Discovering Denial Constraints in Dynamic Datasets

Eduardo H. M. Pena
Federal University of Technology - Paraná
Campo Mourão-PR, Brazil
eduardopena@utfpr.edu.br

Fabio Porto
LNCC-DEXL
Petrópolis-RJ, Brazil
fporto@lncc.br

Felix Naumann
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
felix.naumann@hpi.de

***Abstract*—Denial constraints (DCs) are data dependencies with high expressive power, offering great flexibility for modeling data quality rules. Specifying DCs manually is problematic, as the required domain expertise is expensive and scarce. Moreover, database updates can invalidate DCs thought to hold and simultaneously uncover new DCs. This fact leads to burdensome scenarios where experts must often revisit DC specifications. Several algorithms have been devised to discover DCs from data, among which only one considers DC discovery on data updates. However, that solution underperforms in many scenarios due to long runtime and excessive memory use. Also, it targets database inserts only, so no previous solution covers deletions.**

This paper proposes an efficient and flexible algorithm that covers the earlier limitations regarding performance and scope. The algorithm maintains small-footprint intermediate structures during database updates and a method that exploits the changes in this intermediate to update the DCs incrementally. The results of our extensive experimental evaluation show that our algorithm is orders of magnitude faster than the existing one, with much better scalability in the size of the data updates.

***Index Terms*—Denial constraints, data profiling, data cleaning**

I. Denial Constraints in Dynamic Data

Data profiling refers to processes for discovering metadata, i.e., structures and relationships that help understand and manage data. Data dependencies are one of the most important types of metadata in data management, as they support critical tasks, such as data cleaning [5], database design [10], integration [1], and query optimization [6].

Most dependency discovery algorithms are *static*: they were designed to process given datasets, presumably with no updates. On the other hand, *dynamic* (aka. *incremental*) algorithms have been proposed for the case where the database receives data updates regularly [15]–[17]. The general idea is to evolve dependencies instead of recomputing them from scratch after every change. This paper presents a dynamic algorithm for discovering denial constraints (DCs), a powerful type of data dependency that generalizes keys, functional dependencies, and order dependencies while capturing further constraints that simpler dependencies cannot.

DCs specify relational predicates to indicate potentially inconsistent value combinations. Let φ denote a DC of the form $\varphi\colon \forall \mathtt{t}, \mathtt{t}' \in \mathtt{r}, \neg(\mathtt{p}_1 \wedge \ldots \wedge \mathtt{p}_m)$, where $\mathtt{p}_i$ is a predicate over tuples $\mathtt{t}$ and $\mathtt{t}'$ of a relation instance $\mathtt{r}$. A DC φ is valid in $\mathtt{r}$ if no pair of tuples $(\mathtt{t}, \mathtt{t}')$ satisfies every predicate of φ.

DC discovery algorithms target *minimal* DCs. A DC φ is minimal if there does not exist another DC φ' with a predicate set that is implied by that of φ. For clarity, we defer the remaining DC definitions to Section III.

TABLE I: The `staff` relation under database updates.

		Id	Name	Hired	Level	Mgr
initial	$\mathtt{t}_1$	#1	Ana	2000	5	#1
	$\mathtt{t}_2$	#2	Sam	2001	4	#1
	$\mathtt{t}_3$	#3	Ana	2001	2	#2
	$\mathtt{t}_4$	#4	Kai	2002	2	#2
insert	$\mathtt{t}_5$	#5	Ema	2002	3	#1
delete	~~$\mathtt{t}_4$~~	~~#4~~	~~Kai~~	~~2002~~	~~2~~	~~#2~~

Consider the "initial" (gray) part of the `staff` relation in Table I. Observe that the following constraints are valid: (1) "the `Id` column is a key"; (2) "the staff level determines who their managers are"; (3) "the order of the hire dates ranks the order of the levels"; (4) "staff cannot have levels that are higher than that of their managers." These constraints can be expressed as the following DCs (we omit the tuple and relation quantifiers in the DCs from now on):

$$\varphi_1\colon \neg(\mathtt{t.Id} = \mathtt{t}'.\mathtt{Id})$$
$$\varphi_2\colon \neg(\mathtt{t.Level} = \mathtt{t}'.\mathtt{Level} \wedge \mathtt{t.Mgr} \neq \mathtt{t}'.\mathtt{Mgr})$$
$$\varphi_3\colon \neg(\mathtt{t.Hired} < \mathtt{t}'.\mathtt{Hired} \wedge \mathtt{t.Level} < \mathtt{t}'.\mathtt{Level})$$
$$\varphi_4\colon \neg(\mathtt{t.Mgr} = \mathtt{t}'.\mathtt{Id} \wedge \mathtt{t.Level} > \mathtt{t}'.\mathtt{Level})$$

Tuple insertions may violate the DCs that are valid for this database state, whereas tuple deletions may remove violations of latent DCs that were not valid. As a result, the set of valid DCs may change to accommodate the violations (or their removal thereof) from dynamic data. For instance, the insertion of tuple $\mathtt{t}_5$ in Table I causes the violation of the DC φ_3 due to the inconsistency between tuples $\mathtt{t}_3$ and $\mathtt{t}_5$. After the update, a new DC holds: "The order of the hire dates ranks the order of the levels within staff having the same manager." This constraint, an order dependency, is expressed as:

$$\varphi_5\colon \neg(\mathtt{t.Mgr} = \mathtt{t}'.\mathtt{Mgr} \wedge \mathtt{t.Hired} < \mathtt{t}'.\mathtt{Hired} \wedge \mathtt{t.Level} < \mathtt{t}'.\mathtt{Level}),$$

which is a direct evolution of the DC φ_3. Mind that the DC φ_5 was also valid in the "initial" data, but was not minimal.

Consider the subsequent deletion of tuple $\mathtt{t}_4$ in Table I. Notice that DC φ_2 becomes non-minimal after the deletion,

and the following (minimal) DC emerges, which is as an (accidental) regression of φ_2:

$$\varphi_6\colon \neg(\mathtt{t.Level} = \mathtt{t'.Level})$$

Executing static algorithms after every database update is inefficient: DC discovery is computationally expensive since there is an exponential number of DC candidates. Instead, we propose the Dynamic Discoverer of Denial Constraints (3DC), an algorithm that efficiently computes the set of valid DCs for tables under database inserts and deletes.

A central intuition in 3DC is leveraging "intermediates" from the previous DC discovery. To compute these intermediates quickly, we use specialized indexing and algorithms to handle DC predicates and inference strategies that enable expanding the intermediates for the incremental tuples. To enumerate the DCs from the intermediates, we extend a previous algorithm. This extension needs to consider only the new information from the dynamic data; thus, it usually runs in a fraction of the time compared to the static counterpart.

The only previous solution for dynamic DC discovery is IncDC [15], which uses complex indexes that cover all the DCs from the static part of the data. Since there are many DCs, IncDC incurs a very high memory footprint and many operations on the indexes (probes and updates). Furthermore, IncDC lacks support for tuple deletions, which is common in real-world workloads. In response, our designs in 3DC avoid the performance pitfalls of IncDC and seamlessly accommodate tuple deletions. As a result, we present the first algorithm for dynamic DC discovery in scenarios involving both insertions and deletions. In summary, the contributions of this work are as follows:

- *Flexible DC maintenance.* We present the 3DC algorithm for incorporating data changes into the set of minimal denial constraints valid in a table. We present specialized strategies for handling inserts and deletes, while updates can be modeled as a delete followed by an insert.
- *Detailed evaluation.* We provide an exhaustive evaluation of 3DC on several (real and synthetic) datasets. Our results show that 3DC is orders of magnitude faster than the existing algorithm for dynamic DC discovery.

II. Related Work

Static DC discovery. The first (static) algorithm for DC discovery was the two-phase algorithm FastDC [4]. First, it examines every tuple pair in the input to populate an auxiliary data structure called the *evidence set*. Then, it enumerates DC candidates with a depth-first traversal of the search space, checking DC candidates on the evidence set. Other approaches have adopted this two-phase approach, presenting significant improvements for the individual phases.

For evidence-set building, the algorithm in [3] builds an intermediate evidence set from tuple samples to derive initial DCs. Then, by identifying violations of these DCs, it computes the missing parts of the evidence set. The algorithm in [12] proposes bit-level operations and indexes that avoid excessive tuple pair comparisons. The idea is extended in [11]: based on predicate selectivity, parts of the evidence set are pre-allocated close to their correct state. This last strategy has also been used in [7]. Moreover, it is extended in [18] with a condensed structure having a smaller memory footprint. Finally, the approach in [14] uses a structure that explores data redundancy for faster evidence set building.

For DC enumeration, the algorithms in [11], [12] use the depth-first traversal of FastDC. In turn, [3] proposes the *evidence inversion*: it assumes that DCs with a single predicate are valid. Then, it iterates the evidence, incorporating predicates into the DCs violated by that evidence. The DCs are adjusted with each iteration until the set of DCs is complete and sound. Taking a different direction, the authors in [7] adapted the hitting set enumeration algorithm of [8] for DC enumeration. More recently, the authors in [14] show that all enumeration algorithms perform reasonably well when implemented with proper data structures and optimizations.

DC Ranking. DC discovery often returns many DCs (e.g., thousands or more, even for small datasets). That is expected since DCs have high expressive power, subsuming other dependency types that already entail many results.

To help users explore the results and select *relevant* DCs, *scoring functions for DC ranking* have been proposed [4], [11]. These functions consider the constraint length (e.g., number of symbols within the predicates) and *data coverage* (measures that estimate data support of a constraint). While obtaining DC length is straightforward, obtaining data coverage requires the computation of additional statistics. These statics are the *evidence multiplicity*, which can be computed during evidence-set building. The algorithms in [4], [7], [11], [12], [14], [18] can naturally accommodate this computation. In contrast, the one in [3] cannot: as it leaps the search space, it skips most of the redundancy used to derive the multiplicity.

Qualitative evaluations of DC ranking have been performed in [4], [7], [11]. The findings emphasize the importance of scoring functions based on data coverage. We have intentionally designed 3DC to provide evidence multiplicity so it can support the scoring functions from these previous works.

Approximate DCs. When the input data contain errors, attempting to discover *exact DCs*, which hold across the entire dataset, may lead to overfitting because the DCs have to account for the errors. An alternative is to relax the definition of DCs and discover *approximate* (aka *partial*) *DCs* [4], [7], [11]. Such constraints are guaranteed to "almost" hold, as they must be satisfied by most of the data, given a threshold.

The discovery of approximate DCs has been studied in [4], [7], [11], [12], [18]. The current algorithms are, in essence, extensions of algorithms for enumerating exact DCs, and all require evidence multiplicity to work. As mentioned, 3DC can provide the evidence multiplicity. Thus, it can support approximate DCs. However, to maintain a more focused presentation, we concentrate on the exact DC enumeration in this paper. We defer a more comprehensive evaluation of approximate DC discovery in dynamic scenarios for future research.

Dynamic DC discovery. The only other algorithm for dynamic DC discovery is IncDC, which focuses on discovering exact DCs under tuple insertions [15]. The algorithm takes as input a relation instance, the set Σ of DCs valid in that instance, and a set of tuple insertions. The algorithm builds a set of indexes specialized in DC predicates that covers all DCs in Σ. By probing and updating these indexes, IncDC finds the pairs of tuples that violate Σ, which point to new evidence. Then, by leveraging the updated evidence set and set Σ, IncDC identifies and handles the DCs that need to be removed or updated in Σ to form a set Σ', the set of DCs valid in the (updated) instance.

As verified in our experiments (Section VII), the performance of IncDC is significantly impaired with large sets Σ (typical for many datasets). Moreover, the memory footprint incurred by its indexing scheme is often huge, even for small datasets, as many indexes are required to cover all DCs in Σ. IncDC is similar to the algorithm in [3] in that it cannot be trivially adapted to compute evidence multiplicity, therefore lacking the ability to compute data coverage for DC ranking or enumerating approximate DCs. Also, the algorithm does not target tuple deletions, which are common in production.

III. Preliminaries

A. Notations and basic definitions

1) Notations: We use standard conventions from related work to express DC predicates [3], [7], [11]. We consider a relation instance $\mathtt{r}$ of schema $\mathtt{R}$, tuples $\mathtt{t}, \mathtt{t}' \in \mathtt{r}$, and columns $\mathtt{A}, \mathtt{B}, \ldots \in \mathtt{R}$. A DC predicate has the form $\mathtt{p}\colon \mathtt{t.A}\ \theta\ \mathtt{t}'.\mathtt{B}$ (or $\mathtt{p}\colon \mathtt{t.A}\ \theta\ \mathtt{t}'.\mathtt{A}$), where $\mathtt{t.A}$ is the column value $\mathtt{A}$ in tuple $\mathtt{t}$ (the same for $\mathtt{t}'.\mathtt{B}$), and $\theta \in \{=, \neq, <, \leq, >, \geq\}$ is a comparison operator. Like [3], [7], [11], [14], [18], we focus on predicates over two distinct tuples ($\mathtt{t} \neq \mathtt{t}'$).

2) Semantics: From the definition in Section I, a DC $\varphi\colon \neg(\mathtt{p}_1 \wedge \ldots \wedge \mathtt{p}_m)$ states that at least one of the predicates of φ must be false for every pair of tuples $\mathtt{t}, \mathtt{t}'$ of the instance $\mathtt{r}$. Any pair $\mathtt{t}, \mathtt{t}'$ that satisfies every predicate of a DC φ is considered a *violation* of φ.

3) Minimal (non-trivial) DCs: To avoid redundant and uninteresting DCs, discovery targets minimal DCs (defined in Section I) and *non-trivial* DCs. A DC is trivial if any relation instance satisfies it. For instance, the DC $\varphi_7\colon \neg(\mathtt{t.Id} = \mathtt{t}'.\mathtt{Id} \wedge \mathtt{t.Id} \neq \mathtt{t}'.\mathtt{Id})$ is a trivial DC.

4) Predicate space: The predicate space $\mathtt{P}$ of an instance $\mathtt{r}$ is the set of all possible predicates allowed to express DCs on $\mathtt{r}$. Any subset of $\mathtt{P}$ is a DC candidate, so the bigger the space $\mathtt{P}$, the larger the DC search space.

The authors of [4] show that some predicates hardly contribute to relevant DCs, and propose the following restrictions. The operator set for predicates on categorical columns uses only the operators ($=$, $\neq$), whereas for numeric columns, it uses the operators $\{=, \neq, <, \leq, >, \geq\}$. Predicates on two different columns require the columns to be of the same data type and share a percentage of common values (30% has been shown to work well in practice [4]). Like all existing DC discovery algorithms [4], [7], [11], [12], [14], [18] we adopt these restrictions. Figure 1 illustrates a sample predicate space for staff.

$\mathtt{p}_1 : \mathtt{t.ID} = \mathtt{t}'.\mathtt{ID}$	$\mathtt{p}_2 : \mathtt{t.ID} \neq \mathtt{t}'.\mathtt{ID}$
$\mathtt{p}_3 : \mathtt{t.Name} = \mathtt{t}'.\mathtt{Name}$	$\mathtt{p}_4 : \mathtt{t.Name} \neq \mathtt{t}'.\mathtt{Name}$
$\mathtt{p}_5 : \mathtt{t.Hired} = \mathtt{t}'.\mathtt{Hired}$	$\mathtt{p}_6 : \mathtt{t.Hired} \neq \mathtt{t}'.\mathtt{Hired}$
$\mathtt{p}_7 : \mathtt{t.Hired} < \mathtt{t}'.\mathtt{Hired}$	$\mathtt{p}_8 : \mathtt{t.Hired} \leq \mathtt{t}'.\mathtt{Hired}$
$\mathtt{p}_9 : \mathtt{t.Hired} > \mathtt{t}'.\mathtt{Hired}$	$\mathtt{p}_{10} : \mathtt{t.Hired} \geq \mathtt{t}'.\mathtt{Hired}$
$\mathtt{p}_{11} : \mathtt{t.Level} = \mathtt{t}'.\mathtt{Level}$	$\mathtt{p}_{12} : \mathtt{t.Level} \neq \mathtt{t}'.\mathtt{Level}$
$\mathtt{p}_{13} : \mathtt{t.Level} < \mathtt{t}'.\mathtt{Level}$	$\mathtt{p}_{14} : \mathtt{t.Level} \leq \mathtt{t}'.\mathtt{Level}$
$\mathtt{p}_{15} : \mathtt{t.Level} > \mathtt{t}'.\mathtt{Level}$	$\mathtt{p}_{16} : \mathtt{t.Level} \geq \mathtt{t}'.\mathtt{Level}$
$\mathtt{p}_{17} : \mathtt{t.Mgr} = \mathtt{t}'.\mathtt{Mgr}$	$\mathtt{p}_{18} : \mathtt{t.Mgr} \neq \mathtt{t}'.\mathtt{Mgr}$
$\mathtt{p}_{19} : \mathtt{t.Mgr} = \mathtt{t}'.\mathtt{ID}$	$\mathtt{p}_{20} : \mathtt{t.Mgr} \neq \mathtt{t}'.\mathtt{ID}$

Fig. 1: A sample predicate space for the staff relation.

The predicate space can also include predicates with constants, for example, a predicate like $\mathtt{t.Hired} > 2000$. Including such predicates enables the expression of *conditional DCs* [4], which is not the focus of this paper.

5) Evidence set: The *evidence* $\mathtt{e}(\mathtt{t}, \mathtt{t}')$ is the subset of the predicate space satisfied by a pair of tuples $\mathtt{t}, \mathtt{t}'$ of $\mathtt{r}$, more formally: $\mathtt{e}(\mathtt{t}, \mathtt{t}') = \{\mathtt{p} \mid \mathtt{p} \in \mathtt{P} \text{ and } (\mathtt{t}, \mathtt{t}') \text{ satisfies } \mathtt{p}\}$. In this context, the *evidence set* $\mathtt{E}_\mathtt{r}$ is the set of evidence from every pair of tuples of $\mathtt{r}$, that is, $\mathtt{E}_\mathtt{r} = \{\mathtt{e}(\mathtt{t}, \mathtt{t}') \mid \mathtt{t}, \mathtt{t}' \in \mathtt{r}\}$. This set helps to validate DCs quickly, as they represent the predicate satisfaction regarding the entire table.

6) Minimal cover: As shown in [3], [4], a DC $\varphi\colon \neg(\mathtt{p}_1 \wedge \ldots \wedge \mathtt{p}_m)$ is valid in an instance $\mathtt{r}$ if $\nexists\ \mathtt{e} \in \mathtt{E}_\mathtt{r}\colon \{\mathtt{p}_1 \wedge \ldots \wedge \mathtt{p}_m\} \subseteq \mathtt{e}$. In other words, a DC φ is valid in $\mathtt{r}$ if no pair of tuples in $\mathtt{r}$ satisfies every predicate of φ, i.e., there are no violations of φ in $\mathtt{r}$. Thus, it is possible to check the evidence set for the nonexistence of violations. DC discovery algorithms use this intuition for discovering *minimal set covers* (and non-covers) of evidence sets [3], [4]. Similarly, *hitting set enumeration* algorithms can also be used due to the equivalence between set covers and hitting sets [7], [14].

7) Evidence multiplicity: The *evidence multiplicity* $count(\mathtt{e})$ of each (distinct) evidence $\mathtt{e}$ is given by the number of tuple pairs that produce $\mathtt{e}$. As discussed in Section II, this statistic is useful for DC ranking and approximate DC discovery. Rankings can use *data coverage*, which measures how many tuple pairs satisfy strict subsets of the predicates of a DC [4]. This value can be computed directly from the evidence multiplicity. For approximate DCs, it is possible to enumerate *approximate minimal covers* of the evidence set [4]. The approximation of each candidate cover is based on the evidence multiplicity as the statistic indicates how many tuple pairs satisfy a given predicate set and, thus, can quantify the number of DC violations.

B. Dynamic DC discovery problem

Given a relation instance $\mathtt{r}$, a set $\Delta\mathtt{r}$ of tuples, and the set Σ of DCs that hold in $\mathtt{r}$, the dynamic DC discovery problem is to determine the set Σ' of DCs that hold in $\mathtt{r} \oplus \Delta\mathtt{r}$, where $\oplus$ represents a database update (tuple inserts, deletes, or modifications). For practical purposes, we propose algorithms for the insert and delete cases (i.e., $\mathtt{r} \cup \Delta\mathtt{r}$ and $\mathtt{r} \setminus \Delta\mathtt{r}$,

respectively), as we can split mixed updates into a set of deletes followed by a set of inserts. For the case of inserts, we consider the set of tuples in $\mathtt{r}$ and $\Delta\mathtt{r}$ to be disjoint, whereas, in the case of deletes, we assume $\Delta\mathtt{r} \subseteq \mathtt{r}$.

The complexity of the dynamic DC discovery problem depends on the size of the changes from Σ' to Σ. It can be modeled based on the number of violations of Σ concerning the incremental data $\Delta\mathtt{r}$, and subsequently, in terms of the costs involved in leveraging these violations to transform Σ into Σ' [15]. In the worst-case scenario, there can be an exponential number of new DCs for each violated DC. A similar principle applies for deletes, where the elimination of a violation may lead to the expansion of a DC into an exponential number of candidates.

IV. Overview

Figure 2 provides an overview of 3DC. The algorithm takes as input the set Σ of DCs and the evidence set $\mathtt{E}$ computed in the prior DC discovery; the "static" data from the previous instance state; and the update (deletes or inserts). The algorithm returns the set of DCs that are valid in the instance after the update has been applied (depicted as the Σ', Σ'' symbols), as well as the updated evidence set (the $\mathtt{E}', \mathtt{E}''$ symbols), which are used by 3DC in a subsequent incremental DC discovery. Any static algorithm can feed a first call to the 3DC algorithm if it can compute the evidence set.

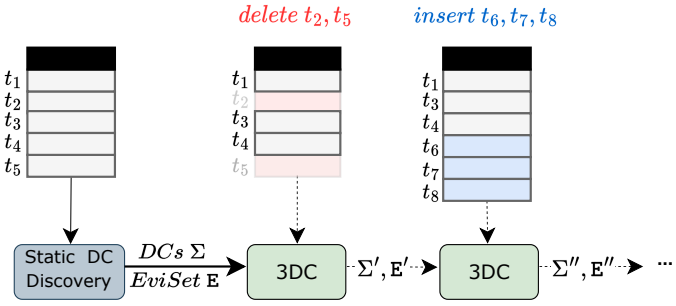


Fig. 2: Operation of 3DC.

Database updates may alter the set of valid minimal DCs in a relation. We rely on two steps to identify these changes quickly: (i) monitoring changes in the evidence set and (ii) translating these changes into the changes in the DC set.

Our reasoning for the first step is that monitoring changes in the evidence set is less costly than directly monitoring changes in the DCs (the approach used in related work [15]). The number of valid DCs is usually high in many datasets, and even a minor update can alter a significant portion of them. On the other hand, the evidence set is more stable across data updates. Inserts can add new evidence, but cannot remove or alter them (element-wise). During our experiments, we observed that inserts often incur only a minor growth in the number of evidences. As a result, new evidence arises less frequently after the datasets reach a certain size because of the considerable redundancy often present in the evidence set. A similar intuition holds for deletes, which can only remove evidence but not add or alter them.

The evidence-set building process is the most time-consuming part of DC discovery for many datasets [3], [4]. Our algorithm leverages the evidence set computed in previous discovery iterations for efficiency, eliminating the need to rerun the entire process. Additionally, 3DC leverages indexes built on the entire (updated) dataset that facilitates the efficient computation of the incremental evidence (Section V).

The second step leverages the changes in the evidence set to update the set of valid DCs. Several approaches exist to enumerate DCs from the evidence set. We observed that running the entire enumeration from scratch can be done fast for several datasets (e.g., in a matter of seconds or less). Still, for better efficiency, we developed an extension of the Hydra algorithm [3], which can significantly reduce recomputation efforts in dynamic settings. The central idea is to process only evidence set increments instead of the entire one (Section VI).

V. Evidence Set Maintenance

A. Fundamental principles and data structures

Our algorithms for evidence sets are inspired by the *evidence presumption* principle of [11]. The basic idea is to initialize evidence closer to its correct state based on *predicate selectivity*, and then use a *reconciliation mechanism* to correct the faulty parts of that evidence.

Predicate spaces typically contain many predicates of low selectivity, which are satisfied by many tuple pairs. Consider a column $\mathtt{A}$ of a relation instance $\mathtt{r}$, and $\mathtt{f}(k)$ the frequency of a column value k of the domain of $\mathtt{A}$. For simplicity, assume that every value k appears more than once and that $\mathtt{A}$ is not a single value column, i.e., $1 < \mathtt{f}(k) < n$. Notice that $\sum_k \mathtt{f}(k)^2$ pairs of tuples satisfy a predicate of the form $\mathtt{p}\colon \mathtt{t.A} = \mathtt{t'.A}$, whereas $\sum_k \mathtt{f}(k) \cdot (|\mathtt{r}| - \mathtt{f}(k))$ pairs of tuples satisfy the "different than" counterpart $\mathtt{p'}\colon \mathtt{t.A} \neq \mathtt{t'.A}$. These formulas show that, between most of the typical equalities and inequalities in predicate spaces, the selectivity of the inequality is much lower. For instance, all tuple pairs from Table I satisfy the predicate $\mathtt{p}_2 : \mathtt{t.ID} \neq \mathtt{t'.ID}$, but none satisfy $\mathtt{p}_1 : \mathtt{t.ID} = \mathtt{t'.ID}$.

Based on this selectivity principle, we instantiate any new evidence with a set of low-selectivity predicates, $ahead = \{\mathtt{p} \mid \mathtt{p} \in \mathtt{P}$ and $\mathtt{p}.\theta \in \{\neq, >, \geq\}\}$. Since many tuple pairs satisfy several predicates of $ahead$, we already start with partially correct evidence. Mind that, although we consider operators $\{\neq, >, \geq\}$ in $ahead$, the computation required for the other direction, $\{\neq, <, \leq\}$, would be equivalent since they reflect the same number of tuple pairs.

The challenge within the evidence presumption principle is reconciling evidence for the complements of $ahead$, that is, correct evidence for tuple pairs that satisfy predicates with operators $\{\neq, <, \leq\}$ or/and $\{=, \leq, \geq\}$. Evidence reconciliation was proposed in [11] and adapted in [14], [18]. In [11], [18], the evidence of all tuple pairs needs to be visited individually after the reconciliation, which can be cost-prohibitive. Instead, we reconcile evidence using the *evidence context* data structure

from [14]. It helps to avoid visiting individual evidence, as it captures the redundancy caused by the many tuple pairs that yield the same evidence.

We represent evidence contexts as mappings $(\mathtt{t}, \mathtt{rids}) \mapsto \mathtt{e}$, where $\mathtt{t}$ is a tuple of the instance $\mathtt{r}$, $\mathtt{rids}$ is a set of tuples of $\mathtt{r}$, and $\mathtt{e}$ is the evidence associated with the pair $(\mathtt{t}, \mathtt{rids})$. Within an evidence context $(\mathtt{t}, \mathtt{rids}) \mapsto \mathtt{e}$, any pair of tuples such that $\mathtt{t}, \mathtt{t}' \in \mathtt{r}$ and $\mathtt{t}' \in \mathtt{rids}$ yield the evidence $\mathtt{e}$. In our example, both tuple pairs $(\mathtt{t}_1, \mathtt{t}_2)$ and $(\mathtt{t}_1, \mathtt{t}_5)$ yield the same evidence, hence an evidence context $(\mathtt{t}_1, \{\mathtt{t}_2, \mathtt{t}_5\}) \mapsto \{\mathtt{p}_2, \mathtt{p}_4, \mathtt{p}_6, \mathtt{p}_7, \mathtt{p}_8, \mathtt{p}_{12}, \mathtt{p}_{15}, \mathtt{p}_{16}, \mathtt{p}_{17}, \mathtt{p}_{20}\}$. Naturally, such redundancy is much higher in larger datasets, which enables processing evidence from many tuples compactly.

Thus, instead of reconciling the evidence directly, we reconcile evidence contexts. We use a pipeline where each stage reconciles evidence for a *predicate group*: a subset of predicates that differ only by the relational operator. For instance, the predicate groups for the predicate space in Figure 1 are as follows: $G_1 = \{\mathtt{p}_1, \mathtt{p}_2\}$, $G_2 = \{\mathtt{p}_3, \mathtt{p}_4\}$, $G_3 = \{\mathtt{p}_5, \ldots, \mathtt{p}_{10}\}$, $G_4 = \{\mathtt{p}_{11}, \ldots, \mathtt{p}_{16}\}$, $G_5 = \{\mathtt{p}_{17}, \mathtt{p}_{18}\}$, and $G_6 = \{\mathtt{p}_{19}, \mathtt{p}_{20}\}$.

The reconciliation adjusts the tuples in the right-hand side of a context' key and its mapped evidence, potentially producing new evidence contexts and eliminating invalid ones. We use the algorithms from [14] for evidence context reconciliation, as they cover various predicate structures of DCs. In a nutshell, the algorithms are based on column indexing and logical operations on compressed bitmaps.

B. Maintaining evidence sets on inserts

From the definitions in Section III, we observe that the evidence set of a relation instance $\mathtt{r}$ determines its DCs. Notice that evidence set $\mathtt{E}_\mathtt{r}$ already holds the evidence from all tuple pairs $\mathtt{t}, \mathtt{t}' \in \mathtt{r}$. Thus, we must compute the incremental evidence set $\mathtt{E}_{\Delta\mathtt{r}}$ from the pairs $\mathtt{t}, \mathtt{t}'$ where $\mathtt{t} \in \Delta\mathtt{r}$ and $\mathtt{t}' \in \mathtt{r}$. The incremental evidence must also regard the symmetric pair $\mathtt{t}', \mathtt{t}$. We propose to use reflexivity and symmetry properties of predicates to infer this latter information. Generally speaking, this intuition can be applied to all existing evidence set-building approaches, e.g., the tuple pair enumeration of `FastDC`. This paper focuses on the evidence context approach. Algorithm 1 shows the procedure to compute the incremental evidence set $\mathtt{E}_{\Delta\mathtt{r}}$.

1) Incremental indexes: We first build equality indexes on the columns of the updated table (Line 1). The indexes consider the entire table as they shape the evidence for each tuple of the insert regarding every other tuple in the entire table. We implement each index as hash maps—similar to the *position list indexes* used in functional dependency discovery [9].

Specifically, for each column $\mathtt{A} \in \mathtt{R}$, we build a hash map that maps each value v of $\mathtt{A}$ to the set of tuples with that value v in $\mathtt{A}$. For instance, for the "initial+insert" part of the `staff` relation, the index on column `Hired` is as follows: $2000 \mapsto \{\mathtt{t}_1\}$, $2001 \mapsto \{\mathtt{t}_2, \mathtt{t}_3\}$, and $2002 \mapsto \{\mathtt{t}_4, \mathtt{t}_5\}$. These maps are employed later in the algorithm for reconciling evidence satisfying predicates with operators $\{=, \leq, \geq\}$.

Algorithm 1: Incremental evidence set building

Input: Relation instance $\mathtt{r}$ of schema $\mathtt{R}$, predicate space $\mathtt{P}$, evidence-set $\mathtt{E}_\mathtt{r}$, set $\Delta\mathtt{r}$ of new tuples
Output: Incremental evidence set $\mathtt{E}_{\Delta\mathtt{r}}$

```
1  Build indexes for columns A ∈ R over r ∪ Δr
2  E_Δr ← ∅
3  foreach tuple t ∈ Δr do
4      EC ← a set of evidence contexts with a single
           context ec = (t, r ∪ Δr \ t) ↦ ahead
5      foreach predicate group S from P do
6          foreach evidence context ec ∈ EC do
7              reconcile ec for S, potentially identifying
                 new contexts EC+ and invalid contexts
                 EC−
8              EC ← (EC ∪ EC+) \ EC−
9      foreach evidence context ec ∈ EC do
10         collect the evidence in ec into E_Δr
11 return E_Δr
```

In addition, we build range indexes as sorted maps on numerical columns. For each numerical column $\mathtt{A} \in \mathtt{R}$, we build a sorted tree map that maps each value v of $\mathtt{A}$ to the set of tuples with a value higher than v in $\mathtt{A}$. For instance, considering tuples $\mathtt{t}_1$–$\mathtt{t}_5$, the index on the `Level` column of `staff` contains the following sorted entries: $2 \mapsto \{\mathtt{t}_1, \mathtt{t}_2, \mathtt{t}_5\}$, $3 \mapsto \{\mathtt{t}_1, \mathtt{t}_2\}$, $4 \mapsto \{\mathtt{t}_1\}$, $5 \mapsto \emptyset$. Such entries help to reconcile evidence for predicates with operators $\{\neq, <, \leq\}$.

2) Incremental evidence contexts: We iterate the incremental tuples to produce the incremental evidence. Based on the selectivity principle, in Line 4 we instantiate a set for each incremental tuple $\mathtt{t}'$ with an initial context stating that "tuple $\mathtt{t}'$, when combined with every other tuple $\mathtt{t}'$, generates the evidence *ahead*." Mind that "every other tuple $\mathtt{t}'$" refers to static and incremental tuples. The initial context set is invalid, hence the need for the reconciliations in Lines 5–8.

Evidence reconciliation is divided into multiple stages, one per predicate group (Line 5). Notice that the evidence contexts after a stage serve as the subsequent stage's input. At each stage, the evidence contexts are reconciled using the algorithms of [14]. The reconciliation can also create new evidence contexts (i.e., EC expansion) and invalidate some (i.e., EC contraction). After a stage for a predicate group G_i, the evidence contexts are reconciled for all predicate groups $\ldots, G_{i-1}, G_i$. After the last stage, all evidence contexts are finally corrected for all predicate groups and can be collected into the incremental evidence set (Lines 9–10).

Figure 3 illustrates the evidence context building for the incremental tuple $\mathtt{t}_5$ on `staff`, and the predicate space subset $\{\mathtt{p}_1, \ldots, \mathtt{p}_{16}\}$. We initiate the evidence context, ec_1, with the respective *ahead* evidence and the tuples from the static part of the data. Mind that when the inserts contain additional tuples, we must include them in such initial context.

Context ec_1 requires reconciling stages for predicate groups G_1, G_2, G_3, and G_4. As ec_1 is already consistent with groups G_1 and G_2, no changes are needed. On the other hand, it has an inconsistency regarding group G_3. Probing the categorical index on the `Hired` column for $\mathtt{t}_5$.`Hired` returns $2002 \mapsto \{\mathtt{t}_4, \mathtt{t}_5\}$, indicating that tuple $\mathtt{t}_4$ (from the static data) should not be part of ec_1. At this point, reconciliation requires removing $\mathtt{t}_4$ from ec_1 and creating a new context regarding tuple $\mathtt{t}_4$ and the `Hired` equality, ec_2. Then, the context ec_2 is consistent with group G_4, but context ec_1 is not. Probing the range index on the `Level` column for $\mathtt{t}_5$.`Level` returns $3 \mapsto \{\mathtt{t}_1, \mathtt{t}_2\}$, indicating that the tuples $\mathtt{t}_1$ and $\mathtt{t}_2$ have a higher `Level` than $\mathtt{t}_5$, and thus are incorrect in ec_1. The final reconciliation transform ec_1, and produces a new context ec_3 that fixes the inconsistency regarding tuples $\{\mathtt{t}_1, \mathtt{t}_2\}$.

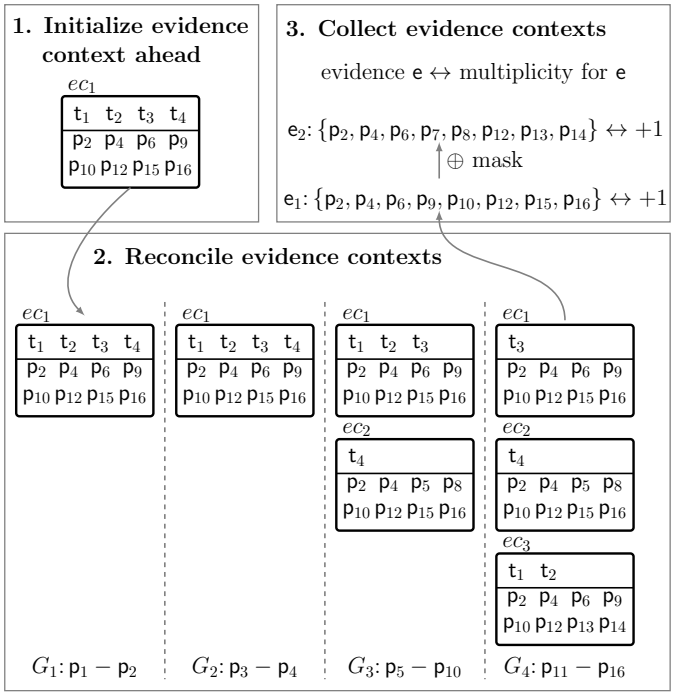


Fig. 3: Incremental evidence context building for tuple $\mathtt{t}_5$ on `staff` and predicate space subset $\{\mathtt{p}_1, \ldots, \mathtt{p}_{16}\}$.

Other types of evidence context reconciliation cover different types of predicate structures. For the detailed reconciliation algorithms, we refer the reader to [14].

3) Evidence collection: Evidence collection (Line 10) maps the evidence from the evidence contexts into the evidence set. It depends on whether or not evidence multiplicity is required. If not, the mapped evidence from the evidence context ec is added to the evidence set, $\mathtt{E}_{\Delta\mathtt{r}} \leftarrow \mathtt{E}_{\Delta\mathtt{r}} \cup ec.value$. Otherwise, the value associated with the evidence multiplicity $count(ec.value)$ is incremented by the size $|\mathtt{rids}|$ in the underlying context ec. Observe that $|\mathtt{rids}|$ indicates how many tuple pairs yield the evidence $ec.value$ within ec.

In addition, we propose a strategy for inferring evidence during evidence collection and, as a result, reduce the overall number of contexts that need to be computed. The idea is to infer evidence $\mathtt{e}(\mathtt{t}', \mathtt{t})$ from each evidence $\mathtt{e}(\mathtt{t}, \mathtt{t}')$ extracted from the evidence contexts.

Notice that predicates of the form $\mathtt{t.A}\ \theta\ \mathtt{t}'\mathtt{.B}$ (`A` and `B` can be the same attribute) have a symmetric relation for $\mathtt{t}, \mathtt{t}'$ when $\theta \in \{=, \neq\}$: if $\mathtt{t.A}\ \theta\ \mathtt{t}'\mathtt{.B}$ then $\mathtt{t}'\mathtt{.A}\ \theta\ \mathtt{t.B}$. In addition, numerical predicates have converse relations: if $\mathtt{t.A} < \mathtt{t}'\mathtt{.B}$, then $\mathtt{t}'\mathtt{.B} > \mathtt{t.A}$ (the result also holds for the corresponding non-strict inequalities). Also, from the context of order relations and the Trichotomy Law, for any two values from the quantifiers $\mathtt{t.A}$ and $\mathtt{t.B}'$ exactly one of the following predicates must be true: $\mathtt{t.A} < \mathtt{t}'\mathtt{.B}$; $\mathtt{t.A} = \mathtt{t}'\mathtt{.B}$; or $\mathtt{t.A} > \mathtt{t}'\mathtt{.B}$. Finally, based on this law and operator implication, tuple pairs can only satisfy a strict subset of each predicate group, namely the predicates with operators $\theta_1 = \{=, \leq, \geq\}$, $\theta_2 = \{\neq, <, \leq\}$, or $\theta_3 = \{\neq, >, \geq\}$. We use these properties for evidence inference.

We illustrate the idea using the evidence context ec_1 of our last example, stating that the tuple pair $(\mathtt{t}_5, \mathtt{t}_3)$ yields the evidence $\mathtt{e}_1 = \{\mathtt{p}_2, \mathtt{p}_4, \mathtt{p}_6, \mathtt{p}_9, \mathtt{p}_{10}, \mathtt{p}_{12}, \mathtt{p}_{15}, \mathtt{p}_{16}\}$.

Notice that the swapped tuple pair $(\mathtt{t}_3, \mathtt{t}_5)$ yields the evidence $\mathtt{e}_2 = \{\mathtt{p}_2, \mathtt{p}_4, \mathtt{p}_6, \mathtt{p}_7, \mathtt{p}_8, \mathtt{p}_{12}, \mathtt{p}_{13}, \mathtt{p}_{14}\}$. As shown in Figure 3, we can infer $\mathtt{e}_2$ from $\mathtt{e}_1$ with logical operations. For categorical predicate groups (e.g., G_1 and G_2), we can copy the respective satisfied predicates due to the symmetric property for operators $\{=, \neq\}$. For numerical predicate groups (e.g., G_3 and G_4), we have two options. If the evidence includes the predicate with operators θ_1, we can simply copy the respective predicates due to reflexivity. Otherwise, we use XOR operations to obtain the converse set of predicates. Specifically, we apply a mask having all range predicates (i.e., with operators $\{<, \leq, >, \geq\}$). Notice that we obtain the converse for each predicate group, from predicates with operators in θ_2 into predicates with operators in θ_3, or vice versa.

Thus, when collecting a context with an evidence $\mathtt{e}$ and multiplicity $|\mathtt{rids}|$, we also infer its equivalent $\mathtt{e}'$ using the reflexivity and converse properties above and add it to the evidence set. Note that $\mathtt{e}'$ has the same multiplicity $|\mathtt{rids}|$ as $\mathtt{e}$ regarding that context, so that we can add this number to the evidence set multiplicity.

The above strategy enables us to dynamically adjust the evidence context initialization to include fewer tuples on the right-hand side of their keys. Suppose an insert with tuples $\{\mathtt{t}_6, \mathtt{t}_7, \mathtt{t}_8\}$ into the `staff` relation. For tuple $\mathtt{t}_6$, we develop contexts from $\mathtt{rids} = \{\mathtt{t}_1 \ldots \mathtt{t}_8\} \setminus \{\mathtt{t}_6\}$ that enable us to collect evidence and multiplicity regarding the pairs $(\mathtt{t}_6, \mathtt{t}_1), (\mathtt{t}_6, \mathtt{t}_2), \ldots, (\mathtt{t}_6, \mathtt{t}_7), (\mathtt{t}_6, \mathtt{t}_8)$ and their swapped counterparts (due to the equivalency). Now, by the time we are handling tuple $\mathtt{t}_7$, we have already handled evidence regarding $(\mathtt{t}_6, \mathtt{t}_7)$ and $(\mathtt{t}_7, \mathtt{t}_6)$, so we can develop contexts $\mathtt{t}_7$ with fewer tuples, that is, $\mathtt{rids} = \mathtt{t}_1 \ldots \mathtt{t}_8 \setminus \{\mathtt{t}_6, \mathtt{t}_7\}$. Notice that handling tuple $\mathtt{t}_8$ requires us to develop contexts regarding only the static data ($\mathtt{rids} = \mathtt{t}_1 \ldots \mathtt{t}_8 \setminus \{\mathtt{t}_6, \mathtt{t}_7, \mathtt{t}_8\}$), as the other contexts were handled for the other tuples in the incremental data.

4) Using the incremental evidence set: Maintaining evidence sets on inserts is straightforward once the incremental evidence $\mathtt{E}_{\Delta\mathtt{r}}$ is computed. We can simply compute $\mathtt{E}_{\mathtt{r}\cup\Delta\mathtt{r}} = \mathtt{E}_{\mathtt{r}} \cup \mathtt{E}_{\Delta\mathtt{r}}$. If evidence multiplicity is considered, we also update the multiplicities of $\mathtt{E}_{\mathtt{r}\cup\Delta\mathtt{r}}$ by the multiplicity of every evidence $\mathtt{e} \in \mathtt{E}_{\Delta\mathtt{r}}$.

C. Maintaining evidence sets on deletes

A direct approach for maintaining evidence sets on deletes is to recompute the evidence for the tuples being deleted, then remove those evidence from the valid evidence set. Specifically, we compute the incremental evidence set $\mathtt{E}_{\Delta\mathtt{r}}$ for the set $\Delta\mathtt{r}$ of deleted tuples. In this case, for every tuple $\mathtt{t} \in \Delta\mathtt{r}$, we process an evidence context with the set $\mathtt{r}\cup\Delta\mathtt{r}\setminus\mathtt{t}$ of tuples using the algorithms discussed in Section V-B. Then, we compute $\mathtt{E}_{\mathtt{r}\setminus\Delta\mathtt{r}} = \mathtt{E}_{\mathtt{r}} \setminus \mathtt{E}_{\Delta\mathtt{r}}$, subtracting the evidence multiplicities of $\mathtt{E}_{\Delta\mathtt{r}}$ from $\mathtt{E}_{\mathtt{r}\setminus\Delta\mathtt{r}}$.

We propose a second approach that improves the performance of maintaining evidence sets on deletes even further. The approach integrates with Algorithm 1 and the static approach in [14], as they are based on evidence contexts. When collecting the evidence set from evidence contexts, we maintain an index where each tuple $\mathtt{t}$ (only the left-hand side in a context's key) is associated with a list of pairs $(\mathtt{e}, count(\mathtt{e}))$, where $\mathtt{e}$ are the evidence generated by $\mathtt{t}$ and $count(\mathtt{e})$ is the respective evidence multiplicity. This index can be implemented as a hash map.

During the deletion of a tuple $\mathtt{t}_j$, we retrieve its evidence from the index and update the evidence set $\mathtt{E}_{\Delta\mathtt{r}}$. Due to the equivalency strategy described earlier, we still need to compute the evidence of tuple $\mathtt{t}_j$ regarding a context with tuples $\mathtt{t}_1, \ldots, \mathtt{t}_{j-1}$. This step can be done similarly to the insert case. However, the number of evidences required to be computed is still smaller when compared to the first strategy.

D. Implementation and Complexity

Evidence reconciliation must be implemented carefully in Algorithm 1 since it is the operation most often called. As detailed in [14], the algorithms for this step involve many set operations between sets `rids` of evidence contexts and column index entries, such as intersections and differences. Thus, we implement `rids` and index entries as compressed bitmaps, as it has been shown to provide good performance and a low memory footprint [13].

Also, we use the two-layered bitmap index with binning to optimize memory usage for columns with high cardinality, as described in [14]. Finally, before building the index and starting Algorithm 1, we perform a multi-column sort of the table based on its numerical columns to enhance bitmap compression and the performance of the set operations— this strategy has also been used in [14].

Regarding the computational complexity, the cost of building the equality indexes for equality is linear in the size of the updated instance, i.e., $\mathcal{O}(|\mathtt{r}\cup\Delta\mathtt{r}|)$. The cost of building range indexes is in $\mathcal{O}(|\mathtt{r}\cup\Delta\mathtt{r}| \cdot \log(|\mathtt{r}\cup\Delta\mathtt{r}|))$, as we use sorted trees. The algorithm handles the tuples in $\Delta\mathtt{r}$ individually, each requiring evidence context reconciliation for $|\mathtt{S}|$ predicate groups. Each reconciliation cost is a factor of the number $|EC|$ of evidence contexts in a current stage and the cost σ of the logical operations for each context. Thus, the total cost of reconciliation is in $\mathcal{O}(|\Delta\mathtt{r}| \cdot |\mathtt{S}| \cdot |EC| \cdot \sigma)$.

The reconciliation costs tend to increase towards the final stages, as EC tends to expand. After the last stage and for a sufficient number of predicate groups, each tuple might have produced $|EC| = |\mathtt{r}| - 1$ contexts in the worst case. However, such cases are rare in practice due to the vast evidence redundancy. In our experiments, typical sizes $|EC|$ were between the hundreds or a few thousand (in the worst cases), even for the largest tables.

VI. Dynamic DC enumeration

As discussed in Section II, three algorithms have been developed for enumerating DCs from evidence sets. Devising dynamic extensions for all these algorithms is not the focus of this paper. Instead, we present `DynEI`, a dynamic version of the *evidence inversion* (`EI`) algorithm, proposed in [3]. We chose to extend the `EI` algorithm because it is often the fastest in previous experimental studies [14], [18].

The three DC enumeration algorithms have been extended to discover approximate DCs in [4], [7], [18]. We leave integrating such extensions in dynamic settings for future work and focus on better evaluating the performance of `DynEI` for exact DCs. Still, the evidence multiplicity computed with the algorithms in Section V provides the basis for such future extensions. For example, the `EI` algorithm is extended for approximate DCs in [18], where candidates are generated and pruned based on the accumulated evidence multiplicity for the violating evidence. This intuition illustrates the importance of having the evidence multiplicity available for DC enumeration.

A. Maintaining DCs on inserts

Algorithm 2 shows our procedure for dynamic DC enumeration on tuple inserts. It operates on the incremental evidence set of a relation rather than on its incremental tuples. Initially, we assume that the DCs Σ from the previous discovery are still valid (Line 1). Recall from Section I that the incremental data (new tuples) might violate these DCs (but cannot remove violations). We capture the potential violations by taking the set difference $\mathtt{E}^{inc}$ between the incremental evidence set $\mathtt{E}_{\Delta\mathtt{r}}$ and the evidence set $\mathtt{E}_{\mathtt{r}}$ of the previous instance state. Each element of $\mathtt{E}^{inc}$ (Line 2) represents a potential violation for the DCs of Σ. Notice that a set $\mathtt{E}^{inc} = \emptyset$ means that Σ is still valid, so nothing needs to be done. Otherwise, we must find and adjust the violated DCs until no violations are left.

Evidences violate a DC if they contain all predicates of that DC. Thus, we iterate each incremental evidence (Line 3) and find the violated DCs Σ^{inv}, which are DCs that fully intersect with that evidence (Line 4). We temporarily remove these invalid DCs from the valid DCs in Line 5 to update them (Lines 6–9) so the violating evidence no longer holds. To do so, we form new DC candidates from the invalid DCs by adding predicates that accommodate the violating

Algorithm 2: Dynamic DC enumeration

Input: evidence sets $\mathtt{E}_r$ and $\mathtt{E}_{\Delta\mathtt{r}}$, predicate space $\mathtt{P}$, and the set Σ of minimal, non-trivial DCs of $\mathtt{r}$
Output: the set Σ' of minimal, non-trivial DCs of $\mathtt{r} \cup \Delta\mathtt{r}$

1 $\Sigma' \leftarrow \{\varphi \mid \varphi \in \Sigma\}$
2 $\mathtt{E}^{inc} \leftarrow \mathtt{E}_{\Delta\mathtt{r}} \setminus \mathtt{E}_{\mathtt{r}}$
3 **foreach** $\mathtt{e} \in \mathtt{E}^{inc}$ **do**
4 $\Sigma^{inv} \leftarrow \{\varphi^{inv} \in \Sigma' \mid \varphi^{inv} \subseteq \mathtt{e}\}$
5 $\Sigma' \leftarrow \Sigma' \setminus \Sigma^{inv}$
6 **foreach** $\varphi^{inv} \in \Sigma^{inv}$ **do**
7 **foreach** $\mathtt{p} \in (\mathtt{P} \setminus \mathtt{e})$ **do**
8 **if** $\nexists \varphi \in \Sigma' : \varphi \subseteq (\varphi^{inv} \cup \{\mathtt{p}\})$ **then**
9 $\Sigma' \leftarrow \Sigma' \cup \{\varphi^{inv} \cup \{\mathtt{p}\}\}$
10 **return** Σ'

part of the evidence. In Line 8, we check the candidates for minimality regarding the DCs currently valid for the visited evidence and add them to the current set of valid DCs in case they are minimal. At the end of each iteration, the iterated evidence no longer violates any current DC. After processing all incremental evidence, no potential violations are left, so the DCs are valid and complete.

Figure 4 illustrates a sample of the operation of Algorithm 2 considering the incremental evidence from Table I. Incremental evidence from tuple $\mathtt{t}_5$ violates a previously valid DC, φ_3. The algorithm accommodates the violation by adding predicates to the violated DC, for instance, forming the DC φ_5 that becomes valid regarding all evidence up to that point.

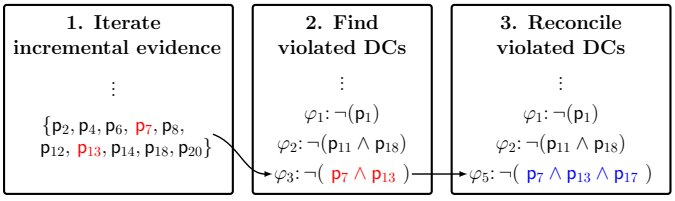


Fig. 4: Dynamic DC enumeration based on evidence inversion.

B. Maintaining DCs on deletes

For delete operations, the set $\mathtt{E}^{inc} = \mathtt{E}_{\Delta\mathtt{r}} \setminus \mathtt{E}_{\mathtt{r}}$ contains the evidence removed from the instance. Each element of $\mathtt{E}^{inc}$ represents a violation that might no longer invalidate subsets of some DCs φ in Σ, in case the evidence removal makes the φ's non-minimal. Our dynamic evidence inversion on deletes uses this intuition and explores a result from hitting set enumeration: Hyperedges can be a critical hyperedge for at most one vertex [7], [8], [19]. In evidence-based DC enumeration, this result means that each evidence can serve as critical evidence for at most one predicate in a DC. If the intersection between an evidence and a DC contains several predicates, the evidence is not a critical evidence for that DC.

Based on this result, we iterate the evidence $\mathtt{E}^{inc}$ and the (previously discovered) DCs Σ to identify DCs Σ^{rem} for which that evidence intersects with all but one predicate. Some predicate subsets of these DCs might form new valid DCs, so we update the current DCs such that $\Sigma = \Sigma \setminus \Sigma^{rem}$. We still need to adequate Σ for the remaining evidence $\mathtt{E}^{left} = \mathtt{E}_{\mathtt{r}} \setminus \mathtt{E}^{inc}$. Thus, we include DCs with a single predicate into Σ to accommodate the evidence in $\mathtt{E}^{left}$. The remaining process follows Algorithm 2, but we use the updated DCs Σ and evidence set $\mathtt{E}^{left}$ as starting points (substitutions for Lines 1 and 2). The new search still needs to visit the elements in $\mathtt{E}^{left}$ but has the pruning potential from the DCs in Σ (in Line 4).

C. Implementation and Complexity

Two operations in Algorithm 2 must be implemented carefully, as they are called often. The first is finding the violated DCs for evidence in Line 4, which can be naively implemented by comparing the evidence to all (current) DCs to find inclusions. The second is checking the minimality of a DC candidate in Line 8, which, similarly, can be implemented by comparing the candidate to all (current) DCs. For efficiency, however, we translate the first operation into a superset query and the second operation into a subset query so that we use the tree structure designed for fast queries on sets [2].

The complexity of Algorithm 2 is influenced by the number $|\Sigma^{inv}|$ of invalid DCs emerging from the insertion/removal of the incremental evidence. Let $|\varphi|$ be the number of predicates of a DC φ. In the worst, there can be $2^{|\mathtt{P}|-|\varphi^{inv}|}$ DC candidates for every invalid DC $\varphi^{inv} \in \Sigma^{inv}$. Thus, the running time depends on the number of candidate DCs checked, which, in turn, is influenced by the incremental evidence's size $|\mathtt{E}^{inc}|$ and the number of invalid DCs each $\mathtt{e} \in \mathtt{E}^{inc}$ yields.

VII. Experimental Evaluation

This section presents an extensive experimental evaluation of 3DC, comparing it to the state-of-the-art and investigating its major components in-depth.

A. Experimental settings

1) Datasets: The datasets in our experiments have already been used to evaluate DC discovery in [3], [7], [11], [12], [14], [15], [18]. For compatibility among the incremental algorithms 3DC and IncDC, we used all the datasets used to evaluate IncDC, found in [15]. In addition, we included the Tax and Hospital datasets provided in [11]. Our experiments used real-world and synthetic datasets that differ in size, domain, and distribution. Table II shows the main properties of these datasets. It also shows the running times of 3DC and baselines in some settings, which are discussed later in this section.

2) Baselines: We compared our 3DC algorithm with the IncDC algorithm [15], which is, to date, the only other solution for dynamic DC discovery. Also, we compared our dynamic DC enumeration solution (denoted as DynEI) to the hitting-set-based solutions in [19] (denoted DynHS), which are dynamic versions of one of the fastest algorithms for hitting set enumeration [8]. Using hitting sets for DCs has

TABLE II: Dataset properties and running times (in seconds). The best results appear in bold, and the "—" mark represents failed executions (i.e., the algorithm exceeded a 24h time limit or threw a memory exception).

Dataset Properties			$\lvert\Delta\mathbf{r}\rvert = 0.1\% \cdot \lvert\mathbf{r}\rvert$			$\lvert\Delta\mathbf{r}\rvert = 1\% \cdot \lvert\mathbf{r}\rvert$			$\lvert\Delta\mathbf{r}\rvert = 10\% \cdot \lvert\mathbf{r}\rvert$			$\lvert\Delta\mathbf{r}\rvert = 30\% \cdot \lvert\mathbf{r}\rvert$		
Name	#Cols	#Rows	3DC	IncDC	ECP	3DC	IncDC	ECP	3DC	IncDC	ECP	3DC	IncDC	ECP
Adult	15	32 561	**0.70**	25.60	93.91	**3.11**	73.92	96.24	**24.10**	261.59	102.52	**69.24**	436.36	127.73
Airport	11	55 113	**0.71**	28.50	2.01	**0.8**	84.56	2.17	**0.25**	786.17	2.18	**0.76**	—	2.82
Atom	13	147 067	**0.12**	109.66	7.25	**0.28**	276.82	7.59	**1.45**	—	7.71	**5.46**	—	11.99
Claim	11	112 000	**0.14**	19.10	22.79	**0.36**	21.33	23.61	**3.21**	27.26	25.14	**11.13**	110.76	30.44
Dit	8	780 000	**3.60**	—	192.58	**8.16**	—	194.15	**54.06**	—	246.14	**163.65**	—	373.16
FD	20	187 500	**0.30**	99.57	6.93	**0.44**	101.04	7.54	**1.94**	106.13	8.02	**4.54**	109.67	10.15
Flight	17	499 308	**0.72**	—	137.01	**4.67**	—	141.09	**47.63**	—	168.82	**182.44**	—	237.04
Hospital	15	114 919	**0.05**	23.48	4.22	**0.11**	24.35	4.82	**0.59**	51.29	4.99	**1.94**	—	5.58
Inspection	13	221 123	**0.39**	53.53	23.59	**1.01**	54.09	24.01	**5.60**	55.73	26.61	**18.71**	69.82	37.81
NCVoter	15	675 000	**1.26**	253.67	217.42	**4.88**	292.17	224.41	**44.83**	—	282.39	**183.77**	—	416.95
Tax	15	100 000	**0.13**	40.57	10.07	**0.42**	123.82	11.27	**2.57**	—	12.74	**7.97**	—	14.27
UCE	11	14 246	**2.87**	429.98	558.18	**16.13**	453.49	543.13	**114.22**	465.54	526.28	**301.82**	537.94	461.52

been explored for static DC discovery in [7]. In [19], the adaption works for dynamic functional dependency discovery. Nevertheless, we can also use the adaptation for DCs: For functional dependencies, difference sets are considered, which have a similar principle as the evidence sets for DCs.

For an in-depth analysis of 3DC (and dynamic DC discovery in general), we also run experiments against a static algorithm, ECP [14]. We chose ECP among the static alternatives as it was consistently the fastest algorithm during our experiments.

3) Implementation: We implemented 3DC in Java; the implementations of IncDC and ECP were also written in Java and were obtained from [15] and [14], respectively. All three algorithms leverage multi-threaded parallelism. For DynHS, we use the Java implementation obtained in [19]. We checked the correctness of the algorithms by verifying the equivalence of the results with the results from the static ECP algorithm.

4) Execution: We conducted the experiments on a server with an Intel Xeon Processor E5-2630 v3 @2.40GHz (8 cores, 16 threads), 96 GB of RAM, running Ubuntu 20.04.6 LTS and OpenJDK 64-Bit Server VM 11.0.19. We limited the JVM heap space to 64 GB, ran each experiment five times, and report the average runtime.

B. 3DC algorithm compared to IncDC

This section compares our 3DC algorithm and the IncDC. Since IncDC supports only DC discovery on inserts, the experiments in this section evaluate the performance of the algorithms in handling only inserts. We evaluate 3DC for the delete case in Section VII-C

1) Scalability with increasing $|\Delta\mathbf{r}|$: We first investigate how the update sizes impact DC discovery performance. To do so, we retained 70% of tuples chosen at random of each dataset $\mathbf{r}$ for each execution. Then, we chose the set $\Delta\mathbf{r}$ of tuples (also at random) from the remaining tuples by varying the ratio λ of incremental data such that $|\Delta\mathbf{r}| = \lambda \cdot |\mathbf{r}|$ and $0.001 \leq \lambda \leq 0.3$. Figure 5 illustrates the scalability of 3DC and IncDC across these increasing ratios.

Mind that IncDC threw memory exceptions or reached a 24h runtime limit in several scenarios, so we could not show the running times for these cases. In general, 3DC presents a better scaling than IncDC in all scenarios. The performance of both algorithms is impacted by the number of rows in the updates. However, IncDC seems profoundly sensitive to higher numbers of DCs (e.g., Adult and Airport). In scenarios where the number of DCs is relatively stable across the updates, the strategy of IncDC seems to suffer less from increasing update sizes.

We compared the dynamic algorithms to the static ECP algorithm to complement our discussion. Table II shows their running times for incremental ratios $\lambda \in \{0.001, 0.01, 0.1, 0.3\}$. Mind that ECP needs to run the discovery on the entire (updated) dataset to produce results. Surprisingly, ECP performed faster than IncDC in many scenarios. On the other hand, ECP is much slower than 3DC. The difference is huge for smaller ratios λ, but it is still relevant for larger ratios.

2) Scalability with increasing $|\mathbb{R}|$: The number of columns significantly impacts DC discovery performance, as each new column adds a set of predicates, thus incurring a larger DC search space. We evaluated the algorithms for increasing numbers of columns to investigate this impact on dynamic DC discovery. Specifically, we randomly selected different column subsets of increasing sizes of each dataset. To account for the variability introduced by the random samples, we computed the average running time based on ten executions for each specific subset size. We used the first 10k rows of each dataset and set the incremental ratio to $\lambda = 0.1$ to minimize interference from many rows. Figure 6 shows the results.

We observe that 3DC performs much better than IncDC. For instance, on the Flight dataset, as the number $|R|$ of columns increases from 5 to 17, the runtime of 3DC increases from 0.09 seconds to 0.50 seconds, whereas that of IncDC increases from 2.42 seconds to 68.90 seconds. The number of columns deeply impacts the efficiency of IncDC, as more columns yield a larger predicate space P and, thus, larger sets Σ of DCs. That, in turn, requires IncDC to maintain many more indexes. The algorithm visits and updates each index for each tuple in $\Delta\mathbf{r}$ to determine the incremental evidence. While a smaller set of DCs may lead to better efficiency, those are rare in practice, given the high expressive power of DCs.

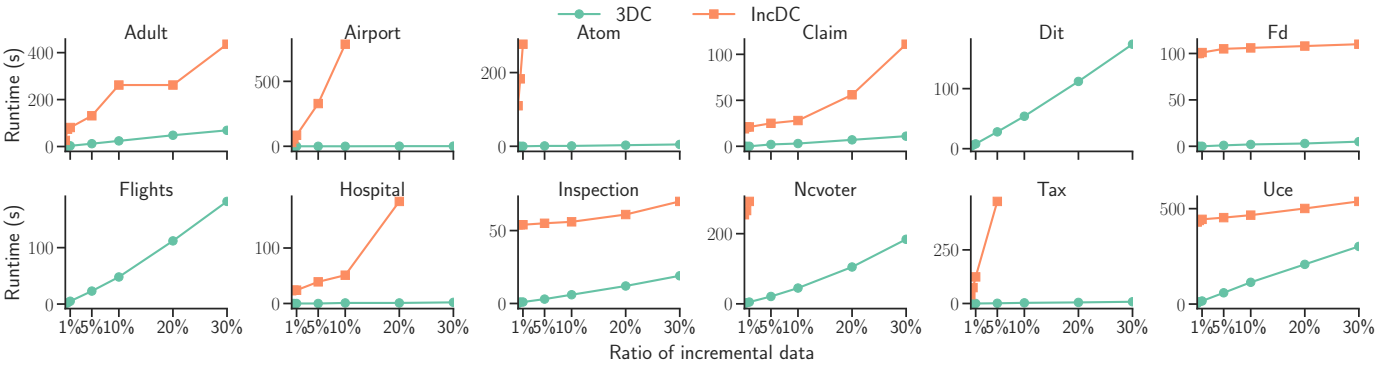


Fig. 5: Scaling of 3DC and IncDC regarding increasing size inserts.

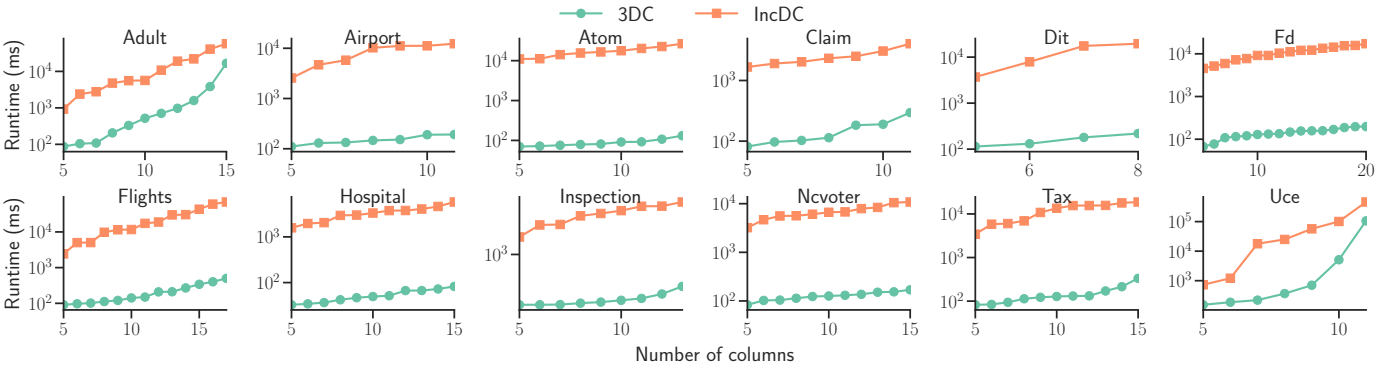


Fig. 6: Scaling of 3DC and IncDC regarding increasing number of columns – the y-axis is in log scale.

On the other hand, the cost incurred by 3DC to find the incremental evidence depends on $|\Delta r|$ and $|R|$ since it adds a pipeline stage per additional column. Notice that $|R| < |P| \ll |\Sigma|$, a fact that greatly contributes to the different scalability of 3DC and IncDC.

3) Memory usage: Figure 7 compares the memory consumption of 3DC and IncDC—it only shows the datasets for which IncDC finishes for the entire dataset (considering a ratio $\lambda = 0.1$ of incremental inserts). The figure indicates the minimum heap size necessary to execute the algorithm with no memory exceptions. To obtain this estimation, we gradually doubled the heap size until the algorithm could successfully process the entire dataset. We started the process with 32 MB.

The size (number of rows and columns) and domain types of the datasets are natural factors that increase memory use. In addition, the number of results does so, too. 3DC is more efficient than IncDC because it does not need to create and maintain indexes on the DCs from the static data. We observed a much higher memory requirement from IncDC, as it required heap sizes of up to 8× larger than that of 3DC.

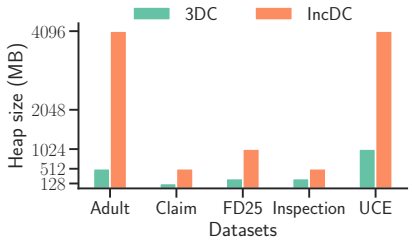


Fig. 7: Memory consumption of 3DC and IncDC.

C. In-depth analysis of the 3DC algorithm

This section presents complementary experiments highlighting the performance and main components of 3DC. For succinctness, we used a mix of datasets that better summarized the behaviors of the algorithms and executions.

1) Impact of different dimensions on 3DC performance: For this next experiment, we maintained the size of the static data at 70%, varying the ratio of the incremental data—like the experiment in Section VII-B1. We seek to investigate how different data dimensions impact the performance of 3DC. Figure 8 illustrates the behavior of 3DC across different dimensions: We ran the discovery with an increasing number of rows, measured the number of newly discovered evidence, and the time required to build the incremental evidence set. We also measured the number of discovered DCs and the number of new DCs compared to the previous set of DCs, along with the runtime for DC enumeration. For additional context, we also plotted the size of the incremental data.

We observe that the size of the incremental data has the most significant impact on evidence-building runtime, as anticipated, given that an evidence context pipeline is processed for each new incremental tuple. Furthermore, the number of new evidences is relatively low in some datasets, such as Atom. The number of DCs varies considerably from one

dataset to another, with some datasets yielding dozens of DCs while others produce thousands. Although the total number of discovered DCs remains stable over incremental ratios, the number of newly discovered DCs increases proportionately to the number of newly discovered evidence. This, in turn, results in an increase in the DC enumeration runtime. This last behavior is also expected, since our DC enumeration solution iterates each new evidence to update the set of valid DCs.

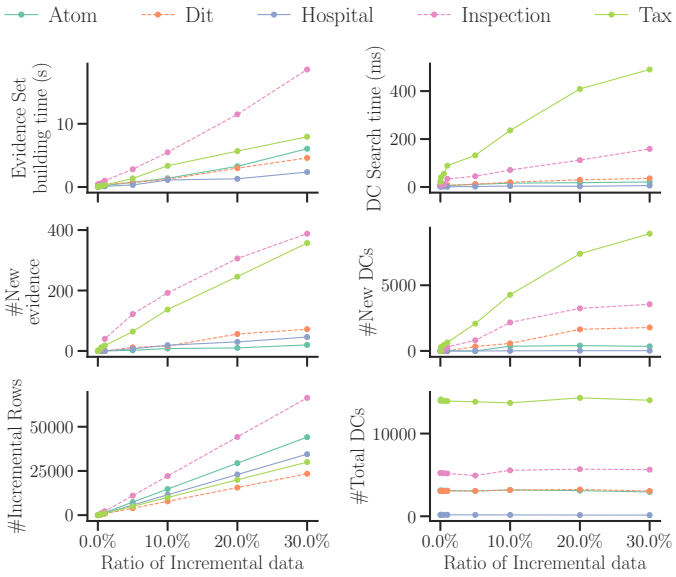


Fig. 8: Impact of different dimensions on the performance of different aspects of 3DC.

2) Evidence set experiments: Only evidence set building runtime is considered for the two following experiments. We first investigate the impact of applying evidence inference regarding the tuples within the update (as explained in Section V-B). Figure 9 shows the results for applying the inference only regarding the static tuples (denoted as DynEvi (Base)) and doing so for static and incremental tuples (DynEvi (Opt)). We use static datasets with 300 000 tuples and inserts of increasing size, up to 120 000 tuples. The results demonstrate that applying the technique improves runtime, particularly when more tuples are considered. While the result of the optimization is modest regarding the dynamic data, the evidence inference technique is crucial for an efficient dynamic evidence enumeration, as the clear difference between the runtimes of the static ECP and 3DC in Table II.

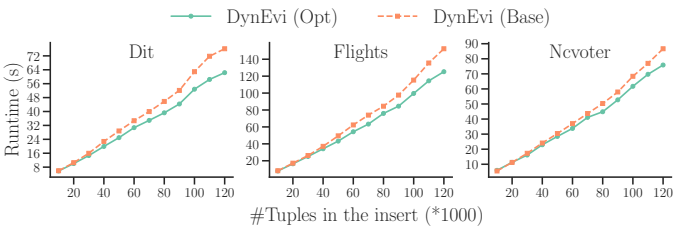


Fig. 9: Performance of dynamic evidence set building with the inference strategy enabled among the incremental data.

Figure 10 compares the efficiency of our two approaches for evidence set maintenance on deletes. We use static datasets with 70 000 tuples and inserts of increasing size, up to 30 000 tuples. The results demonstrate that using the evidence indexes slightly outperforms the evidence recomputing. Regarding the overhead of building the indexes during the previous discoveries (not shown in the plot), we notice only a slight increase in runtime. For instance, on NCVoter, the static discovery runtime increased from 11.9 to 12.9 seconds (when building indexes), whereas the dynamic discovery one decreased from 4.7 to 3.4 seconds (when leveraging indexes). This result suggests that maintaining evidence indexes might be viable for large batches of deletes. Mind that we only build indexes for the evidence at the left-hand side of evidence contexts' keys to avoid large memory footprints. An interesting topic for future work is investigating possible forms of compression for these indexes so that it is possible to leverage indexes on entire evidence contexts.

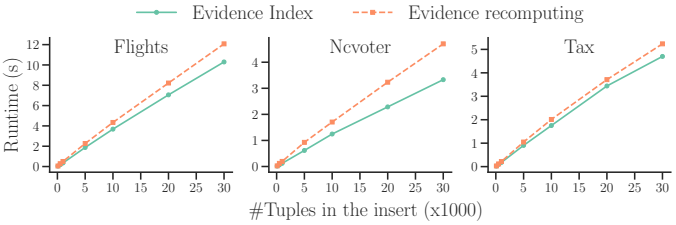
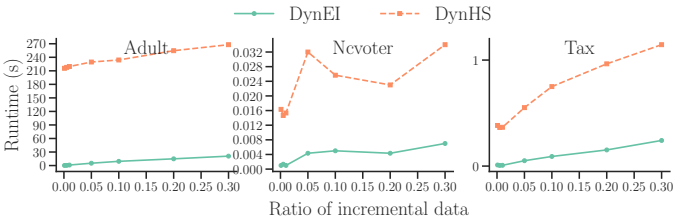


Fig. 10: Performance of evidence set maintenance on deletes using evidence indexes vs. evidence recomputing.

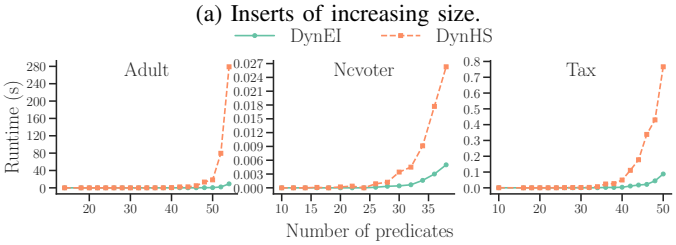
3) DC enumeration experiments: For the following experiments, we considered the first 20 000 tuples of each dataset as static data—we considered only DC enumeration runtime. Figure 11 compares our dynamic DC enumeration algorithm, DynEI, with the dynamic hitting set approach, DynHS, on insertions. The results for insertions of increasing size are depicted in Figure 11a. Increasing the number of tuples produces more evidence to be covered, but this does not necessarily lead to the discovery of more DCs. Doing so might result in more general DCs that subsume others, thus reducing the number of DCs discovered. This fact affects the search space and may even reduce runtime, as observed for the NCVoter dataset. DynEI is much faster than DynHS, particularly for datasets producing many DCs (e.g., Adult).

DC enumeration performance is much more affected by the number of columns/predicates, as observed in Figure 11b. That is because more columns result in more predicates, hence a large search space. The exponential scalability trend hits both algorithms but much harder on DynHS, with DynEI being much faster for more columns.

The results in Figure 12 complement our discussion on dynamic DC enumeration, as it shows the performance of DynEI and DynHS on deletions. The results for deletes of increasing sizes are depicted in Figure 12a, whereas the ones for deletes on datasets with increasing numbers of columns/predicates are in Figure 12b. The increased number of deleted tuples potentially reduces the remaining evidence DCs must cover,



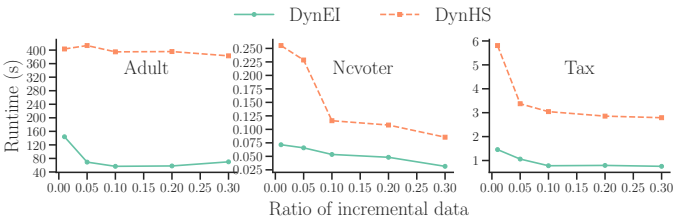
(a) Inserts of increasing size.



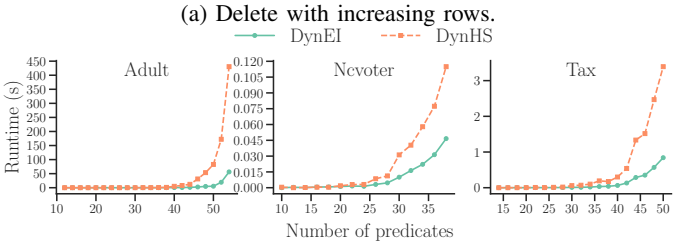
(b) Insert size fixed at %10 and increasing number of predicates.

Fig. 11: Performance of dynamic DC enumeration on inserts.

decreasing runtime for both algorithms. However, we observed that the runtimes of handling deletions are higher than those of inserts. That is expected as, for deletes, DynEI must first identify non-minimal DCs and then find DCs that cover the remaining evidence. For inserts, it only needs to accommodate the new evidence. Generally speaking, handling deletions is more expensive than insertions, as also indicated in evaluations of other dependency discovery algorithms [19].



(a) Delete with increasing rows.

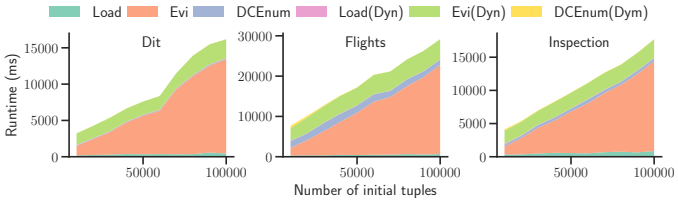


(b) Delete size fixed at %10 and increasing number of predicates.

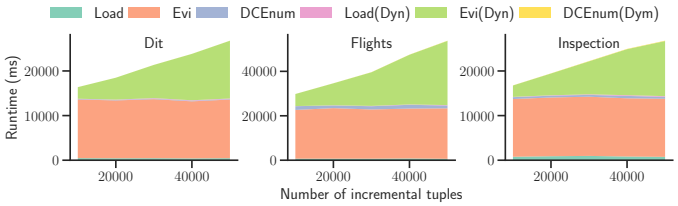
Fig. 12: Performance of dynamic DC enumeration on deletes.

4) Runtime breakdown: The following experiment analyses the runtime proportions of DC discovery, from the initial (static) discovery (with ECP) to the dynamic discovery (with 3DC). Figure 13 shows a runtime breakdown, including the time spent in static discovery phases: loading the dataset (Load), building the evidence set (Evi), and enumerating DCs (DCEnum). It also shows the time spent on the relative phases of dynamic discovery using 3DC (denoted with (Dyn)). In general, we observe that evidence-building dominates runtime, both in static and dynamic portions.

For Figure 13a, we maintained inserts of fixed sizes, $|\Delta \mathtt{r}| = 10\,000$, while progressively increasing the size of the initial tuples $|\mathtt{r}|$. We observe that the runtime of our dynamic solutions scales very well with increasing $|\mathtt{r}|$, indicating that the solution can be efficient even when the initial datasets become large. Loading and DC enumeration in the dynamic phase is much faster than the other phases, so their runtimes are not visible in the plot. For Figure 13b, we investigate the other way around by maintaining the initial tuples fixed, $|\mathtt{r}| = 100\,000$, while progressively increasing the size of the inserts $|\Delta \mathtt{r}|$. In this case, the performance of the dynamic solution is impacted by larger updates. Mind that it does not necessarily mean that rerunning the static algorithm in the entire updated instance is faster, as the runtime would also greatly increase due to the additional data.



(a) Increasing initial data, and fixed inserts at 10,000.



(b) Initial data with 100 000 rows, and increasing inserts.

Fig. 13: Runtime proportions of static and dynamic DC discovery phases in two scenarios.

VIII. Conclusion

This paper presented the 3DC algorithm, which introduces an approach to maintaining dynamic datasets' denial constraints (DCs). By continuously adapting the intermediate structures (the evidence sets) and employing a fast evidence inversion, 3DC achieves fast performance in maintaining the set of DCs with each batch of inserts, deletes, and thus updates. The evaluation shows that the dynamic algorithm significantly outperforms the current state-of-the-art in dynamic DC discovery. In future work, we plan to focus on the enumeration of different forms of approximate DCs in dynamic settings, either by adapting existing solutions or developing new ones. Additionally, we aim to conduct a comprehensive qualitative study of DC discovery in dynamic scenarios.

Acknowledgments

This work was partially supported by the *Serrapilheira Institute* (grant number R−2211−41919).

References

[1] N. Ayat, H. Afsarmanesh, R. Akbarinia, and P. Valduriez, "Pay-as-you-go data integration using functional dependencies," in *Multidisciplinary Research and Practice for Information Systems*, 2012, pp. 375–389.

[2] T. Bleifuß, S. Bülow, J. Frohnhofen, J. Risch, G. Wiese, S. Kruse, T. Papenbrock, and F. Naumann, "Approximate discovery of functional dependencies for large datasets," in *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. Association for Computing Machinery, 2016, p. 1803–1812.

[3] T. Bleifuß, S. Kruse, and F. Naumann, "Efficient denial constraint discovery with Hydra," *PVLDB*, vol. 11, no. 3, pp. 311–323, 2017.

[4] X. Chu, I. F. Ilyas, and P. Papotti, "Discovering denial constraints," *PVLDB*, vol. 6, no. 13, pp. 1498–1509, 2013.

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

[6] J. Kossmann, F. Naumann, D. Lindner, and T. Papenbrock, "Workload-driven, lazy discovery of data dependencies for query optimization," in *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12*, 2022.

[7] E. Livshits, A. Heidari, I. F. Ilyas, and B. Kimelfeld, "Approximate denial constraints," *PVLDB*, vol. 13, no. 10, p. 1682–1695, 2020.

[8] K. Murakami and T. Uno, "Efficient algorithms for dualizing large-scale hypergraphs," *Discrete Applied Mathematics*, vol. 170, pp. 83–94, 2014.

[9] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann, "Functional dependency discovery: An experimental evaluation of seven algorithms," *PVLDB*, vol. 8, no. 10, pp. 1082–1093, 2015.

[10] T. Papenbrock and F. Naumann, "Data-driven schema normalization," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2017, pp. 342–353.

[11] E. H. M. Pena, E. C. de Almeida, and F. Naumann, "Discovery of approximate (and exact) denial constraints," *PVLDB*, vol. 13, no. 3, pp. 266–278, 2019.

[12] E. H. M. Pena and E. C. de Almeida, "BFASTDC: A bitwise algorithm for mining denial constraints," in *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*, 2018, pp. 53–68.

[13] E. H. M. Pena, E. C. de Almeida, and F. Naumann, "Fast detection of denial constraint violations," *PVLDB*, vol. 15, no. 4, p. 859–871, dec 2021.

[14] E. H. M. Pena, F. Porto, and F. Naumann, "Fast algorithms for denial constraint discovery," *PVLDB*, vol. 16, no. 4, p. 684–696, dec 2022.

[15] C. Qian, M. Li, Z. Tan, A. Ran, and S. Ma, "Incremental discovery of denial constraints," *VLDB Journal*, vol. 32, no. 6, Mar 2023.

[16] P. Schirmer, T. Papenbrock, S. Kruse, F. Naumann, D. Hempfing, T. Mayer, and D. Neuschäfer-Rube, "Dynfd: Functional dependency discovery in dynamic datasets," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2019, pp. 253–264.

[17] Z. Tan, A. Ran, S. Ma, and S. Qin, "Fast incremental discovery of pointwise order dependencies," *PVLDB*, vol. 13, no. 10, p. 1669–1681, 2020.

[18] R. Xiao, Z. Tan, H. Wang, and S. Ma, "Fast approximate denial constraint discovery," *PVLDB*, vol. 16, no. 2, p. 269–281, oct 2022.

[19] R. Xiao, Y. Yuan, Z. Tan, S. Ma, and W. Wang, "Dynamic functional dependency discovery with dynamic hitting set enumeration," in *Proceedings of the International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 286–298.