Fast Algorithms for Denial Constraint Discovery

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

Fabio Porto
LNCC-DEXL
Petropolis, Rio de Janeiro, Brazil
fporto@lncc.br

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

ABSTRACT

Denial constraints (DCs) are an integrity constraint formalism widely used to detect inconsistencies in data. Several algorithms have been devised to discover DCs from data, as manually specifying them is burdensome and, worse yet, error-prone. The existing algorithms follow two basic steps: building an intermediate data structure from records, then enumerating the DCs from that intermediate. However, current algorithms are often inefficient in computing these intermediates. Also, it is still unclear which enumeration algorithm performs best since some of the available algorithms have not yet been compared to each other.

In response, we present a set of new algorithms with improved design choices. We introduce a parallel pipeline for rapidly computing the intermediate using custom data representations, algorithms, and indexes. For DC enumeration, we propose an inverted index, pruning, and parallel search strategies. We present hybrid approaches that integrate our techniques with previous enumeration algorithms, improving their performance in many scenarios. Our experimental study shows that the proposed DC discovery algorithms are consistently much faster (up to an order of magnitude) than the current state-of-the-art.

PVLDB Reference Format:
Eduardo H. M. Pena, Fabio Porto, and Felix Naumann. Fast Algorithms for Denial Constraint Discovery. PVLDB, 16(4): 684-696, 2022.
doi:10.14778/3574245.3574254

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/eduardopena/fdcd.

1 INTRODUCTION

Data cleaning, database design, and query optimization exemplify data management tasks that can exploit several facets of integrity constraints. Specifying such constraints is required, but manually doing so is unlikely to work for large databases and complex constraints, for instance because the required expertise might not always be available. Even if it is, manually designing constraints is error-prone and time-consuming since the constraints must be kept up to date for the ever-evolving semantics of data and applications.

Instead, one can discover the constraints from data [1]. Of particular interest are *denial constraints* (DCs), as they generalize several intra-relation constraints (i.e., over single tables) and can capture complex data quality rules that the generalized constraints cannot. Consider the `employee` relation in Table 1 and observe that (non-supervisor) employees never earn salaries higher than their supervisors (referenced by `SID`). Due to limited expressive power, constraints such as keys, functional dependencies, and order dependencies cannot model this observation. In turn, a DC can do so with a set of predicates that specify the inconsistent value combinations for this observation, as follows:

$$\varphi_1 : \forall \mathtt{t}, \mathtt{t}' \in \mathtt{employee}\ \neg(\mathtt{t.SID} = \mathtt{t}'\mathtt{.ID} \wedge \mathtt{t.Salary} > \mathtt{t}'\mathtt{.Salary})$$

DC φ_1 defines that there cannot exist any tuple pair in `employee` satisfying all predicates of φ_1 simultaneously. We introduce DCs more formally in Section 2.

Table 1: The `employee` relation.

	ID	Name	Salary	SID
t_1	#1	Caruso	10 000	#1
t_2	#2	Zhang	5500	#1
t_3	#3	Schneider	6000	#1
t_4	#4	Smith	11 000	#4
t_5	#5	Caruso	6000	#4
t_6	#6	Souza	7000	#4
t_7	#6	Souza	7000	#4

Recent works have shown how DCs help with data consistency [10, 11, 18]. For example, it is possible to augment relational tuples by quantifying their degrees of inconsistency concerning a set of DCs, then use the augmented tuples during query answering to compute top-k results [11]. It is also possible to introduce cleaning operators into the query processing to track and repair DC violations on demand [10]. Overall, DCs are essential building blocks in quantifying database inconsistency [18], being at the core of several state-of-the-art data cleaning systems [6, 9, 24].

The higher the expressive power of a constraint type, the higher the computational complexity of its discovery. For functional dependencies, this complexity is already quadratic in the number of tuples and exponential in the number of columns [16]. In the case of DCs, each column can generate several predicates. Although the complexity of DC discovery is still quadratic in the number of tuples, it is exponential in the number of predicates [5].

If the input contains errors, discovering exact DCs, which hold for the entire dataset, may result in overfitting because the DCs need to accommodate the errors. A common workaround is to relax the DC definition. For example, *approximate* DCs (aka. *partial* DCs) must hold for at least a fixed percentage of the data [5]. Observe that the following DC could conveniently specify a key for `employee`:

$$\varphi_2 : \forall \mathtt{t}, \mathtt{t}' \in \mathtt{employee}\ \neg(\mathtt{t.ID} = \mathtt{t}'\mathtt{.ID})$$

Proceedings of the VLDB Endowment, Vol. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574254

Although the duplicate tuples t_6 and t_7 prevent DC φ_2 from holding, it is still possible to discover this approximate DC with an algorithm that accounts for the small number of violations. It turns out that computing and leveraging the violations of DC candidates increases the discovery costs, making the discovery of approximate DCs even more expensive than the discovery of exact DCs [17, 21].

There are a number of algorithms for DC discovery [4, 5, 17, 20, 21]. However, a deeper analysis of these algorithms shows several design choices that limit their performance and applicability. The algorithm proposed first, FastDC, requires enumerating every pair of tuples of the input relation, which incurs a high computational cost [5]. In response, the algorithms in [17, 20, 21] use column indexing and logical operations to reduce this cost. Still, their approach needs to visit a quadratic number of intermediates, suffering significant performance degradation for larger inputs. Hydra is a sample-based algorithm that, unlike the other previous algorithms, can discover only exact DCs [4]. As it leaps across the search space for performance, it loses track of the additional information required for discovering approximate DCs. Datasets in production often contain data errors, so focusing only on exact DCs might not be appropriate due to DC overfitting. A second shortcoming of Hydra is that its sample-based approach may induce expensive computations for datasets containing many DCs, significantly hindering performance. Such datasets are common in practice due to the large DC search space.

All prior algorithms first build an intermediate called *evidence set* from the input and then enumerate the DCs based on this intermediate. Some even share the same algorithm for one of the two phases: [17, 21] use the same algorithm for evidence set building, whereas [5, 20, 21] use the same one for DC enumeration. In this paper, we use this DC discovery framework but present much more efficient algorithms for both phases.

First, we introduce intermediate representations, column indexes, and algorithms to build a parallel pipeline for building evidence sets suitable for discovering exact and approximate DCs. We show experimentally that our approach is much faster than the ones provided by previous alternatives. This contribution is essential for DC discovery since building evidence sets consumes most of the runtime for many datasets [5, 17].

We present novel techniques also for DC enumeration, in particular an inverted index, pruning strategies, and a DC search scheme that can run in parallel. We have applied these techniques to redesign one of the previous algorithms completely [5], thereby providing an alternative that is much faster than the original. Also, we combine our redesign with the other previous DC enumeration algorithms in hybrid approaches. Our experiments show that these hybrids often result in the fastest enumeration strategies.

We organize the rest of this paper as follows. We provide the background on DCs and an overview of the state-of-the-art in DC discovery in Section 2. We present our solution for evidence set building in Section 3, followed by our DC enumeration algorithms in Section 4. Then, we provide an extensive experimental evaluation in Section 5. We compare our algorithms to the previous ones on various datasets, showing that our algorithms consistently perform much faster than the previous state-of-the-art (up to an order of magnitude). Finally, we discuss our conclusions and future directions in Section 6.

2 BACKGROUND AND RELATED WORK

2.1 Denial constraints

DCs express *predicate conjunctions* to determine conflicting combinations of column values. They generalize other integrity constraints, including unique column combinations, functional dependencies, and order dependencies. Let A and B be two columns of a relation schema R having a relation instance r with n tuples. Following [4, 5, 17, 21], we consider predicates of the form $p\colon t.A\ \theta\ t'.B$, where t and t' are tuples of r, and $\theta \in \{=, \neq, <, \leq, >, \geq\}$ is a comparison operator of the database. A DC φ defines a set of predicates that cannot be true at the same time, as the following:

$$\varphi\colon \forall t, t' \in r, \neg(p_1 \wedge \ldots \wedge p_m)$$

A DC φ holds in r if no pair of tuples t, t' satisfy all the predicates of φ simultaneously. To put it another way, a DC φ is valid in r if, for every pair of tuples in r, there is at least one predicate of φ that returns false.

We call a pair of tuples t, t' that satisfy all predicates of a DC φ a *violation*. A DC φ that holds in r entirely, i.e., there are no violations, is usually called an *exact DC*. A DC φ is called *trivial* if it is satisfied by any relation instance. For example, any instance of the employee relation would satisfy the trivial DC $\varphi_3\colon \neg(t.\text{Name} = t'.\text{Name} \wedge t.\text{Name} \neq t'.\text{Name})$. A DC φ is *set-minimal* if no proper subset of its predicates form a valid DC in r. For example, observe that the predicates of a DC $\varphi_4\colon \neg(t.\text{SID} = t'.\text{ID} \wedge t.\text{Salary} > t'.\text{Salary} \wedge t.\text{Name} \neq t'.\text{Name})$ form a superset of the minimal DC φ_1, therefore, DC φ_4 is considered non-minimal.

An *approximate* DC is a DC partially satisfied in a relation instance. The term "partial" refers to a relaxation in the constraint definition. Following related work [17], we use *approximation functions* f to relax the DC definition. These functions map a relation instance r and a DC φ into a value in the interval $[0, 1]$, representing the degree to which that instance violates that DC. Given a threshold $0 \leq \varepsilon < 1$, a DC φ is a minimal approximate DC if $f(r, \varphi) \leq \varepsilon$ and there is no DC φ' whose predicates are a proper subset of the predicates of φ and such that $f(r, \varphi') \leq \varepsilon$.

The existing approximation functions may produce complementary results, so it is unclear which function works best [17]. In this paper, we use the g_1 function [13], as it has been used to define approximate DCs in most related work [5, 17, 20, 21]. This function captures the proportion between the number of violating tuple pairs and the number of distinct tuple pairs of the instance relation:

$$g_1(r, \varphi) = \frac{|\{(t, t') \in r \mid (t, t') \not\models \varphi\}|}{n \cdot (n-1)}$$

For example, observe that tuples t_6 and t_7 in employee form a pair of violations for the DC φ_2. By allowing a small number of violations, say with a threshold $\varepsilon = 0.05$, we can obtain φ_2 as an approximate DC since $g_1(\text{employee}, \varphi_2) = 0.047$. This example illustrates how approximate DCs help in scenarios where the input datasets contain errors or exceptions.

2.2 DC discovery

In general, existing DC discovery algorithms follow three main steps [4, 5, 17, 20, 21], as explained next.

1. Predicate space building. The first step defines the set of predicates that the discovery algorithms can use to derive DCs, that is, building the predicate space P. In principle, any subset of P is a DC candidate. Including predicates into P increases the DC search space, and hence computational costs. In particular, including some predicates into P would only increase the number of uninteresting DCs in the output. For example, including the predicate t.Name = t′.Salary into P would result in a DC φ_5: ¬(t.Name = t′.Salary) with low semantic value.

Chu et al. proposed predicate restrictions to reduce computational costs and uninteresting results [5]. Predicates on categorical columns use the operator set {=, ≠}, whereas predicates on numeric columns use {=, ≠, <, ≤, >, ≥}. Predicates on two different columns are allowed only if the columns share the type and a percentage of common values. All DC discovery algorithms use this approach, and so do ours. Also, our algorithm in Section 3 rearranges the predicate space into a list of *predicate groups*. Each predicate group is the subset of P whose predicates differ from each other solely by the operator. Figure 1 illustrates the predicate space and group arrangement for employee relation.

p_1 : t.ID = t′.ID	p_2 : t.ID ≠ t′.ID
p_3 : t.Name = t′.Name	p_4 : t.Name ≠ t′.Name
p_5 : t.Salary = t′.Salary	p_6 : t.Salary ≠ t′.Salary
p_7 : t.Salary < t′.Salary	p_8 : t.Salary ≤ t′.Salary
p_9 : t.Salary > t′.Salary	p_{10} : t.Salary ≥ t′.Salary
p_{11} : t.SID = t′.SID	p_{12} : t.SID ≠ t′.SID
p_{13} : t.ID = t′.SID	p_{14} : t.ID ≠ t′.SID
p_{15} : t.ID = t.SID	p_{16} : t.ID ≠ t.SID

Figure 1: Predicate space (and groups) of employee.

2. Evidence set building. The next step is building the *evidence set*, a data structure used to validate DC candidates quickly. The term "piece of evidence" e refers to the predicate subset of P that is satisfied by a pair of tuples t, t′, that is, $e_{t,t'} = \{p \mid p \in P, t, t' \models p\}$. The evidence set E_r is the set of pieces of evidence from all pairs of tuples of the instance r, given a predicate space P. In practice, the size of evidence sets is only a tiny fraction of the total number of tuple pairs [17, 21]: distinct tuples pairs can produce the very same piece of evidence, and they often do. As a result, building evidence sets is a way of compacting the information on predicate satisfaction of the entire input into a single and smaller data structure.

For approximate DC discovery, algorithms need the number of tuple pairs that produced each piece of evidence $e \in E_r$, i.e., the *multiplicity* of each evidence e. We denote the multiplicity value as *count*(e). This value serves as a proxy for determining the violation number of each approximate DC candidate. It is also required for ranking DCs regarding their coverage of the datasets. We refer to related work for details on DC ranking [5, 21].

Evidence set building can dominate discovery runtime for many datasets, as it is significantly impacted by the number of tuples in the input [4, 5, 17, 21]. A few algorithms have been proposed to mitigate exhaustive enumerations of tuple pairs, thereby reducing the performance impact for large datasets. We discuss these previous algorithms in Section 2.3 and present our own in Section 3.

3. DC enumeration. Discovering DCs is equivalent to discovering predicate sets that cover evidence sets [4, 5]. DC discovery is connected also to enumerating hitting sets in hypergraphs: the predicate space regards the vertex set, and each piece of evidence regards a hyperedge [17]. The problem equivalencies are based on the following insight. A DC φ is valid if no tuple pair in r satisfies all predicates of φ, so it is possible to verify the evidence set for the absence of a piece of evidence that contains all predicates of φ.

The runtime of enumerating DCs grows exponentially in the number of columns, regardless of the enumeration method [4, 5, 17]. Still, prior algorithms perform well for many practical datasets, often finishing in a matter of seconds. Unfortunately, it is still an open question how to efficiently enumerate DCs (or any other dependency type) for wider datasets (i.e., many columns) [2]. We review the prior DC enumeration algorithms in Section 2.3, and present our own in Section 4.

2.3 Related work

FastDC was the first algorithm for DC discovery [5]. First, it iterates every pair of tuples of the input to compute the evidence set [5]. Then, it enumerates DCs with a heuristic-driven depth-first traversal of the DC search space, which validates the DC candidates using evidence set intersections. This enumeration scheme works for exact and approximate DCs (with adaptations). Our enumeration algorithm in Section 4 follows a similar framework, but introduces auxiliary data structures and pruning techniques that significantly reduce the cost of checking DC candidates.

Evidence set building in FastDC is computationally expensive—it may take several days to process even medium-size datasets [5]. BFastDC [20] and DCFinder [21] algorithms use faster strategies for this phase, and then use the DC enumeration strategy of FastDC. The algorithms operate on small blocks of evidence at a time, and use custom column indexes to derive predicate satisfaction. DCFinder additionally uses information on predicate selectivity to reduce the number of evidence manipulations. The strategies in BFastDC and DCFinder greatly improve runtimes compared to the approach of FastDC, but can still be penalized for large instances. The main shortcoming is that these algorithms need to collect each piece of evidence in each block to compute the evidence set and accumulate its multiplicity. This step requires visiting a quadratic number of elements, which can be cost-prohibitive. Our approach uses a similar intuition as DCFinder regarding predicate selectivity. However, it introduces an entirely different framework that avoids visiting each piece of evidence individually.

The Hydra algorithm focuses on exact DCs [4]. First, it discovers DCs from an intermediate evidence set built from tuple pair samples. Then, it detects the violations of these DCs regarding the relation instance, as each violation points to a missing piece of evidence. A shortcoming is that the time to detect violations and complete the evidence drastically increases with the number of intermediate DCs—high DC numbers are common in many datasets. A similarity between our approach and Hydra is that both exploit the evidence redundancy across tuple pairs for performance. However, Hydra uses sampling to leap across the evidence search space. This strategy loses evidence multiplicity, which is critical for discovering approximate DCs and ranking DCs based on coverage. In turn,

our approach exploits evidence redundancy using efficient data structures and algorithms that do capture evidence multiplicity.

To enumerate the DCs, Hydra initially assumes single predicate DCs to be valid. Then, it iterates each piece of evidence in the evidence set and adds predicates to the DCs violated by that piece of evidence. Each iteration conforms the current DCs to the current piece of evidence, so at the end of the process, the set of DCs is valid and complete regarding the entire evidence set. This enumeration scheme works only for exact DCs, and it is unclear how to adapt it for the approximate DC case [4].

Livshits et al. [17] integrate the evidence set building of DCFinder with a hitting set enumeration algorithm called Minimal-to-Maximal Conversion Search with pruning (MMCS) [19]. The MMCS algorithm uses a set system based on essential properties of hitting sets, which enables the algorithm to perform a depth-first search with several pruning strategies. These properties also enable fast operations on the data structures that check and validate the hitting sets (i.e., DCs). The authors also adapt the MMCS algorithm for the approximate DC by changing the base case and including additional routines.

In Section 4, we propose to integrate our enumeration techniques with the enumeration scheme of Hydra and the MMCS algorithm in hybrid approaches, which can improve enumeration performance.

3 FAST EVIDENCE SET BUILDING

This section presents the evidence context pipeline (ECP), our solution for efficiently building evidence sets with multiplicity. The main shortcomings of previous solutions lie in the quadratic number of evidence they compute (in [5]) or allocate (in [17, 20, 21]). With ECP, we can process evidence from multiple tuples simultaneously, reducing the overall number of computations and memory allocations required. A key insight implemented in ECP is to operate on compact data structures that bind each distinct evidence to the tuples that produce that evidence, the *evidence contexts*.

Each ECP comprises a series of stages that perform *evidence context corrections*. These corrections take as input a predicate group and a set of evidence contexts whose evidence and tuples are not yet correct according to that predicate group. They operate the evidence and tuples in each evidence context and produce a new set of evidence contexts, corrected according to the underlying predicate group. The idea is to transform evidence contexts incrementally until they represent the correct and complete evidence set. For performance, we have designed correction algorithms that exploit a strong relationship between the evidence context representation and logical (bitwise) operations.

3.1 Evidence Context

There is often much redundancy in the evidence space in the sense that many tuples pairs produce the same evidence [17, 21]. As a result, the number of elements in the evidence set is much smaller than the total number of tuple pairs. We capture such redundancy by processing evidence contexts, thereby avoiding processing or visiting each piece of evidence individually.

An evidence context is represented as a triple $\texttt{ect} = \langle \texttt{t}, \texttt{tids}, \texttt{e} \rangle$ comprising a tuple t, a set tids of tuples, and a piece of evidence e. Each triple is interpreted as follows: "tuple t, when combined with any tuple $\texttt{t}' \in \texttt{tids}$, produces the piece of evidence e". In other words, each triple ect denotes that the tuple t produces the same piece of evidence e by |tids| times. Thus, the evidence multiplicity $count(\texttt{e})$ in r is given by summing the number of tuples |tids| of every triple ect from r having the piece of evidence e.

Evidence contexts provide an efficient way to store evidence. Consider tuple t_4 from the employee relation and the predicate space in Figure 1. We obtain among others the following evidence context: $\langle t_4, \{t_1, t_2, t_3\}, \{p_2, p_4, p_6, p_9, p_{10}, p_{12}, p_{14}, p_{15}\}\rangle$. This evidence context represents the tuple pairs (t_4, t_1), (t_4, t_2), and (t_4, t_3) but requires us to store only four integers (tuple ids) plus a single piece of evidence instance. For comparison, the FastDC algorithm would enumerate the three pairs, computing the same piece of evidence each time. The BFastDC and DCFinder algorithms reduce the computation time but still need to allocate and visit three pieces of evidence individually. ECP avoids the high overhead from pairwise iteration and memory allocations with algorithms exploiting the highly compressible evidence contexts. Naturally, the evidence context efficiency becomes much more evident as the number of rows increases.

3.2 Pipeline

For each tuple t, we run a pipeline that incrementally forms a set ECTs of all evidence contexts for t. First, we initialize ECTs with a single evidence context $\texttt{ect} = \langle \texttt{t}, \texttt{tids}, \texttt{e} \rangle$, where tids is the set of every tuple of r except t, and $\texttt{e} = \{\texttt{p} \in \texttt{P} \mid \texttt{p.op} \in \{\neq, >, \geq\}\}$. We use this initial set tids because we need the evidence of every $\texttt{t} \in \texttt{r}$ combined with every other tuple of r. As for the initial piece of evidence e, the goal is to improve performance by minimizing the number and costs of updates in the evidence contexts of each stage. At a high level, we initialize evidence contexts with predicates that are more likely to require less data movement from evidence context corrections (see Sections 3.3 and 3.5).

The set ECTs goes through a pipeline, where each stage implements the evidence correction for a predicate group. The corrections update ECTs such that each element conforms to the current predicate group. For example, we may need to remove tuples from tids that do not satisfy predicates on $\{\neq, >, \geq\}$; or create new evidence contexts whose tids satisfy predicates on $\{\neq, <, \leq\}$ or $\{=, \leq, \geq\}$. Due to the updates, new evidence contexts might arise, while others might disappear. The set ECTs of each tuple t is complete at the end of the pipeline, since it contains all (corrected) evidence contexts associated with t. At this stage, we extract the pieces of evidence with their multiplicities. Since the sets ECTs are independent of each other for different tuples, we can run multiple ECPs in parallel by synchronizing the concurrent access to the final evidence set.

As an example, Figure 2 illustrates the ECP execution for tuple t_1, considering the subset of predicates p_1–p_{12} of Figure 1. The final pieces of evidence for t_1 are identified at the last stage. The evidence e_4 also appears in the contexts: $\langle t_4, \{t_1, t_2, t_3\}, e_4\rangle$, $\langle t_5, \{t_2\}, e_4\rangle$, $\langle t_6, \{t_2, t_3\}, e_4\rangle$, and $\langle t_7, \{t_2, t_3\}, e_4\rangle$. By summing the number of elements in the sets tids of the contexts having e_4 we obtain the multiplicity of e_4 in employee, that is, $count(e_4) = 10$.

3.3 The role of predicate selectivity

Predicates with the operator $\neq$ tend to be much less selective than predicates with the operator = (unless for particular distributions,

Stage	Evidence Contexts
	$\langle t_1, \{t_2, t_3, t_4, t_5, t_6, t_7\}, \{p_2, p_4, p_6, p_9, p_{10}, p_{12}\}\rangle$
$p_1 - p_2$	$\langle t_1, \{t_2, t_3, t_4, t_5, t_6, t_7\}, \{p_2, p_4, p_6, p_9, p_{10}, p_{12}\}\rangle$
$p_3 - p_4$	$\langle t_1, \{t_2, t_3, t_4, t_6, t_7\}, \{p_2, p_4, p_6, p_9, p_{10}, p_{12}\}\rangle$, $\langle t_1, \{t_5\}, \{p_2, p_3, p_6, p_9, p_{10}, p_{12}\}\rangle$
$p_5 - p_{10}$	$\langle t_1, \{t_2, t_3, t_6, t_7\}, \{p_2, p_4, p_6, p_9, p_{10}, p_{12}\}\rangle$, $\langle t_1, \{t_4\}, \{p_2, p_4, p_6, p_7, p_8, p_{12}\}\rangle$, $\langle t_1, \{t_5\}, \{p_2, p_3, p_6, p_9, p_{10}, p_{12}\}\rangle$
$p_{11} - p_{12}$	$\langle t_1, \{t_2, t_3\}, e_1\colon \{p_2, p_4, p_6, p_9, p_{10}, p_{11}\}\rangle$, $\langle t_1, \{t_4\}, e_2\colon \{p_2, p_4, p_6, p_7, p_8, p_{12}\}\rangle$, $\langle t_1, \{t_5\}, e_3\colon \{p_2, p_3, p_6, p_9, p_{10}, p_{12}\}\rangle$, $\langle t_1, \{t_6, t_7\}, e_4\colon \{p_2, p_4, p_6, p_9, p_{10}, p_{12}\}\rangle$

Figure 2: ECP for tuple t_1 and predicates p_1–p_{12} of employee.

e.g., single-valued columns). For instance, for tuple t_1, we find only tuple pairs satisfying the predicate p_2 : t.ID ≠ t′.ID. By prioritizing the predicate p_2 over p_1, there is no need to reconcile any evidence context for p_1. Suppose we had prioritized the other way around (p_1 over p_2), we would need to move six tuples to correct the evidence context at the $p_1 - p_2$ stage. Of course, some cases do require moving tuples for equality predicates. However, the selectivity of such predicates is often much smaller than the selectivity of the "different than" counterpart. Thus, we significantly increase the chances of the correction operating on much smaller set of tuples.

When we prioritize the operator ≠ for predicates on numeric columns, we must choose the inequality direction. Since there is a selectivity equivalency between directions $\{<, \leq\}$ and $\{>, \geq\}$, we can expect the correction algorithms to perform equivalent work either way. Thus, we choose to initiate the initial predicates in the $\{>, \geq\}$ direction and provide algorithms to reconcile evidence contexts for the opposite direction. Notice that by doing so, we already have a significant portion of the evidence contexts in the correct state, so the correction algorithms have less data to move.

3.4 Indexes for Evidence context correction

Given a column A and a value v, our correction algorithms use specialized indexes for two operations: equals(A, v), which returns the set tids of every tuple t having t[A] = v; and greaters(A, v), which returns the set tids of every tuple t having t[A] > v. To implement equals, we build a hash table that maps each unique column value into the set of tuples having that value. For instance, the hash table for the Salary column of the employee relation contains the entries: $\langle 5500, \{t_2\}\rangle$, $\langle 6000, \{t_3, t_5\}\rangle$,$\langle 7000, \{t_6, t_7\}\rangle$, $\langle 10000, \{t_1\}\rangle$, and $\langle 11000, \{t_4\}\rangle$.

We build an additional index for the greaters operation. For low cardinality columns, we use the sorted set from the keys of the equals hash table. Let the sorted set of keys for a column A be $v_1, \ldots, v_k$. The bitmap entry for each column value v_j ($1 \leq j < k$) is computed with the union equals(A, v_{j+1}) ∪ . . . ∪ equals(A, v_k). For example, from the Salary column, we obtain the following entries: $\langle 11000, \{\}\rangle$,$\langle 10000, \{t_4\}\rangle$,$\langle 7000, \{t_1, t_4\}\rangle$, $\langle 6000, \{t_1, t_4, t_6, t_7\}\rangle$, and $\langle 5500, \{t_1, t_3, t_4, t_5, t_6, t_7\}\rangle$.

For high cardinality columns (boundaries defined later), we use a two-layered bitmap index with binning to save memory. We split the column values into ranges, i.e., bins, and use a bitmap to represent each range rather than distinct values. Like the procedure for low cardinality columns, we build a bitmap index using the bitmaps of (sorted) ranges—the first layer. We then build a bitmap index for each range, using only the tuples within that range—the second layer. We opt for equi-depth binning to divide the ranges, so that each bin contains approximately the same number of tuples. The goal is to make each check in the second layer index equally expensive. Each call to greaters requires computing the union between the bitmap entries of the first and second layers.

Consider we are using the binning approach with the Salary column and two bins. The first layer of the index is built from entries: $\langle [5500, 7000), \{t_2, t_3, t_5\}\rangle$ and $\langle [7000, 11000], \{t_1, t_4, t_6, t_7\}\rangle$. Suppose we are looking for key 5500, that is, all tuples having a Salary value higher than 5500. The first layer returns $\text{tids}_1 = \{t_1, t_4, t_6, t_7\}$. The second layer is built only from tuples $\{t_2, t_3, t_5\}$, and returns $\text{tids}_2 = \{t_3, t_5\}$ when it is probed. Observe that with $\text{tids}_1 \cup \text{tids}_2 = \{t_1, t_3, t_4, t_5, t_6, t_7\}$ we obtain precisely the result for greaters(Salary, 5500).

In the first layer, the number of bitmaps is limited by the number of bins. On the other hand, the bitmaps in the second layer are much more compact than a non-binned counterpart because they store only the identifiers of tuples in a specific range. We observed during experiments that enabling binning for columns with more than 2000 distinct values and setting the number of bins to around 500 works well for all tested datasets. Similar results have also been obtained in related work using bitmap indexes [22].

3.5 Algorithms for evidence context correction

Each stage of an ECP takes as input a set of evidence contexts and a predicate group and then processes each context to correct its evidence and tids according to that group. Predicates of the form t.A θ t.B are straightforward to handle, as they involve only one tuple. For such cases, we evaluate the predicates once per ECP and then include the result in each evidence context at the end of the pipeline. Observe, for example, that all evidence contexts for tuple t_4 can include only the predicate p_{15}. In turn, the correction for other predicate groups requires algorithms that find the parts of the evidence context currently incorrect and fix these parts according to the current predicate group.

Correction for categorical predicates. Algorithm 1 shows the evidence context correction for predicates of the form t.A θ t′.A where A is a categorical column. Each evidence context ect ∈ ECTs already includes the inequality t.A ≠ t′.A, so our first goal is to identify the parts of tids in each ect that should include the equality counterpart. Tuples t with a unique A value are different from every other tuple for A, so there is no need for correction (case in Line 2). The variable *fixEquals* extracts from each ect the tuple subset with the A value equal to the A value of the current tuple t. This subset contains the tuples that should include t.A = t′.A. Whenever such subsets are non-empty, we create a new evidence context with the new piece of evidence holding the equality configuration (Lines 12 and 13). We include this new context for the next pipeline stages in Line 14. Also, we must correct the tids of

each ect by removing from them the tuples satisfying the equality (Lines 8–11). This step causes evidence contexts to disappear when there is no tuple partner to satisfy the inequality (Line 9).

Algorithm 1: CategoricalStage(t, A, ECTs)

```
1  equals ← equals(A, t[A])
2  if equals.size = 1 then
3      return
4  newECTs ← ∅
5  foreach ect ∈ ECTs do
6      fixEquals ← equals ∩ ect.tids
7      if fixEquals.size > 0 then
8          if ect.tids = fixEquals then
9              remove ect from ECTs
10         else
11             ect.tids ← ect.tids \ fixEquals
12         e ← CopyReconcile(ect.e, t.A θ t′.A, {=})
13         newECTs ← newECTs ∪ ⟨t, fixEquals, e⟩
14 ECTs ← ECTs ∪ newECTs
```

For predicates of the form $t.A\ \theta\ t'.B$ where both A and B are categorical columns, the algorithm above requires minor changes. First, the lookup is equals(B, t[A]). Observe that an empty set of tids means that column B does not contain the value t[A]. In this case, since $t.A \neq t'.B$ is true for every tuple pair where $t \neq t'$, so no correction is required. Also, no correction is required whenever equals(B, t[A]) returns a set containing only tuple t. In this case, the predicate $t.A = t.B$ is true but $t.A = t'.B$ is not.

Correction for numerical predicates. Algorithm 2 shows the evidence context correction for predicates of the form $t.A\ \theta\ t'.A$ where A is a numerical column. The intuition is similar to the categorical case, but now evidence contexts initially cover a predicate direction with operators $\{\neq, >, \geq\}$. As a result, we need to correct pieces of evidence for the portions of tids in each ect that instead should include either the direction $\{=, \leq, \geq\}$ or $\{\neq, <, \leq\}$. The variable *fixEquals* covers the $\{=, \leq, \geq\}$ case, whereas *fixGreaters* covers $\{\neq, <, \leq\}$. Mind that *fixGreaters* contains the tuples with an A-value greater than t[A]. Whenever all the tuples in ect.tids have an A value equal to t[A] (case in Line 8), we need to correct ect only for $\{=, \leq, \geq\}$ because no tuple in ect.tids can satisfy neither the $\{\neq, <, \leq\}$ nor the $\{\neq, >, \geq\}$ direction. Otherwise, we remove from the current ect.tids all tuples satisfying the equality for A (Line 12). Then, we reconcile this new set ect.tids for the $\{\neq, <, \leq\}$ direction through Lines 16–22. The case in Line 21 occurs when all the tuples in the remaining ect.tids have an A-value less than t[A].

For predicates of the form $t.A\ \theta\ t'.B$ where both A and B are numerical columns, the search for variable *equals* is equals(B, t[A])—the cases with no join partner being like the case for categorical columns. As for variable *greaters* the searches are *greaters*(B, t[A]), but we need to account for three base cases. First, the value t[A] is in the domain of B, so the return is straightforward. Otherwise, we try returning *greaters*(B, v) for a value v in B that is strictly less than t[A]. If such value does not exist, than t[A] is less than every other value in B, and the result is all but tuple t.

Algorithm 2: NumericalStage(t, A, ECTs)

```
1  equals ← equals(A, t[A])
2  greaters ← greaters(A, t[A])
3  newECTs ← ∅
4  foreach ect ∈ ECTs do
5      skipRange ← false
6      fixEquals ← equals ∩ ect.tids
7      if fixEquals.size > 0 then
8          if ect.tids = fixEquals then
9              skipRange ← true
10             remove ect from ECTs
11         else
12             ect.tids ← ect.tids \ fixEquals
13         e ← CopyReconcile(ect.e, t.A θ t′.A, {=, ≤, ≥})
14         newECTs ← newECTs ∪ ⟨t, fixEquals, e⟩
15     if not skipRange then
16         fixGreaters ← greaters ∩ ect.tids
17         if fixGreaters.size > 0 then
18             e ← CopyReconcile(ect.e, t.A θ t′.A,
                 {=, <, ≤})
19             newECTs ← newECTs ∪ ⟨t, fixGreaters, e⟩
20         ect.tids ← ect.tids \ fixGreaters
21         if ect.tids.size = 0 then
22             remove ect from ECTs
23 ECTs ← ECTs ∪ newECTs
```

3.6 Complexity and Optimizations

The complexity of Algorithms 1 and 2 is dominated by the number of evidence contexts along each pipeline. This number starts at one but grows throughout the pipeline stages. The size of the predicate space influences the growth rate, as each predicate group requires corrections that may split evidence contexts. In our experiments, we did not observe a significant influence from the number of records on the growth rate. In general, we observed that the number of evidence contexts would usually grow to hundreds or, in the worst cases, to a few thousand. Such a result reflects the evidence set redundancy discussed in Section 3.1.

Another computational cost factor is the implementation of the intersections and differences between integer sets; for instance, in Lines 6 and 11 of Algorithm 1. We use roaring bitmaps to store and operate these sets, as they provide one of the best trade-offs between performance and storage requirements [15]. The data structure enables us to store the evidence contexts from each pipeline using a couple of megabytes, as we show in our experiments. Also, we can apply the following heuristics to improve performance.

We sort the table on the numerical columns, as row-reordering can improve bitmap compression [12, 14]. Observe that the entries of greaters indexes are built from one another incrementally. By sorting the entire table on a column list $[A_1, A_2, \ldots]$, we increase the potential for each index entry to contain more compressible tids. This potential is high for column A_1, but wears off as we move to the next columns. Although determining the optimal sorting order is

NP-hard [23], sorting based on column cardinality has been shown to work well [12, 14, 23]. Thus, we sort the table on columns in decreasing cardinality (estimated using the accurate HyperLogLog sketches [7]). The higher the cardinality, the higher the number of index entries. So, this sorting scheme makes more index entries more likely to benefit from better bitmap compression.

Our other heuristics prioritize index probes that are more likely to return tids with fewer tuples, as we seek evidence corrections that move fewer data. We correct evidence for categorical predicate groups first, as the equality predicates are much more selective than "less than" inequalities. Among predicate groups, we correct evidence for predicate groups on high-cardinality columns first, since their predicates tend to be more selective. Notice that this last heuristic is directly connected with the sorting one.

4 FAST DC ENUMERATION

Previous works have shown that it is possible to enumerate all *non-trivial* and *minimal* DCs of a relation instance r from its underlying evidence set E_r [4, 5]. The main intuition is that a DC $\varphi: \neg(p_1 \wedge \ldots \wedge p_m)$ holds in r if there exists no evidence violating φ, i.e., no evidence satisfying all predicates of φ. Thus, discovering DCs is equivalent to enumerating all (minimal) predicate sets $\{p_1 \wedge \ldots \wedge p_m\}$ such that $\nexists e \in E_r: \{p_1 \wedge \ldots \wedge p_m\} \subseteq e$. We refer to such predicate sets as *negative covers*. In this section, we present DC enumeration algorithms based on the negative cover property.

In Section 4.1, we present a redesign of the minimal cover search (MCS) used in [5, 20, 21]. In Section 4.2, we present hybrid algorithms that combine our redesign with the two other algorithms from the state-of-the-art [4, 17]. We refer to the original publications for proofs and complexity analysis of the base algorithms we extend.

4.1 Indexed Negative Cover Search (INCS)

Our INCS algorithm is based on the MCS algorithm, which uses a depth-first search to enumerate the negative covers of the evidence set, as shown in Algorithm 3. It takes as input: a predicate set *path* representing a DC candidate; the list preds of predicates used to create new nodes; and the evidence set E whose evidence violate the DC candidate relative to *path*.

The first call is INCS($\emptyset$, P, E_r) with an empty *path*, the predicate space P as a list, and the evidence set E_r of the relation instance. Predicates are added from preds into *path'*, and the recursive calls in Line 14 form the nodes of the search tree. Suppose we add a predicate p into *path'*. In that case, we must create its respective evidence set E' by removing from the current evidence set E the evidence that no longer violates the DC candidate relative to *path'*. That is, the set E' is given by the subset of E of all evidences that do not contain p. The leaf nodes represent the two base cases. The first may occur in Line 1, where the empty evidence set implies that, for the current *path* = $\{p_1, \ldots, p_m\}$, $\nexists e \in E_r$ such that $\{p_1 \wedge \ldots \wedge p_m\} \subseteq e$. In this case, we found a negative cover, which is a DC $\varphi: \neg(p_1 \wedge \ldots \wedge p_m)$. The second base case may occur in Line 4, where there are still violating evidences to cover, but no remaining predicates to add; hence there are no DCs down that branch.

We use two optimizations from [5, 26]. First, we sort the list of remaining predicates for each new node in Line 6. Predicates that intersect the least with the current evidence set come first, as we prioritize predicates tied to the least number of violating evidence. As a result, we form branches where the longest lists preds' are tied with the smallest set E'. The longer preds' is, the higher the number of branches. As the branches are tied to a smaller E', we are more likely to require fewer iterations. Second, we use the modulo evidence set principle from [5] in Lines 9 and 11 to decompose the search space into subspaces based on whether paths can include a specific predicate or not. The subspaces have the advantage of leveraging fewer predicates and smaller evidence sets.

Algorithm 3: NegativeCoverSearch(*path*, preds, E)

```
1  if E = ∅ then
2      Add path as a DC to the output
3      return
4  else if E ≠ ∅ and preds.length = 0 then
5      return
6  sort preds on ascending order of intersection regarding E
7  for k=0 to preds.length −1 do
8      p ← preds[k]
9      preds' ← preds[k + 1, preds.length −1]
10     remove trivial and implied predicates of preds' w.r.t. p
11     E' ← filtered E (w.r.t. p and preds')
12     if E' ≠ NIL then
13         path' ← path ∪ p
14         NegativeCoverSearch (path', preds', E')
```

The routines in Algorithm 3 need to be efficient since they are called many times due to the high number of DC candidates. In the following, we describe our strategies for efficiency and how to adapt the algorithm for approximate DC discovery.

Late minimality check. The two optimizations described earlier speed up the search, but they may include redundant nodes for which the negative covers are non-minimal. The MCS algorithm checks candidate minimality based on the DCs previously discovered. Such an approach is computationally expensive because of the subset lookups for every candidate, even candidates whose branches never form DCs. Instead, our INCS algorithm allows redundant nodes, but once the INCS is finished, it checks the minimality of the negative covers discovered.

We initiate a search tree with every negative cover discovered. Then, for each negative cover, we check if there is no proper subset of that cover in the tree. If not, that negative cover is minimal and can be retained. After this check, only the minimal DCs remain in the result. We use the binary tree proposed in [3] to perform fast subset lookups. In practice, the overhead of minimality checking is low compared to the entire enumeration runtime, as it regards only the discovered DCs.

Parallel search. The late minimality check enables processing the subspaces independently using multiple threads. We use a hash set supporting concurrent inserts to receive the negative covers from these subspaces, which are minimized after all threads finish. In principle, we can start the parallel threads at any level down the tree. We tested the algorithm by starting the threads at different levels and found that doing so once after one-level decomposition

performs best. The reason is that the number of individual predicates is already high, so concurrent threads already find room to improve upon a sequential execution. If we start the threads further down the tree, the number of subspaces becomes large, and the multi-threading overhead quickly starts hurting performance.

Evidence set filtering. The filtering routine in Line 11 arranges the evidence sets for the branching paths. Instead of iterating the evidence set every time, we propose using an inverted index. We assign an identifier (id) to each piece of evidence. Then, we associate with each predicate a bitmap with the ids of pieces of evidence in which it occurs. In this context, evidence sets are represented as id bitmaps. Thus, in Line 11, we filter the current evidence set using an intersection (i.e., a logical "AND" operation) between its bitmap and the bitmap associated with the predicate for the next node.

In principle, we could use the same index for the entire search, In practice, rebuilding the index after filtering the evidence set may improve performance, as the number of elements in the evidence set often reduces. We found that rebuilding the index works well with the initial evidence set filtering (for one predicate). Rebuilding the index in deeper levels of the search tree does not pay off, as there are many more nodes, and rebuilding the index takes time proportional to the number of elements in the evidence set.

Also, we can remove the predicates $\texttt{preds}[0, k]$ from each piece of evidence of the current evidence set since these predicates are handled in other branches. As we remap these modified pieces of evidence, we are likely to obtain an even smaller evidence set: the number of possible distinct pieces of evidence reduces with fewer predicates. After remapping, we assign new ids to each (reshaped) piece of evidence, associate these ids to the bitmap of the remaining predicates, and continue the search from that branch forward using the new index system.

Pruning. We can prune unnecessary branches as follows. First, in the filtering phase, we check if the filtered evidence set references any piece of evidence that contains all predicates of $\texttt{preds}'$ (the remaining predicates to be added). If such a piece of evidence is found, the branches from the current node can never reach a point where $\texttt{E} = \emptyset$. In this case, we return NIL to skip those branches. Second, whenever we choose a predicate p in Line 8, we remove from $\texttt{preds}'$ all predicates from the predicate group of p, that is, the predicates that differ from p only by the operator. Those predicates can only result in non-minimal or trivial DCs. Third, while sorting in Line 6, we remove from preds any predicate p whose bitmap includes all the pieces of evidence of the current evidence set E. In this case, p can never contribute to removing any piece of evidence of E, so adding p to the path would only result in non-minimal DCs.

Approximate negative covers. The authors of [5] have shown that discovering approximate DCs is equivalent to enumerating *approximate covers* of the evidence set. Also, they have shown how to modify the MCS algorithm to enumerate these covers. The idea is to use the evidence multiplicity to compute the approximation function g_1 (Section 2). Similarly, we propose AINCS, an adaption of the INCS algorithm that enumerates approximate DCs, as follows.

First, we modify the base case in Line 1 to support approximate covers. Instead of empty evidence sets ($\texttt{E} = \emptyset$), we now search for predicate sets *path* such that $g_1(\texttt{r}, path) \leq \varepsilon$. The g_1 value for each *path* is the sum of the multiplicities in the current evidence set, as each piece of evidence $\texttt{e} \in \texttt{E}$ represents a violation of the current *path*, and each *count*(e) represents how many tuple pairs produced that violation. Second, we modify the pruning strategy in the evidence set filtering. In the exact DC case, a single piece of evidence subsuming $\texttt{preds}'$ is enough to prune the current branch. For the approximate DC case, we iterate each piece of evidence subsuming $\texttt{preds}'$, summing their multiplicities. When this sum exceeds ε, no further *path* could ever be able to satisfy the approximation criteria, so we can prune that branch by returning NIL.

4.2 Hybrid methods for DC enumeration

We now show how to integrate the INCS algorithm with the two other DC enumeration algorithms in related work: evidence inversion (EI) from [4] and the MMCS from [19]. We observed that the performance of these other algorithms is significantly impacted by the evidence set size, i.e., number of elements. Our insight is first to call INCS to obtain a set of search paths, each path connected with a smaller evidence set. Then, we feed these search paths into the other algorithms, which can perform the search fast due to the smaller evidence sets. We denote our hybrid version of EI as HEI, and our hybrid version of MMCS as HMMCS.

Observe in Lines 8–11 of Algorithm 3 that each predicate p derives a subspace of predicates $\texttt{preds}'$ that further forms new branches with its respective evidence set $\texttt{E}'$. Like for the INCS algorithm, we can apply the subspace concept from [5] to discover DCs from the smaller evidence sets $\texttt{E}'$ and spaces $\texttt{preds}'$. First, we perform the first level of INCS, but instead of continuing with the recursive calls of INCS, we can call either EI or MMCS algorithms.

For each predicate p, we collect the filtered (and reshaped) evidence set $\texttt{E}'$ and predicate list $\texttt{preds}'$, giving them as input to EI/MMCS calls. Each call to EI returns a set NC of negative covers for $\texttt{E}'$. From each $nc \in \texttt{NC}$, we derive a valid (but not always minimal) DC $\varphi\colon nc \cup \texttt{p}$. In that way, we guarantee output minimality. For MMCS calls, the results are sets of (minimal) positive covers for $\texttt{E}'$, that is, predicate sets that intersect with every element of $\texttt{E}'$. In this case, the derived DCs are given by the inverse of the covers [5]. The final DCs are given after we apply the late minimization strategy.

One advantage of the hybrid approach is that the input sizes for EI/MMCS algorithms are smaller due to the reduced evidence sets $\texttt{E}'$ and remaining predicate spaces in $\texttt{preds}'$. Of course, the counterpoint is that we now require many executions of the algorithms. If we call these algorithms further down the search tree (i.e., higher than the first level), we significantly increase the number of calls, worsening performance. Thus, we only perform the calls only once after the first level. The second advantage is that, as we do with INCS, we can execute the calls to EI/MMCS in parallel because we use the late DC minimization.

5 EXPERIMENTAL EVALUATION

The experiments presented in this section investigate four main aspects of the proposed algorithms: (i) performance regarding runtime and memory consumption; (ii) comparison to the fastest algorithms in the state-of-the-art; (iii) scalability with the number of rows and columns of the input; and (iv) impact of the design decisions and optimizations to performance.

5.1 Experimental setting

Table 2 shows the datasets used in our experiments (which were also used in [17, 21]), along with exact DC discovery runtimes (discussed in Section 5.2) of three different algorithms. The ECP/HEI-P is our ECP algorithm combined with the parallel version of our HEI algorithm, denoted as HEI-P. We used HEI-P, as it was often the fastest DC enumeration (as experimented in Section 5.4). Hydra [4] and DCFinder [21] algorithms represent the previous state-of-the-art. ECP/HEI-P and DCFinder run on parallel threads. In turn, the several phases of Hydra make it non-trivial to parallelize. While such an extension is an interesting subject for future work, our experiments use the algorithm the way the authors proposed it.

Table 2: Datasets used for evaluation and (exact) DC discovery runtimes (in seconds) of state-of-the-art algorithms.

Dataset	#Cols	#Rows	Size	ECP/ HEI-P	Hydra*	DCFinder
Adult	15	32k	3.6 MB	**53**s	1 089s	324s
Airport	18	55k	7.1 MB	**19**s	789s	71s
Flight	20	500k	53.6 MB	**251**s	6 346s	3 958s
Food	19	174k	194.9 MB	**163**s	1 017s	477s
Hospital	15	114k	30.6 MB	**5**s	19s	69s
NCVoter	22	938k	191.4 MB	**1 555**s	7 895s	29 703s
Tax	15	1M	73 MB	**224**s	1 149s	11 816s

*Hydra is limited to exact DC discovery.

We compare our algorithms for evidence set building and DC enumeration to the equivalent ones in the state-of-the-art. We use only DCFinder and Hydra for evidence building because they are consistently much faster than the other options, FastDC and BFastDC [4, 21]. We use all existing algorithms for DC enumeration since we aim to investigate how they benefit from the strategies in Section 4. In most experiments, we used the exact DC discovery, so that Hydra could be included. However, recall from Section 2 that approximate DC discovery is computationally much harder. We discuss such computation differences in Sections 5.3 and 5.4.

We used the Java implementation of DCFinder and Hydra provided by the authors [4, 21]. The MCS implementation of DCFinder includes two optimizations that have not been described in its original proposal [5]: a prefix tree for DC lookup and the first pruning rule described in Section 4.1. We left this optimization as is when comparing the entire algorithms, but disabled it when comparing DC enumeration only (Section 5.4). We used the C++ implementation of the MMCS from [8]. It includes a parallel version of MMCS that suits our comparisons. We implemented most of our algorithms in Java. The exception is the HMMCS algorithm, which requires us to write the in-memory evidence set as a disk file and call its C++ implementation with this file as input. Other than that (and data loading), all algorithms run in main memory.

For consistency with the previous work [4, 21], we replace the NIL values of column values with default values. We used empty strings for categorical columns and $-\infty$ for numeric columns. Although null semantics for integrity constraints is a vibrant line of research (e.g., [25]), it is not the focus of this study, hence the practical definition. Also, for consistency, we follow the same predicate space restrictions used in previous studies [4, 5, 21].

All experiments were run on a Dell PowerEdge R730 server with two Intel Xeon E5-2690 v3 @2.60GHz CPUs (12 cores, 24 threads), 768 GB of RAM, running Linux CentOS 7.7.1908 and OpenJDK 64-Bit Server VM 1.8.0_232 with the JVM heap space limited to 32 GB. The numbers reported are the average of (at least) three executions.

5.2 Overview of the results

The runtimes in Table 2 show that, in absolute terms, ECP/HEI-P algorithm is consistently faster than all competitors (by an order of magnitude in several cases). Switching our DC enumeration algorithm in this experiment would not have changed the general result order because (i) for most datasets, the runtime is dominated by evidence set building, and (ii) the performances of the other algorithms are often close to each other, as discussed in Section 5.4.

The overall performance of DC discovery algorithms can be broken down into two pieces, (1) the efficiency in building the evidence set and (2) the efficiency in enumerating the DCs from the evidence set. For a clearer perspective, we investigate each of these pieces individually in the following sections.

5.3 Evidence set building evaluation

Figure 3 depicts the runtime (only evidence set building) of ECP, Hydra and DCFinder, for an increasing number of rows. It also depicts the number of DCs discovered. Overall, ECP exhibits the best row scaling behavior.

DCFinder is significantly affected by a higher number of rows, for instance, on *Tax* dataset. In turn, ECP scales much more smoothly. The different scaling behavior of these algorithms is expected: DCFinder visits a quadratic number of intermediates to compute evidence multiplicity, whereas ECP computes the same information with evidence contexts—which are much more efficient. We observed that only for a small number of rows (e.g., *Adult*), DCFinder might be slightly faster (in evidence building). The turning point was usually about 50k rows. For small datasets, the evidence context finds not much room for compression, so the initial overhead from ECP (input sorting and index building) does not pay off.

Hydra is impacted by the number of rows and, in particular, by the number of DCs the dataset contains. It requires detecting the tuple pairs that violate an initial set of DCs, discovered from a sample. The higher the number of DCs, the more time is spent detecting DC violations. In cases where the number of DCs decreases with the increasing number of rows, this fact counterbalances the higher row count. It might sometimes even benefit performance (e.g., the negative slope for *Tax* with 1M rows). Also, Hydra presents high runtime variance: some samples produce DCs whose violation detection cost is higher than the cost of DCs of other samples. The runtime stabilized for *NCVoter* as a high number of rows do not produce much new evidence. Unfortunately, such behaviors are difficult to predict due to the natural behavior of sampling. On the other hand, ECP follows a more predictable scaling behavior. The runtimes are much lower and generally increase more gradually, which enlightens the efficiency of our approach.

The difference between ECP and Hydra also regards the amount of information they need to produce. We observed that Hydra "touches" only a fraction of all tuple pairs (e.g., less than 0.1% in many inspected datasets). Also, it can often visit the same tuple pair

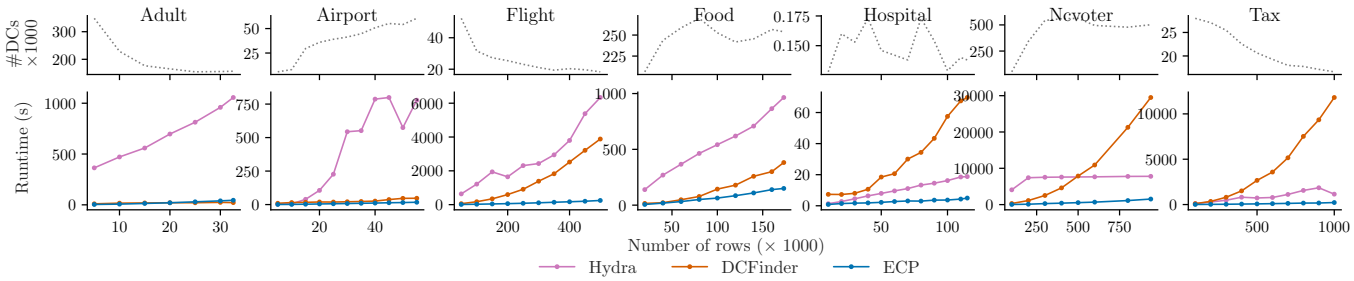

Figure 3: Row scaling of the three fastest algorithms for evidence set building.

many times due to sampling. As a result, its evidence set multiplicity distribution becomes distorted compared to that of a specialized algorithm such as ECP, DCFinder, or FastDC.

For example, we ran the algorithm ECP–AINCS, on all tested datasets on many violation thresholds, to obtain gold standards of approximate DCs. Then, we ran Hydra–AINCS to compare its outputs with the gold standards. For Hydra, the thresholds needed to be applied for the total amount of tuple pairs visited by the algorithm, to adjust tuple pair proportions. The fraction of (correct) approximate DCs was always low, e.g., nearly 30% in the best scenario, illustrating the unreliability of Hydra for tasks requiring evidence multiplicity.

Next, we compare the parallel scalability of ECP and DCFinder for the three datasets with the highest number of rows. Figure 4 shows the parallel speedup obtained by these algorithms relative to a single-threaded execution. Both algorithms show good parallel scalability, with better scalability up to the maximum number of physical cores. These results demonstrate the parallel efficiency of ECP. For example, we observe self-relative speedups of up to 15.74× with ECP and up to 14, 27× with DCFinder (both on *Flight*).

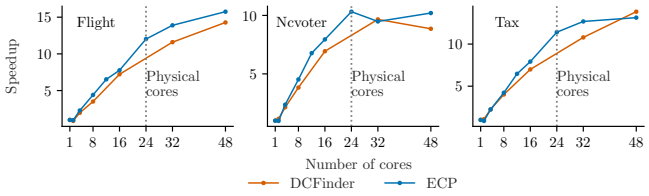

Figure 4: Speedup of parallel evidence building strategies.

5.4 DC enumeration evaluation

We used the following protocol for the experiments in this section. With the ECP algorithm, we built the evidence set of the first 10 000 records of the datasets and then gave it as input to the DC enumeration algorithms. In general, we observed that DC enumeration is not much affected by the number of rows, but it is very much affected by the number of columns, hence predicates. Therefore, we evaluated each algorithm across increasing numbers of columns and predicates. We varied the number of columns in the datasets by randomly selecting different column combinations. To accommodate the randomness of the samples, we report the average running time of ten executions for each column number. Mind that, in this section, we report only DC enumeration runtime and we use a log scale in the vertical axes of Figures 5, 6, 7, and 8.

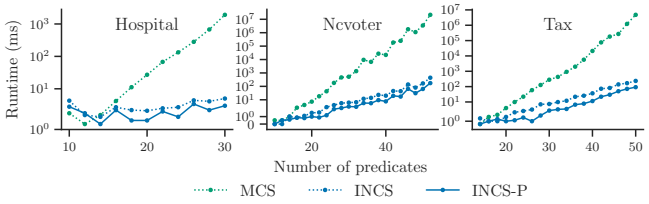

Figure 5: Predicate and column scaling of MCS vs. INCS.

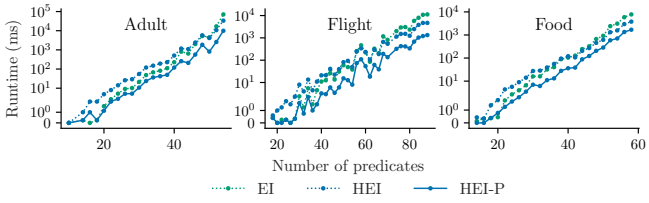

Figure 6: Predicate and column scaling of EI vs. HEI.

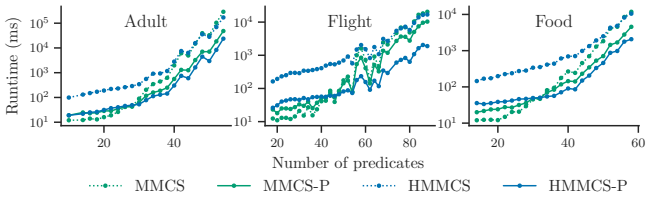

Figure 7: Predicate and column scaling of MMCS vs. HMMCS.

Figure 5 shows the runtime of our indexed negative cover search INCS, its parallel version INCS-P, and the standard minimal cover search MCS as proposed in [5]. We used the datasets where MCS yields its lowest runtimes for this comparison. Still, INCS is orders of magnitude faster: where MCS takes more than one hour to complete, INCS takes under a second. This result shows the significant performance gains from a lighter candidate checking using evidence indexing, late minimization, and evidence set pruning. Since INCS was consistently faster than MCS, we compared the other enumeration algorithms to only INCS in the remaining experiments.

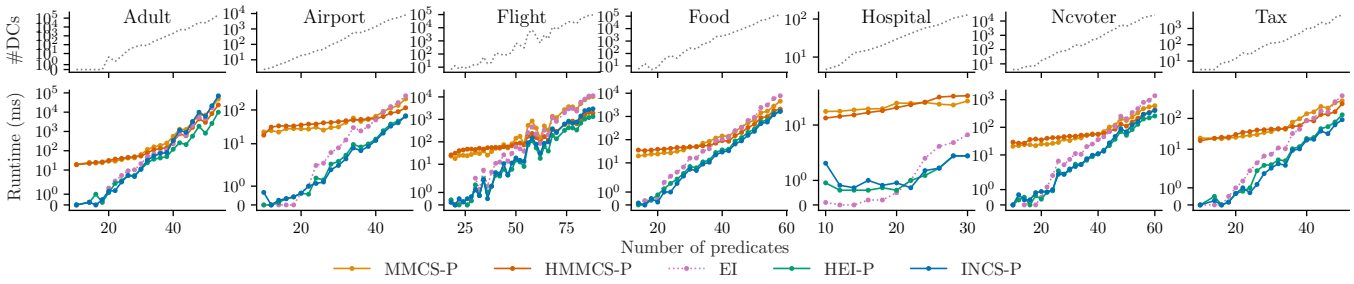

Figure 8: Predicate and column scaling of five DC enumeration algorithms—all but EI run in parallel threads.

The next results depict the performance impact of the hybrid approach on EI (Figure 6) and MMCS (Figure 7). The plots show the "hardest" datasets, that is, datasets where the algorithms yield the highest runtimes. As the number of predicates increases, the hybrid approach starts paying off by improving performance in many scenarios, even in its non-parallel version. For example, on *Adult* dataset with 54 predicates, our hybrid approach improves the EI algorithm by a factor of 2.12 (HEI) and 7.12 (HEI-P). In the same scenario, our HMMCS-P version was 2.03 times faster than the (also parallel) MMCS-P, even with HMMCS-P having the additional time of writing disk files. For the *Flight* dataset with 88 predicates, the improvement is 5.46 from MMCS-P to HMMCS-P, 2.40 from EI to HEI, and 8.4 for EI to HEI-P. However, the hybrid approach does not find much room for improvement for low predicate numbers since all algorithms execute in a few milliseconds. In such scenarios, the performance of HMMCS/HMMCS-P is even impaired due to the disk accesses that do not pay off.

Figure 8 compares the scalability of the enumeration algorithms that performed best in the previous experiments. We include EI in this comparison since it does not originally offer a parallel version. The plot shows the number of DCs (#DCs) as the mean of the total DCs discovered across the ten executions on random column subsets. The scaling of the algorithms seems to follow the number of predicates exponentially, which is expected considering the exponentially increasing number of DCs discovered. Also, DC enumeration takes longer on datasets with many DCs. Take *Adult* and *Tax* for example. These datasets have about the same number of predicates, but the number of DCs for *Tax* is about 34 times lower, and the runtimes (of all algorithms) were much faster (e.g., 75-fold using HEI-P).

In general, the fastest algorithm was HEI-P for larger numbers of predicates and resulting DCs, and INCS-P for other cases. Nonetheless, all algorithms could enumerate the results fast: in just over a minute in the worst cases. With better design choices, the proposed INCS-P was able to compete with the previous state-of-the-art enumeration algorithms. This fact highlights the importance of our design choices in Section 4 because, as shown in previous studies, the original MCS algorithm could not yield such a result.

Next, we evaluate the scalability of the (parallel) DC enumeration algorithms on an increasing number of threads. Figure 9 shows the parallel speedup from each algorithm relative to its single-threaded execution. We observe that parallel scaling in DC enumeration is poorer than the scaling on evidence set building. Nonetheless, some algorithms still present good parallel scalability, generally up to the maximum number of physical cores. In general, the better scaling appears from INCS-P and HMMCS-P. For example, INCS-P achieves a self-relative speedup of 10.06× on *Adult* with 48 threads, and HMMCS-P achieves a self-relative speedup of 5, 82× on *Flight* with 24 threads. These results illustrate the improvements by late DC minimization and the hybrid approach.

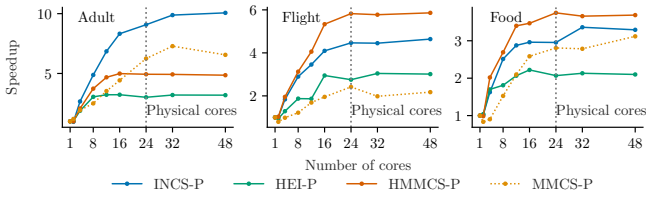

Figure 9: Parallel speedup of DC enumeration algorithms.

5.5 Additional analysis

Next, we investigate the impact of the heuristic optimizations of Section 3.6 on evidence set building. Figure 10 shows the relative speedup from (1) using each input dataset as is; (2) sorting the input on their numerical columns; (3) prioritizing categorical stages; (4) prioritizing high-cardinality columns; and combinations of heuristic 2 with heuristics 3 and 4.

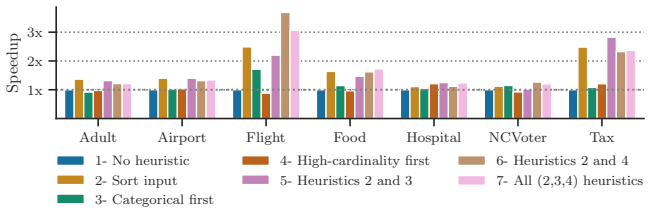

Figure 10: Speedup from heuristic optimizations (and their combinations) on evidence set building.

Sorting the input has a clear beneficial performance impact. On the other hand, heuristics 3 and 4 do not yield much improvement on their own. However, when combined with sorting, they yielded the highest speedups, particularly for datasets with many rows and many numerical predicates (e.g., *Flight*). These results align with previous studies that have shown how predicate evaluation order and input sorting can impact performance [12, 22, 23]. We kept all heuristics on in all remaining experiments.

Figure 11 shows the relative impact of INCS vs. AINCS on runtime and the number of DCs. We set AINCS with typical thresholds (in parentheses) from previous works [5, 21]. All numbers are relative to the exact DC discovery, i.e., INCS. We observe an increase (up to 1.8×) in runtime in some scenarios, e.g., *Adult* and *NCVoter*. That is because AINCS needs to compute the g_1 value for the DC candidates, so it needs to iterate intermediate evidence sets to collect evidence multiplicities. Also, pruning based on subsuming predicates is less effective for AINCS, as more evidence may be required to prune branches. We observe that the number of approximate DCs is significantly lower than that of exact DCs, up to orders of magnitude, e.g., for *NCVoter* with larger thresholds. That is because approximate DCs are often less specialized (fewer predicates). A single approximate DC can be the prefix of many (specialized) exact DCs. The larger the thresholds, the lower the number of branches, so the faster the DC enumeration. Notice that AINCS(0.0) and INCS produce the same results as they are equivalent. However, the former algorithm incurs the performance overhead described above.

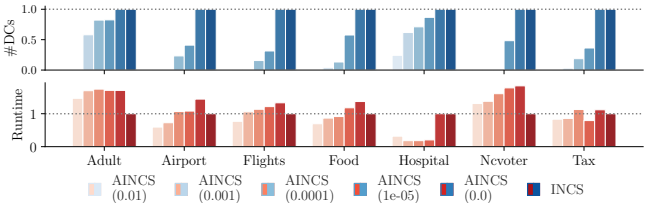

Figure 11: Relative impact of approximate DC discovery on runtime and number of DCs discovered.

Figure 12 shows the runtime breakdown of the ECP/HEI-P algorithm for all tested datasets. Evidence building dominates discovery runtime for all datasets, whereas DC enumeration makes a considerable part of the runtime for datasets with many DCs, such as *Adult*. In general, the time required for input preparation (table sorting and index building) is just a small fraction of the runtime. This result shows the importance of fast evidence set building.

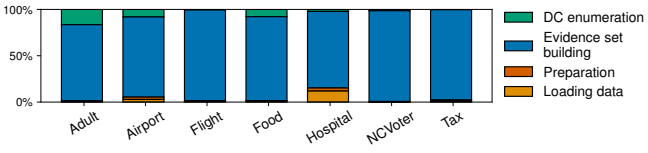

Figure 12: Runtime breakdown of ECP/HEI-P.

Figure 13 illustrates the memory requirement of our approach (mind the log-scale axis). It shows the minimum heap size required to run ECP/HEI-P (right-most bar for each dataset). We obtained this estimate by doubling the heap-size, starting with 32MB, until the algorithm could finish processing the entire dataset. We observe a low memory requirement, as the algorithm can process most of the datasets with a heap-size of 256MB or less. As a comparison, ECP/HEI-P uses 256MB to process the 1M rows of *Flight*, whereas Hydra and DCFinder uses 1024MB. The higher memory requirement for *Food* (1GB) and *NCVoter* (4GB) regard specific parts of the execution, as discussed next.

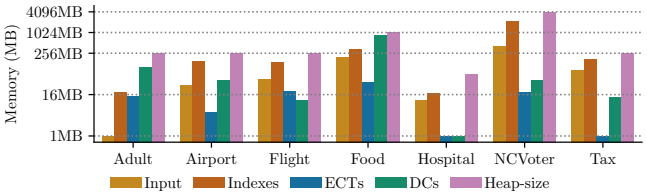

Figure 13: Memory requirement of ECP-HEI-P.

Figure 13 also shows the memory used by the key data structures in our algorithms. "Input" regards the in-memory dataset, whereas "Indexes" regards the column indexes for evidence context correction. In general, the index size overhead is larger for datasets having large column domains and wide column values, such as Food and NCVoter with many (lengthy) strings. The index sizes being greater than the in-memory dataset sizes is expected, because we build indexes on all columns. Index structures consume relatively large portions of memory when tuples are small, and tables have many indexes [27]. For "ECTs", we monitored the ECP execution to measure the largest set of evidence contexts (the ECTs in Algorithms 1 and 2). We observe relatively low footprints due to the pipeline model with compressed evidence contexts. Finally, the value of "DCs" regards the in-memory representation of the discovered DCs, and the data structures used to support the EI algorithm and the late minimality checks. We observed numbers within the same range as the other DC enumeration methods for this last parameter. For the Food dataset, memory consumption is dominated by the number of DCs in the result set. Upon closer inspection, we noticed that the binary tree used for subset lookup might increase its footprint when DCs are longer and sparser (not many shared predicates across DCs). We intend to investigate more space-efficient structures in future work.

6 CONCLUSION

This paper presented several algorithms to address the algorithmic challenges of discovering all minimal, non-trivial (approximate and exact) denial constraints of relational datasets. First, we proposed the evidence context pipeline with data representations, indexes, and algorithms designed to compute evidence sets efficiently. Second, we introduced inverted indexes, pruning, late minimization, and parallel execution for DC enumeration. We showed how to use these techniques to improve the previous DC enumeration algorithms. Our experimental study shows that our algorithms can consistently outperform the previous state-of-the-art.

In future work, we intend to investigate how to extend or adapt the solutions proposed in this paper to discover other forms of partial DCs, such as constant DCs (with constant predicates) or approximate DCs using other approximation functions.

ACKNOWLEDGMENTS

This study was funded in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

REFERENCES

[1] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling Relational Data: A Survey. *VLDB Journal* 24, 4 (2015), 557–581.

[2] Thomas Bläsius, Tobias Friedrich, and Martin Schirneck. 2022. The Complexity of Dependency Detection and Discovery in Relational Databases. *Theor. Comput. Sci.* 900, C (2022), 79–96. https://doi.org/10.1016/j.tcs.2021.11.020

[3] Tobias Bleifuß, Susanne Bülow, Johannes Frohnhofen, Julian Risch, Georg Wiese, Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. 2016. Approximate Discovery of Functional Dependencies for Large Datasets. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*. Association for Computing Machinery, 1803–1812. https://doi.org/10.1145/2983323.2983781

[4] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. 2017. Efficient Denial Constraint Discovery with Hydra. *PVLDB* 11, 3 (2017), 311–323.

[5] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Discovering Denial Constraints. *PVLDB* 6, 13 (2013), 1498–1509.

[6] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *Proceedings of the International Conference on Data Engineering (ICDE)*. 458–469.

[7] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms*, Vol. DMTCS Proceedings vol. AH, Conference on Analysis of Algorithms (AofA). Discrete Mathematics and Theoretical Computer Science, 137–156.

[8] Andrew Gainer-Dewar and Paola Vera-Licona. 2017. The Minimal Hitting Set Generation Problem: Algorithms and Computation. *SIAM Journal on Discrete Mathematics* 31, 1 (2017), 63–100. https://doi.org/10.1137/15M1055024

[9] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2020. Cleaning data with Llunatic. *VLDB Journal* 29, 4 (2020), 867–892. https://doi.org/10.1007/s00778-019-00586-5

[10] Stella Giannakopoulou, Manos Karpathiotakis, and Anastasia Ailamaki. 2020. Cleaning Denial Constraint Violations through Relaxation. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. ACM, 805–815. https://doi.org/10.1145/3318464.3389775

[11] Ousmane Issa, Angela Bonifati, and Farouk Toumani. 2020. Evaluating Top-k Queries with Inconsistency Degrees. *PVLDB* 13, 12 (2020), 2146–2158. https://doi.org/10.14778/3407790.3407815

[12] Andreas Kipf, Damian Chromejko, Alexander Hall, Peter Boncz, and David G. Andersen. 2020. Cuckoo Index: A Lightweight Secondary Index Structure. *PVLDB* 13, 13 (2020), 3559–3572. https://doi.org/10.14778/3424573.3424577

[13] Jyrki Kivinen and Heikki Mannila. 1995. Approximate Inference of Functional Dependencies from Relations. *Theoretical Computer Science* 149, 1 (1995), 129–149.

[14] Daniel Lemire, Owen Kaser, and Kamel Aouiche. 2010. Sorting Improves Word-Aligned Bitmap Indexes. *Data and Knowledge Engineering (DKE)* 69, 1 (2010), 3–28. https://doi.org/10.1016/j.datak.2009.08.006

[15] Daniel Lemire, Gregory Ssi-Yan-Kai, and Owen Kaser. 2016. Consistently Faster and Smaller Compressed Bitmaps with Roaring. *Softw. Pract. Exper.* 46, 11 (2016), 1547–1569.

[16] Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. 2012. Discover Dependencies from Data - A Review. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 24, 2 (2012), 251–264.

[17] Ester Livshits, Alireza Heidari, Ihab F. Ilyas, and Benny Kimelfeld. 2020. Approximate Denial Constraints. *PVLDB* 13, 10 (2020), 1682–1695. https://doi.org/10.14778/3401960.3401966

[18] Ester Livshits, Rina Kochirgan, Segev Tsur, Ihab F. Ilyas, Benny Kimelfeld, and Sudeepa Roy. 2021. Properties of Inconsistency Measures for Databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1182–1194. https://doi.org/10.1145/3448016.3457310

[19] Keisuke Murakami and Takeaki Uno. 2014. Efficient algorithms for dualizing large-scale hypergraphs. *Discrete Applied Mathematics* 170 (2014), 83–94. https://doi.org/10.1016/j.dam.2014.01.012

[20] Eduardo H. M. Pena and Eduardo Cunha de Almeida. 2018. BFASTDC: A Bitwise Algorithm for Mining Denial Constraints. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)*. 53–68.

[21] Eduardo H. M. Pena, Eduardo C. de Almeida, and Felix Naumann. 2019. Discovery of Approximate (and Exact) Denial Constraints. *PVLDB* 13, 3 (2019), 266–278.

[22] Eduardo H. M. Pena, Eduardo C. de Almeida, and Felix Naumann. 2021. Fast Detection of Denial Constraint Violations. *PVLDB* 15, 4 (2021), 859–871. https://doi.org/10.14778/3503585.3503595

[23] Elaheh Pourabbas, Arie Shoshani, and Kesheng Wu. 2012. Minimizing Index Size by Reordering Rows and Columns. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*. 467–484.

[24] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *PVLDB* 10, 11 (2017), 1190–1201.

[25] Ziheng Wei and Sebastian Link. 2019. Embedded Functional Dependencies and Data-completeness Tailored Database Design. *PVLDB* 12, 11 (2019), 1458–1470. https://doi.org/10.14778/3342263.3342626

[26] Catharine Wyss, Chris Giannella, and Edward L. Robertson. 2001. FastFDs: A Heuristic-Driven, Depth-First Algorithm for Mining Functional Dependencies from Relation Instances - Extended Abstract. In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*. 101–110.

[27] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 1567–1581. https://doi.org/10.1145/2882903.2915222