Database Syst. **45**(3), 13:1-13:33 (2020)
51. Schirmer, P., Papenbrock, T., Kruse, S., Naumann, F., Hempfing, D., Mayer, T., Neuschäfer-Rube, D.: Dynfd: functional dependency discovery in dynamic datasets. In EDBT, pp. 253–264 (2019)
52. Schmidl, S., Papenbrock, T.: Efficient distributed discovery of bidirectional order dependencies. VLDB J. **31**(1), 49–74 (2022)
53. Shaabani, N., Meinel, C.: Incrementally updating unary inclusion dependencies in dynamic data. Distrib. Parallel Databases **37**(1), 133–176 (2019)
54. Simmen, D. E., Shekita, E. J., Malkemus, T.: Fundamental techniques for order optimization. In SIGMOD, pp. 57–67 (1996)
55. Song, S., Chen, L.: Discovering matching dependencies. In CIKM, pp. 1421–1424 (2009)
56. Song, S., Chen, L.: Efficient discovery of similarity constraints for matching dependencies. Data Knowl. Eng. **87**, 146–166 (2013)
57. Song, S., Gao, F., Huang, R., Wang, C.: Data dependencies extended for variety and veracity: A family tree. IEEE Trans. Knowl. Data Eng. **34**(10), 4717–4736 (2022)
58. Szlichta, J., Godfrey, P., Golab, L., Kargar, M., Srivastava, D.: Effective and complete discovery of order dependencies via set-based axiomatization. PVLDB **10**(7), 721–732 (2017)
59. Szlichta, J., Godfrey, P., Golab, L., Kargar, M., Srivastava, D.: Effective and complete discovery of bidirectional order dependencies via set-based axioms. VLDB J. **27**(4), 573–591 (2018)
60. Szlichta, J., Godfrey, P., Gryz, J.: Fundamentals of order dependencies. PVLDB **5**(11), 1220–1231 (2012)
61. Szlichta, J., Godfrey, P., Gryz, J., Ma, W., Qiu, W., Zuzarte, C.: Business-intelligence queries with order dependencies in DB2. In EDBT, pp. 750–761 (2014)
62. Szlichta, J., Godfrey, P., Gryz, J., Zuzarte, C.: Expressiveness and complexity of order dependencies. PVLDB **6**(14), 1858–1869 (2013)
63. Tan, Z., Ran, A., Ma, S., Qin, S.: Fast incremental discovery of pointwise order dependencies. PVLDB **13**(10), 1669–1681 (2020)
64. Tschirschnitz, F., Papenbrock, T., Naumann, F.: Detecting inclusion dependencies on very many tables. ACM Trans. Database Syst. **42**(3), 18:1-18:29 (2017)
65. Vazirani, V.V.: Approximation algorithms. Springer, Heidelberg (2001)
66. Wei, Z., Hartmann, S., Link, S.: Algorithms for the discovery of embedded functional dependencies. VLDB J. **30**(6), 1069–1093 (2021)
67. Wei, Z., Link, S.: Discovery and ranking of functional dependencies. In ICDE, pp. 1526–1537 (2019)
68. Weise, J., Schmidl, S., Papenbrock, T.: Optimized theta-join processing through candidate pruning and workload distribution. In BTW, pp. 59–78 (2021)
69. Xiao, R., Tan, Z., Wang, H., Ma, S.: Fast approximate denial constraint discovery. Proc. VLDB Endow. **16**(2), 269–281 (2022)
70. Xiao, R., Yuan, Y., Tan, Z., Ma, S., Wang, W.: Dynamic functional dependency discovery with dynamic hitting set enumeration. In ICDE, pp. 286–298 (2022)
71. Lin Z., Xu, S., Zijing T., Yang, K., Yang, W., Zhou, X., Tian, Y.: Incremental discovery of order dependencies on tuple insertions. In DASFAA, pp. 157–174 (2019)