Efficient Denial Constraint Discovery with Hydra

Tobias Bleifuß
Hasso-Plattner-Institut
Prof.-Dr.-Helmert-Str. 2–3
14482 Potsdam, Germany
tobias.bleifuss@hpi.de

Sebastian Kruse
Hasso-Plattner-Institut
Prof.-Dr.-Helmert-Str. 2–3
14482 Potsdam, Germany
sebastian.kruse@hpi.de

Felix Naumann
Hasso-Plattner-Institut
Prof.-Dr.-Helmert-Str. 2–3
14482 Potsdam, Germany
felix.naumann@hpi.de

ABSTRACT

Denial constraints (DCs) are a generalization of many other integrity constraints (ICs) widely used in databases, such as key constraints, functional dependencies, or order dependencies. Therefore, they can serve as a unified reasoning framework for all of these ICs and express business rules that cannot be expressed by the more restrictive IC types. The process of formulating DCs by hand is difficult, because it requires not only domain expertise but also database knowledge, and due to DCs' inherent complexity, this process is tedious and error-prone. Hence, an automatic DC discovery is highly desirable: we search for all valid denial constraints in a given database instance. However, due to the large search space, the problem of DC discovery is computationally expensive.

We propose a new algorithm Hydra, which overcomes the quadratic runtime complexity in the number of tuples of state-of-the-art DC discovery methods. The new algorithm's experimentally determined runtime grows only linearly in the number of tuples. This results in a speedup by orders of magnitude, especially for datasets with a large number of tuples. Hydra can deliver results in a matter of seconds that to date took hours to compute.

PVLDB Reference Format:
Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. Efficient Denial Constraint Discovery with Hydra. *PVLDB*, 11(3): 311-323, 2017.
DOI: 10.14778/3157794.3157800

1. KILLING N BIRDS WITH ONE STONE

The research area of data profiling [1] has borne several efficient algorithms to discover structural properties of datasets. In particular, the discovery of functional dependencies (FDs) [13] and unique column combinations (UCCs) [8] have attracted great attention. Lately, also order dependencies (ODs) [15] have been in focus. Each of these integrity constraints (ICs) serves different applications, such as schema normalization (FDs), key discovery (UCCs), and query optimization (ODs).

However, this state of the art suffers from two major problems. First, each type of IC is of strongly limited expressiveness. The types of rules one can observe in actual datasets are much greater, though, as we demonstrate later in this section. This imbalance is accounted for by proposing new IC types every now and then. Second, the different ICs are logically isolated. That is, to reason on the interaction among ICs, e.g., for data cleaning, one has to explicitly provide interaction rules. Not only is this unwieldy — it also does not scale well with the number of considered IC types, but DCs can bridge that gap [4, 7].

Then, why are there so many different ICs rather than a single, general IC capable of avoiding such problems? A major reason is plain efficiency. For that matter, discovering FDs, UCCs, or ODs alone is already computationally expensive and the respective discovery algorithms employ highly specific data structures and pruning rules to operate efficiently. That being said, being able to efficiently discover more general ICs could solve aforementioned problems and would thus be very valuable.

To this end, we propose Hydra, an efficient algorithm to discover all valid *denial constraints (DCs)* in a given relational dataset. But let us first define DCs and show that their expressiveness goes beyond UCCs, FDs, and ODs. In few words, a DC defines a set of predicates on n tuples. A relational instance *satisfies* that DC if for any n distinct tuples of that instance at least one predicate is violated. In the following, we set $n = 2$ and refer to the two involved tuples as t and t'. As we show by examples below, DCs over two tuples are sufficient to express all of the aforementioned integrity constraints. For reasons of rapidly increasing computational complexity for $n > 2$, as well as questionable gains, related work applies the same restriction [3].

Definition 1 (Denial constraint). *Let r be a relational instance with schema $R(A_1, \dots, A_n)$. A denial constraint ψ is a statement of the form*

$$\forall t, t' : \neg \left(t^1[A^1]\, \phi_1\, u^1[B^1] \wedge \dots \wedge t^k[A^k]\, \phi_k\, u^k[B^k] \right)$$

with $t^i, u^i \in \{t, t'\}$, $A^i, B^i \in \{A_1, \dots, A_n\}$ and $\phi_i \in \{=, \neq, <, \leq, >, \geq\}$. The different clauses in the conjunction are referred to as predicates. *The instance r satisfies ψ if and only if ψ is satisfied for any two distinct tuples t and $t' \in r$. A DC is* trivial *if it is valid for any possible instance. Furthermore, a DC ψ is* (set-)minimal, *if no DC whose predicates are a subset of ψ's predicates is valid on r.*

Let us demonstrate the expressiveness of DCs with the example dataset from Table 1 in combination with some valid

$$\begin{aligned}
\psi_1 : \forall t, t' : &\quad \neg\big(t[\mathsf{Name}] = t'[\mathsf{Name}] \wedge t[\mathsf{Hired}] = t'[\mathsf{Hired}]\big)\\
\psi_2 : \forall t, t' : &\quad \neg\big(t[\mathsf{Department}] = t'[\mathsf{Department}] \wedge t[\mathsf{D\text{-}Code}] \neq t'[\mathsf{D\text{-}Code}]\big)\\
\psi_3 : \forall t, t' : &\quad \neg\big(t[\mathsf{ID}] \leq t'[\mathsf{ID}] \wedge t[\mathsf{Hired}] > t'[\mathsf{Hired}]\big)\\
\psi_4 : \forall t, t' : &\quad \neg\big(t[\mathsf{D\text{-}Code}] = t'[\mathsf{D\text{-}Code}] \wedge t[\mathsf{Hired}] < t'[\mathsf{Hired}] \wedge t[\mathsf{Salary}] > t'[\mathsf{Salary}]\big)
\end{aligned}$$

Figure 1: A number of example DCs that hold in our example dataset (Table 1).

Table 1: Example dataset with staff data.

	ID	Name	Department	D-Code	Hired	Salary
t_1	103	J. Miller	Sales	SAL	2008	3,200
t_2	197	J. Miller	Accounting	ACT	2012	3,500
t_3	338	P. Smith	Sales	SAL	2012	2,900
t_4	631	S. Blake	Accounting	ACT	2016	3,200
t_5	664	P. Dean	Sales	SAL	2016	2,700

DCs shown in Figure 1. Amongst others, Name and Hired form a key candidate, i.e., a UCC. This fact is captured by the DC ψ_1. Further, we observe that Department determines D-Code, that is, the FD Department → D-Code holds. This FD is equivalent to the DC ψ_2. As a third IC, we note that the staff IDs reflect when an employee has been hired. Thus, we have the OD ID $\rightarrow_{\leq}$ Hired, which can be expressed as ψ_3. In addition, a closer look reveals a further, more intricate constraint: For any two employees from the same department, the newer employee should not earn more than the older. While neither UCCs, nor FDs, nor ODs can express this constraint, the DC ψ_4 naturally captures it. This shows the superiority of DCs over the specific IC types.

Problem statement. Nevertheless, automatically discovering all DCs in a dataset is a highly challenging task: given a relational instance r and a predicate space P, we want to find all non-trivial, set-minimal DCs valid on r that can be expressed using only predicates in P. At first, the number of candidate DCs grows extremely fast with the number of attributes — in fact, super-exponential! Furthermore, comparing all tuple pairs to verify DCs can be prohibitive: Although, this is only of quadratic complexity w.r.t. the number of tuples, there are usually orders of magnitudes more tuples than attributes in a dataset. Our algorithm HYDRA mitigates these scalability traps. In fact, we make the following major contributions:

1. We propose to first approximate the DCs for a dataset and then correct any incorrect results (Section 3). This avoids the comparison of all tuple pairs. In particular, we prove this idea to be sound and complete (Section 6.1).
2. We devise a technique to quickly approximate the DCs in a dataset while looking only at a small fraction of all tuple pairs. This comprises the careful choice of tuples to compare (Section 4) and the efficient conversion of comparison results to approximated DCs (Section 5).
3. We explain a proceeding to efficiently determine all violations of a dataset w.r.t. the approximated DCs. Combined with the above mentioned conversion algorithm, we quickly turn the set of approximate DCs to the set of actual DCs (Section 6).
4. Finally, we exhaustively evaluate the efficiency of HYDRA (Section 7).

In addition, we discuss related work in Section 2 and summarize as well as conclude in Section 8.

2. RELATED WORK

Although the research area of data profiling has produced various dependency discovery algorithms [1], the discovery of denial constraints (DCs) has not received much attention so far. Therefore, this section looks at related dependency discovery algorithms in a broader sense.

DC discovery. The first and only other DC discovery algorithm is FASTDC [3], which is also our evaluation baseline. It discovers DCs in two major phases: At first, all tuples in a given dataset are compared pairwise to form the *evidence set*. Then, by searching a minimal cover for this evidence set, FASTDC constructs DCs that are not violated by any found evidence. This strategy is much slower than that of HYDRA for two reasons. First, the comparison of all tuple pairs is very expensive. Second, HYDRA proposes a novel, more efficient algorithm to derive DCs from the evidence set. Besides their basic approach, the authors of FASTDC provide two modifications of their algorithm. A-FASTDC is able to detect *approximate DCs*, which may be partially violated, and C-FASTDC allows for constants in the predicates of the DCs. We note that the modifications for C-FASTDC are compatible with HYDRA. Furthermore, HYDRA could be in principle adapted to discover approximate DCs. We outline the necessary changes in Section 8.

UCC/FD/OD discovery. As explained in Section 1, DCs subsume various other integrity constraints, in particular UCCs, FDs, and ODs. Hence, HYDRA is a valid substitute for any of their discovery algorithms. Nonetheless, we can identify commonalities of HYDRA and those specifically tailored algorithms. Generally speaking, HYDRA belongs to the class of algorithms that are based on pairwise tuple comparisons, which comprises, e.g., GORDIAN for UCC discovery [14], and FDEP for FD discovery [6], but no known OD discovery algorithm. In particular, the idea to approximate ICs by repeatedly sampling tuple pairs has been successfully applied for FDs [2, 13]. However, sampling for DCs is more complex, because DCs involve inequality predicates. Also, there is no equivalent for the completion strategy for FDs via *stripped partitions* [9]. Hence, a completely new completion strategy is required. Another recent incentive for efficiency improvements are *holistic profiling algorithms* that discover multiple IC types in a single run, thereby saving overhead [5]. This property applies to HYDRA, too.

3. OVERVIEW OF HYDRA

Many IC discovery algorithms compare all tuple pairs in a dataset to collect every possible violation of their respective IC type. From those violations, the actual ICs can then be inferred [3, 6, 14]. However, those algorithms scale poorly with the number of tuples — all tuple pairs incur quadratic complexity. HYDRA is specifically designed to *not* compare all tuple pairs and, hence, to avoid this quadratic trap when discovering all DCs.

Essentially, this leap is possible, because any two tuple pairs that violate the exact same candidate DCs are redundant and it suffices to consider only one of them. A tuple pair is also redundant if all predicates that are fulfilled by that pair are fulfilled by another pair (and possibly other predicates), so only the maximum predicate sets are relevant. Thus, we can improve performance considerably by identifying a subset of tuple pairs from a dataset, such that all other tuple pairs are redundant to this subset, and compute the DCs from this subset only. While in theory it is possible that no pair is redundant (e.g., in a diagonal matrix), our experiments show that in practice the vast majority of tuple pairs is in fact redundant. In consequence, HYDRA adapts to the characteristics of the input data and rapidly converges to the relevant area of the search space. In contrast, the non-adaptive FASTDC always incurs a quadratic runtime regardless of the input data.

Our general idea is to determine a preliminary set of DCs on a sample of tuple pairs. Then, we need to spot all tuple pairs violating the preliminary DCs. Eventually, the combination of sampled and violating tuple pairs — or rather their implied violations — exactly determine *all* valid DCs. To put this approach into practice, HYDRA carefully employs efficient algorithms and data structures in each step. In the following, we describe HYDRA's workflow and components in more detail.

Consider the overview in Figure 2 depicting HYDRA's basic steps. HYDRA takes as input a relational table and a *predicate space*, i.e., the set of predicates that can appear in DCs of that table (see Definition 1). At first, HYDRA samples tuple pairs from the relational instance for the preliminary DCs (Step (1)). For each tuple pair, an *evidence* is computed, which captures all predicates in the predicate space that are violated by the tuple pair. Because duplicate evidences (which arise from redundant tuple pairs) do not provide new information regarding the DCs, HYDRA aims to maximize the number of non-redundant tuple pairs in the sample. As a result of its focused sampling stage, the algorithm obtains a relatively complete *evidence set* from only a fraction of all tuple pairs in the dataset. We describe the sampling in detail in Section 4.

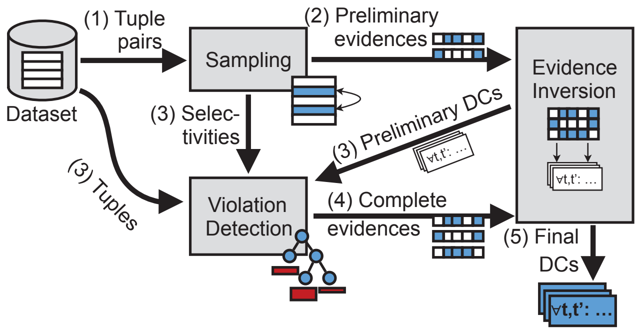

Figure 2: Overview of Hydra.

These *preliminary evidences* are then used to compute the *preliminary DCs* using a novel, efficient *evidence inversion* algorithm (Step (2)), as detailed in Section 5. The preliminary DCs might of course be violated by tuple pairs that were not included in the sample. Therefore, HYDRA employs a highly efficient scheme to determine all tuple pairs that violate the preliminary DCs (Step (3)). In particular, HYDRA tries to avoid duplicate work that arises from predicates recurring in multiple DCs. Moreover, specific checking operators detect the violating tuple pairs *without* comparing *all* tuple pairs. Section 6 details this checking scheme.

Finally, the preliminary evidences are unioned with those from the violating tuple pairs (Step (4)). From this completed evidence set, the evidence inversion can now compute the final set of DCs (Step (5)) in exactly the same way as for the preliminary evidences in Step (2) before.

4. SAMPLING

A key challenge of DC discovery is the quadratic complexity w.r.t. the number of tuples in the profiled dataset: To verify a DC, we must ensure that it is satisfied by *all tuple pairs*. In related work, this is achieved by evaluating all predicates of the predicate space on all tuple pairs [3]. HYDRA, in contrast, saves a lot of these predicate evaluations by initially operating on only a sample of tuple pairs, as we detail in this section. Because the resulting DCs might be violated by an unseen tuple pair, though, HYDRA subsequently employs efficient strategies to discover all those violating tuple pairs.

4.1 Evidence Set

Before explaining *how* we sample, let us describe more thoroughly the output of the sampling component, namely an *evidence set* [3]. Given a predicate space P, which is merely a set of predicates as defined in Definition 1, an evidence $e(t,\ t') \subseteq P$ is the subset of predicates that are satisfied by the tuple pair $(t,\ t')$. For a given pair of tuples, $e(t,\ t')$ is easy to calculate by evaluating each predicate in P. An evidence set is, as the name gives away, a set of such evidences. In particular, we refer to the *full evidence set* E_{full} as the set of evidences for *all* tuple pairs in a dataset. There are two things to note about this evidence set. First, the full evidence set most likely contains fewer evidences than the dataset has tuple pairs, because duplicate evidences are eliminated. Second, the full evidence set immediately determines the valid DCs in a dataset, as we show in the following lemma.

LEMMA 1. *Assume a dataset r with predicate space P and its full evidence set E_{full}. Then the DC $\psi := \forall t, t' \colon \neg\big(p_1(t,t') \wedge \cdots \wedge p_n(t,t')\big)$ with predicates $p_i \in P$ is valid w.r.t. r, if and only if $\nexists e \in E_{full} \colon \{p_1, \ldots, p_n\} \subseteq e$.*

PROOF. The DC ψ is valid exactly when no tuple pair in r satisfies all the p_i occurring in ψ, which is equivalent to the absence of an evidence containing all the p_i. □

Based on the insights presented above, we can now precisely state the goal of the sampling component: Given a dataset r and its predicate space P, we want to determine a *preliminary evidence set* $E_{\text{pre}} \subseteq E_{\text{full}}$ that approximates the full evidence set. We observed that this can be achieved by sampling only very few tuple pairs from r (compared to all tuple pairs) and computing their evidences. Since string comparisons are expensive, we employ a dictionary encoding for strings during load to speed up the evidence generation: we need only simple integer comparisons instead of operations on strings. In the remainder of this section, we present HYDRA's complementary sampling stages, namely *random sampling* and the subsequent *focused sampling* phase.

4.2 Random Sampling

HYDRA starts by populating the preliminary evidence set with evidences from randomly sampled tuple pairs. However, the main purpose of this sampling stage is to estimate the *selectivity* of predicates, i.e., the number of tuple pairs that satisfy the predicate. That is, given a random pair of tuples, how likely is it to satisfy a certain predicate from the predicate space? HYDRA's violation detection can employ the selectivities to optimize its efficiency (see Section 6). While a focused sample is biased and, thus, not well-suited to determine selectivities, we can directly extrapolate the selectivities from a random sample.

To obtain the random sample, HYDRA chooses for every tuple in the dataset k other random tuples and calculates the k associated evidences. Our experiments show that a value of $k = 20$ is sufficient for a good selectivity estimation (Section 7.5.1). Consequently, the sampling effort is linear in the number of tuples — we retrieve $k \cdot |r|$ instead of $|r| \cdot (|r| - 1)$ pairs. Recall that usually an evidence set does not contain duplicate evidences. In case that two or more randomly sampled tuples yield the same evidence, HYDRA also maintains a counter for that evidence to properly employ them for selectivity estimation.

4.3 Focused Sampling

The random sample might not yet be sufficient to obtain a good approximation of the full evidence set. Hence, we complement it with a focused sampling that tries to sample tuple pairs that yield unseen evidences. Similar focused sampling strategies have been devised in the context of FD discovery [2, 13]. However, those are not sufficient here, because — speaking in DC vocabulary — their evidences consider only predicates of the form $t[A] = t'[A]$ for some attribute A. In contrast, DC discovery has to consider other types of predicates, such as inequalities and comparisons across attributes.

Therefore, we devise a new focused sampling algorithm inspired by FD discovery. Its core idea is to maintain a set of *sampling foci*, each of which is associated with a predicate from the predicate space. Each sampling focus further specifies a *sampling strategy* that samples only tuple pairs satisfying that predicate. Now, we repeatedly sample according to whatever sampling focus we currently believe to have the highest chance of yielding an unseen evidence. At some point, that belief falls below a certain threshold for all sampling foci. At that point, we terminate the sampling phase. In few words, the key points of this algorithm are to (i) ensure that all predicates are considered by the sample and (ii) making the sampling sensitive to the yield of the different sampling foci. We now explain how to technically realize this sampling.

Sampling strategies. HYDRA defines four sampling strategies: WITHIN enforces predicates of the form $t[A] = t'[A]$; OTHER enforces predicates of the form $t[A] \neq t'[A]$; and BACK and FORTH enforce predicates of the form $t[A] > t'[A]$ and $t[A] < t'[A]$, respectively. Note that WITHIN, BACK, and FORTH are implicit strategies for the predicates with $\leq$ and $\geq$.

To efficiently sample according to these four strategies, HYDRA initially clusters tuples individually according to each attribute. That is, for each attribute we obtain a set of clusters, each of which is a set of references to tuples that share the same value in the respective attribute. If the operators $<$ and $>$ are defined on the considered attribute, the clusters are further ordered according to the values they represent. For instance, the attribute Salary from Table 1 defines the ordered clustering $(\{t_5\}, \{t_3\}, \{t_1, t_4\}, \{t_2\})$.

Given an attribute A with its clustering, we can now easily apply any sampling strategy: For each tuple index t_i, we pick a random partner tuple index t_j (if any) from the same (WITHIN) or different clusters (OTHER, BACK, FORTH), retrieve the actual tuples for each (t_i, t_j), and calculate the evidences. For instance, if we want to apply the BACK strategy to the attribute Salary and currently have $t_i = t_4$, then the clustering tells us that we have to sample t_j from the preceding clusters $\{t_5\}$ or $\{t_3\}$. Assume, we sample t_3, we then need to calculate the evidence $e(t_4, t_3)$. The reader might have noticed that we do not define sampling strategies for predicates that compare two distinct attributes A and B. While technically this would need only a mapping of the clusters of A to their position in the ordered cluster of B, we omit it, simply because the sampling works sufficiently well with the four presented strategies.

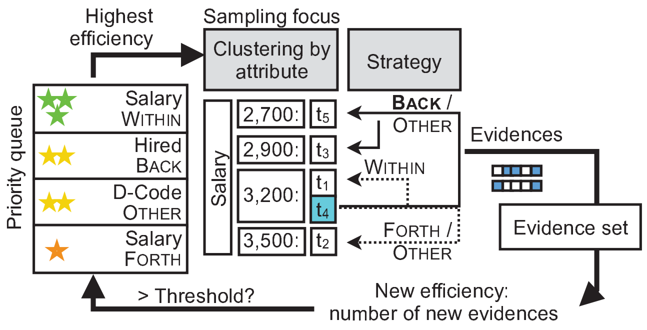

Figure 3: Overview of the focused sampling.

Sampling efficiency. Having explained sampling strategies, we can precisely describe a sampling focus as a pair of an attribute and a sampling strategy. As described above, HYDRA's focused sampling phase repeatedly applies the most promising sampling focus. We define this *sampling efficiency* of a sampling focus as the number of previously unseen evidences it has contributed (the growth in size of the distinct evidence set), divided by the total number of evidence calculations (as a measure of effort) during its last application. To obtain an initial efficiency, we apply all sampling foci once in the beginning, and then insert them into a priority queue that orders the sampling foci descending by their sampling efficiency.

Now, we just poll a sampling focus from that priority queue (for example Salary WITHIN in Figure 3), apply it (calculate for example $e(t_4, t_1)$), and obtain as a byproduct a new efficiency value. If that value is below a certain threshold (0.005), we discard the sampling focus — otherwise we put it back into the priority queue. Note that this threshold can influence only HYDRA's runtime, but never its result quality. Additionally, the algorithm is quite robust for the choice of this parameter, as shown in Section 7.5.1.

5. EVIDENCE INVERSION

An evidence set determines a set of valid DCs, as shown in Lemma 1. However, actually computing those DCs is a challenging problem, because even the output size can be exponential in the number of attributes n. This is the case, because the set of DCs generalizes the set of minimal keys, and this set can be of size $\binom{n}{n/2}$ [8]. What is more, Hydra needs to perform such *evidence inversion* at two points (cf. Figure 2): First, it needs to invert the preliminary evidence set and, eventually, the full evidence set. It is therefore crucial to perform the inversion with a high efficiency.

Intuitively, Hydra achieves that goal as follows: Initially, we assume the most basic DCs, which contain only a single predicate, to be true. Then, we iterate over the evidence set. For each evidence, we update all the DCs by adding predicates, such that the evidence no longer violates the DC. Once all evidences have been processed, the set of DCs is valid and complete.

Algorithm 1 shows this proceeding in more detail. At first, it initializes the basic candidate DCs (Line 1) and represents them as the sets of their predicates. For instance for Table 1, we have the candidate DC $\forall t, t'\colon \neg(t[\mathsf{Name}] = t'[\mathsf{Name}])$ and represent it as $\{p\}$ (where $p(t, t') :\equiv t[\mathsf{Name}] = t'[\mathsf{Name}]$).

The function `handleEvidence` now iteratively updates the tentative candidate DCs, so that they conform to the evidence set (Line 3). The function begins by detecting all candidate DCs $\psi^- \in \Psi^-$ that are violated by the current evidence e (Line 6). This is the case when e satisfies all predicates of ψ^-. Our above example candidate DC would be violated by, e.g., the evidence $e(t_1, t_2)$, because t_1 and t_2 agree in the attribute Name. However, violated candidate DCs in Ψ^- can be reconciled with e by simply adding a predicate that is not satisfied by e (Lines 9–11). For instance, our example candidate DC can be extended to $\forall t, t'\colon \neg(t[\mathsf{Name}] = t'[\mathsf{Name}] \wedge t[\mathsf{Department}] = t'[\mathsf{Department}])$. Note that there are multiple ways to reconcile the candidate DCs so that Ψ might eventually grow. However, the reconciliation might yield redundant DC candidates, which is the case when there is a valid, more general DC. For that matter, our reconciled candidate DC might be subsumed by $\forall t, t'\colon \neg(t[\mathsf{Department}] = t'[\mathsf{Department}])$ (assuming that we have not yet processed an evidence violating it). Thus, our algorithm actively checks and removes such redundant DC candidates (Line 10). Let us show that this algorithm is sound and complete.

Theorem 1. *Given a predicate space P and evidence set E, Algorithm 1 determines a sound, complete, and minimal set of DCs w.r.t. E.*

Proof. *(Sound)* Suppose a DC ψ in the output is not correct, i.e., an evidence $\hat{e} \in E$ fulfills all of its predicates ($\psi \subseteq \hat{e}$). `handleEvidence` exchanges only DCs in Ψ with superset DCs, i.e., valid DCs can only grow due to an evidence, never shrink. Thus, for ψ to be part of the result, a subset of ψ must be present in Ψ at all times. However, each call for `handleEvidence` ensures that no DC invalidated by the current evidence e remains in Ψ by removing all subsets of e from Ψ and only adding proper supersets of e. As $\psi \subseteq \hat{e}$, the call for `handleEvidence` for $e = \hat{e}$ removes all subsets of ψ and therefore ψ cannot be part of the output.

(Complete) Throughout each iteration of the loop in Line 2 the following property holds: If a DC ψ is a valid, non-trivial DC, Ψ always contains a DC whose predicates are a subset of ψ. This property obviously holds at the initialization of Ψ in Line 1. If, at some point, the last subset of ψ is removed in Line 7, then a new subset of ψ would be added in Line 11, as at least one superset must not be contradictory to e, otherwise ψ would not be valid. This property immediately leads to completeness in the sense that all valid DCs can be derived from the output.

(Minimal) Suppose the DC ψ_1 is valid and minimal, but the result contains ψ_2, i.e., the predicate set of ψ_1 is a subset of ψ_2's predicate set and therefore ψ_2 is not minimal. The check in Line 10 ensures that Ψ contains no subsets of new DCs before adding them to the result. The completeness loop invariant guarantees that Ψ always contains a subset of ψ_1, so whenever ψ_2 should be added to the result, this subset check prevents it. □

Algorithm 1: Evidence inversion

```
Data: evidence set $E$, predicate space $P$
Result: the set of minimal, non-trivial DCs $\Psi$
1  $\Psi \leftarrow \{\{p\} \mid p \in P\}$
2  for $e \in E$ do
3      handleEvidence $(e, \Psi, P)$
4  return $\Psi$
5  Function handleEvidence($e, \Psi, P$)
6      $\Psi^- \leftarrow \{\psi^- \in \Psi \mid \psi^- \subseteq e\}$
7      $\Psi \leftarrow \Psi \setminus \Psi^-$
8      for $\psi^- \in \Psi^-$ do
9          for $p \in (P \setminus e)$ do
10             if $\nexists \psi \in \Psi\colon \psi \subseteq (\psi^- \cup \{p\})$ then
11                 $\Psi \leftarrow \Psi \cup \{\psi^- \cup \{p\}\}$
```

6. EVIDENCE SET COMPLETION

The output of the previous step – the preliminary DCs – can contain DCs that are not valid on the entire relational instance, because the evidence that violates them was not included in the preliminary evidence set. Additionally, as the inversion step outputs a minimal set of DCs, some DCs that are actually correct can be missing in the output. This can happen, because a more general, but in fact invalid DC is present in the output and thus the missing DCs are not considered minimal. In this section, we describe how to arrive at all minimal DCs given the preliminary evidence set and DCs. At first, we characterize which evidences the preliminary evidence set lacks to entail the complete and correct set of DCs (Section 6.1). Then, we show how to efficiently collect exactly those missing evidences (Sections 6.2–6.4). Applying once more the evidence inversion from Section 5 to the preliminary *and* newly collected evidences finally gives us the correct and complete set of DCs for the entire dataset.

6.1 The Delta Evidence Set

A *delta evidence set* E_Δ is a set of evidences that entails the correct and complete DCs of a dataset when combined with the preliminary evidence set. That is, the delta evidence set is precisely the set of evidences that are missing from the preliminary set to make the result correct. Recall that the preliminary evidence set itself entails a set of preliminary DCs. Interestingly, the evidences for all tuple pairs that violate any of the preliminary DCs form such a delta evidence set.

Theorem 2. *Assume a preliminary evidence set E_{pre} and its associated DCs Ψ_{pre}. Further assume an evidence set E_Δ*

that contains the evidences for all tuple pairs that violate Ψ_{pre}. Then an evidence inversion of $E_{pre} \cup E_{\Delta}$ leads to the complete and minimal set of denial constraints.

PROOF. The evidence inversion's result is unambiguously determined by the maximal evidences $e_{\max}$, which are not a subset of a further evidence in the same evidence set. This is because any DC that is precluded by some $e' \subseteq e_{\max}$ is also precluded by $e_{\max}$. Now, let E_{full} denote the full evidence set and assume that some maximal evidence $e_{\max} = \{p_1, \ldots, p_n\}$ is contained in E_{full} but not in E_{pre}. In other words, E_{full} precludes the DC $\psi = \forall t, t'\colon \neg(p_1 \wedge \cdots \wedge p_n)$, while E_{pre} does not. Hence, there must be some DC $\psi_{\text{pre}} \in \Psi_{\text{pre}}$ that comprises only predicates from $e_{\max}$. Because the tuple pair yielding $e_{\max}$ violates ψ_{pre}, $e_{\max}$ is in E_{Δ}. So, we can conclude that all maximal evidences from E_{full} are either included in E_{pre} or E_{Δ}. Therefore, E_{full} and $E_{\text{pre}} \cup E_{\Delta}$ entail the same DC set. $\square$

EXAMPLE 1. *Let us assume that our sampling did not compare t_1 and t_2. Therefore, we did not find any evidences that contradict the DC $\forall t, t'\colon \neg t[Name] = t'[Name]$ and the DC is thus contained in our preliminary DC set Ψ_{pre}. When we check this DC, we discover that the tuple pairs (t_1, t_2) and (t_2, t_1) violate that DC and we can add two new elements $e(t_1, t_2)$ and $e(t_2, t_1)$ to our evidence set. As shown in the proof above, the evidence set is complete if we repeat this process for every DC in Ψ_{pre}.*

To employ Theorem 2 for DC discovery, we need to discover E_{Δ} and, hence, all tuple pairs that violate any of the preliminary DCs in Ψ_{pre}. A naïve approach to discover all violating tuple pairs for some DC $\psi_{\text{pre}} \in \Psi_{\text{pre}}$ (i) inspects each tuple pair; (ii) determines, whether the tuple pair violates ψ_{pre}; and, if so, (iii) saves this tuple pair for the calculation of E_{Δ}. However, this approach has a runtime complexity of $\mathcal{O}(|\Psi_{\text{pre}}| \cdot |r|^2)$, where $|r|$ is the number of tuples in the dataset, and is clearly too inefficient.

HYDRA tackles this challenging task to efficiently discover E_{Δ} on two levels. At first, it provides an efficient scheme to discover the violating tuple pairs for a single DC, by employing specific data structures (Section 6.2) and dedicated evaluation algorithms for the various predicate types (Section 6.3). Second, HYDRA optimizes the checking of multiple DCs at once. This is possible, because DCs often share some predicates, which might then need to be checked only once (Section 6.4).

6.2 Checking a single DC

Assume a DC $\psi = \forall t, t'\colon \neg(p_1 \wedge \cdots \wedge p_n)$ whose violating tuple pairs we would like to determine. Intuitively, we can attain this as follows: We initially assume that all tuple pairs $TP := \{(t, t') \in r^2 \mid t \neq t'\}$ violate ψ. Then, we iterate the predicates in ψ in some order and for each predicate p_i, we retain only those tuple pairs that satisfy p_i using a *refiner* function $\text{ref}_{p_i}(TP) := \{(t, t') \in TP \mid (t, t') \text{ satisfies } p_i\}$. Finally, we obtain all tuple pairs $(\text{ref}_{p_1} \circ \cdots \circ \text{ref}_{p_n})(TP)$ that satisfy all predicates in ψ — these are exactly the violating tuple pairs. We now explain how to efficiently implement this proceeding.

Data Structures. Obviously, representing sets of tuple pairs by explicitly materializing them all is neither memory efficient nor allows for efficient processing. Instead, HYDRA uses three data structures to represent sets of tuple pairs: A *cluster* is an unordered set of tuple indices. A *cluster pair* is an ordered pair (c_1, c_2) of two clusters c_1 and c_2. It represents all tuple pairs (t, t'), such that $t \in c_1$ and $t' \in c_2$ and $t \neq t'$. As an example, the cluster pair $(\{t_1, t_2\}, \{t_2, t_3\})$ represents the tuple pairs $(t_1, t_2), (t_1, t_3), (t_2, t_3)$. A *partition* is a set of cluster pairs, which represents all tuple pairs contained in any of those cluster pairs.

The partition representation has the advantage that many partitions can be represented using much less memory than would be required by the enumeration of every tuple pair. For instance, all possible (ordered) tuple pairs in a dataset with $|r|$ tuples, can be represented as one partition with only one cluster pair, where each cluster contains all tuples. We call this partition that represents the cross-product of all tuples the *full partition* of r. This representation requires only $2n$ integers – so only linear space, as opposed to to $n^2 - n$ tuple pairs for a materialization of the cross product.

Operations. Besides being a compact representation, partitions can be employed to efficiently implement the refinement operation defined above. We extend the definition above to a new operator called *partition refiner*, that takes as input a set of cluster pairs and yields a set of cluster pairs that represents all tuples pairs of the input cluster pair that also satisfy a predicate p_i. Again, the repeated application for all predicates of a DC, starting with the full partition, yields the set of all tuple pairs violating this DC. Thus, it is of major importance for HYDRA to efficiently execute the refinement. The actually employed refinement strategies are determined by the type of the processed predicate and are described in detail in Section 6.3.

Pipelining. Although partitions are a compact representation of tuple pairs, they still can consume a considerable amount of memory — in particular, when there are very many very small cluster pairs. HYDRA attenuates this fact by pipelining: Instead of applying the partition refiner for each predicate one after another, HYDRA rather operates on the granularity of cluster pairs. That is, as soon as a partition refiner yields a refined cluster pair, this cluster pair is directly fed into the partition refiner of the next predicate. As an example, consider the full partition for Table 1, which we want to refine with the predicate $t[\mathsf{Name}] = t'[\mathsf{Name}]$. The refined partition would contain four cluster pairs, one for each name. However, as soon as the partition refiner has determined one of the cluster pairs, e.g., $(\{t_1, t_2\}, \{t_1, t_2\})$ for *J. Miller*, this cluster pair can immediately be refined further with other predicates (if any exist). Note that this optimization technique is independent of the actual refinement strategy.

6.3 Refinement strategies

Having explained the utility of partition refinement to discover the violating tuple pairs for a given DC, let us now discuss the actual refinement strategies. In brief, a refinement strategy takes as input a predicate and a partition and outputs a refined partition that contains only tuple pairs that satisfy the given predicate. A naïve solution to this problem could be an algorithm that iterates over all tuple pairs in the partition and adds each of them to the resulting partition if the tuple pair satisfies the predicate. While this solution would work correctly, it certainly is not efficient. As a matter of fact, this strategy would compare all tuple pairs – this is exactly what HYDRA attempts to avoid!

To solve this problem more efficiently, Hydra divides the predicates into the following categories: *filters*, *equi-join*, *anti-join*, and *inequality* joins. For each category we provide an algorithm that, given a predicate of this category, solves the problem by taking the nature of the predicate into account. All cluster pairs can be treated independently and, consequently, the algorithm refines only individual cluster pairs. If a partition is to be refined, the result is the union of all results of the refinement operations for every cluster pair. All of the presented algorithms take a cluster pair as well as a predicate as input and yield new cluster pairs that contain all tuple pairs included in the input cluster pair and satisfy the predicate. Note that the refinement algorithms may yield empty cluster pairs (that describe no tuple pairs), which Hydra immediately discards.

Filters. Filters are all predicates that involve only one tuple (either t or t') and therefore are of the form $\hat{t}[A]\,\theta\,\hat{t}[B]$ for $\theta \in \{=, \neq, <, \leq, >, \geq\}$, any attributes A, B, and $\hat{t} \in \{t, t'\}$. An example for a filter is $t[\mathsf{Hired}] = t[\mathsf{Salary}]$. To refine a cluster pair by a filter on t (or t', respectively), Hydra iterates over all tuples in the first (or second, respectively) cluster and keeps only those tuples that satisfy the predicate. The other cluster is not altered. Thus, the complexity of processing filters is linear in the number of tuples of the filtered cluster.

Equi-join. Equi-join predicates are all predicates of the form $t[A] = t'[B]$ for any attributes A and B (A and B can be equal). Hydra processes them with a technique akin to hash joins in relational databases. It splits the clusters by the values of A and B, and then combines those clusters that result from the same values. In contrast to a classical hash-join algorithm, Hydra needs to build a hash map for both sides of the join condition. The purpose is to output cluster pairs instead of tuple pairs and, hence, to keep the number of cluster pairs small.

Consider this example: given the predicate $t[\mathsf{D\text{-}Code}] = t'[\mathsf{D\text{-}Code}]$ and the cluster pair $(c_1, c_2) = (\{t_1, t_2, t_3\}, \{t_1, t_2, t_5\})$, Hydra constructs a hash-map with D-Code as key: D-Code: $\text{SAL} \rightarrow \{t_1, t_3\}, \text{ACT} \rightarrow \{t_2\}$. Then it constructs the equivalent hash-map for c_2: $\text{SAL} \rightarrow \{t_1, t_5\}, \text{ACT} \rightarrow \{t_2\}$. Combining the clusters for SAL yields the cluster pair $(\{t_1, t_3\}, \{t_1, t_5\})$ and for ACT it would yield $(\{t_2\}, \{t_2\})$. The latter can be ignored as it does not contain a pair of distinct tuples. The complexity is $\mathcal{O}(|c_1| + |c_2|)$, because building the indexes is in $\mathcal{O}(|c_1|)$ and $\mathcal{O}(|c_2|)$ and iterating over the second index is in $\mathcal{O}(|c_2|)$.

Anti-join. Anti-join predicates are all predicates of the form $t[A] \neq t'[B]$ for any attributes A and B (where A and B can be equal). Again, Hydra uses a hashing algorithm to calculate the resulting cluster pairs. The overall approach is similar to the algorithm for equi-join predicates, but instead of combining clusters with equal values, Hydra removes the tuples with equal values from the original clusters. A special case are those tuples in the first cluster that lack a join partner in the second cluster: all of these tuples can be combined into one cluster pair, because they need no further refinement.

Let the predicate $t[\mathsf{Salary}] \neq t'[\mathsf{Salary}]$ and the cluster pair $(c_1, c_2) = (\{t_1, t_2, t_3\}, \{t_4, t_5\})$ serve as an example. Constructing a hash-map for Table 1 with Salary as key leads to $3{,}200 \rightarrow \{t_1\}$, $3{,}500 \rightarrow \{t_2\}$, $2{,}900 \rightarrow \{t_3\}$ and $3{,}200 \rightarrow \{t_4\}$, $2{,}700 \rightarrow \{t_5\}$. As $\{t_1\}$ has a corresponding cluster $\{t_4\}$, it can only be joined with $c_2 \setminus \{t_4\}$, which yields the result $(\{t_1\}, \{t_5\})$. t_2 and t_3 do not have any join partners so they can be combined with the whole c_2 cluster: $(\{t_2, t_3\}, \{t_4, t_5\})$. The worst-case time complexity here is $\mathcal{O}(|c_1| \cdot |c_2|)$. If each tuple in c_1 has a unique value for A and for each there is one tuple in t_2, such that $t[A] = t'[B]$, then a total of $|c_1|$ arrays of size $|c_2| - 1$ need to be constructed.

Inequality Join. Inequality join predicates are all predicates of the form $t[A]\,\theta\,t'[B]$ for $\theta \in \{<, \leq, >, \geq\}$, any attributes A and B (where A and B can be equal). To evaluate single inequality joins, Hydra uses a variant of a sort-merge join: Hydra first sorts both clusters according to the values of the predicate's attributes (the first cluster by A and the second cluster by B). If $\theta \in \{<, \leq\}$ the clusters are sorted in ascending order, and in descending order otherwise. Because θ is transitive, these sorted arrays, denoted by a and b, satisfy the following properties:

(i) if the tuple pair (a_i, b_j) satisfies the predicate, then all tuple pairs (a_i, b_k) $\forall k \geq j$ also satisfy the predicate;
(ii) if for (a_i, b_j) the predicate is not satisfied, then it cannot be satisfied for any (a_l, b_j) $\forall l \geq i$ either.

As an example, consider the predicate $t[\mathsf{Salary}] \geq t'[\mathsf{Salary}]$ along with the cluster pair $(c_1, c_2) = (\{t_3, t_4\}, \{t_1, t_2, t_5\})$. In this example, Hydra creates $a = [t_4, t_3]$ and $b = [t_2, t_1, t_5]$. Then, the algorithm iterates both a and b in a nested loop. That is, in our example it starts by comparing t_4 and t_2, for which the predicate evaluates to false. However, the next tuple pair (t_4, t_1) does satisfy the predicate, so Hydra can instantly yield the cluster pair $(\{t_4\}, \{t_1, t_5\})$ due to property (i) without any further checks. In the second iteration of the outer loop, property (ii) allows the algorithm to skip all tuples in the second cluster that did not satisfy the predicate in the last iteration (so t_2 in this case). The check for (t_3, t_1) is without success and only the final check returns the second result $(\{t_3\}, \{t_5\})$. Note that if two subsequently yielded cluster pairs have identical "right" clusters (which is not the case in our example), the two can be merged into a single cluster pair to speed up subsequent refiners. The worst-case complexity of this algorithm is $\mathcal{O}(|c_1| \cdot \log|c_1| + |c_2| \cdot \log|c_2| + |c_1| \cdot |c_2|)$. The reason is similar to that for the anti-join refinement, but in this case the clusters additionally need to be sorted.

IEJoin. Inequality joins usually are computationally expensive and have a low selectivity, thereby putting a lot of work to subsequent refiners. To attenuate this issue, Hydra tries to evaluate inequality joins as pairs of two, whenever possible, using the more efficient IEJoin [10], which evaluates two inequality predicates in a single pass. In this section, we convey the basic idea of this join algorithm.

In contrast to the regular inequality join, both clusters need to be sorted twice instead of only once (once for each involved attribute). At first, IEJoin iterates over a sorted version of the first cluster to find join partners that satisfy the first predicate. As these join partners do not necessarily also satisfy the second predicate, the join partners are not generated immediately, but instead marked in a bitset that is initially empty. Through the use of an index structure, the algorithm determines the part of the bitset that also satisfies the second predicate. Finally, the algorithm iterates over that part, and for every marked bit the original version would yield individual tuple pairs that satisfy both predicates, but we show next how to obtain cluster pairs instead. With similar arguments as for the previous algorithm, the set bits in the bitset are valid for the next iterations as well.

As also shown in [10], the algorithm has a worst-case complexity of $\mathcal{O}(|c_1| \cdot \log|c_1| + |c_2| \cdot \log|c_2| + |c_1| \cdot |c_2|)$. However, because the original IEJoin yields tuples rather than cluster pairs, a few optimizations are possible. When iterating over the bitset (inner loop) to find join partners, the modified algorithm collects all eligible tuples into one cluster instead of yielding them all separately. Additionally, the resulting cluster pair of the previous iteration in the outer loop is saved and not yielded directly. If the next tuple has the same join partners as the previous ones, then the current tuple is added to the last cluster. Only if they do not match or if the algorithm finishes, the last result is returned. Both optimizations reduce the number of cluster pairs.

6.4 Checking many DCs

The fact that Hydra has to check many DCs at once allows further optimizations. First, to reduce the number of DCs that need to be checked, we use implication testing and remove redundant DCs from Ψ_{pre}. Additionally, many partition refinement operations can be combined by organizing the predicates of the DCs in a tree. Finally, the number of partition refinements can be further optimized by changing the shape of that tree through modifying the order of predicates in the DCs.

Redundant DCs. Usually, Ψ_{pre} contains redundant DCs in the sense that all tuple pairs that violate them also violate another DC, because they are implied by that other DC. So checking these implied DCs does not augment the evidence set and thereby incurs computational overhead: removing them from Ψ_{pre} reduces the number of DCs that need to be checked, and, hence, the number of necessary refinement operations. For this purpose, Hydra artificially extends all DCs $\psi \in \Psi_{\text{pre}}$ to ψ^* by adding implied predicates that must be satisfied by any tuple pair that satisfies all predicates of ψ. For example, the DC $\psi = \forall t, t' \colon \neg\big(t[\mathsf{Id}] < t'[\mathsf{Id}] \wedge t[\mathsf{Hired}] > t'[\mathsf{Hired}]\big)$ can be extended by the predicate $t[\mathsf{Id}] \le t'[\mathsf{Id}]$. Note that ψ and ψ^* are logically equivalent despite their different notations, because any evidence e that violates a DC ψ also violates ψ^*. We can further conclude that any DC $\psi' \in \Psi_{\text{pre}}$, if it comprises exclusively predicates from ψ^*, logically implies ψ^*, i.e., ψ. For instance, ψ_3 in Figure 1 satisfies this condition, although it is not a subset of ψ itself. As a result, to complete the evidence set, it is sufficient to check only ψ', which allows Hydra to skip ψ. To calculate ψ^*, Hydra uses a set of sound implication rules that are repeatedly applied until no new predicates are found: general implications (i), transitivity (ii), anti-symmetry (iii), and other rules (iv) and (v).

$$\begin{array}{rll}
\text{i)} & t[A] < t'[B] & \Rightarrow t[A] \le t'[B], t[A] \ne t'[B] \\
\text{ii)} & t[A] < t'[B] \wedge t'[B] < t[C] & \Rightarrow t[A] < t[C] \\
\text{iii)} & t[A] \le t'[B] \wedge t'[B] \le t[A] & \Rightarrow t[A] = t'[B] \\
\text{iv)} & t[A] \ne t'[B] \wedge t[A] \le t'[B] & \Rightarrow t[A] < t'[B] \\
\text{v)} & t[A] \ne t'[B] \wedge t'[B] = t'[C] & \Rightarrow t[A] \ne t'[C]
\end{array}$$

Hydra reduces the number of DCs to be checked even further by considering symmetry: for each DC ψ it calculates a symmetric DC ψ_{sym} by swapping t and t'. Any tuple pair that violates ψ violates ψ_{sym} with inverse roles of t and t'. Consider $\forall t, t' \colon \neg\big(t[\mathsf{Id}] > t'[\mathsf{Id}] \wedge t[\mathsf{Hired}] < t'[\mathsf{Hired}]\big)$ as an example for the symmetric DC of ψ above. If ψ_{sym} is redundant according to the rules above, Hydra removes ψ from Ψ_{pre} in favor of the DC that makes it redundant, so in our example ψ_3. Instead of checking symmetrical DCs, Hydra calculates both $e(t, t')$ and $e(t', t)$ for every discovered violating tuple pair and adds them to E_Δ. As Hydra needs to access both tuples t and t' for the calculation of $e(t, t')$ anyway, the calculation for inverse roles is very cheap, but can save a high number of refinement operations.

Tree-based checking. Many DCs in the preliminary DC set have predicates in common with some other DCs. If they were each checked sequentially, many calculations for these predicates would be repeated. Instead, Hydra performs the DC checking in a depth-first tree traversal and combines DCs that share a common prefix, for an example see Figure 4. In the tree there is one path from root to leaf for every DC and each of the path's edges represents one predicate in the DC. The construction starts in the root node and the set of DCs is split by their first predicate (the order of predicates is assumed random for now), so for example $t[\mathsf{Hired}] = t'[\mathsf{Hired}]$ and $t[\mathsf{Name}] = t'[\mathsf{Name}]$. For each of the resulting DC sets one edge labeled by their shared predicate and a node that represents the DC set is added. Then, for each of the newly constructed nodes, the remaining set of DCs is split by the second predicate and so on. In our example only the first child ($t[\mathsf{Hired}] = t'[\mathsf{Hired}]$) needs to be further split, as the second node already represents the contained DC.

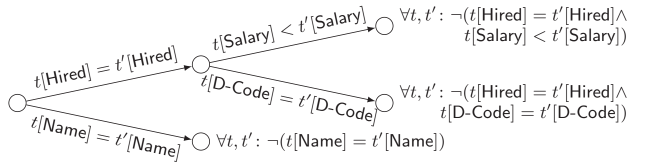
$t[\mathsf{Hired}] = t'[\mathsf{Hired}]$
$t[\mathsf{Salary}] < t'[\mathsf{Salary}]$
$\forall t, t' \colon \neg(t[\mathsf{Hired}] = t'[\mathsf{Hired}] \wedge t[\mathsf{Salary}] < t'[\mathsf{Salary}])$
$t[\mathsf{D\text{-}Code}] = t'[\mathsf{D\text{-}Code}]$
$\forall t, t' \colon \neg(t[\mathsf{Hired}] = t'[\mathsf{Hired}] \wedge t[\mathsf{D\text{-}Code}] = t'[\mathsf{D\text{-}Code}])$
$t[\mathsf{Name}] = t'[\mathsf{Name}]$
$\forall t, t' \colon \neg(t[\mathsf{Name}] = t'[\mathsf{Name}])$

Figure 4: Example tree for a set of three DCs. Changing the order of predicates for one of the DCs that contain the predicate $t[\mathsf{Hired}] = t'[\mathsf{Hired}]$ would result in larger tree, but depending on the selectivity estimates it may still pay off, if the intermediate results are smaller.

Once the tree is constructed, Hydra uses it to find violations of each DC: cluster pairs traverse the tree in a depth-first manner, starting with the full partition in the root node. Whenever a cluster pair traverses an edge the refiner for the predicate of this edge is applied to the cluster pair. As soon as the refiner yields a new cluster pair, a new cluster pair is added to the child node. Deeper nodes have a higher priority and therefore this cluster pair is now further refined before the evaluation of the first refiner is continued. Once a cluster pair hits a leaf, all of its tuple pairs satisfy all predicates in a DC and, hence, are violating tuple pairs that can contribute to E_Δ.

Order of evaluation. Until now the order of predicates was assumed to be random for the tree construction, but in fact the order of evaluation has a high performance impact. There are two aims that need to be considered: (i) the intermediate results should be as small as possible, because the fewer tuple pairs a cluster pair contains, the cheaper are subsequent refinement operations and the less memory is used; (ii) the evaluation tree should be as small as possible, so we can save on repeated refinement operations.

Aim (i) can be achieved by sorting the predicates by their *selectivity*, so the refiners filter out more tuple pairs early. Although the true value of the predicate selectivity remains unknown, we obtain a good estimation in the random sampling phase. To obtain these selectivity estimations, HYDRA counts the occurrences of any predicate as well as any pair of inequality predicates in the sample evidence set. In contrast to that, a heuristic to attain a small tree of refinement operations (Aim (ii)) is to split on *frequent* predicates in the (approximate) DC set first. However, this strategy could lead to large intermediate results, because it can pick predicates with a low selectivity too early. Instead, to combine both aims, HYDRA maximizes the quotient of frequency and selectivity. Thus, HYDRA prefers refinement operations with a high frequency and a low selectivity.

For IEJOIN, inequality joins need to be combined into disjoint pairs of two. So for every DC that contains at least three inequality predicates, HYDRA needs to decide which of them should be combined into one partition refiner. Here however, the same considerations as for the predicate selection in general apply: HYDRA maximizes the total frequency divided by the total selectivity of these combined partition refiners. The sample can also be used to estimate the selectivity of those pairs (counting the tuple pairs for which both predicates are true). These pairs of inequality joins are then treated like the other refiners and also sorted by their combined priority weight.

Final result. At the end of the whole depth-first traversal, we obtain the complete delta evidence set. HYDRA could continue the previous evidence inversion if it saved the original Ψ_{pre} (including all redundant DCs) and the index structure for the subset checking. For memory efficiency and due to the fact that experiments show that evidence inversion is rather cheap compared to HYDRA's other phases, HYDRA does a new evidence inversion for the complete evidence set, which yields the complete set of minimal DCs (see Theorem 2).

7. EVALUATION

In this section, we compare HYDRA with the only other published DC discovery algorithm FASTDC [3]. In particular, (i) we demonstrate the improved efficiency of our algorithm; (ii) we investigate its scalability over growing dataset sizes; (iii) we explore its memory consumption; and (iv) we provide a number of in-depth experiments that allow a better understanding of the interaction of HYDRA's different phases thereby justifying our design decisions.

7.1 Setup and Datasets

For our experiments, we implemented both HYDRA and FASTDC in Java and integrated them into Metanome [11], a data profiling framework. For FASTDC no prior implementation was publicly available, but the observations with our best-effort implementation generally agree with those of Chu et al. [3]. All experiments were run on a Dell PowerEdge R620 with two Intel Xeon E5-2650 CPUs (2.00 GHz, Octa-Core) and 128 GB of DDR3 RAM running CentOS release 6.8 and OpenJDK 64-Bit Server JVM 1.8.0_101. Furthermore, all measurements of HYDRA were repeated at least three times to accommodate the randomness of the samples and the mean values are reported. HYDRA conducts a random sampling with 20 random samples per tuple as more samples do not seem to improve the selectivity estimations any further (Section 7.5.1). The focused sampling continues until the efficiency drops below a threshold of 0.005.

The authors of FASTDC evaluated their algorithm against three datasets [3]: *Tax*, *Hospital*, and *Stock*. We reuse these datasets to ensure that our implementation is comparable in terms of performance to the original implementation. *Tax* is a generated dataset with personal data, such as salary, zip code, and city; *Hospital* features information about hospital provider, type, etc.; and *Stock* contains historical data for the S&P 500 stocks. Table 2 provides more details on the datasets. Furthermore, our repeatability page[1] provides our implementations and pointers to the datasets.

Table 2: The three datasets used for evaluation.

Name	#attr.	#tuples	#pred.	Type
Hospital	15	114,919	34	real-world
Stock	7	122,496	82	real-world
Tax	15	1,000,000	50	synthetic

Predicate spaces. To allow for a fair comparison of HYDRA and FASTDC, both algorithms consistently apply the predicate space restrictions from [3]: textual attributes are compared only for (in)equality, while for numeric attributes all six operators are allowed ($=, \neq, <, \leq, >, \geq$). Furthermore, the predicates that involve two different attributes are restricted: For *Hospital* and *Stock*, two distinct attributes A and B, need to have at least 20% common values to allow the predicates $t[A]\,\theta\,t[B]$ and $t[A]\,\theta\,t'[B]$ for $\theta \in \{=, \neq\}$. If A and B are numeric attributes and their arithmetic means are in the same order of magnitude, we furthermore add $t[A]\,\theta\,t'[B]$ for $\theta \in \{<, \leq, >, \geq\}$. For *Hospital* this leads to 34 predicates, while for *Stock* there are 82 predicates. For *Tax*, the predicate space P contains only predicates that compare the same attribute for different tuples, which leads to 50 predicates. In [3], the authors already showed that these restrictions are quite reasonable, because they prune uninteresting DCs and at the same time speed up discovery. Of course, we can also run HYDRA on a predicate space without restrictions: although the discovery takes up to three times longer on these datasets, we mainly gain uninteresting new results, such as $\forall t, t' : \neg(t.\mathsf{zipCode} = t'.\mathsf{HospitalOwner})$.

7.2 Tuple Scalability

This experiment evaluates the tuple scalability by measuring the runtime for the same dataset but different numbers of tuples. The aim is to receive an impression of the growth behavior and, hence, to facilitate an estimation on how the algorithm behaves on larger datasets. We vary the number of tuples by gradually considering more tuples starting from the beginning of the dataset. Figure 5 shows the scaling behavior of both FASTDC and HYDRA. For all three datasets we see that HYDRA's runtime scales almost linearly in the number of tuples. For *Tax*, an inflection point occurs at around 400,000 tuples, but the overall scaling behavior still seems linear. The increased gradient might be attributed to low-level caching effects. On contrast, for *Hospital* and

[1]`https://hpi.de/naumann/projects/repeatability/data-profiling/dc.html`

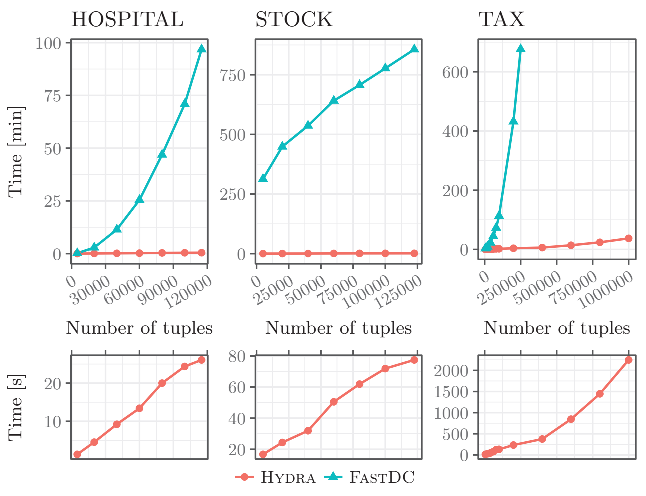

Figure 5: Comparison of Hydra's and FastDC's scalability with an increasing number of tuples. The upper plots show the runtimes for both in minutes, while the lower plots give a more detailed view for Hydra's runtime in seconds.

Tax, the quadratic complexity of FastDC is evident. The quadratic runtime complexity of FastDC does not show for the *Stock* dataset, because here FastDC's runtime is dominated by its second phase (the minimal cover search, comparable to our evidence inversion). This phase is highly sensitive to the number of predicates and *Stock* combines a big predicate space with a relatively small number of tuples.

Hydra's almost linear scaling is a clear improvement over the quadratic complexity of FastDC, which becomes increasingly evident for larger datasets, because the speedup of Hydra in comparison to FastDC grows linear in the number of tuples. Due to this quadratic scaling, FastDC's runtime hits the time limit of 15 hours before it can process the complete *Tax* dataset, which Hydra processes in about 40 minutes. The estimated runtime of FastDC for this dataset is more than a week and even a parallel version of FastDC takes 16 hours using all 16 cores of the CPU, which results in an estimated speedup factor of about 360 times for Hydra over a single-threaded version of FastDC. Hydra finishes the two smaller datasets *Hospital* and *Stock* at a maximum of 2 minutes, while FastDC takes about 90 minutes for *Hospital* and more than 14 hours for *Stock*. To conclude, we can say that Hydra not only scales better, but also shows a much better runtime behavior in absolute numbers. Due to these results, we focus only on Hydra in the following.

7.3 Predicate Scalability

The maximum size of the predicate space grows exponentially in the number of attributes. Because the number of attributes widely differs between different datasets, we investigate how well Hydra scales to large predicate spaces. For this scaling experiment, we ran Hydra only on the first 20,000 tuples of *Hospital*, *Stock* and *Tax* to reduce the influence exerted by the number of tuples. In order to vary the number of predicates, we considered different attribute subsets of the datasets. In detail, we started with only one

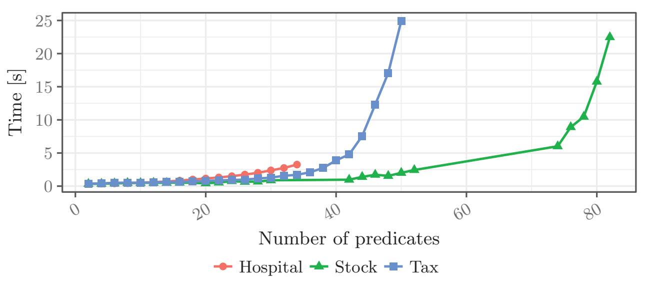

Figure 6: Scalability of Hydra with an increasing number of predicates on the first 20,000 tuples.

attribute and successively added attributes chosen at random until the full set of attributes was reached. This process was repeated multiple times and the average runtimes are reported in Figure 6.

Apparently, Hydra's runtime increases exponentially with the number of predicates. However, a better scaling behavior is precluded, because we observe an exponential growth of the results as well. That being said, while Hydra easily scales to datasets with up to 15 columns, in situations with more columns one might consider to tighten the predicate space restrictions to allow further scaling. The larger predicate space is also the reason, why even Hydra cannot compete against more specialized algorithms for, e.g., FD discovery. But although the FD discovery algorithm HyFD [13] finishes FD discovery in less than 15s on all of these datasets, a large share of valid DCs (up to 99 %) on these datasets cannot be expressed as FDs.

7.4 Memory Usage

While one might assume that especially the DC checking of Hydra requires a high amount of memory for storing the involved cluster pairs, we observed Hydra to have a low memory footprint. To determine its peak memory usage, we ran Hydra with different main memory capacities, starting from 128 MB and successively doubling that value until the algorithm finished three times in a row for one limit. We found that Hydra can process the *Hospital* and *Stock* datasets with 256 MB of main memory. In contrast, the *Tax* dataset required 2 GB. A reason for this demand is the fact that our implementation loads the full dataset into main memory. Even though the *Tax* dataset's CSV file takes up only 73 MB, it effectively requires much more main memory when loaded into the JVM, in particular due to the overhead of string objects and their UTF-16 encoding. Using a heap space of 120 GB, Hydra can process a *Tax* instance of 50 M tuples (in 7h), while it runs out of main memory for 75 M tuples while loading the input data.

7.5 Phases of Hydra

Having demonstrated Hydra's performance and scalability characteristics, we proceed to evaluate its various phases in more detail, thereby accounting for our design decisions. Figure 7 gives an overview of which phases of Hydra dominate in different scenarios and aims to establish an intuition on which phases are the most critical for efficiency. Overall, the evidence set completion appears to be the most expensive part. It dominates especially for datasets with a lot of tuples (*Tax* dataset). The runtimes of the linear and focused sampling grow linear in the number of tuples. In addition,

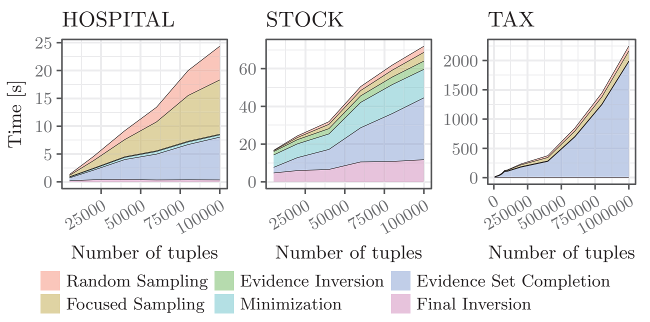

Figure 7: Runtime distribution for the different phases of Hydra while scaling the number of tuples.

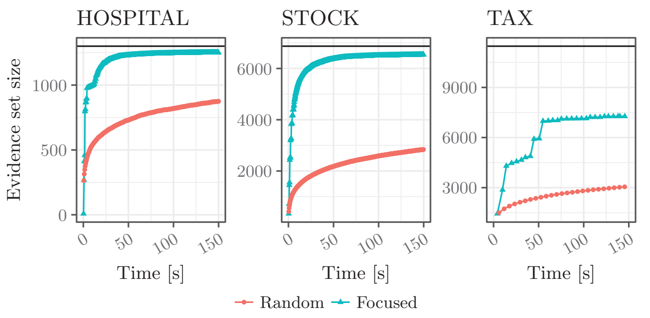

Figure 8: Effectiveness of focused sampling. The black horizontal line marks the size of the complete evidence set. The random sampling serves here as a baseline.

the experiments on the *Stock* dataset with its 82 predicates give a notable impression of the influence of a high number of predicates: in such circumstances the evidence inversion and the minimization take a much longer time than for datasets with fewer predicates.

7.5.1 Sampling

In the following, the focused sampling's effectiveness is evaluated w.r.t. its recall on E_{full} over time, as well as how the specific sampling configurations influence the total runtime of Hydra. Note that although random and focused sampling serve different purposes in Hydra, the random sampling still can serve as a baseline to show how much faster Hydra's focused sampling completes the evidence set.

Figure 8 evaluates the effectiveness of the focused sampling method described in Section 4. At different points in time, the size of the evidence set was measured up to a time limit of 150 seconds. The focused sampling method achieves a much higher recall on the evidence set than the random sampling methods in a short amount of time. This fact shows that our focused sampling is working well and delivers a diversity of tuple pairs.

Figure 9 gives insight on the influence of Hydra's two sampling parameters on Hydra's total runtime: the number of samples per tuple k in the random sampling and the efficiency threshold in the focused sampling phase. The overall result of these experiments is that Hydra is relatively robust for the choice of both parameters. Hydra performs a random sampling to estimate predicate selectivity. Figure 9a) shows the impact of different numbers of samples per tuple k for this phase: the runtime of the random sampling

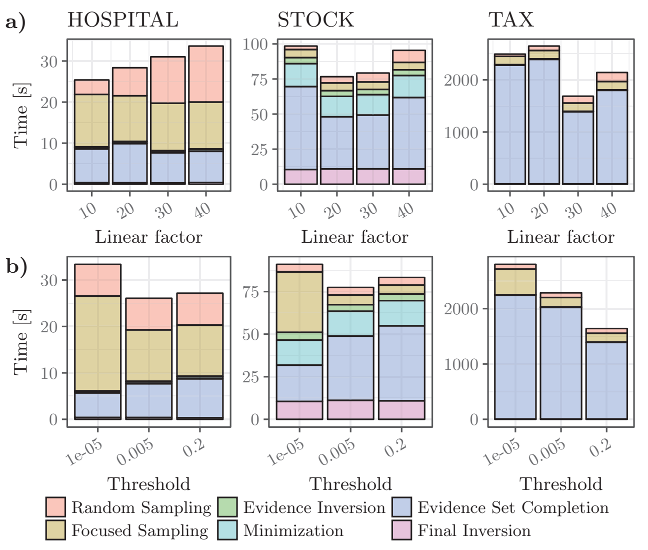

Figure 9: Influence of a) the samples per tuple k in the random sampling and of b) the focused sampling's efficiency threshold on Hydra's total runtime.

grows obviously linear with k, but due to better selectivity estimations, the algorithm can save time in the evidence completion phase. The influence of this parameter choice is small and 20 samples per tuple seem to work well for both real-world datasets and show only slight disadvantages for the synthetic *Tax* dataset.

In Figure 9b) we can observe the influence of the efficiency threshold of the focused sampling with interesting results. A smaller efficiency threshold causes more sampling iterations and therefore of course a longer sampling phase. This increase does not come as a surprise, and on the *Hospital* and *Stock* datasets the additional sampling pays off by reducing the time spent on the evidence completion. For both datasets the threshold 0.005 works best, and the performance decreases in both directions (although only slowly). Unexpectedly, for the *Tax* dataset a smaller threshold and, hence, more sampling, does not lower the time needed for the result completion. One might think that this phenomenon occurs because the tree of DCs that needs to be checked grows with an increasing sample. However, we could not measure a substantial growth of the tree size. This suggests that due to the more extensive sampling more complex predicate combinations arise that need to be checked in the evidence set completion. Nevertheless, this observation is actually less of a problem, because Hydra's default threshold of 0.005 for the sampling saves in this case on both, the time for the sampling as well as the result completion. As the result for the *Tax* dataset seems to constitute an exception and as it is also a generated dataset, the best threshold of 0.005 observed on the other datasets was chosen for Hydra. Overall, Hydra is not very sensitive to the parameters and hence has proven its robustness.

7.5.2 Evidence Inversion

This experiment evaluates the gains of Hydra's evidence inversion in comparison to FastDC's minimal cover search, which serves the same purpose. The underlying set cover

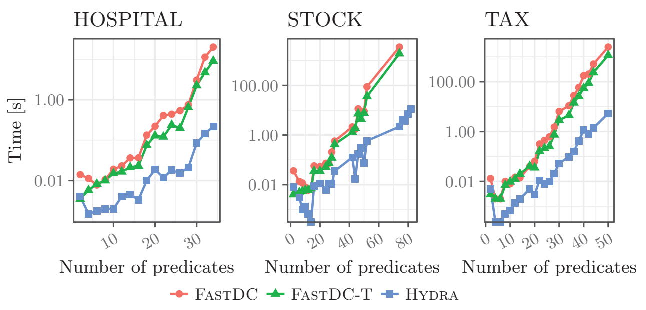

Figure 10: Scalability of different evidence inversion methods with an increasing number of predicates compared on the full evidence set (log-scale).

Table 3: Effect of different refiner weights on the runtime of the result completion.

Order criterion	*Hospital*	*Stock*	*Tax*
Selectivity only	7.5s	422.8s	$> 2h$
Frequency / selectivity	**6.1s**	**36.5s**	**38m**
Frequency only	81.6s	$> 2h$	$> 2h$

problem is NP-complete, so it is not surprising that Figure 10 shows an exponential runtime behavior for both methods with an increasing number of predicates (mind that the y-axis for this graph is in log scale). However, as it is an NP-complete problem we should pay special attention and design the algorithm with great care as even a constant factor speedup can have serious performance impacts. The general testing procedure is similar to that for the predicate scaling experiment with a time limit of 60 minutes. It is interesting that FastDC's cover inversion is faster if one of its optimizations (the transitivity pruning rule) is disabled (*FastDC-T*) and we therefore disabled it in general scaling experiments. Nevertheless, Hydra's evidence inversion is faster by orders of magnitude in all scenarios. The observed speedup is consistent with experiments in [12] for the domain of functional dependency discovery: Fdep [6], whose cover inversion inspired that of Hydra, scales better with the number of columns than FastFD [16], which in turn inspired FastDC.

7.5.3 Evidence Set Completion

Hydra considers both the selectivity and the frequency of partition refiners (predicates or pairs of predicates) to determine their order of evaluation in the evidence completion phase. Table 3 shows that it is indeed important to take the selectivity of the refiners into account as well as their frequency in the DC set. If only the selectivity is considered or only the frequency, then the DC checking on the *Tax* dataset does not finish within a time limit of two hours. However, if we optimize the ratio of both, the checking finishes in about 40 minutes. Additionally, we can see that the selectivity is a more important criterion than the frequency. If only the frequency is taken into account the resulting runtimes are worse for every dataset compared to if only the selectivity is considered.

Finally, Figure 11 stresses the importance of minimizing the set of DCs prior to checking them. The minimization introduces a small extra cost, but the total runtime can be up to four times smaller than without the minimization (*NOMIN*). The effect is bigger for datasets with a larger number of predicates and therefore also a larger number of DCs. The minimization not only reduces the number of DCs to be checked, but, as it also prefers smaller representations, the number of necessary refinement operations decrease as well. In the same figure the effect of the pipelining (described in Section 6.2) on the runtime is shown by comparing Hydra to a version with disabled pipelining (*NOPIPE*). Although the pipelining's effect on the runtime is negligible, it serves its purpose very well: it reduces the required memory significantly. Without pipelining, *Hospital* can still be processed with 256 MB of RAM, but for *Stock* the required memory doubles (512 MB) and *Tax* even requires four times the original amount (16 GB instead of 2 GB).

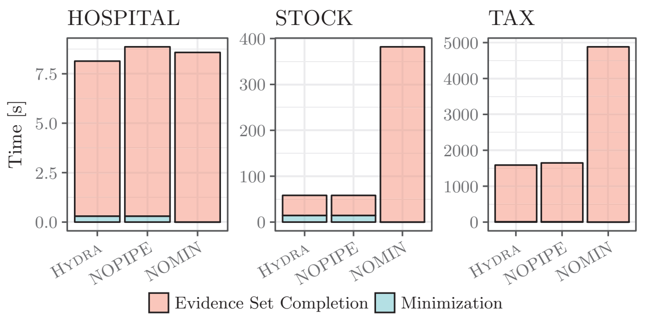

Figure 11: Evaluation of variants for the result completion comparing original Hydra with a version of disabled pipelining (*NOPIPE*) and no prior minimization (*NOMIN*).

8. CONCLUSIONS

We presented the novel denial constraint discovery algorithm Hydra. Hydra first approximates the set of valid denial constraints through intelligent sampling, and then efficiently checks and corrects invalid results to finally obtain a complete and correct result, thereby avoiding to compare all tuple pairs in the given dataset. The algorithm's experimentally determined runtime grows only linearly in the number of tuples, and thus overcomes the quadratic runtime complexity of state-of-the-art DC discovery methods. This results in a speedup by orders of magnitude, especially for datasets with a large number of tuples. In fact, Hydra can deliver results in a matter of seconds that to date took hours to compute.

While Hydra discovers exact DCs, real-world data might contain errors, so that an approximate version is a next step. An adapted version needs to count the number of times it saw each evidence, ensuring not to count twice any evidence originating from the same tuple pair. Due to the approximation, more evidences are necessary to preclude a DC. An adjusted efficiency definition in the focused sampling phase takes this into account and results in more sampling. Hydra's cover inversion technique needs major changes to incorporate the observed frequency of evidences, while the evidence set completion requires almost no changes.

Acknowledgements. We thank the X. Chu, I. F. Ilyas, and P. Papotti for their help in starting this project. This research was partly funded by the German Research Society (grant no. FOR 1306).

9. REFERENCES

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

[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)*, pages 1803–1812, 2016.

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

[4] X. Chu, I. F. Ilyas, and P. Papotti. Holistic Data Cleaning: Putting Violations Into Context. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 458–469, 2013.

[5] J. Ehrlich, M. Roick, L. Schulze, J. Zwiener, T. Papenbrock, and F. Naumann. Holistic data profiling: Simultaneous discovery of various metadata. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 305–316, 2016.

[6] P. A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI Communications*, 12(3):139–160, 1999.

[7] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC data-cleaning framework. *PVLDB*, 6(9):625–636, 2013.

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

[9] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[10] Z. Khayyat, W. Lucia, M. Singh, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and P. Kalnis. Lightning Fast and Space Efficient Inequality Joins. *PVLDB*, 8(13):2074–2085, 2016.

[11] T. Papenbrock, T. Bergmann, M. Finke, J. Zwiener, and F. Naumann. Data Profiling with Metanome. *PVLDB*, 8(12):1860–1863, 2015.

[12] 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*, 8(10):1082–1093, 2015.

[13] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 821–833, 2016.

[14] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. Gordian: Efficient and scalable discovery of composite keys. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 691–702, 2006.

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

[16] C. Wyss, C. Giannella, and E. Robertson. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In *Proceedings of the International Conference of Data Warehousing and Knowledge Discovery (DaWaK)*, pages 101–110, 2001.