Discovery of Approximate (and Exact) Denial Constraints

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

Eduardo C. de Almeida
Federal University of Paraná
Curitiba, Brazil
eduardo@inf.ufpr.br

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

ABSTRACT

Maintaining data consistency is known to be hard. Recent approaches have relied on integrity constraints to deal with the problem – correct and complete constraints naturally work towards data consistency. State-of-the-art data cleaning frameworks have used the formalism known as denial constraint (DC) to handle a wide range of real-world constraints. Each DC expresses a relationship between predicates that indicate which combinations of attribute values are inconsistent. The design of DCs, however, must keep pace with the complexity of data and applications.

The alternative to designing DCs by hand is automatically discovering DCs from data, which is computationally expensive due to the large search space of DCs. To tackle this challenging task, we present a novel algorithm to efficiently discover DCs: DCFINDER. The algorithm combines data structures called position list indexes with techniques based on predicate selectivity to efficiently validate DC candidates. Because the available data often contain errors, DCFINDER is especially designed to discovering approximate DCs, i.e., DCs that may partially hold. Our experimental evaluation uses real and synthetic datasets and shows that DCFINDER outperforms all the existing approximate DC discovery algorithms.

PVLDB Reference Format:
Eduardo H. M. Pena, Eduardo C. de Almeida and Felix Naumann. Discovery of Approximate (and Exact) Denial Constraints. *PVLDB*, 13(3): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/3368289.3368293

1. DENIAL CONSTRAINT DISCOVERY

Integrity constraints are fundamental rules for ensuring that data updates do not cause inconsistencies in databases. Typical applications of constraints include database design [30, 36], data integration [35], query optimization [23], and data cleaning [9, 33], to name but a few. There are different formalisms to define them. Each one has its own level of expressiveness. For example, *functional dependencies* express relationships between two attribute sets. *Conditional functional dependencies* also do so, but for subsets of tuples with specific constant patterns [13]. The values of *unique column combinations* uniquely identify all tuples of a relational instance [17]. *Order dependencies* are a generalization of functional dependencies, and can define ordering relationships between attributes [10]. Readers can find pointers to the literature on various types of data dependencies in the comprehensive studies [1, 19].

This paper concerns the automatic discovery of *denial constraints* (DCs), a generalization of all formalism above-mentioned. Each DC defines a set of predicates for which its predicates cannot hold true simultaneously. The idea is to use relational predicate relationships to determine inconsistent combinations of attribute values.

Let us walk through two DCs for the tuples of the *employees* relation in Table 1. First, any two employees that have the same $\{\mathsf{Name}, \mathsf{Phone}\}$ values have the same $\{\mathsf{Position}\}$ value. This statement is in fact a functional dependency, which is translated into a DC as follows: If a tuple pair $\mathsf{t}_x, \mathsf{t}_y$ of *employees* satisfies the predicates $\mathsf{t}_x.\mathsf{Name} = \mathsf{t}_y.\mathsf{Name}$ and $\mathsf{t}_x.\mathsf{Phone} = \mathsf{t}_y.\mathsf{Phone}$, it cannot satisfy the predicate $\mathsf{t}_x.\mathsf{Position} \neq \mathsf{t}_y.\mathsf{Position}$. Second, the relationship between the attributes Position, Salary and Hired shows that for any two employees with the same position, the longer-standing employee always earns the highest salary. If a tuple pair $\mathsf{t}_x, \mathsf{t}_y$ of *employees* has the same position, then the predicate $\mathsf{t}_x.\mathsf{Position} = \mathsf{t}_y.\mathsf{Position}$ is true. If that is the case and $\mathsf{t}_x.\mathsf{Hired} < \mathsf{t}_y.\mathsf{Hired}$ is true, then $\mathsf{t}_x.\mathsf{Salary} < \mathsf{t}_y.\mathsf{Salary}$ is false. Among the constraint formalisms we discussed, DCs are the only one capable of capturing this last constraint.

Table 1: An instance of the relation *employees*.

	Name	Phone	Position	Salary	Hired
t_0	W. Jones	202-222	Developer	$2.000	2012
t_1	B. Jones	202-222	Developer	$3.000	2010
t_2	J. Miller	202-333	Developer	$4.000	2010
t_3	D. Miller	202-333	DBA	$8.000	2010
t_4	W. Jones	202-555	DBA	$7.000	2010
t_5	W. Jones	202-222	Developer	$1.000	2012

The DCs in the previous examples are fully satisfied by the data in Table 1. DCs with this feature are called *exact* DCs. In ideal settings, data is error-free and the constraints are fully satisfied. In reality, data all too often present inconsistencies. The root cause of inconsistencies vary greatly, for instance, from schema evolution to erroneous data imputation not caught by the (un)defined constraints. The workaround for potential data errors is to relax the constraints so that they admit a certain degree of inconsistency, but still hold for most of the data [6]. DCs with this relaxation feature are called relaxed or, here, *approximate DCs*.

Suppose we want to find approximate DCs in *employees* that are violated by at most two tuple pairs. There are two (non-reflexive)

*Work done while at Hasso Plattner Institute.

Proceedings of the VLDB Endowment, Vol. 13, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3368289.3368293

tuple pairs that satisfy the predicates $t_x.\mathsf{Name} = t_y.\mathsf{Name}$ and $t_x.\mathsf{Phone} = t_y.\mathsf{Phone}$ simultaneously, t_0, t_5 and t_5, t_0. These predicates define an approximate DC, which reads: there cannot exist any two employees with the same values of $\{\mathsf{Name}, \mathsf{Phone}\}$. This constraint seems a reasonable key candidate for the *employees* instance and reveals the potential inconsistency between tuples t_0 and t_5 as duplicates. The example shows how meaningful DCs may be "hidden" amid inconsistent data.

Defining integrity constraints by hand requires judging the structure and content of a database. The task requires expertise and time, and furthermore, it is error-prone considering how complex and dynamic datasets can be. But profiling datasets to discover constraints has emerged as a feasible approach [1]. Profiling datasets to discover DCs is particularly appealing. First, DCs subsume various types of integrity constraints [8]. For example, DCs can express relationships between attribute sets, such as functional dependencies, or define ordering relationships between attributes, i.e., order dependencies. A single DC discovery algorithm substitutes the various algorithms required to discover the constraints of different types found in real datasets. Second, DCs advance constraint expressiveness by capturing rules that cannot be expressed by other attribute dependencies. Third, recent studies have used DCs as the *de facto* integrity constraint language [11, 14, 33]. For example, HOLOCLEAN [33] is a data cleaning tool whose inputs are DC sets, dirty datasets, and optionally master data information. Discovered DCs naturally serve as the input of such a tool.

The computational complexity of discovering attribute dependencies regards the number of tuples and attributes of a relation [1]. The complexity of discovering DCs, in turn, has additional challenges because each DC is expressed as a set of predicates rather than a set of attributes. The DC search space consists of any subset of the predicates drawn for a relation. Each attribute adds many DC candidates to the search space, because each additional attribute can generate predicates of various types: Equalities, inequalities, and comparisons across attributes. Therefore, discovering DCs requires efficient techniques to traverse the search space and validate DC candidates. Discovering approximate DCs is even more challenging than discovering exact DCs: It requires keeping track of the number of tuple pairs that violate each candidate. This requirement prohibits the use of pruning techniques based on the fact that a single violation is enough to invalidate a candidate.

We present denial constraint finder (DCFINDER), a novel algorithm to efficiently discover both approximate and exact DCs. DCFINDER iterates over the data records to build auxiliary data structures that guide the discovery. Based on predicate selectivity, the algorithm uses the auxiliary structures to build representations of tuple pairs and their satisfied predicates. With this representation, DCFINDER can directly generate and validate DC candidates.

The ability to measure the interestingness of discovered DCs is important as it helps users decide which DCs are relevant for their application. Because interestingness is a broad concept [15], we use different measures on the discovered DCs. We have designed DCFINDER to output this additional information, giving users different perspectives on the interestingness of DCs.

In summary, our contributions are the following:

- The novel DCFINDER algorithm for the discovery of approximate and exact DCs.
- Experimental comparison of DCFINDER to all known state-of-the-art DC discovery algorithms, showing that DCFINDER is the most efficient algorithm for the discovery of approximate DCs and at times better than the state of the art even for the discovery of exact DCs.
- The use of interestingness measures for DC-selection; for instance, for data cleaning or query optimization.

The rest of the paper is organized as follows: In Section 2, we discuss related work, followed by key definitions and notations in Section 3. In Section 4 we present an overview of DCFINDER. We split the description of our algorithm into preprocessing (Section 5); evidence set building (Section 6); and DC search, followed by DC interestingness (Section 7). In Section 8 we present our experimental evaluation, followed by our conclusions and suggestions for future work in Section 9.

2. RELATED WORK

It is fundamental for users (or software) to know how datasets are structured. Data profiling helps to acquire such knowledge by uncovering important assets from data, such as attribute value statistics, correlation between attributes and businesses rules [1]. The field has been gaining momentum in the database research community, because many data management tasks benefit from its results. A recurring theme is the discovery of integrity constraints [5, 8, 17, 24, 28, 32, 34].

FASTDC was the first algorithm for DC discovery [8]. It compares every tuple pair of the input dataset to compute which predicates each tuple pair satisfies. The result of this computation is called *evidence set*. FASTDC uses the evidence set to guide a depth-first search. The goal is to quickly find *covers* for the evidence set. The authors of [8] have shown that the problem of DC discovery can be transformed into the problem of discovering covers. BFASTDC algorithm presents substantial improvements over FASTDC [23]. The algorithm builds associations between attribute values so that it is not necessary to check every tuple pair on every predicate to build the evidence set. However, BFASTDC requires many logical operations to calculate which predicates are satisfied by tuple pairs, which hinders performance.

The reason why the approaches based on evidence sets are suitable for DC discovery is that they scale well with the number of attributes. Lattice traversal algorithms arrange all possible dependency candidates in a lattice of attribute combinations, and then use the data instance to validate the candidates [18, 26]. Extensive experimental evaluation have shown how these algorithms quickly run into memory or performance issues for datasets with a relative big number of attributes [28]. The search space is even larger for DC discovery, so building lattices of predicate combinations might be prohibitive. Instead of building huge lattices, DCFINDER follows the evidence set approach of FASTDC (and BFASTDC). Evidence sets are comparable to the difference-sets used in the discovery of functional dependencies [28]. These structures help to define the search space based on instance observations rather than exhaustive candidate enumeration. As observed in [28], the algorithms based on difference-sets can keep reasonable memory footprints while generating and validating candidates. With this in mind, efficiently building evidence sets plays a major role in DC discovery. DCFINDER uses attribute value indexing to avoid the expensive tuple pair comparison of FASTDC. To drive efficiency even further, it uses predicate selectivity to avoid the unnecessary large number of logical operations required by BFASTDC.

Having perfect data to derive DCs is unrealistic and it is reasonable to relax their satisfiability criterion. Our study considers the possibility that a few tuple pairs may not satisfy a valid DC due to imperfect data. Still, the discovery algorithm should be able to find that valid (but approximate) DC. It turns out that discovering approximate DCs is even more challenging than discovering exact DCs. For every approximate DC discovered, the algorithm must

guarantee that the number of violations for that DC is no greater than a given threshold. To do so, it needs to known how many tuple pairs may still violate a candidate DC. It is possible to obtain this information from the evidence sets, as long as the algorithm keeps information on evidence multiplicity. FASTDC, BFASTDC and DCFINDER can integrate a few modifications in their operation to provide such information and discover approximate DCs.

HYDRA, yet another DC discovery algorithm, samples tuple pairs to build an intermediary set of DCs [5]. From such a set, the algorithm corrects the tuple pair samples and determines the complete evidence set. In an approach comparable to [8], the algorithm extracts the final DCs from the evidence set. However, HYDRA can use its evidence sets to compute only exact DCs. It is based on the assumption that a DC is valid if there does not exist one single tuple pair violating that DC. Such an assumption does not hold for approximate DCs. HYDRA leaps over the evidence search space to save computations on duplicate pieces of evidence. The technique may reduce computation time, but loses the evidence multiplicity. We observed that the number of evidences produced by HYDRA is only a fraction of the evidences required to discover approximate DCs (more details in Section 8). Adapting HYDRA to discover approximate DCs requires major changes in the algorithm, which is beyond the scope of this paper. In our experiments, however, we use the algorithm as a baseline to evaluate how DCFINDER compares to an specialized exact DC discovery solution.

A direct application for DCs is data cleaning: violations of DCs usually point to inconsistent data. The authors in [9] present a database repairing algorithm based on correcting DC violations. HoloClean brings together DC formalism and statistical analysis to generate a probabilistic model for data repairing [33]. Fan et al. use DCs to express data currency relationships [14]. Fagin et al. use DCs to find inconsistencies in the results of information extraction routines [11]. Because DCs subsume various types of ICs, they inherit applications; for instance, just as functional dependencies, DCs can be used in query optimization [23] or data exchange [25].

3. BACKGROUND

Before stating our problem definition, let us first revisit the syntax and semantics of DCs. They can be expressed as sets of relational predicates. Let r denote a relational instance with schema $\mathsf{R}(\mathsf{A}_1, ..., \mathsf{A}_n)$, t a tuple of r, and $\mathsf{O} = \{=, \neq, <, \leq, >, \geq\}$ a set of built-in operators of the database. A predicate p is a comparison atom with the form $\mathsf{t}_x.\mathsf{A}_i \ \mathsf{o} \ \mathsf{t}_y.\mathsf{A}_j$: $\mathsf{A}_i, \mathsf{A}_j \in \mathsf{R}$; $\mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}$; and $\mathsf{o} \in \mathsf{O}$. Predicates can compare two tuples for the same attribute, so it is possible that the two attributes in a predicate are the same ($i = j$). DCs can include predicates that compare attribute values to constants; they also can be defined using predicates on many tuples at a time. We narrow this study to variable DCs (i.e., DCs without constants) defined over two distinct tuples (i.e., $\mathsf{t}_x \neq \mathsf{t}_y$). We denote $\mathsf{p}.\mathsf{A}_\mathsf{i}$ and $\mathsf{p}.\mathsf{A}_\mathsf{j}$ the attributes of a predicate p, and $\mathsf{p}.\mathsf{o}$ its operator. We write $\mathsf{p}_1 \sim \mathsf{p}_2$ to say that the attributes $\mathsf{p}_1.\mathsf{A}_i = \mathsf{p}_2.\mathsf{A}_i$ and $\mathsf{p}_1.\mathsf{A}_j = \mathsf{p}_2.\mathsf{A}_j$, and $\mathsf{p}_1 \not\sim \mathsf{p}_2$ to say otherwise.

The predicate space P of a relation schema R is the set of all predicates that can form DCs for R.

DEFINITION 1 (DENIAL CONSTRAINT). *A denial constraint φ over relational instance r is a statement of the form*

$$\varphi\colon \forall \mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}, \neg(\mathsf{p}_1 \wedge \ldots \wedge \mathsf{p}_m)$$

where φ is satisfied *by r if and only if for any tuple pair $\mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}$ at least one of the predicates $\mathsf{p}_1, \ldots, \mathsf{p}_m$ is false. A DC φ_1 is* minimal *if there does not exist a φ_2 such that both φ_1 and φ_2 are satisfied by r, and the predicates of φ_2 are a subset of φ_1.*

In other words, a DC φ_1 does not hold if there exists any tuple pair of r that satisfies all the predicates of φ_1, and φ_1 is not minimal if it is simply a generalization of another DC φ_2.

We write $\mathsf{t}_x, \mathsf{t}_y \models \varphi$ to say that the tuple pair $\mathsf{t}_x, \mathsf{t}_y$ satisfies φ, and $\mathsf{t}_x, \mathsf{t}_y \not\models \varphi$ to say otherwise. For simplicity, we omit the tuple quantifiers $\mathsf{t}_x, \mathsf{t}_y$ of φ hereafter. The following DCs express the exact constraints discussed in Section 1:

$$\varphi_1\colon \neg(\mathsf{t}_x.\mathsf{Name} = \mathsf{t}_y.\mathsf{Name} \wedge \mathsf{t}_x.\mathsf{Phone} = \mathsf{t}_y.\mathsf{Phone} \wedge \mathsf{t}_x.\mathsf{Position} \neq \mathsf{t}_y.\mathsf{Position})$$

$$\varphi_2\colon \neg(\mathsf{t}_x.\mathsf{Position} = \mathsf{t}_y.\mathsf{Position} \wedge \mathsf{t}_x.\mathsf{Hired} < \mathsf{t}_y.\mathsf{Hired} \wedge \mathsf{t}_x.\mathsf{Salary} < \mathsf{t}_y.\mathsf{Salary})$$

We are also interested in relaxing the DC satisfiability constraint so that if a DC has just a small number of violations, it still can be considered valid. The proportion between the number of violating tuple pairs and the total number of tuple pairs can be used to quantify the degree of approximation g_1 of a DC φ in r [20]:

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

DEFINITION 2 (APPROXIMATE DENIAL CONSTRAINT). *Given an error threshold ε, $0 \leq \varepsilon < 1$, a denial constraint φ is ε-approximate in r if and only if its degree of approximation $g_1(\varphi, \mathsf{r})$ is below ε.*

The following DC expresses the approximate constraint discussed in Section 1:

$$\varphi_3\colon \neg(\mathsf{t}_x.\mathsf{Name} = \mathsf{t}_y.\mathsf{Name} \wedge \mathsf{t}_x.\mathsf{Phone} = \mathsf{t}_y.\mathsf{Phone})$$

Let $\mathsf{e}_{\mathsf{t}_x,\mathsf{t}_y}$ be the set of predicates that tuple pair $\mathsf{t}_x, \mathsf{t}_y$ satisfies, that is, $\mathsf{e}_{\mathsf{t}_x,\mathsf{t}_y} = \{\mathsf{p} \mid \mathsf{p} \in \mathsf{P}, \mathsf{t}_x, \mathsf{t}_y \models \mathsf{p}\}$. We refer to these subsets as tuple pairs evidence e (or simply evidence e when the context is clear) [8]. Given a relation instance r and a predicate space P, the evidence set E_r is the set of evidence w.r.t. r and P, that is, $\mathsf{E}_\mathsf{r} = \{\mathsf{e}_{\mathsf{t}_x,\mathsf{t}_y} \mid \forall \mathsf{t}_x, \mathsf{t}_y \in \mathsf{r}\}$. The authors in [8] have shown that it is possible to obtain the set of minimal DCs from the evidence set E_r. The evidence set is also used to efficiently calculate the degree of approximation of each candidate DC.

Given a relation instance r, and an error threshold ε, the approximate DC discovery problem is to find all ε-approximate minimal DCs that hold on r. The discovery of exact DCs is a special case of this problem, where the error threshold is set to zero.

4. OVERVIEW OF DCFINDER

Figure 1 depicts the building blocks of our DC discovery algorithm. From the dataset schema, DCFINDER defines a predicate space; and from the dataset records, the algorithm assembles data structures called position list indexes (PLIs). Some types of predicates are most likely to have low selectivity (i.e., when a predicate is satisfied by many tuple pairs). DCFINDER takes this into account to divide the predicate space into likely/unlikely predicate sets. The idea is to presume that an evidence satisfies the least selective predicates. DCFINDER then allocates arrays of evidence where every element holds the set of "most likely satisfied" predicates. The algorithm uses PLIs to compute references to tuple pairs that do satisfy the "unlikely satisfied" predicates. Performing simple logical operations for each of these references brings the arrays of evidence to their consistent state. Finally, the algorithm uses a simple hash table to map the elements of these arrays into the final evidence set.

The evidence set is a compact representation of tuple pairs and their satisfied predicate sets. It allows efficient validation of DC

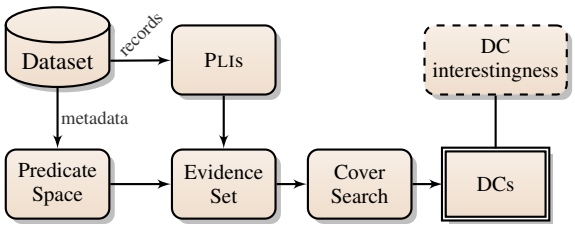


Figure 1: Building blocks of DCFINDER.

candidates. To discover all minimal DCs, DCFINDER uses a depth-first search (DFS) strategy based on evidence set coverage of DC candidates. The last, optional, step is to rank DCs based on interestingness measures to help users filter the discovered results.

5. DATASET TRANSFORMATION

DCFINDER transforms a relational dataset into a predicate space and PLI index structures, as described next.

5.1 From schema into predicate space

Any subset of the predicate space P is a DC candidate, and the DC search space is of size $2^{|\mathsf{P}|}$. We follow related work and apply some restrictions to our predicate space [5, 8]. As showed in [8], restricting the predicate space helps prune meaningless results and reduces computational costs. We distinguish the attribute types whether they are character strings, longs, or doubles, and we use the set of built-in operators $\mathsf{O} = \{=, \neq, <, \leq, >, \geq\}$. For numeric attributes, we define predicates with all operators $\mathsf{o} \in \mathsf{O}$; for non-numeric attributes, we define only predicates with operators $\mathsf{o} \in \{=, \neq\}$. Predicates across two different attributes are regarded only as long as their attributes have the same type and share at least 30% of common values [8]. Figure 2 illustrates the predicate space defined for the relation *employees* of Section 1.

$\mathsf{p}_1 : \mathsf{t}_x.\mathsf{Name} = \mathsf{t}_y.\mathsf{Name}$	$\mathsf{p}_{10} : \mathsf{t}_x.\mathsf{Salary} \leq \mathsf{t}_y.\mathsf{Salary}$
$\mathsf{p}_2 : \mathsf{t}_x.\mathsf{Name} \neq \mathsf{t}_y.\mathsf{Name}$	$\mathsf{p}_{11} : \mathsf{t}_x.\mathsf{Salary} > \mathsf{t}_y.\mathsf{Salary}$
$\mathsf{p}_3 : \mathsf{t}_x.\mathsf{Phone} = \mathsf{t}_y.\mathsf{Phone}$	$\mathsf{p}_{12} : \mathsf{t}_x.\mathsf{Salary} \geq \mathsf{t}_y.\mathsf{Salary}$
$\mathsf{p}_4 : \mathsf{t}_x.\mathsf{Phone} \neq \mathsf{t}_y.\mathsf{Phone}$	$\mathsf{p}_{13} : \mathsf{t}_x.\mathsf{Hired} = \mathsf{t}_y.\mathsf{Hired}$
$\mathsf{p}_5 : \mathsf{t}_x.\mathsf{Position} = \mathsf{t}_y.\mathsf{Position}$	$\mathsf{p}_{14} : \mathsf{t}_x.\mathsf{Hired} \neq \mathsf{t}_y.\mathsf{Hired}$
$\mathsf{p}_6 : \mathsf{t}_x.\mathsf{Position} \neq \mathsf{t}_y.\mathsf{Position}$	$\mathsf{p}_{15} : \mathsf{t}_x.\mathsf{Hired} < \mathsf{t}_y.\mathsf{Hired}$
$\mathsf{p}_7 : \mathsf{t}_x.\mathsf{Salary} = \mathsf{t}_y.\mathsf{Salary}$	$\mathsf{p}_{16} : \mathsf{t}_x.\mathsf{Hired} \leq \mathsf{t}_y.\mathsf{Hired}$
$\mathsf{p}_8 : \mathsf{t}_x.\mathsf{Salary} \neq \mathsf{t}_y.\mathsf{Salary}$	$\mathsf{p}_{17} : \mathsf{t}_x.\mathsf{Hired} > \mathsf{t}_y.\mathsf{Hired}$
$\mathsf{p}_9 : \mathsf{t}_x.\mathsf{Salary} < \mathsf{t}_y.\mathsf{Salary}$	$\mathsf{p}_{18} : \mathsf{t}_x.\mathsf{Hired} \geq \mathsf{t}_y.\mathsf{Hired}$

Figure 2: Predicate space for the *employees* relation.

5.2 From tuples into PLIs

PLIs represent the unique values of a dataset [28]. Consider the attribute $\mathsf{A}_i \in \mathsf{R}$. A cluster is an entry $c = \langle k, l \rangle$, where key k is a value from the projection operation $\pi(\mathsf{A}_i)$ and value l is a list of tuple identifiers of the relation instance having the same value k, i.e., $\forall x \in l$ then $t_x[\mathsf{A}_i] = k$. The list l maintains its elements in ascending order. A PLI $\Pi(\mathsf{A}_i)$ is the set of all cluster entries of A_i in r. The numeric PLIs are sorted by the entry keys in descending order. Figure 3 shows the PLIs of the *employees* relation.

PLIs are commonly used in attribute dependency discovery [29], and are also known as *stripped partitions* [18]. In these works,

Name

k	l
W. Jones	{0, 4, 5}
B. Jones	{1}
J. Miller	{2}
D. Miller	{3}

Position

k	l
Developer	{0, 1, 2, 5}
DBA	{3, 4}

Salary

k	l
8,000	{3}
7,000	{4}
4,000	{2}
3,000	{1}
2,000	{0}
1,000	{5}

Phone

k	l
202-222	{0, 1, 5}
202-333	{2, 3}
202-555	{4}

Hired

k	l
2012	{0, 5}
2010	{1, 2, 3, 4}

Figure 3: Transformation of the records of *employees* into PLIs.

intersecting the values of PLIs helps to validate dependency candidates. In our context, PLIs are used as an intermediate data structure that helps generating evidence sets. With PLIs, we can efficiently answer the question: which tuple pairs satisfy a given predicate p? DCFINDER simply iterates over cluster combinations to generate these tuple pairs; the details are given in Section 6.

Building PLIs takes linear time as it requires only projection operations to collect the distinct attribute values and their associated tuple identifiers. PLIs are used to look clusters up. Non-numeric clusters are stored in hash tables so looking them up takes constant time. Numeric clusters are stored as sorted arrays so that it is possible to look keys up using binary search. The binary search is required for looking up inequalities. For instance, given a key k, we can ask what is the next cluster whose key is greater than k.

6. EVIDENCE SET GENERATION

One may think that storing evidence sets requires significant resources, because they represent all tuple pairs. However, different tuple pairs may draw redundant evidence, i.e., they may satisfy the very same set of predicates. As a matter of fact, the number of distinct pieces of evidence was just a fraction of the total number of tuple pairs of the datasets in our experiments. As a result, keeping only the distinct evidence saves a huge amount of space. But the computational costs of materializing tuple pair evidences may still be high. To significantly reduce also these costs, DCFINDER uses attribute indexing and predicate selectivity with a novel approach.

Let us first assume that the pieces of evidence of r are stored into a virtual array B. Each tuple pair is assigned an identifier tpid to index B as in Equation 1.

$$\mathsf{tpid}(\mathsf{t}_x, \mathsf{t}_y, \mathsf{r}) = |\mathsf{r}|\, x + y \tag{1}$$

We use $B[\mathsf{tpid}]$ to refer to the tuple pair evidence at position tpid of B, and $\mathbf{T}$ to denote tpid sets. For now, assume that a single array B can store all the evidence of the relation instance.

Our goal is to put B into a consistent state. Every element $B[\mathsf{tpid}]$ must hold only the predicates satisfied by tpid. The naive approach would fill each evidence of B by evaluating every tuple pair for every predicate. This approach performs poorly due to the high number of tuple pair accesses and predicate evaluations. DCFINDER avoids directly comparing every tuple pair by benefiting from two main insights: First, some predicates may have low selectivity, and if so, are satisfied by many tuple pairs. Second, we can efficiently build attribute value associations between tuple pairs and their satisfied predicates using PLIs. DCFINDER is designed based on these two insights to minimize the number of operations

within the evidence array B. This drastically reduces the performance penalties from the quadratic tuple pair space, thus helping the efficiency of DCFINDER.

DCFINDER builds evidence sets, in the three stages: Evidence initialization, reconstruction, and counting.

6.1 Evidence initialization

DCFINDER initializes an array B so that many of the elements of B are close to their consistent state. Consider an evidence e to be stored in $B[\mathsf{tpid}]$. The probability of a predicate p to occur in e is simply the probability of tpid to satisfy p, i.e., the selectivity of predicate p. Tuple pairs are more likely to satisfy the least selective predicates. Under this assumption, DCFINDER fills in a general evidence $\mathsf{e}_{\mathsf{ahead}}$ with some of the least selective predicates, and then instantiates every element of B as a copy of $\mathsf{e}_{\mathsf{ahead}}$. The chances are high that many elements of B are already consistent for some $\mathsf{e}_{\mathsf{ahead}}$ predicates. For instance, all the tuple pairs of the employees relation satisfy the predicate $\mathsf{t}_x.\mathsf{Salary} \neq \mathsf{t}_y.\mathsf{Salary}$.

Figure 4 shows the implication of each operator $\mathsf{o} \in \mathsf{O}$. A predicate $\mathsf{p}_1 : \mathsf{A}_i\ \mathsf{o}\ \mathsf{A}_j$ implies every predicate $\mathsf{p}_2 : \mathsf{A}_i\ \mathsf{o}'\ \mathsf{A}_j$, where $\mathsf{o}' \in \mathsf{o}_{\Rightarrow}$. If predicate p_1 is true, then every implication of p_2 is true. Therefore, DCFINDER also includes p implications $\mathsf{p}_{\Rightarrow}$ into $\mathsf{e}_{\mathsf{ahead}}$, for every p it has included into $\mathsf{e}_{\mathsf{ahead}}$. Figure 4 also shows the logical complement $\bar{\mathsf{o}}$ of each operator. The complement of a predicate $\mathsf{p} : \mathsf{t}_x.\mathsf{A}_i\ \mathsf{o}\ \mathsf{t}_y.\mathsf{A}_j$ is the predicate $\bar{\mathsf{p}} : \mathsf{t}_x.\mathsf{A}_i\ \bar{\mathsf{o}}\ \mathsf{t}_y.\mathsf{A}_j$, where $\bar{\mathsf{o}}$ is the logical complement (or negation) of operator o. If predicate p is true, then $\bar{\mathsf{p}}$ is false.

Operator (o)	$=$	$\neq$	$<$	$\leq$	$>$	$\geq$
Implication ($\mathsf{o}_{\Rightarrow}$)	$=, \leq, \geq$	$\neq$	$<, \leq, \neq$	$\leq$	$>, \geq, \neq$	$\geq$
Negation ($\bar{\mathsf{o}}$)	$\neq$	$=$	$\geq$	$>$	$\leq$	$<$

Figure 4: Implications and complement of operators.

The selectivities of both $<, \leq$ and $>, \geq$ predicates are equivalent. For each tuple pair $\mathsf{t}_x, \mathsf{t}_y$ that satisfies the predicates $\mathsf{p}_1 : \mathsf{t}_x.\mathsf{A}_i < \mathsf{t}_y.\mathsf{A}_j$ (and its implied predicates $\mathsf{p}_{1\Rightarrow}$), there is the tuple pair $\mathsf{t}_y, \mathsf{t}_x$ that satisfies the predicates $\mathsf{p}_2 : \mathsf{t}_x.\mathsf{A}_i > \mathsf{t}_y.\mathsf{A}_j$ (and its implied predicates $\mathsf{p}_{2\Rightarrow}$). The selectivity of a predicate $\bar{\mathsf{p}}$ is given simply by subtracting the selectivity of p from the total number of tuple pairs. Out of the 30 tuple pairs in the *employees* instance, only 6 tuple pairs satisfy the predicate $\mathsf{t}_x.\mathsf{Name} = \mathsf{t}_y.\mathsf{Name}$, but $30 - 6 = 24$ tuple pairs satisfy the predicate $\mathsf{t}_x.\mathsf{Name} \neq \mathsf{t}_y.\mathsf{Name}$.

Let us assume uniform distribution of attribute values and high attribute cardinality (i.e., number of distinct values). The predicates with operators $(\neq, <, \leq, >, \geq)$ have low selectivity compared to equality predicates $(=)$. Framing $\mathsf{e}_{\mathsf{ahead}}$ to hold inequality predicates $(\neq)$ minimizes the number of inconsistent evidence, and therefore, the number of evidence reconstruction required. We can choose whether $\mathsf{e}_{\mathsf{ahead}}$ should hold $<, \leq$ or $>, \geq$ predicates without increasing the number of reconstructions. The evidence $\mathsf{e}_{\mathsf{ahead}}$, however, should not include both $<, \leq$ and $>, \geq$ predicates because that would only increase the number of inconsistent evidence. If $\mathsf{e}_{\mathsf{ahead}}$ holds $<, \leq$ predicates, then array B must be reconstructed for the correspondent $>, \geq$ predicates, or vice versa. Reconstructing single evidence requires accessing the array of evidence B and performing simple set operations. Because array B reflects the quadratic tuple pair space, minimizing the number of evidence reconstruction considerably reduces the overall runtime.

Evidence ahead initialization. For ease of exposition, let $\mathsf{e}_{\mathsf{ahead}}$ denote a general evidence that includes every predicate $\mathsf{p} \in \mathsf{P}$ such that $\mathsf{p.o} \in \{\neq, <, \leq\}$. DCFINDER initializes an evidence array B of size $|\mathsf{r}| \cdot |\mathsf{r}|$, and instantiate every element of B as a copy of $\mathsf{e}_{\mathsf{ahead}}$. We next describe how the algorithm reconstructs the array B for predicates with operators $\{=, >, \geq\}$, so that B represents a consistent state with regard to the predicate space and dataset tuple pairs. These procedures can be straightforwardly adjusted to use other settings of the evidence $\mathsf{e}_{\mathsf{ahead}}$.

6.2 Evidence reconstruction

DCFINDER uses PLIs to find the inconsistent tpid's of B, and then iterates over those elements to perform evidence reconstructions. We can find inconsistent tpid's from combinations of ordered pairs (l_1, l_2). The procedures to define and combine pairs of tuple identifiers (l_1, l_2) are based on the types of each predicate.

Consider the case for predicates of the form $\mathsf{p}\colon \mathsf{t}_x.\mathsf{A}_i = \mathsf{t}_y.\mathsf{A}_i$. Recall that PLIs are sets of clusters $c = \langle k, l \rangle$, and each cluster c keeps track of all tuples identifiers l with the same value k. From each cluster $c = \langle k, l \rangle \in \Pi(\mathsf{A}_i)$, DCFINDER builds ordered pairs (l_1, l_2), where $l_1 = l$ and $l_2 = l$. The tuple pairs with $t_x \in l_1$, $t_y \in l_2$, and $t_x \neq t_y$ are precisely those tuple pairs that satisfy the equality predicate p. Each of these tuple pairs is assigned a tpid (Equation 1), which is stored in an ordered set $\mathbf{T}$. Consider the cluster $\langle \text{DBA}, \{3, 4\} \rangle$ of $\Pi(\mathsf{Position})$ for instance. It gives us the ordered pair $(\{3, 4\}, \{3, 4\})$, and therefore, tuple pairs $\mathsf{t}_3, \mathsf{t}_4$ and $\mathsf{t}_4, \mathsf{t}_3$. These are exactly some of the tuple pairs that satisfy the predicate $\mathsf{p}_5 : \mathsf{t}_x.\mathsf{Position} = \mathsf{t}_y.\mathsf{Position}$. From Equation 1, and tuple pairs $\mathsf{t}_3, \mathsf{t}_4$ and $\mathsf{t}_4, \mathsf{t}_3$, we get tpid's 22 and 27. These tpid's point to evidence in the array B that are incorrectly holding the predicate $\mathsf{p}_6 : \mathsf{t}_x.\mathsf{Position} \neq \mathsf{t}_y.\mathsf{Position}$, so we must reconstruct these pieces of evidence to hold p_5 instead. Following the above procedures for every cluster of $\Pi(\mathsf{Position})$ gives us every piece of evidence we must reconstruct for predicate p_5.

Finding tuple pairs that satisfy other types of predicates follows a similar principle, but with a slight change on how ordered pairs (l_1, l_2) are arranged. The procedure for predicates on different attributes, $\mathsf{p}\colon \mathsf{t}_x.\mathsf{A}_i = \mathsf{t}_y.\mathsf{A}_j$ where $i \neq j$, is as follows: For each cluster $c = \langle k, l \rangle \in \Pi(\mathsf{A}_i)$, DCFINDER probes $\Pi(\mathsf{A}_j)$ for a cluster $c' = \langle k', l' \rangle \in \Pi(\mathsf{A}_j)$ such that $k = k'$. If there is a match, DCFINDER builds an ordered pair (l_1, l_2), where $l_1 = l$ and $l_2 = l'$. Building the tuple pair representation from (l_1, l_2) follows the same principle described before. Finally, the procedure for greater-than predicates with the form $\mathsf{t}_x.\mathsf{A}_i > \mathsf{t}_y.\mathsf{A}_j$ is as follows. For each cluster $c = \langle k, l \rangle \in \Pi(\mathsf{A}_i)$, DCFINDER looks up every cluster $c' = \langle k', l' \rangle \in \Pi(\mathsf{A}_j)$ such that $k > k'$. For each match, a new ordered pair (l_1, l_2) is built. DCFINDER transforms these tuple pair representations into the tpid's, just as described before. The algorithm keeps a map $\mathcal{T}$ of associations between a predicate p and the ordered set of tuple pair identifiers that satisfy p.

Algorithm 1 shows the steps to find all the tuple pair identifiers that point to inconsistent evidence in array B, given a predicate space and relation instance. DCFINDER calculates tuple pair identifiers only for $\{=, >\}$ predicates. By minding the implication property, the algorithm reconstructs B for $\{\geq\}$ predicates as well.

Algorithm 2 shows how DCFINDER materializes and reconstructs tuple pairs evidence. Evidence array B is initialized with copies of $\mathsf{e}_{\mathsf{ahead}}$. For each pair $\langle \mathsf{p}, \mathbf{T} \rangle$ in the mapping $\mathcal{T}$, DCFINDER performs a sequence of reconstructions. Given a tpid set $\mathbf{T}$, the algorithm updates $B[\mathsf{tpid}]$ for each $\mathsf{tpid} \in \mathbf{T}$. The operations slightly differ from each other depending on the type of the predicate.

For now, let p be a *non-numeric* equality $(=)$ predicate, and $B[\mathsf{tpid}]$ an evidence we need to reconstruct for p. At this stage, $B[\mathsf{tpid}]$ holds the inequality complement $(\neq)$ $\bar{\mathsf{p}}$ of p. But we want $B[\mathsf{tpid}]$ to hold p, not $\bar{\mathsf{p}}$. Let fix denote a predicate set that includes

Algorithm 1: Find the identifiers of inconsistent tuple pairs

Data: Relation instance r, and predicate space P
Result: A mapping $\mathcal{T}$ from predicates to tuple pair identifiers

for $A_i \in R$ **do**
  build PLI $\Pi(A_i)$
  if A_i *is numeric* **then**
    sort $\Pi(A_i)$ in descending order of keys k
$\mathcal{T} \leftarrow \emptyset$
foreach $p \in P$ *where* p.o $\in \{=, >\}$ **do**
  Use PLIs to compute $\mathbf{T}$ of p
  $\mathcal{T}\{p\} \leftarrow \mathbf{T}$
return $\mathcal{T}$

Algorithm 2: Materialization and reconstruction of evidence

Data: Mapping $\mathcal{T}$, relation instance r, and predicate space P
Result: Evidence array B

$e_{ahead} \leftarrow$ every $p \in P$ where p.o $\in \{\neq, <, \leq\}$
initialize array B, each element is a copy of e_{ahead}
foreach $p \in P$ *where* p.o $\in \{=, >\}$ **do**
  fix $\leftarrow$ build predicate mask of p
  foreach tpid $\in \mathcal{T}\{p\}$ **do**
    B[tpid] $\leftarrow B$[tpid] $\oplus$ fix
return B

both p and $\bar{p}$, that is, fix $\leftarrow \{p, \bar{p}\}$. The symmetric difference[1] between B[tpid] and fix, denoted as B[tpid] $\leftarrow B$[tpid] $\oplus$ fix, gives us a consistent B[tpid] with regard to p. If p is a *numeric* equality ($=$) predicate, fix must also include the correspondent $<, \geq$ predicates of p. Once the symmetric difference has been applied, B[tpid] satisfies p and its correspondents $\leq, \geq$. That fulfills the implication requirement for p.

Finally, let p be a greater than ($>$) predicate, and an evidence B[tpid] be inconsistent for p. B[tpid] holds the correspondent $\{\neq, <, \leq\}$ predicates of p, but should hold $\{\neq, >, \geq\}$ predicates, instead. To reconstruct B[tpid], we need to set fix to hold $\{<, \leq, >, \geq\}$ and calculate the symmetric difference B[tpid] $\leftarrow B$[tpid] $\oplus$ fix. This operation removes the correspondent $\{<, \leq\}$ predicates of p, but includes the correspondent $\{>, \geq\}$ ones. Figure 5 illustrates part of the reconstruction for the evidence of *employees* with regard to the inequalities predicates on attribute Hired. The cluster $\langle 2012, \{0, 5\}\rangle$ pairs with cluster $\langle 2010, \{1, 2, 3, 4\}\rangle$ to form tpid*s* $1, 2, 3, 4, 31, 32, 33, 34$. These elements initially hold p_{15} and p_{16}, but are reconstructed to correctly hold p_{17} and p_{18}.

6.3 How to scale up to large datasets

Storing arrays of evidence B incurs a quadratic space overhead in the number of tuples because each array B represents evidence of all tuple pairs. Also of quadratic space are the sets of tuple pair identifier $\mathbf{T}$ used to reconstruct B because they grow as a function of the number of tuple pairs. Storing all the data of B and $\mathbf{T}$ at once may be infeasible as it can sooner or later exhaust any memory limit. It turns out that slightly modifying how these structures are built enables DCFINDER to scale up for larger datasets. DCFINDER uses a multi-level partitioning scheme based on the range of tuple pair identifiers. The idea is to create a partial evidence set for each

[1]The symmetric difference is implemented as a simple exclusive or operation (XOR).

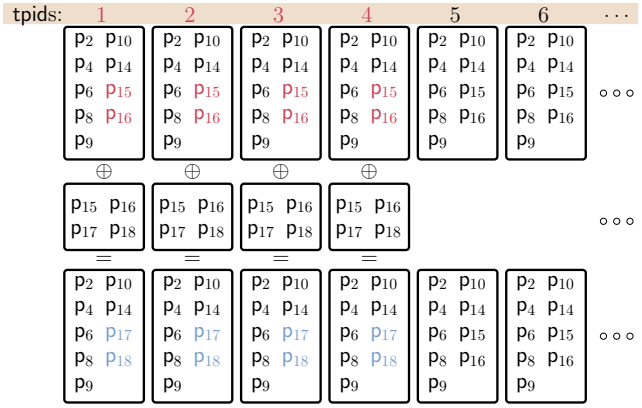


Figure 5: Part of the reconstruction for the evidence of *employees* and predicates p_{17} : t_x.Hired $>$ t_y.Hired and p_{18} : t_x.Hired $\geq$ t_y.Hired.

range, and then merge these sets into the final and correct evidence set. The scheme allows DCFINDER to: (i) handle larger relation instances, and (ii) use multiple parallel threads.

Figure 6 illustrates the partitioning scheme. Instead of materializing whole sets of tuple pair identifiers $\mathbf{T}$, DCFINDER processes only fractions of $\mathbf{T}$ at a time. Virtual sets of tuple pair identifiers $\mathbf{T}$ are partitioned into chunks $\mathbf{T} = \{T_0, T_1, \ldots, T_s, \ldots\}$, $s \in \mathbb{N}$. Partitioning is based on the disjoint ranges of tpid values. Assuming a maximum chunk length ω, chunk T_0 can store any tpid $\in [0,\omega)$, tpid $\in \mathbb{N}$. Chunk T_s can store any tpid $\in$ [low,high), where low $= s \cdot \omega$, and high $= (s+1) \cdot \omega$. In a similar fashion, DCFINDER processes all the evidence of B using small evidence fragments. Each fragment stores at most λ evidence elements. This two-tier partitioning scheme benefits from data locality, as we show in our experimental evaluation.

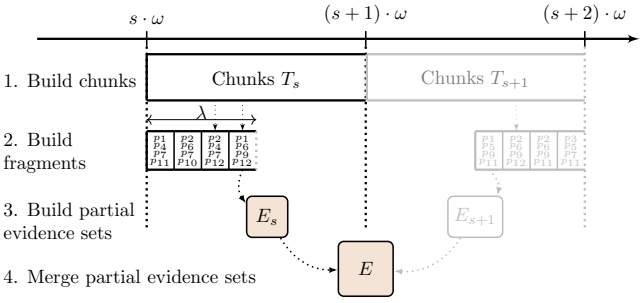


Figure 6: Evidence set building: Partitioning of tuple pair identifiers into chunks, and splitting of tuple pair evidence into evidence fragments.

Let us consider the s-th run. We build tuple pair identifier sets T_s for every predicate required to materialize the evidence set. We want each chunk T_s to hold every tpid associated to $\mathbf{T}$ such that low $\leq$ tpid $<$ high. Recall that tuple pair identifiers tpids are drawn from ordered pairs $\langle l, l'\rangle$. DCFINDER shrinks pairs $\langle l, l'\rangle$ so they yield tpids within the range of chunk T_s. From Equation 1, we see that any tuple pair t_x, t_y such that $t_x \in l$, and $t_x >$ high$/|r|$ yields a tpid that is greater or equal to high, and therefore t_x, t_y falls outside the range of T_s. Depending on the size of chunks and relation instances, other t_x, t_y settings may also yield tpids outside the range of T_s. DCFINDER removes such tuple settings from ordered pairs $\langle l, l'\rangle$. Any tpid from $\langle l, l'\rangle$ is guaranteed to

fall within the range of T_s after $\langle l, l' \rangle$ has been shrunk. DCFINDER proceeds to reconstruct evidence after all chunks T_s are created.

DCFINDER follows Algorithm 2, but reconstructs small evidence fragments instead of the potentially huge evidence array B. The algorithm initializes a fragment using $\mathsf{e}_{\mathsf{ahead}}$. Then it iteratively consumes tpids from chunks to perform the reconstructions. It stops consuming tpids if a tpid is no longer within the fragment range. The current fragment is consistent after all chunks within the same range have been processed. DCFINDER then iterates over the evidence of the current fragment to retain two information: (i) distinct evidence, and (ii) evidence multiplicity. Evidence of reflexive tuple pairs, i.e., $\{\mathsf{t}_x, \mathsf{t}_x\}$, are skipped. The evidence set produced at that point is partial, because it regards only tuple pairs within a given range. DCFINDER requires an additional step to merge all partial evidence sets. As discussed before, the number of distinct evidence is very small compared to the number of total tuple pairs. Thus, merging partial evidence sets does not incur significant overhead.

The primary computational pattern for evidence reconstruction is the sequential read of chunks followed by symmetric difference computations. If the chunk is too small, the number of runs increases. On the other hand, if the chunk is too large, memory may end up exhausted. The symmetric difference operation is implemented as an XOR operation, which is usually optimized in modern CPU architectures. Because DCFINDER needs to perform many of these operations, improving data locality helps reducing cache miss penalty. We performed micro-benchmarks to verify the influence of chunk size ω and fragment size λ parameters in runtime. Our experiments (Section 8) show that using relatively small evidence fragments decreases cache misses, and thus improve runtime. We observed that settings where the fragment size is just a fraction of the chunk size yields better runtime than the settings where the size of chunks and fragments are the same.

Keeping a simple counter for each distinct evidence suffices, so we are able to accommodate the cover search (Section 7) to discover approximate DCs. The final evidence set E is a simple hash map with evidence as keys, and evidence frequency as values. We use counter to denote a function $\mathsf{E} \to \mathbb{N}$ such that $\mathsf{counter}(\mathsf{e})$ returns the frequency of evidence e.

DCFINDER can build partial evidence sets independently of each other, because chunks $\{T_0, T_1, \ldots, T_s, \ldots\}$ are disjoint. It picks up available threads from a thread pool to serve as workers. The only data shared across workers is the data from PLIs, and from the final evidence set. Multiple workers can safely read PLIs because they never change once built. Each worker operates on its own chunks and fragments to generate its partial evidence set. The concurrent access to the final evidence set is synchronized via latches. This last operation does not impose significant overhead: most time is spent finding the inconsistent tpids and fixing pieces of evidence. As we show in Section 8, the evidence set building phase of DCFINDER scales (almost) linearly in the number of CPU cores.

7. DC SEARCH

This section describes how DCFINDER uses the evidence set to discover minimal approximate (and exact) DCs. It also describes three measures to score the interestingness of the discovered DCs.

7.1 Minimal covers

A DC can be any subset of the predicate space P, so entirely traversing the search space with $2^{|\mathsf{P}|}$ candidates is infeasible. Discovering attribute dependencies is likely an intractable problem [4,16]. For example, the authors of [4] have recently shown that detecting functional dependencies is a $W[2]$-complete problem. The result directly impacts the computational hardness of DC discovery, because DCs subsume functional dependencies. Despite their computational complexity, data profiling algorithms have managed to perform quite well on various real-world datasets [21, 29, 34].

The problem of discovering all minimal DCs can be transformed into the problem of finding all *minimal covers* of the evidence set [8]. The latter problem is cognate with other problems, such as enumerating *hitting sets* or *hypergraph traversals* [22]. These problems have been studied under a variety of domains for their wide range of applications [2, 22]. We make use of the approach of [8], because it easily accommodates the search of approximate (partial) covers, and therefore, approximate DCs. The approach works well in practice, as discussed in Section 8.

An evidence $\mathsf{e} \in \mathsf{E_r}$ cannot hold predicates $\{\mathsf{p}_1, \ldots, \mathsf{p}_m\}$ and $\{\bar{\mathsf{p}}_1, \ldots, \bar{\mathsf{p}}_m\}$ simultaneously. If e holds $\{\mathsf{p}_1, \ldots, \mathsf{p}_m\}$, any DC φ containing at least one predicate of $\{\bar{\mathsf{p}}_1, \ldots, \bar{\mathsf{p}}_m\}$ could not be violated by the tuple pairs that yield evidence e. For φ to be exact, that intuition must apply for every evidence $\mathsf{e} \in \mathsf{E_r}$. That is why we find covers of the full evidence set $\mathsf{E_r}$. A cover Q_1 is a set of predicates that intersects with every evidence of $\mathsf{E_r}$, i.e., $\forall \mathsf{e} \in \mathsf{E_r}, Q_1 \cap \mathsf{e} \neq \emptyset$. The cover Q_1 is minimal if there does not exist a Q_2 that is a strict subset of Q_1 and intersects with the same elements of Q_1, i.e., $\nexists Q_2 \subset Q_1$ such that $\forall \mathsf{e} \in \mathsf{E_r}, Q_2 \cap \mathsf{e} \neq \emptyset$. The following theorem holds for discovering DCs (see [8] for proof).

THEOREM 1. *A DC $\varphi\colon \neg(\bar{\mathsf{p}}_1 \wedge \ldots \wedge \bar{\mathsf{p}}_m)$ holds in relational instance r if the set $Q\colon \{\mathsf{p}_1, \ldots, \mathsf{p}_m\}$ is a cover of the evidence set $\mathsf{E_r}$. The DC φ is minimal if Q is minimal.*

In addition, we must be able to discover approximate DCs. Recall that the degree of approximation ε of a DC φ is based on the number of tuple pairs that do not satisfy φ. The multiplicity of an evidence set is given by $\|\mathsf{E}\| = \sum_{\mathsf{e} \in \mathsf{E}} \mathsf{counter}(\mathsf{e})$, that is, how many tuple pairs yielded all evidence of E. The multiplicity $\|\mathsf{E}\|$ is equal to $|\mathsf{r}| \cdot (|\mathsf{r}| - 1)$ if $\mathsf{E} = \mathsf{E_r}$. Consider again the set $Q\colon \{\mathsf{p}_1, \ldots, \mathsf{p}_m\}$, but assume that E is only a subset of the full evidence set $\mathsf{E} \subseteq \mathsf{E_r}$ such that $\forall \mathsf{e} \in \mathsf{E}, Q \cap \mathsf{e} = \emptyset$. The set Q approximately covers the full evidence set $\mathsf{E_r}$ if $\|\mathsf{E}\| \leq \varepsilon \cdot |\mathsf{r}| \cdot (|\mathsf{r}| - 1)$. If so, the predicate set Q is an ε-approximate cover of $\mathsf{E_r}$, and it is minimal if there does not exist a strict subset of Q that is also an ε-approximate cover of $\mathsf{E_r}$.

Algorithm 3 presents the minimal cover search. It is a heuristic-based depth-first search for which nodes are recursively formed based on evidence set coverage. Each node maintains a path of the search tree $Q \subseteq \mathsf{P}$, the set of evidence not covered by the current path $\mathsf{E}_{path} \subseteq \mathsf{E}$, the set of predicates that can be included in further branches $\mathsf{P}_{path} \subseteq \mathsf{P}$, and all minimal covers MC found in prior branches. Every path is a cover candidate. At first $Q = \emptyset$, $\mathsf{E}_{path} = \mathsf{E_r}$, $\mathsf{P}_{path} = \mathsf{P}$, and $\mathsf{MC} = \emptyset$. To unfold a new branch, the algorithm adds a predicate p_{add} to the new path and updates the information for the child node. The child evidence set E_{new} is the result of removing all evidence that contain p_{add} from the parent evidence set E_{path}. The child predicate set P_{new} is every predicate $p \in \mathsf{P}_{path}$ such that $\mathsf{p} \not\sim \mathsf{p}_{add}$.

Two base cases stop the recursion. First, the algorithm finds an approximate cover if the path Q removes large pieces of evidence of E_r such that $\|\mathsf{E}_{path}\| \leq \varepsilon \cdot |\mathsf{r}| \cdot (|\mathsf{r}| - 1)$. Consequently, the corresponding DC of Q could be violated by no more than $\|\mathsf{E}_{path}\|$ tuple pairs. If $\mathsf{E}_{path} = \emptyset$, Q is an exact cover. To ensure minimality, the algorithm tests whether there exists an immediate subset of Q that also (approximately) covers E_r. If it does not find such a subset, the predicate set Q is added to the result MC as a minimal cover. Second, if the search reaches a node for which there are still

Algorithm 3: Find Minimal Covers [8]

Data: Evidence set $\mathsf{E_r}$, Predicate space P, Error threshold ε
Result: Set of minimal covers MC

$\mathsf{MC} \leftarrow \emptyset$
`findCover` $(\emptyset, \mathsf{E_r}, \mathsf{P}, \mathsf{MC})$

Function `findCover` $(Q, \mathsf{E}_{path}, \mathsf{P}_{path}, \mathsf{MC})$
  if $\|\mathsf{E}_{path}\| \leq \varepsilon \cdot |\mathsf{r}| \cdot (|\mathsf{r}| - 1)$ **then**
    if *no subset of size* $|Q| - 1$ *of* Q ε*-covers* $\mathsf{E_r}$ **then**
      $\mathsf{MC} \leftarrow \mathsf{MC} \cup Q$
    return
  else if $\mathsf{P}_{path} = \emptyset$ **then**
    return
  else
    sort P_{path} based on tuple pair coverage of E_{path}
    for $\mathsf{p}_{add} \in \mathsf{P}_{path}$ **do**
      $Q \leftarrow Q \cup p_{add}$
      if Q *is implied by* MC **then**
        $Q \leftarrow Q \setminus p_{add}$
        continue
      $\mathsf{E}_{new} \leftarrow \{\mathsf{e} \mid \mathsf{e} \in \mathsf{E}_{path}$ **and** $\mathsf{p}_{add} \notin \mathsf{E}_{path}\}$
      $\mathsf{P}_{new} \leftarrow \{\mathsf{p} \mid \mathsf{p} \in \mathsf{P}_{path}$ **and** $\mathsf{p} \not\sim \mathsf{p}_{add}\}$
      `findCover` $(Q, \mathsf{E}_{new}, \mathsf{P}_{new}, \mathsf{MC})$

enough evidence to cover, but there are no predicates to form new branches, then there is no valid cover in that branch.

The tuple pair coverage of a predicate p is the multiplicity of the evidence set in which all evidence contain p, that is, $\|\mathsf{E}\|$ such that $\mathsf{e} \in \mathsf{E}$ and $\mathsf{p} \in \mathsf{e}$. The heuristic to unfold new paths is to include predicates in dynamic ordering of tuple pair coverage. The search adds predicates satisfied by most tuple pairs first, i.e., those predicates that reduce the evidence set size the most. Removing predicates from E_{new} changes the tuple pair coverage distribution for the remaining candidate predicates P_{new}, so the algorithm needs to compute a new predicate ordering for each new branch. The sooner the evidence set becomes small enough, the sooner the algorithm finds minimal covers. The algorithm uses these covers MC to reduce the number of searches. Before updating the information for a new path (E_{new} and P_{new}), the algorithm checks if that path is already in the cover. If so, there is no need to unfold that branch.

Once Algorithm 3 is finished, each minimal cover in MC is translated into a minimal DC by inverting its predicates (Theorem 1). The output may contain implied DCs, so we need to test whether each DC is implied by the remaining discovered DCs. This implication testing is known to be a coNP-complete problem [3]. The authors of [8] introduced an inference system for DCs and describe an algorithm to test DC implication with it. We use this algorithm to remove implied DCs from the output of all DC algorithms. Although not complete, the implication testing algorithm is correct and helps to remove many implied DCs from the output, which helps with user verification. More details on the static analyses of DCs and other constraints can be found in [3, 12].

7.2 DC interestingness

DCFINDER discovers all minimal DCs in a dataset. But in all likelihood, not all of them are equally useful. DCFINDER optionally estimates three interestingness measures: succinctness, coverage, and degree of approximation. We use these measures to: (i) pruning DC candidates that fall beneath interestingness thresholds, and (ii) ranking DCs to help users selecting relevant ones.

Succinctness has been used to rank DCs in [8]. It is inversely proportional to the number of distinct symbols (attributes and operators) in the predicates of a DC: the fewer symbols a DC has, the more succinct it is. The measure is based on the minimum description length principle: data representations with fewer symbols are more succinct. DCFINDER can use succinctness to prune DCs during cover search. To do so, it simply counts how many symbols a candidate DC expresses before checking it. If the quantity is greater than a given threshold, there is no need to check further paths from that candidate DC – the succinctness can only decrease.

Coverage is described in [8] as the statistical significance of a DC based on the proportion of tuple pairs that satisfy a given set of predicates. It is given by a weighted sum of tuple pairs scores. Given a DC φ with $|\varphi|$ predicates, each tuple pair scores the DC φ based on how many predicates that tuple pair satisfies. The larger the amount of tuple pairs satisfying a number of predicates close to $|\varphi| - 1$, the higher the coverage of φ. There is no guarantee that coverage always decreases for a given path, so we used this measure only during post-processing to rank DCs according to their coverage scores. Estimating the coverage of a DC requires iterating over the evidence set and evidence frequency counters. Because many DCs have predicates in common to each other, this estimation can be performed in a depth-first tree traversal to save computation for DCs sharing a common prefix.

We can additionally use the degree of approximation, defined in Section 3, to measure the interestingness of approximate DCs. It follows from Definition 2 that the number of tuple pairs allowed to violate an approximate DC is always bounded by the error threshold. But the number of actual violations varies between the discovered DCs. The degree of approximation simply shows how many tuple pairs are inconsistent with regard to an approximate DC. After a minimal (approximate) cover is found, the degree of approximation is simply the multiplicity of the remainder evidence set.

8. EXPERIMENTAL EVALUATION

We present an experimental evaluation of DCFINDER. We used all DC algorithms known to date as baselines: FASTDC [8] and BFASTDC [31] for the discovery of approximate and exact DCs; and HYDRA [5] for the discovery of exact DCs.

8.1 Experimental setup

We used the code provided by the authors of [5] for HYDRA and FASTDC. The code of BFASTDC was provided by the authors of [31]. We implemented DCFINDER from scratch. All implementations were written in Java and run in main memory after dataset loading. We integrated all implementations with the data profiling framework Metanome [27] to guarantee a unified testing environment. To keep consistent comparisons, we set all algorithms to replace NULL values with default values (i.e., empty strings for non-numeric attributes, or $-\infty$ for numeric attributes). This approach has been used also in the implementations of [5].

The strategies that FASTDC, BFASTDC and DCFINDER use to build evidence sets are designed to run over multiple threads. Therefore, unless stated otherwise, the reports for these algorithms are from multi-thread executions. The authors of [5] do not present a parallel version of HYDRA, so we use the implementation of the algorithm just as it is described in the paper. In addition, we implemented a new version of HYDRA, namely HYDRA+, so the algorithm can benefit from parallel execution in its systematic tuple pair sampling phase. This parallel step is implemented in similar fashion to the grid scheme used in FASTDC.

The experiments were run on an Intel Core i7-7700HQ machine (2.8 GHz, 4 physical cores/8 logical cores, 32 KB for L1, 256 KB

for L2, and 6 MB for shared L3); 16 GB RAM; 256GB SSD; Ubuntu 16.04; and Java 1.8 with the JVM heap space limited to 8 GB. The runtime reports are the average measurement of three independent runs.

Table 2 shows the main characteristics of the datasets used in our experiments. The majority of these datasets have been used in related work. The *Hospital* and *Tax* datasets have been used to evaluate DC discovery algorithms in [5, 8]. The *Adult*, *Flight*, and *NCVoter* datasets have been used to evaluate FD discovery algorithms in [21]. The *Inspection* dataset has been used to evaluate data cleaning systems in [33]. We additionally used the *Airport* dataset, which contains a list of airport codes and locations. Our repeatability page provides the implementation of our algorithm and pointers to all datasets[2].

Table 2: Datasets used to evaluate the DC discovery algorithms.

Name	Type	#tuples	#attributes	#predicates
Adult	real-world	32,561	15	54
Airport	real-world	55,113	18	48
Flight	real-world	500,000	20	88
Hospital	real-world	114,919	15	44
Inspection	real-world	170,000	19	74
NCVoter	real-world	938,085	22	60
Tax	synthetic	100,000	15	58

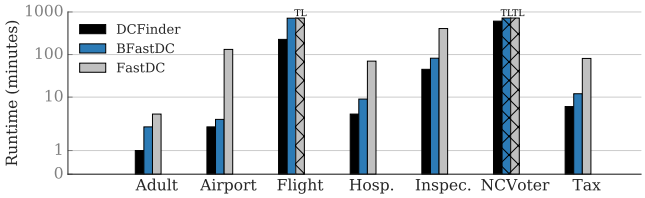
8.2 Discover of approximate DCs

We ran DCFINDER, FASTDC, and BFASTDC for all datasets shown in Table 2. We used degrees of approximation $\varepsilon = 0.01$ and $\varepsilon = 0.05$; these values have been previously used to evaluate the discovery of approximate dependencies [21]. We set the chunk and fragment lengths of DCFINDER to 5×10^6 and 5×10^3, respectively. We evaluate varying chunk and fragment lengths in Section 8.6, and varying degrees of approximation in Section 8.7.

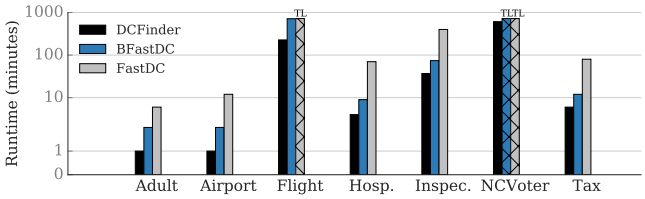
The results in Figure 7 show that DCFINDER is the fastest algorithm among the competitors. For *Tax* and *Hospital*, DCFINDER is at least 2× as fast as BFASTDC, and at least 13× times faster than FASTDC. The performance gains of our algorithm is higher for larger datasets. Using a degree of approximation $\varepsilon = 0.01$, for instance, DCFINDER took approximately 228 minutes to process *Flight*, BFASTDC took nearly 715 minutes, but FASTDC could not finish within the time limit of 12 hours. DCFINDER was the only algorithm able to process *NCVoter* within the time limit. The three algorithms use the same minimal cover search strategy; thus, the difference in their performance is a reflection of how efficiently they build evidence sets. Here, a good efficiency indicator is tuple pair throughput; i.e., how many tuple pairs an algorithm processes in a fixed amount of time. DCFINDER achieved better throughput than the competitors, especially for large datasets. This shows that, in terms of performance, DCFINDER improves the state of the art for the discovery of approximate DCs.

The algorithms discovered the largest sets of DCs in *Inspection* and *Adult*, respectively. Interestingly, the evidence sets for these two datasets were also the largest among all. With bigger evidence sets, the algorithms iterate over more evidence in each path of the cover search, which hinders runtime. For *Adult* and *Inspection*, a major part of the runtime was spent searching for minimal covers. The cover search for *Flight*, for example, was much faster than the cover search for *Inspection*. The *Flight* dataset has a bigger predicate space, but draws an evidence set that is only a fraction (nearly a thirtieth) of the evidence set drawn from *Inspection*.

[2] http://hpi.de/naumann/projects/repeatability/data-profiling/metanome-dc-algorithms.html



(a) Degree of approximation $g_1 = 0.01$.



(b) Degree of approximation $g_1 = 0.05$.

Figure 7: Runtime of approximate DC discovery. The crossed bars indicate that an algorithm did not terminate within the time limit (TL) of 12 hours. The Y-axes are in log-scale.

8.3 Discover of exact DCs

The next experiment focuses on the discovery of exact DCs; therefore, our comparisons additionally include the specialized algorithms, HYDRA and HYDRA+.

From Figure 8 we see that DCFINDER is faster than FASTDC and BFASTDC in every scenario. The algorithm even outperforms HYDRA and HYDRA+ in four out of seven datasets. For instance, DCFINDER was approximately 4.5× faster than HYDRA in *Airport*. But the sampling approach helped HYDRA to process some datasets faster than DCFINDER: For instance, HYDRA processed *NCVoter* approximately 3.5× faster than DCFINDER did.

DCFINDER materializes every tuple pair evidence to output evidence multiplicity, whereas HYDRA processes a fraction of tuple pairs to find only the distinct evidence. In a more detailed investigation, we found that HYDRA processed less than 0.1% of the total tuple pairs of each dataset. That is why HYDRA cannot produce the evidence multiplicity of the full dataset, which is required for discovering approximate covers, or calculating DC coverage. HYDRA spent a significant amount of time correcting tuple pair samples to complete the evidence set – similar observations were made in the experimental evaluation of HYDRA. The correction was particularly efficient for datasets that draw a small evidence set, e.g., *Hospital*. However, it performed poorly for datasets with large evidence sets. HYDRA+ improved the sampling phase of HYDRA, but had minor influence on the overall runtime.

HYDRA iterates over each evidence to dynamically update the set of candidate DCs, so they no longer violate such evidence. The depth-first search of FASTDC, BFASTDC and DCFINDER starts from DC candidates, and then updates the evidence set. Such a strategy is also penalized by large evidence sets; however, it uses the minimal covers to prune the search space as soon as they are discovered. For *Adult* and *Inspection*, the depth-first search was faster than the equivalent strategy of HYDRA. For the remaining datasets, all algorithms took less than two minutes to complete the search. This indicates that, in many cases, being able to build the evidence set in an efficient manner is crucial for the performance of the evaluated DC discovery algorithms.

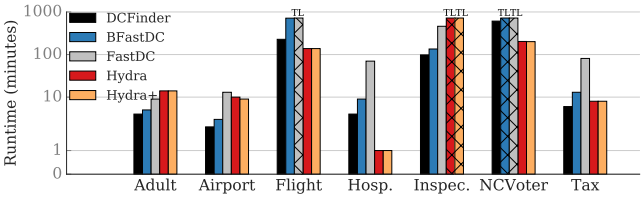


Figure 8: Runtime of exact DC discovery. The crossed bars indicate that an algorithm did not terminate within the time limit (TL) of 12 hours. The Y-axis is in log-scale.

8.4 Scalability

To evaluate the scalability in the number of tuples, we started at the beginning of a dataset and incrementally added more tuples to each execution. Figure 9 depicts the scaling behavior for *Tax* and *Flight* datasets. All algorithms are sensitive to the number of tuples. DCFINDER, however, seems to suffer less than FASTDC and BFASTDC. The algorithm has an advantage over FASTDC because it avoids the tuple pair comparison overhead. The evidence set building strategy of DCFINDER is faster than the one of BFASTDC for two reasons. First, it does not need to calculate tpids for the inverse and implied predicates, as BFASTDC does. Second, it reduces the number of accesses to the evidence elements due to the ahead evidence allocation. For small numbers of tuples, DCFINDER may be faster than HYDRA (e.g., as in *Tax* dataset). As the number of tuples increases, Hydra starts benefiting from tuple pair sampling (e.g., when we consider more than two hundred thousand tuples for *Flight* dataset). There is an important trade-off from this improvement though: HYDRA could not be tested if we had set the degree of approximation to a value other than $\varepsilon = 0.0$. DCFINDER, on the other hand, materializes all pieces of evidence to calculate the evidence counters. That is necessary not only for discovering approximate DCs, but measuring the interestingness of the results based on coverage and degree of approximation.

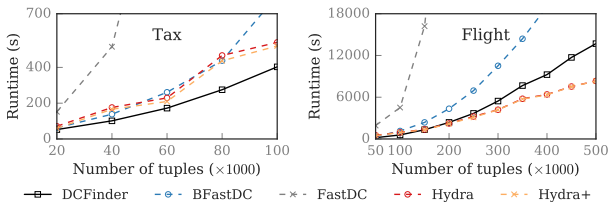


Figure 9: Runtime scalability in the number of rows.

To check scalability in the number of attributes, we began with the five initial attributes in the dataset schema. Then we incrementally added more attributes, using schema order, until every attribute of the dataset had been added. Figure 10 depicts the scaling behavior we obtained for *Tax* and *Flight* datasets. We used only the first 20,000 tuples of each dataset to avoid expensive computations in the number of tuples. The runtime of all algorithms increases exponentially in the number of attributes: as the predicate space grows, so does the number of DC candidates and the evidence set. Since DCFINDER, FASTDC and BFASTDC share the same cover search, the difference in their scalability is from how efficiently they build evidence sets for bigger predicate spaces. Out of these three algorithms, DCFINDER shows a slightly smoother scalability. On the other hand, FASTDC seems to have the worst performance degradation. The results in Figure 10 show that the performance of HYDRA is abruptly penalized when more attributes are added to its executions.

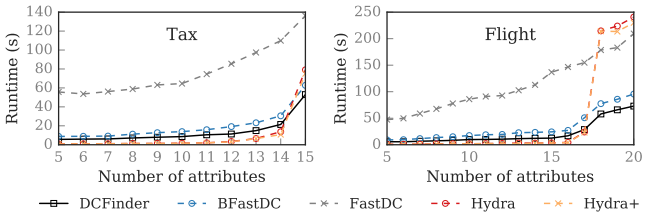


Figure 10: Runtime scalability in the number of attributes.

We evaluate predicate scalability using the first 20,000 tuples of *Adult*. The experiment chose different combinations of attributes at random. The goal is to check, for different combinations of predicates, how long DC discovery takes and how many DCs are discovered. We executed the experiment twenty times and report the average values in Figure 11. As expected, the predicate scaling of all algorithms behaves in a similar way to their attribute scaling. Just as there is exponential growth in runtime, there is exponential growth in the number of DCs.

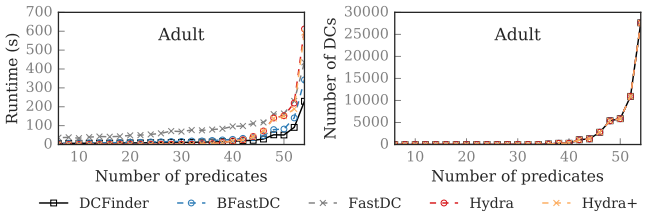


Figure 11: Runtime scalability in the number of predicates.

8.5 Memory consumption

The next experiment measures how much memory is required by the different DC discovery algorithms. For the largest datasets, *Flight* and *NCVoter*, we executed each algorithm using a maximum heap size of 64MB. Then, we repeatedly doubled this value until the respective algorithm was able to actually process that dataset (up to the time limit for slower algorithms). All algorithms had similar memory footprints. To process *Flight*, BFASTDC required 2048MB, whereas the other algorithms required 1024MB. All algorithms required 4GB to process *NCVoter*.

The main reason for this high demand is that our implementations load the full dataset into main memory to provide a fair comparison of the in-memory processing of the algorithms. This full loading incurs the overhead of encoding many attribute values as string objects. The main data structures used by DCFINDER are PLIs, chunks of tuple pair identifiers, and evidence fragments. PLIs are integer-based compact representations of datasets, and their sizes grow as a function of the number of distinct attribute values. Chunks and fragments have constant size defined by the parameters ω and λ, respectively. While these structures can be set to be as high as the available memory allows, we performed micro-benchmarking and found DCFINDER to perform better with relatively small values of ω and λ (as discussed in the next section).

8.6 DCFinder in-depth experiments

Figure 12 illustrates the runtime breakdown on each phase of DCFINDER. A large part of the runtime is shared between finding tpids and correcting evidence, which is expected as these phases

are the core of producing accountable evidence sets. Initializing and accumulating evidence also takes a considerable amount of the runtime: This is a reflection of the quadratic complexity that the problem has in the number of tuples. For *Adult* and *Inspection*, DCFINDER spent a major part of the runtime in cover search, as explained in Section 8.2. The overhead from the remaining phases is relatively small compared to the overall runtime.

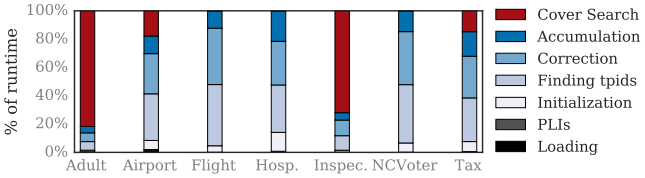


Figure 12: Runtime breakdown of DCFINDER ($\varepsilon = 0.01$): relative time the algorithm spent on loading datasets, building PLIs, initializing evidence, calculating tpids, correcting evidence, accumulating (hashing) evidence, and searching for minimal covers.

The next experiment focuses on the evidence set building phase of DCFINDER (Section 6) to highlight the scalability of DCFINDER in the number of threads. Such scaling is possible because the algorithm splits the tuple pair space into chunks, which can be processed independently of each other. The measurements are over the first $100,000$ tuples of each dataset, or over the total number of tuples for *Adult* and *Airport*. Figure 13 shows the scalability of DCFINDER in the number of threads. The algorithm scales (almost) linearly up to the number of physical cores (4); from there, it scales narrowly up to the number of logical cores (8). That behavior is expected as the cache resources are shared among the hyper-threads. Increasing the number of threads for more than the available logical cores does not improve runtime. Doing so is likely to increase the complexity of coordinating competing accesses to data, which may even hinder performance.

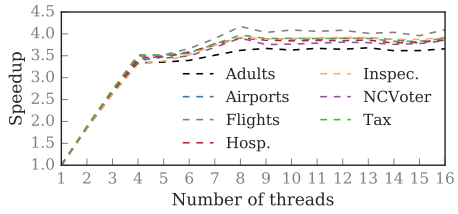


Figure 13: Relative runtime speedup in the number of threads (evidence set building only).

How DCFINDER splits the tuple pair space influences its efficiency. Figure 14 compares the behavior of the algorithm for varying sizes of chunks and fragments. We use *Tax* dataset to show this behavior, but the same trend was observed across all the evaluated datasets. The metrics of interest are runtime and cache misses (both L1 and LLC): the arrows in Figure 14 indicate the lowest measurements. The smaller the chunks, the more often DCFINDER iterates over PLIs to generate tpids, and the lower the tuple pair throughput (i.e., how many tuple pairs the algorithm processes in a fixed amount of time). The left plot in Figure 14 shows that DCFINDER runs faster as we increase chunk lengths, up until it nearly stabilizes its performance. From there, the fragment lengths at the edge (i.e., 10^2 and 10^5) negatively influenced runtime. This shows that DCFINDER is robust to the two parameters, for sizable ranges. For all datasets, DCFINDER was stable with chunk lengths around $10^6 \leq \omega \leq 10^7$ and fragments lengths at the few thousands region. After runtime inflection, the algorithm obtained no performance improvement, but increased its memory requirement.

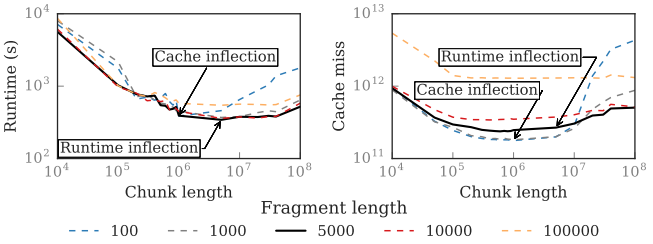


Figure 14: Influence of chunk and fragment length on DCFINDER runtime and cache misses. The axes are in log scale.

We observed that the cache miss ratio of the settings for which DCFINDER had the best runtime was at the same level of the best cache miss ratio we measured. Recall that DCFINDER operates on two pieces of data, tpids and evidence fragments, and that it implements the correction operation as an XOR, which is directly supported by the CPU. The runtime inflection reflects a sweet-spot where DCFINDER benefits from cache locality and achieves high tuple pair throughput without exhausting main memory. We observed very small variations in the runtime inflections of the evaluated datasets. In our experiments, setting chunk length to 5×10^6 and fragment length 5×10^3 worked very well across the evaluated datasets. BFASTDC also required us to set these two parameters, so we also tried different values to tune its execution. We observed that BFASTDC works best with chunks that are slightly smaller than the chunks of DCFINDER, because BFASTDC stores tpids of all predicates of the predicate space in memory.

8.7 DC interestingness

The following experiment shows how different degrees of approximation impact DC discovery. The approximation parameter has no influence on the evidence set building phase (for all algorithms), so we analyze only the minimal cover search behavior. We gradually increased the parameter for different executions of DCFINDER to measure how many DCs the algorithm returns, and how much time is spent in the minimal cover search. Figure 15 shows the results of these executions. The number of discovered DCs varies greatly between datasets. The predominant behavior is that for larger degrees of approximation the minimal cover search runs faster. The search may find approximate DCs sooner for larger degrees of approximation, even when there are still many evidence to cover. The number of discovered DCs decreases, in most cases, with larger degrees of approximation. But the number of DCs may also increase because discovering specializations of more general DCs may change the general paths followed by the cover search.

Figure 16 shows how DCFINDER behaves with different succinctness thresholds. We restricted the discovery to DCs with up to a varying number of symbols (attributes and operators). As expected, there are fewer short DCs – with predicates involving a few attributes and operators. This result is reflected in the cover search runtime since there are far fewer short DC candidates to check. Most of the DCs discovered for *Hospital* are functional dependencies with a few attributes, therefore, increasing the succinctness threshold for this dataset did not affect the result.

The evaluated datasets have no gold standard with a complete set of "interesting" DCs, so reporting the recall of the discovered DCs would be subjective. In an approach similar to [7], we report the precision of the top-k DCs. For this experiment, we used the first

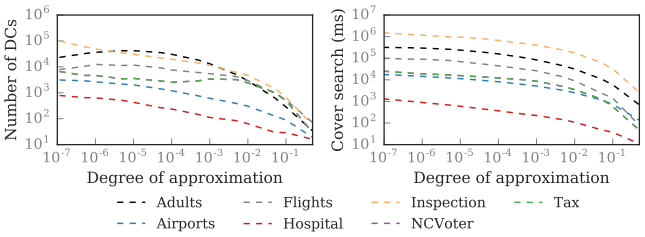


Figure 15: Influence of different degrees of approximation in the number of discovered DCs (left) and cover search time (right). The axes are in log scale.

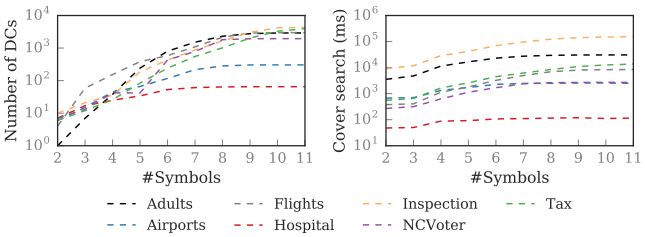


Figure 16: Influence of different succinctness thresholds in the number of discovered DCs (left) and cover search time (right). The Y axis is in log scale.

50,000 tuples of each dataset. We rank all DCs by either coverage or succinctness, in ascending order; or degree of approximation, in descending order. Then, we empirically verify each of the top-k DCs to mark it as meaningful or not. The precision of each interestingness measure at k is given by the number of relevant DCs found in the top-k divided by k. We inspected approximate DCs of *Flight* and *Inspection*; and exact DCs of *Tax*, because of its synthetic nature. As seen in Table 3, the interestingness measures generally achieved good precision rates. The exception was the succinctness measure for *Inspection*, because some rules were under-fitted due to the approximate cover search.

Table 3: Precision of the interestingness measures at $k = 10$.

Dataset	ε	Coverage	Succinctness	Degree of approximation
Flight	0.0001	1.0	1.0	1.0
Inspection	0.0001	0.7	0.5	0.8
Tax	0.0	0.8	0.8	-

Table 4 reports a sample of the discovered DCs. Both coverage and succinctness put the entry φ_4 at the top. The DC φ_4 has no violations, and it expresses an order relationship between attributes originairportid and originairportseqid. Such a relationship is a good opportunity for query optimization. The entry φ_5 is an approximate DC with relatively low succinctness, and low coverage. But because it has a small number of violations (i.e., low degree of approximation), it was straightforward to verify its correctness. The rule has a potential use for data cleaning, because it reveals problems with regard to the operating names of a company and their facility type. The DC φ_6 is a meaningful business rule that did not show up at the top ranked DCs of *Tax*, which shows that the interestingness measures are sometimes imperfect. The DC has predicates with many different symbols and, therefore, low succinctness. The more predicates a DC has, the less likely a tuple pair is to add high coverage scores to that DC.

Table 4: A sample of the discovered DCs.

Dataset	Denial constraint
Flight	φ_4: $\neg(t_x.\text{originairportid} \geq t_y.\text{originairportid} \wedge t_x.\text{originairportseqid} < t_y.\text{originairportseqid})$
Inspection	φ_5: $\neg(t_x.\text{dbaname} = t_y.\text{akaname} \wedge t_x.\text{address} = t_y.\text{address} \; t_x.\text{facilitytype} \neq t_y.\text{facilitytype})$
Tax	φ_6: $\neg(t_x.\text{state} = t_y.\text{state} \wedge t_x.\text{singleexemp} < t_y.\text{childexemp} \; t_x.\text{childexemp} > t_y.\text{childexemp})$

Overall, it is possible to quickly find relevant DCs by using the interestingness measures we discussed in this section. Coverage and degree of approximation are particular useful to spot records that do not follow constraints satisfied by most of the data. The degrees of approximation and succinctness have a high impact on the runtime of cover search, and number of discovered DCs. Of course, this brief analysis only scratches the surface of the problem of ranking discovered DCs for further use. It does show the potential though, and the ability of DCFINDER to incorporate relevance measures to speed up execution.

9. CONCLUSION AND FUTURE WORKS

Motivated by the continuous need for maintaining the consistency of data, we investigated the problem of discovering consistency rules expressed as DCs. We presented the DCFINDER algorithm for discovering all minimal, approximate or exact, DCs of relational datasets. In DCFINDER, building a complete, but compact, evidence set is broken down into: (i) creating PLIs; (ii) partitioning tuple pairs based on their ranges; (iii) preparing evidence based on predicate selectivity; and (iv) completing evidence based on PLI relationships. DCFINDER uses evidence distribution to efficiently explore the large DC search space, and to calculate two measures: the number of violations of approximate DCs, and the statistical significance of DCs based on data coverage. Our performance evaluation shows that DCFINDER is faster than all prior state-of-the-art for the discovery of approximate DCs. The algorithm is, at times, even faster than the algorithms specialized in discovering exact DCs only. Our brief study on DC interestingness indicates that it is possible to quickly spot interesting DCs out of the many DCs discovered.

We envision two promising directions for future work. First, devising a theory to calculate statistical guarantees of approximate DCs may open opportunities for aggressive pruning in the evidence set building phase. Second, ranking DCs is still limited to humans-in-the-loop: DC scores are tied to the weights chosen by users. Devising new techniques to dynamically calculate interestingness weights may help users to better explore and judge the results.

Acknowledgments

We thank Tobias Bleifuß and Thorsten Papenbrock for their help in starting this project and the authors of [5] for providing code for our comparative evaluation.

10. REFERENCES

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

[2] J. Bailey, T. Manoukian, and K. Ramamohanarao. A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns. In *Proceedings of the International Conference on Data Mining (ICDM)*, pages 485–488, 2003.

[3] M. Baudineta, J. Chomicki, and P. Wolper. Constraint-generating dependencies. *Journal of Computer and System Sciences*, 59(1):94 – 115, 1999.

[4] T. Bläsius, T. Friedrich, and M. Schirneck. The parameterized complexity of dependency detection in relational databases. In *International Symposium on Parameterized and Exact Computation (IPEC)*, pages 6:1–6:13, 2016.

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

[6] L. Caruccio, V. Deufemia, and G. Polese. Relaxed functional dependencies – a survey of approaches. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(1):147–165, 2016.

[7] F. Chiang and R. J. Miller. Discovering data quality rules. *PVLDB*, 1(1):1166–1177, 2008.

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

[9] 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.

[10] C. Consonni, P. Sottovia, A. Montresor, and Y. Velegrakis. Discovering order dependencies through order compatibility. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 409–420, 2019.

[11] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Cleaning inconsistencies in information extraction via prioritized repairs. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 164–175, 2014.

[12] W. Fan. Data quality: From theory to practice. *SIGMOD Record*, 44(3):7–18, 2015.

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

[14] W. Fan, F. Geerts, and J. Wijsen. Determining the currency of data. *ACM Transactions on Database Systems (TODS)*, 37(4):25:1–25:46, 2012.

[15] L. Geng and H. J. Hamilton. Interestingness measures for data mining: A survey. *ACM Computing Surveys*, 38(3), 2006.

[16] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems (TODS)*, 28(2):140–174, 2003.

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

[18] 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.

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

[20] J. Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. *Theoretical Computer Science*, 149(1):129–149, 1995.

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

[22] L. Lin and Y. Jiang. The computation of hitting sets: Review and new algorithms. *Information Processing Letters*, 86(4):177 – 184, 2003.

[23] H. Liu, D. Xiao, P. Didwania, and M. Y. Eltabakh. Exploiting soft and hard correlations in big data query optimization. *PVLDB*, 9(12):1005–1016, 2016.

[24] J. Liu, J. Li, C. Liu, and Y. Chen. Discover dependencies from data - a review. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 24(2):251–264, 2012.

[25] B. Marnette, G. Mecca, and P. Papotti. Scalable data exchange with functional dependencies. *PVLDB*, 3(1-2):105–116, 2010.

[26] N. Novelli and R. Cicchetti. FUN: An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 189–203, 2001.

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

[28] 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.

[29] 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.

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

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

[32] J. Rammelaere and F. Geerts. Revisiting conditional functional dependency discovery: Splitting the "C" from the "FD". In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD)*, pages 552–568, 2019.

[33] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. HoloClean: Holistic data repairs with probabilistic inference. *PVLDB*, 10(11):1190–1201, 2017.

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

[35] D. Z. Wang, X. L. Dong, A. D. Sarma, M. J. Franklin, and A. Y. Halevy. Functional dependency generation and applications in pay-as-you-go data integration systems. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 2009.

[36] Z. Wei and S. Link. Embedded functional dependencies and data-completeness tailored database design. *PVLDB*, 12(11):1458–1470, 2019.