Discovering Denial Constraints

Xu Chu*
University of Waterloo
x4chu@uwaterloo.ca

Ihab F. Ilyas
QCRI
ikaldas@qf.org.qa

Paolo Papotti
QCRI
ppapotti@qf.org.qa

ABSTRACT

Integrity constraints (ICs) provide a valuable tool for enforcing correct application semantics. However, designing ICs requires experts and time. Proposals for automatic discovery have been made for some formalisms, such as functional dependencies and their extension conditional functional dependencies. Unfortunately, these dependencies cannot express many common business rules. For example, an American citizen cannot have lower salary and higher tax rate than another citizen in the same state. In this paper, we tackle the challenges of discovering dependencies in a more expressive integrity constraint language, namely Denial Constraints (DCs). DCs are expressive enough to overcome the limits of previous languages and, at the same time, have enough structure to allow efficient discovery and application in several scenarios. We lay out theoretical and practical foundations for DCs, including a set of sound inference rules and a linear algorithm for implication testing. We then develop an efficient instance-driven DC discovery algorithm and propose a novel scoring function to rank DCs for user validation. Using real-world and synthetic datasets, we experimentally evaluate scalability and effectiveness of our solution.

1. INTRODUCTION

As businesses generate and consume data more than ever, enforcing and maintaining the quality of their data assets become critical tasks. One in three business leaders does not trust the information used to make decisions [12], since establishing trust in data becomes a challenge as the variety and the number of sources grow. Therefore, data cleaning is an urgent task towards improving data quality. Integrity constraints (ICs), originally designed to improve the quality of a database schema, have been recently repurposed towards improving the quality of data, either through checking the validity of the data at points of entry, or by cleaning the dirty data at various points during the processing pipeline [10, 13].Traditional types of ICs, such as key constraints, check constraints, functional dependencies (FDs), and their extension conditional functional dependencies (CFDs) have been proposed for data quality management [7]. However, there is still a big space of ICs that cannot be captured by the aforementioned types.

EXAMPLE 1. *Consider the US tax records in Table 1. Each record describes an individual address and tax information with 15 attributes: first and last name (*FN, LN*), gender (*GD*), area code (*AC*), mobile phone number (*PH*), city (*CT*), state (*ST*), zip code (*ZIP*), marital status (*MS*), has children (*CH*), salary (*SAL*), tax rate (*TR*), tax exemption amount if single (*STX*), married (*MTX*), and having children (*CTX*).*

Suppose that the following constraints hold: (1) area code and phone identify a person; (2) two persons with the same zip code live in the same state; (3) a person who lives in Denver lives in Colorado; (4) if two persons live in the same state, the one earning a lower salary has a lower tax rate; and (5) it is not possible to have single tax exemption greater than salary.

Constraints (1), (2), and (3) can be expressed as a key constraint, an FD, and a CFD, respectively.

$(1) : Key\{AC, PH\}$
$(2) : ZIP \rightarrow ST$
$(3) : [CT = \text{'Denver'}] \rightarrow [ST = \text{'CO'}]$

Since Constraints (4) and (5) involve order predicates ($>, <$), and (5) compares different attributes in the same predicate, they cannot be expressed by FDs and CFDs. However, they can be expressed in first-order logic.

$c_4 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ST = t_\beta.ST \wedge t_\alpha.SAL < t_\beta.SAL \wedge t_\alpha.TR > t_\beta.TR)$
$c_5 : \forall t_\alpha \in R, \neg(t_\alpha.SAL < t_\alpha.STX)$

Since first-order logic is more expressive, Constraints (1)-(3) can also be expressed as follows:

$c_1 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.AC = t_\beta.AC \wedge t_\alpha.PH = t_\beta.PH)$
$c_2 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ZIP = t_\beta.ZIP \wedge t_\alpha.ST \neq t_\beta.ST)$
$c_3 : \forall t_\alpha \in R, \neg(t_\alpha.CT = \text{'Denver'} \wedge t_\alpha.ST \neq \text{'CO'})$

The more expressive power an IC language has, the harder it is to exploit it, for example, in automated data cleaning algorithms, or in writing SQL queries for consistency checking. There is an infinite space of business rules up to ad-hoc programs for enforcing correct application semantics. It is easy to see that a balance should be achieved between the expressive power of ICs in order to deal with a broader space of business rules, and at the same time, the restrictions required to ensure adequate static analysis of ICs and the development of effective cleaning and discovery algorithms.

Denial Constraints (DCs) [5, 13], a universally quantified first order logic formalism, can express all constraints in Example 1 as they are more expressive than FDs and CFDs. To clarify the connection between DCs and the different classes of ICs we show in

*Work done while interning at QCRI.

Proceedings of the VLDB Endowment, Vol. 6, No. 13

TID	FN	LN	GD	AC	PH	CT	ST	ZIP	MS	CH	SAL	TR	STX	MTX	CTX
t_1	Mark	Ballin	M	304	232-7667	Anthony	WV	25813	S	Y	5000	3	2000	0	2000
t_2	Chunho	Black	M	719	154-4816	Denver	CO	80290	M	N	60000	4.63	0	0	0
t_3	Annja	Rebizant	F	636	604-2692	Cyrene	MO	64739	M	N	40000	6	0	4200	0
t_4	Annie	Puerta	F	501	378-7304	West Crossett	AR	72045	M	N	85000	7.22	0	40	0
t_5	Anthony	Landram	M	319	150-3642	Gifford	IA	52404	S	Y	15000	2.48	40	0	40
t_6	Mark	Murro	M	970	190-3324	Denver	CO	80251	S	Y	60000	4.63	0	0	0
t_7	Ruby	Billinghurst	F	501	154-4816	Kremlin	AR	72045	M	Y	70000	7	0	35	1000
t_8	Marcelino	Nuth	F	304	540-4707	Kyle	WV	25813	M	N	10000	4	0	0	0

Table 1: Tax data records.

Figure 1 a classification based on two criteria: (i) single tuple level vs table level, and (ii) with constants involved in the constraint vs with only column variables. DCs are expressive enough to cover interesting ICs in each quadrant. DCs serve as a great compromise between expressiveness and complexity for the following reasons: (1) they are defined on predicates that can be easily expressed in SQL queries for consistency checking; (2) they have been proven to be a useful language for data cleaning in many aspects, such as data repairing [10], consistent query answering [5], and expressing data currency rules [13]; and (3) while their static analysis turns out to be undecidable [3], we show that it is possible to develop a set of sound inference rules and a linear implication testing algorithm for DCs that enable an efficient adoption of DCs as an IC language, as we show in this paper.

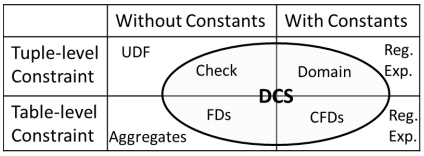


Figure 1: The ICs quadrant.

While DCs can be obtained through consultation with domain experts, it is an expensive process and requires expertise in the constraint language at hand as shown in the experiments. We identified three challenges that hinder the adoption of DCs as an efficient IC language and in discovering DCs from an input data instance:
(1) Theoretical Foundation. The necessary theoretical foundations for DCs as a constraint language are missing [13]. Armstrong Axioms and their extensions are at the core of state-of-the-art algorithms for inferring FDs and CFDs [15, 17], but there is no similar foundation for the design of tractable DCs discovery algorithms.

EXAMPLE 2. *Consider the following constraint, c_6, which states that there cannot exist two persons who live in the same zip code and one person has a lower salary and higher tax rate.*
$c_6 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ZIP = t_\beta.ZIP \wedge t_\alpha.SAL < t_\beta.SAL \wedge t_\alpha.TR > t_\beta.TR)$
c_6 is implied by c_2 and c_4: if two persons live in the same zip code, by c_2 they would live in the same state and by c_4 one person cannot earn less and have higher tax rate in the same state.

In order to systematically identify implied DCs (such as c_6), for example, to prune redundant DCs, a reasoning system is needed.
(2) Space Explosion. Consider FDs discovery on schema R, let $|R| = m$. Taking an attribute as the right hand side of an FD, any subset of remaining $m-1$ attributes could serve as the left hand side. Thus, the space to be explored for FDs discovery is $m*2^{m-1}$. Consider discovering DCs involving at most two tuples without constants; a predicate space needs to be defined, upon which the space of DCs is defined. The structure of a predicate consists of two different attributes and one operator. Given two tuples, we have $2m$ distinct cells; and we allow six operators ($=, \neq, >, \leq, <, \geq$). Thus the size of the predicate space $\mathbf{P}$ is: $|\mathbf{P}| = 6 * 2m * (2m-1)$. Any subset of the predicate space could constitute a DC. Therefore, the search space for DCs discovery is of size $2^{|\mathbf{P}|}$.

DCs discovery has a much larger space to explore, further justifying the need for a reasoning mechanism to enable efficient pruning, as well as the need for an efficient discovery algorithm. The problem is further complicated by allowing constants in the DCs.
(3) Verification. Since the quality of ICs is crucial for data quality, discovered ICs are usually verified by domain experts for their validity. Model discovery algorithms suffer from the problem of overfitting [6]; ICs found on the input instance I of schema R may not hold on future data of R. This happens also for DCs discovery.

EXAMPLE 3. *Consider DC c_7 on Table 1, which states that first name determines gender.*
$c_7 : \forall t_\alpha, t_\beta \in R, \neg(t_\alpha.FN = t_\beta.FN \wedge t_\alpha.GD \neq t_\beta.GD)$
Even if c_7 is true on current data, common knowledge suggests that it does not hold in general.

Statistical measures have been proposed to rank the constraints and assist the verification step for specific cases. For CFDs it is possible to count the number of tuples that match their tableaux [8]. Similar support measures are used for association rules [2].

Unfortunately, discovered DCs are more difficult to verify and rank than previous formalisms for three reasons: (1) similarly to FDs, in general it is not possible to just count constants to measure support; (2) given the explosion of the space, the number of discovered DCs is much larger than the size of discovered FDs; (3) the semantics of FDs/CFDs is much easier to understand compared to DCs. A novel and general measure of interestingness for DCs is therefore needed to rank discovered constraints.

Contributions. Given the DCs discovery problem and the above challenges, we make the following three contributions:

1. We give the formal problem definition of discovering DCs (Section 3). We introduce static analysis for DCs with three sound axioms that serve as the cornerstone for our implication testing algorithm as well as for our DCs discovery algorithm (Section 4).

2. We present FASTDC, a DCs discovery algorithm (Section 5). FASTDC starts by building a predicate space and calculates evidence sets for it. We establish the connection between discovering minimal DCs and finding minimal set covers for evidence sets. We employ depth-first search strategy for finding minimal set covers and use DC axioms for branch pruning. To handle datasets that may have data errors, we extend FASTDC to discover approximate constraints. Finally, we further extend it to discover DCs involving constant values.

3. We propose a novel scoring function, the *interestingness* of a DC, which combines succinctness and coverage measures of discovered DCs in order to enable their ranking and pruning based on thresholds, thus reducing the cognitive burden for human verification (Section 6).

We experimentally verify our techniques on real-life and synthetic data (Section 7). We show that FASTDC is bound by the number of tuples $|I|$ and by the number of DCs $|\Sigma|$, and that the polynomial part w.r.t. $|I|$ can be parallelized. We show that the implication test substantially reduces the number of DCs in the output, thus reducing users' effort in verifying DCs. We also verify how effective our scoring function is at identifying interesting constraints.

2. RELATED WORK

Our work finds similarities with several bodies of work: static analysis of ICs, dependency discovery, and scoring of ICs.

Whenever a dependency language is proposed, the static analysis should be investigated.Static analysis for FDs has been laid out long ago [1], in which it is shown that static analysis for FDs can be done in linear time w.r.t. the number of FDs and three inference rules are proven to be sound and complete. Conditional functional dependencies were first proposed by Bohannon et al. [7], where implication and consistency problems were shown to be intractable. In addition, a set of sound and complete inference rules were also provided, which were later simplified by Fan [14]. Though denial constraints have been used for data cleaning as well as consistent query answering [5, 10], static analysis has been done only for special fragments, such as currency rules [13].

In the context of constraints discovery, FDs attracted the most attention and whose methodologies can be divided into schema-driven and instance-driven approaches. TANE is a representative for the schema-driven approach [17]. It adopts a level-wise candidate generation and pruning strategy and relies on a linear algorithm for checking the validity of FDs. TANE is sensitive to the size of the schema. FASTFD is a an instance-driven approach [19], which first computes agree-sets from data, then adopts a heuristic-driven depth-first search algorithm to search for covers of agree-sets. FASTFD is sensitive to the size of the instance. Both algorithms were extended in [15] for discovering CFDs. CFDs discovery is also studied in [8], which not only is able to discover exact CFDs but also outputs approximate CFDs and dirty values for approximate CFDs, and in [16], which focuses on generating a near-optimal tableaux assuming an embedded FD is provided. The lack of an efficient DCs validity checking algorithm makes the schema-driven approach for DCs discovery infeasible. Therefore, we extend FASTFD for DCs discovery.

Another aspect of discovering ICs is to measure the importance of ICs according to a scoring function. In FDs discovery, Ilyas et al. examined the statistical correlations for each column pair to discover soft FDs [18]. In CFDs discovery some measures have been proposed, including support, which is defined as the percentage of the tuples in the data that match the pattern tableaux, conviction, and χ^2 test [8, 15]. Our scoring function identifies two principles that are widely used in data mining, and combines them into a unified function, which is fundamentally different from previous scoring functions for discovered ICs.

3. DENIAL CONSTRAINTS AND DISCOVERY PROBLEM

In this section, we first review the syntax and semantics of DCs. Then, we define minimal DCs and state their discovery problem.

3.1 Denial Constraints (DCs)

Syntax. Consider a database schema of the form $\mathbb{S} = (\mathbb{U}, \mathbb{R}, \mathbb{B})$, where $\mathbb{U}$ is a set of database domains, $\mathbb{R}$ is a set of database predicates or relations, and $\mathbb{B}$ is a set of finite built-in operators. In this paper, $\mathbb{B} = \{=, <, >, \neq, \leq, \geq\}$. $\mathbb{B}$ must be *negation closed*, such that we could define the *inverse* of operator ϕ as $\overline{\phi}$.

We support the subset of integrity constraints identified by *denial constraints* (DCs) over relational databases. We introduce a notation for DCs of the form $\varphi : \forall t_\alpha, t_\beta, t_\gamma, \ldots \in R, \neg(P_1 \wedge \ldots \wedge P_m)$, where P_i is of the form $v_1 \phi v_2$ or $v_1 \phi c$ with $v_1, v_2 \in t_x.A, x \in \{\alpha, \beta, \gamma, \ldots\}, A \in R$, and c is a constant. For simplicity, we assume there is only one relation R in $\mathbb{R}$.

For a DC φ, if $\forall P_i, i \in [1, m]$ is of the form $v_1 \phi v_2$, then we call such DC *variable denial constraint* (VDC), otherwise, φ is a *constant denial constraint* (CDC).

The *inverse* of predicate P: $v_1 \phi_1 v_2$ is $\overline{P}$: $v_1 \phi_2 v_2$,with $\phi_2 = \overline{\phi_1}$. If P is true, then $\overline{P}$ is false. The set of *implied* predicates of P is $Imp(P) = \{Q | Q : v_1 \phi_2 v_2\}$, where $\phi_2 \in Imp(\phi_1)$. If P is true, then $\forall Q \in Imp(P)$, Q is true. The inverse and implication of the six operators in $\mathbb{B}$ is summarized in Table 2.

ϕ	$=$	$\neq$	$>$	$<$	$\geq$	$\leq$
$\overline{\phi}$	$\neq$	$=$	$\leq$	$\geq$	$<$	$>$
$Imp(\phi)$	$=, \geq, \leq$	$\neq$	$>, \geq, \neq$	$<, \leq, \neq$	$\geq$	$\leq$

Table 2: Operator Inverse and Implication.

Semantics. A DC states that all the predicates cannot be true at the same time, otherwise, we have a violation. Single-tuple constraints (such as check constraints), FDs, and CFDs are special cases of unary and binary denial constraints with equality and inequality predicates. Given a database instance I of schema $\mathbb{S}$ and a DC φ, if I satisfies φ, we write $I \models \varphi$, and we say that φ is a *valid DC*. If we have a set of DC Σ, $I \models \Sigma$ if and only if $\forall \varphi \in \Sigma, I \models \varphi$.

A set of DCs Σ implies φ, i.e., $\Sigma \models \varphi$, if for every instance I of $\mathbb{S}$, if $I \models \Sigma$, then $I \models \varphi$.

In the context of this paper, we are only interested in DCs with at most two tuples. DCs involving more tuples are less likely in real life, and incur bigger predicate space to search as shown in Section 5. The universal quantifier for DCs with at most two tuples are $\forall t_\alpha, t_\beta$. We will omit universal quantifiers hereafter.

3.2 Problem Definition

Trivial, Symmetric, and Minimal DC. A DC $\neg(P_1 \wedge \ldots \wedge P_n)$ is said to be *trivial* if it is satisfied by any instance. In the sequel, we only consider nontrivial DCs unless otherwise specified. The *symmetric* DC of a DC φ_1 is a DC φ_2 by substituting t_α with t_β, and t_β with t_α. If φ_1 and φ_2 are symmetric, then $\varphi_1 \models \varphi_2$ and $\varphi_2 \models \varphi_1$. A DC φ_1 is *set-minimal*, or *minimal*, if there does not exist φ_2, s.t. $I \models \varphi_1, I \models \varphi_2$, and $\varphi_2.Pres \subset \varphi_1.Pres$. We use $\varphi.Pres$ to denote the set of predicates in DC φ.

EXAMPLE 4. *Consider three additional DCs for Table 1.*
$c_8 : \neg(t_\alpha.SAL = t_\beta.SAL \wedge t_\alpha.SAL > t_\beta.SAL)$
$c_9 : \neg(t_\alpha.PH = t_\beta.PH)$
$c_{10} : \neg(t_\alpha.ST = t_\beta.ST \wedge t_\alpha.SAL > t_\beta.SAL \wedge t_\alpha.TR < t_\beta.TR)$
c_8 is a trivial DC, since there cannot exist two persons that have the same salary, and one's salary is greater than the other. If we remove tuple t_7 in Table 1, c_9 becomes a valid DC, making c_1 no longer minimal. c_{10} and c_4 are symmetric DCs.

Problem Statement. Given a relational schema R and an instance I, the *discovery problem* for DCs is to find all valid minimal DCs that hold on I. Since the number of DCs that hold on a dataset

is usually very big, we also study the problem of ranking DCs with an objective function described in Section 6.

4. STATIC ANALYSIS OF DCS

Since DCs subsume FDs and CFDs, it is natural to ask whether we can perform reasoning the same way. An inference system for DCs enables pruning in a discovery algorithm. Similarly, an implication test is required to reduce the number of DCs in the output.

4.1 Inference System

Armstrong Axioms are the fundamental building blocks for implication analysis for FDs [1]. We present three symbolic inference rules for DCs, denoted as $\mathcal{I}$, analogous to such Axioms.

Triviality: $\forall P_i, P_j$, if $\overline{P_i} \in Imp(P_j)$, then $\neg(P_i \wedge P_j)$ is a trivial DC.

Augmentation: If $\neg(P_1 \wedge \ldots \wedge P_n)$ is a valid DC, then $\neg(P_1 \wedge \ldots \wedge P_n \wedge Q)$ is also a valid DC.

Transitivity: If $\neg(P_1 \wedge \ldots \wedge P_n \wedge Q_1)$ and $\neg(R_1 \wedge \ldots \wedge R_m \wedge Q_2)$ are valid DCs, and $Q_2 \in Imp(\overline{Q_1})$, then $\neg(P_1 \wedge \ldots \wedge P_n \wedge R_1 \wedge \ldots \wedge R_m)$ is also a valid DC.

Triviality states that, if a DC has two predicates that cannot be true at the same time ($\overline{P_i} \in Imp(P_j)$), then the DC is trivially satisfied. Augmentation states that, if a DC is valid, adding more predicates will always result in a valid DC. Transitivity states, that if there are two DCs and two predicates (one in each DC) that cannot be false at the same time ($Q_2 \in Imp(\overline{Q_1})$), then merging two DCs plus removing those two predicates will result in a valid DC.

Inference system $\mathcal{I}$ is a syntactic way of checking whether a set of DCs Σ implies a DC φ. It is sound in that if by using $\mathcal{I}$ a DC φ can be derived from Σ, i.e., $\Sigma \vdash_{\mathcal{I}} \varphi$, then Σ implies φ, i.e., $\Sigma \models \varphi$. The completeness of $\mathcal{I}$ dictates that if $\Sigma \models \varphi$, then $\Sigma \vdash_{\mathcal{I}} \varphi$. We identify a specific form of DCs, for which $\mathcal{I}$ is complete. The specific form requires that each predicate of a DC is defined on two tuples and on the same attribute, and that all predicates must have the same operator θ except one that must have the reverse of θ.

THEOREM 1. *The inference system $\mathcal{I}$ is sound. It is also complete for VDCs of the form $\forall t_\alpha, t_\beta \in R, \neg(P_1 \wedge \ldots \wedge P_m \wedge Q)$, where $P_i = t_\alpha.A_i \theta t_\beta.A_i, \forall i \in [1, m]$ and $Q = t_\alpha.B \overline{\theta} t_\beta.B$ with $A_i, B \in \mathbb{U}$.*

Formal proof for Theorem 1 is reported in the extended version of this paper [9]. The completeness result of $\mathcal{I}$ for that form of DCs generalizes the completeness result of Armstrong Axioms for FDs. In particular, FDs adhere to the form with θ being $=$. The partial completeness result for the inference system has no implication on the completeness of the discovery algorithms described in Section 5. We will discuss in the experiments how, although not complete, the inference system $\mathcal{I}$ has a huge impact on the pruning power of the implication test and on the FASTDC algorithm.

4.2 Implication Problem

Implication testing refers to the problem of determining whether a set of DCs Σ implies another DC φ. It has been established that the complexity of the implication testing problem for DCs is coNP-Complete [3]. Given the intractability result, we have devised a linear, sound, but not complete, algorithm for implication testing to reduce the number of DCs in the discovery algorithm output.

In order to devise an efficient implication testing algorithm, we define the concept of *closure* in Definition 1 for a set of predicates $\mathbf{W}$ under a set of DCs Σ. A predicate P is in the closure if adding $\overline{P}$ to $\mathbf{W}$ would constitute a DC implied by Σ. It is in spirit similar to the closure of a set of attributes under a set of FDs.

DEFINITION 1. *The closure of a set of predicates $\mathbf{W}$, w.r.t. a set of DCs Σ, is a set of predicates, denoted as $Clo_\Sigma(\mathbf{W})$, such that $\forall P \in Clo_\Sigma(\mathbf{W})$, $\Sigma \models \neg(\mathbf{W} \wedge \overline{P})$.*

Algorithm 1 GET PARTIAL CLOSURE:

Input: Set of DCs Σ, Set of Predicates $\mathbf{W}$
Output: Set of predicates called closure of $\mathbf{W}$ under Σ : $Clo_\Sigma(\mathbf{W})$

```
1: for all P ∈ W do
2:     Clo_Σ(W) ← Clo_Σ(W) + Imp(P)
3:     Clo_Σ(W) ← Clo_Σ(W) + Imp(Clo_Σ(W))
4: for each P, create a list L_P of DCs containing P
5: for each φ, create a list L_φ of predicates not yet in the closure
6: for all φ ∈ Σ do
7:     for all P ∈ φ.Pres do
8:         L_P ← L_P + φ
9: for all P ∉ Clo_Σ(W) do
10:    for all φ ∈ L_P do
11:        L_φ ← L_φ + P
12: create a queue J of DC with all but one predicate in the closure
13: for all φ ∈ Σ do
14:    if |L_φ| = 1 then
15:        J ← J + φ
16: while |J| > 0 do
17:    φ ← J.pop()
18:    P ← L_φ.pop()
19:    for all Q ∈ Imp(P̄) do
20:        for all φ ∈ L_Q do
21:            L_φ ← L_φ − Q
22:            if |L_φ| = 1 then
23:                J ← J + φ
24:    Clo_Σ(W) ← Clo_Σ(W) + Imp(P̄)
25:    Clo_Σ(W) ← Clo_Σ(W) + Imp(Clo_Σ(W))
26: return Clo_Σ(W)
```

Algorithm 1 calculates the partial closure of $\mathbf{W}$ under Σ, whose proof of correctness is provided in [9]. We initialize $Clo_\Sigma(\mathbf{W})$ by adding every predicate in $\mathbf{W}$ and their implied predicates due to Axiom Triviality (Line 1-2). We add additional predicates that are implied by $Clo_\Sigma(\mathbf{W})$ through basic algebraic transitivity (Line 3). The closure is enlarged if there exists a DC φ in Σ such that all but one predicates in φ are in the closure (Line 15-23). We use two lists to keep track of exactly when such condition is met (Line 3-11).

EXAMPLE 5. *Consider Σ=$\{c_1, \ldots, c_5\}$ and $\mathbf{W}$ = $\{t_\alpha.ZIP = t_\beta.ZIP, t_\alpha.SAL < t_\beta.SAL\}$.*

The initialization step in Line(1-3) results in $Clo_\Sigma(\mathbf{W}) = \{t_\alpha.ZIP = t_\beta.ZIP, t_\alpha.SAL < t_\beta.SAL, t_\alpha.SAL \leq t_\beta.SAL\}$. As all predicates but $t_\alpha.ST \neq t_\beta.ST$ of c_2 are in the closure, we add the implied predicates of the reverse of $t_\alpha.ST \neq t_\beta.ST$ to it and $Clo_\Sigma(\mathbf{W}) = \{t_\alpha.ZIP = t_\beta.ZIP, t_\alpha.SAL < t_\beta.SAL, t_\alpha.SAL \leq t_\beta.SAL, t_\alpha.ST = t_\beta.ST\}$. As all predicates but $t_\alpha.TR > t_\beta.TR$ of c_4 are in the closure (Line 22), we add the implied predicates of its reverse, $Clo_\Sigma(\mathbf{W}) = \{t_\alpha.ZIP = t_\beta.ZIP, t_\alpha.SAL < t_\beta.SAL, t_\alpha.SAL \leq t_\beta.SAL, t_\alpha.TR \leq t_\beta.TR\}$. No more DCs are in the queue (Line 16).

Since $t_\alpha.TR \leq t_\beta.TR \in Clo_\Sigma(\mathbf{W})$, we have $\Sigma \models \neg(\mathbf{W} \wedge t_\alpha.TR > t_\beta.TR)$, i.e., $\Sigma \models c_6$.

Algorithm 2 tests whether a DC φ is implied by a set of DCs Σ, by computing the closure of $\varphi.Pres$ in φ under Γ, which is Σ enlarged with symmetric DCs. If there exists a DC ϕ in Γ, whose predicates are a subset of the closure, φ is implied by Σ. The proof of soundness of Algorithm 2 is in [9], which also shows a counterexample where φ is implied by Σ, but Algorithm 2 fails.

EXAMPLE 6. *Consider a database with two numerical columns, High (*H*) and Low (*L*). Consider two DCs c_{11}, c_{12}.*

Algorithm 2 IMPLICATION TESTING

Input: Set of DCs Σ, one DC φ
Output: A boolean value, indicating whether $\Sigma \models \varphi$
1: **if** φ is a trivial DC **then**
2: **return** true
3: $\Gamma \leftarrow \Sigma$
4: **for** $\phi \in \Sigma$ **do**
5: $\Gamma \leftarrow \Gamma +$ symmetric DC of ϕ
6: $Clo_\Gamma(\varphi.Pres) = getClosure(\varphi.Pres, \Gamma)$
7: **if** $\exists \phi \in \Gamma$, s.t. $\phi.Pres \subseteq Clo_\Gamma(\varphi.Pres)$ **then**
8: **return** true

$c_{11} : \forall t_\alpha, (t_\alpha.H < t_\alpha.L)$
$c_{12} : \forall t_\alpha, t_\beta, (t_\alpha.H > t_\beta.H \wedge t_\beta.L > t_\alpha.H)$

Algorithm 2 identifies that c_{11} implies c_{12}. Let $\Sigma = \{c_{11}\}$ and $\mathbf{W} = c_{12}.Pres$. $\Gamma = \{c_{11}, c_{13}\}$, where c_{13}: $\forall t_\beta, (t_\beta.H < t_\beta.L)$. $Clo_\Gamma(\mathbf{W}) = \{t_\alpha.H > t_\beta.H, t_\beta.L > t_\alpha.H, t_\beta.H < t_\beta.L\}$, because $t_\beta.H < t_\beta.L$ is implied by $\{t_\alpha.H > t_\beta.H, t_\beta.L > t_\alpha.H\}$ through basic algebraic transitivity (Line 3).

Since $c_{13}.Pres \subset Clo_\Gamma(\mathbf{W})$, the implication holds.

5. DCS DISCOVERY ALGORITHM

Algorithm 3 describes our procedure for discovering minimal DCs. Since a DC is composed of a set of predicates, we build a predicate space **P** based on schema R (Line 1). Any subset of **P** could be a set of predicates for a DC.

Algorithm 3 FASTDC

Input: One relational instance I, schema R
Output: All minimal DCs Σ
1: **P** $\leftarrow$ BUILD PREDICATE SPACE(I, R)
2: $Evi_I \leftarrow$ BUILD EVIDENCE SET(I, **P**)
3: **MC** $\leftarrow$ SEARCH MINIMAL COVERS($Evi_I, Evi_I, \emptyset, >_{init}, \emptyset$)
4: **for all** **X** $\in$ ***MC*** **do**
5: $\Sigma \leftarrow \Sigma + \neg(\overline{\mathbf{X}})$
6: **for all** $\varphi \in \Sigma$ **do**
7: **if** $\Sigma - \varphi \models \varphi$ **then**
8: remove φ from Σ

Given **P**, the space of candidate DCs is of size $2^{|\mathbf{P}|}$. It is not feasible to validate each candidate DC directly over I, due to the quadratic complexity of checking all tuple pairs. For this reason, we extract evidence from I in a way that enables the reduction of DCs discovery to a search problem that computes valid minimal DCs without checking each candidate DC individually.

The evidence is composed of sets of satisfied predicates in **P**, one set for every pair of tuples (Line 2). For example, assume two satisfied predicates for one tuple pair: $t_\alpha.A = t_\beta.A$ and $t_\alpha.B = t_\beta.B$. We use the set of satisfied predicates to derive the valid DCs that do not violate this tuple pair. In the example, two sample DCs that hold on that tuple pair are $\neg(t_\alpha.A \neq t_\beta.A)$ and $\neg(t_\alpha.A = t_\beta.A \wedge t_\alpha.B \neq t_\beta.B)$. Let Evi_I be the sets of satisfied predicates for all pairs of tuples, deriving valid minimal DCs for I corresponds to finding the minimal sets of predicates that cover Evi_I (Line 3)[1]. For each minimal cover **X**, we derive a valid minimal DC by inverting each predicate in it (Lines 4-5). We remove implied DCs from Σ with Algorithm 2 (Lines 6-8).

Section 5.1 describes the procedure for building the predicate space **P**. Section 5.2 formally defines Evi_I, gives a theorem that reduces the problem of discovering all minimal DCs to the problem of finding all minimal covers for Evi_I, and presents a procedure for building Evi_I. Section 5.3 describes a search procedure for finding minimal covers for Evi_I. In order to reduce the execution time, the search is optimized with a dynamic ordering of predicates and branch pruning based on the axioms we developed in Section 4. In order to enable further pruning, Section 5.4 introduces an optimization technique that divides the space of DCs and performs DFS on each subspace. We extend FASTDC in Section 5.5 to discover approximate DCs and in Section 5.6 to discover DCs with constants.

5.1 Building the Predicate Space

Given a database schema R and an instance I, we build a predicate space **P** from which DCs can be formed. For each attribute in the schema, we add two equality predicates ($=, \neq$) between two tuples on it. In the same way, for each numerical attribute, we add order predicates ($>, \leq, <, \geq$). For every pair of attributes in R, they are *joinable* (*comparable*) if equality (order) predicates hold across them, and add cross column predicates accordingly.

Profiling algorithms [11] can be used to detect joinable and comparable columns. We consider two columns joinable if they are of same type and have common values[2]. Two columns are comparable if they are both of numerical types and the arithmetic means of two columns are within the same order of magnitude.

EXAMPLE 7. *Consider the following Employee table with three attributes: Employee ID (*I*), Manager ID (*M*), and Salary(*S*).*

TID	*I(String)*	*M(String)*	*S(Double)*
t_9	*A1*	*A1*	*50*
t_{10}	*A2*	*A1*	*40*
t_{11}	*A3*	*A1*	*40*

*We build the following predicate space **P** for it.*

$P_1 : t_\alpha.I = t_\beta.I$	$P_5 : t_\alpha.S = t_\beta.S$	$P_9 : t_\alpha.S < t_\beta.S$
$P_2 : t_\alpha.I \neq t_\beta.I$	$P_6 : t_\alpha.S \neq t_\beta.S$	$P_{10} : t_\alpha.S \geq t_\beta.S$
$P_3 : t_\alpha.M = t_\beta.M$	$P_7 : t_\alpha.S > t_\beta.S$	$P_{11} : t_\alpha.I = t_\alpha.M$
$P_4 : t_\alpha.M \neq t_\beta.M$	$P_8 : t_\alpha.S \leq t_\beta.S$	$P_{12} : t_\alpha.I \neq t_\alpha.M$
$P_{13} : t_\alpha.I = t_\beta.M$	$P_{14} : t_\alpha.I \neq t_\beta.M$	

5.2 Evidence Set

Before giving formal definitions of Evi_I, we show an example of the satisfied predicates for the Employee table Emp above:
$Evi_{Emp} = \{\{P_2, P_3, P_5, P_8, P_{10}, P_{12}, P_{14}\}, \{P_2, P_3, P_6, P_8, P_9, P_{12}, P_{14}\}, \{P_2, P_3, P_6, P_7, P_{10}, P_{11}, P_{13}\}\}$.
Every element in Evi_{Emp} has at least one pair of tuples in I such that every predicate in it is satisfied by that pair of tuples.

DEFINITION 2. *Given a pair of tuple $\langle t_x, t_y \rangle \in I$, the satisfied predicate set for $\langle t_x, t_y \rangle$ is $SAT(\langle t_x, t_y \rangle) = \{P | P \in \mathbf{P}, \langle t_x, t_y \rangle \models P\}$, where **P** is the predicate space, and $\langle t_x, t_y \rangle \models P$ means $\langle t_x, t_y \rangle$ satisfies P.*

The evidence set *of I is $Evi_I = \{SAT(\langle t_x, t_y \rangle) | \forall \langle t_x, t_y \rangle \in I\}$.*

A set of predicates $\mathbf{X} \subseteq \mathbf{P}$ is a minimal set cover *for Evi_I if $\forall E \in Evi_I, \mathbf{X} \cap E \neq \emptyset$, and $\nexists \mathbf{Y} \subset \mathbf{X}$, s.t. $\forall E \in Evi_I, \mathbf{Y} \cap E \neq \emptyset$.*

The minimal set cover for Evi_I is a set of predicates that intersect with every element in Evi_I. Theorem 2 transforms the problem of minimal DCs discovery into the problem of searching for minimal set covers for Evi_I.

THEOREM 2. *$\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$ is a valid minimal DC if and only if $\mathbf{X} = \{X_1, \ldots, X_n\}$ is a minimal set cover for Evi_I.*

[1]For sake of presentation, parameters are described in Section 5.3

[2]We show in the experiments that requiring at least 30% common values allows to identify joinable columns without introducing a large number of unuseful predicates. Joinable columns can also be discovered from query logs, if available.

Proof. Step 1: we prove if $\mathbf{X} \subseteq \mathbf{P}$ is a cover for Evi_I, $\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$ is a valid DC. According to the definition, Evi_I represents all the pieces of evidence that might violate DCs. For any $E \in Evi_I$, there exists $X \in \mathbf{X}$, s.t. $X \in E$; thus $\overline{X} \notin E$. I.e., the presence of $\overline{X}$ in $\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$ disqualifies E as a possible violation.

Step 2: we prove if $\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$ is a valid DC, then $\mathbf{X} \subseteq \mathbf{P}$ is a cover. According to the definition of valid DC, there does not exist tuple pair $\langle t_x, t_y \rangle$, s.t. $\langle t_x, t_y \rangle$ satisfies $\overline{X_1}, \ldots, \overline{X_n}$ simultaneously. In other words, $\forall \langle t_x, t_y \rangle$, $\exists \overline{X_i}$, s.t. $\langle t_x, t_y \rangle$ does not satisfy $\overline{X_i}$. Therefore, $\forall \langle t_x, t_y \rangle$, $\exists \overline{X_i}$, s.t. $\langle t_x, t_y \rangle \models X_i$, which means any tuple pair's satisfied predicate set is covered by $\{X_1, \ldots, X_n\}$.

Step 3: if $\mathbf{X} \subseteq \mathbf{P}$ is a minimal cover, then the DC is also minimal. Assume the DC is not minimal, there exists another DC φ whose predicates are a subset of $\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$. According to Step 2, $\varphi.Pres$ is a cover, which is a subset of $\mathbf{X} = \{X_1, \ldots, X_n\}$. It contradicts with the assumption that $\mathbf{X} \subseteq \mathbf{P}$ is a minimal cover.

Step 4: if the DC is minimal, then the corresponding cover is also minimal. The proof is similar to Step 3. ◇

EXAMPLE 8. *Consider Evi_{Emp} for the table in Example 7.*

$\boldsymbol{X}_1 = \{P_2\}$ is a minimal cover, thus $\neg(\overline{P_2})$, i.e., $\neg(t_\alpha.I = t_\beta.I)$ is a valid DC, which states I *is a key.*

$\boldsymbol{X}_2 = \{P_{10}, P_{14}\}$ is another minimal cover, thus $\neg(\overline{P_{10}} \wedge \overline{P_{14}})$, i.e., $\neg(t_\alpha.S < t_\beta.S \wedge t_\alpha.I = t_\beta.M)$ is another valid DC, which states that a manager's salary cannot be less than her employee's.

The procedure to compute Evi_I follows directly from the definition: for every tuple pair in I, we compute the set of predicates that tuple pair satisfies, and we add that set into Evi_I. This operation is sensitive to the size of the database, with a complexity of $O(|\mathbf{P}| \times |I|^2)$. However, for every tuple pair, computing the satisfied set of predicates is independent of each other. In our implementation we use the *Grid Scheme* strategy, a standard approach to scale in entity resolution [4]. We partition the data into B blocks, and define each task as a comparison of tuples from two blocks. The total number of tasks is $\frac{B^2}{2}$. Suppose we have M machines, we need to distribute the tasks evenly to M machines so as to fully utilize every machine, i.e., we need to ensure $\frac{B^2}{2} = w \times M$ with w the number of tasks for each machine. Therefore, the number of blocks $B = \sqrt{2wM}$. In addition, as we need at least two blocks in memory at any given time, we need to make sure that $(2 \times \frac{|I|}{B} \times$ *Size of a Tuple)* $<$ *Memory Limit*.

5.3 DFS for Minimal Covers

Algorithm 4 presents the depth-first search (DFS) procedure for minimal covers for Evi_I. Ignore Lines (9-10) and Lines (11-12) for now, as they are described in Section 5.4 and in Section 6.3, respectively. We denote by Evi_{curr} the set of elements in Evi_I not covered so far. Initially $Evi_{curr} = Evi_I$. Whenever a predicate P is added to the cover, we remove from Evi_{curr} the elements that contain P, i.e., $Evi_{next} = \{E | E \in E_{curr} \wedge P \notin E\}$ (Line 23). There are two base cases to terminate the search:

(i) there are no more candidate predicates to include in the cover, but $Evi_{curr} \neq \emptyset$ (Lines 14-15); and

(ii) $Evi_{curr} = \emptyset$ and the current path is a cover (Line 16). If the cover is minimal, we add it to the result ***MC*** (Lines 17-19).

We speed up the search procedure by two optimizations: dynamic ordering of predicates as we descend down the search tree and branching pruning based on the axioms in Section 4.

Opt1: Dynamic Ordering. Instead of fixing the order of predicates when descending down the tree, we dynamically order the remaining candidate predicates, denoted as $>_{next}$, based on the number of remaining evidence set they cover (Lines 23 -24). Formally, we define the cover of P w.r.t. Evi_{next} as $Cov(P, Evi_{next}) = |\{P \in E | E \in Evi_{next}\}|$. And we say that $P >_{next} Q$ if $Cov(P, Evi_{next}) > Cov(Q, Evi_{next})$, or $Cov(P, Evi_{next}) = Cov(Q, Evi_{next})$ and P appears before Q in the preassigned order in the predicate space. The initial evidence set Evi_I is computed as discussed in Section 5.2. To computer Evi_{next} (Line 21), we scan every element in Evi_{curr}, and we add in Evi_{next} those elements that do not contain P.

Algorithm 4 SEARCH MINIMAL COVERS

Input: 1. Input Evidence set, Evi_I
2. Evidence set not covered so far, Evi_{curr}
3. The current path in the search tree, $\mathbf{X} \subseteq \mathbf{P}$
4. The current partial ordering of the predicates, $>_{curr}$
5. The DCs discovered so far, Σ

Output: A set of minimal covers for Evi, denoted as ***MC***

1: *Branch Pruning*
2: $P \leftarrow \mathbf{X}.last$ // Last Predicate added into the path
3: **if** $\exists Q \in \mathbf{X} - P$, s.t. $P \in Imp(Q)$ **then**
4: **return** //Triviality pruning
5: **if** $\exists \mathbf{Y} \in$ ***MC***, s.t. $\mathbf{X} \supseteq \mathbf{Y}$ **then**
6: **return** //Subset pruning based on ***MC***
7: **if** $\exists \mathbf{Y} = \{Y_1, \ldots, Y_n\} \in$ ***MC***, and $\exists i \in [1, n]$, and $\exists Q \in Imp(Y_i)$, s.t. $\mathbf{Z} = \mathbf{Y}_{-i} \cup \overline{Q}$ and $\mathbf{X} \supseteq \mathbf{Z}$ **then**
8: **return** //Transitive pruning based on ***MC***
9: **if** $\exists \varphi \in \Sigma$, s.t. $\overline{\mathbf{X}} \supseteq \varphi.Pres$ **then**
10: **return** //Subset pruning based on previous discovered DCs
11: **if** $Inter(\varphi) < t, \forall \varphi$ of the form $\neg(\overline{\mathbf{X}} \wedge \mathbf{W})$ **then**
12: **return** //Pruning based on $Inter$ score
13: *Base cases*
14: **if** $>_{curr} = \emptyset$ and $Evi_{curr} \neq \emptyset$ **then**
15: **return** //No DCs in this branch
16: **if** $Evi_{curr} = \emptyset$ **then**
17: **if** no subset of size $|\mathbf{X}| - 1$ covers Evi_{curr} **then**
18: ***MC*** $\leftarrow$ ***MC*** $+ \mathbf{X}$
19: **return** //Got a cover
20: *Recursive cases*
21: **for all** Predicate $P \in >_{curr}$ **do**
22: $\mathbf{X} \leftarrow \mathbf{X} + P$
23: $Evi_{next} \leftarrow$ evidence sets in Evi_{curr} not yet covered by P
24: $>_{next} \leftarrow$ total ordering of $\{P' | P >_{curr} P'\}$ wrt Evi_{next}
25: SEARCH MINIMAL COVERS($Evi_I, Evi_{next}, \mathbf{X}, >_{next}, \Sigma$)
26: $\mathbf{X} \leftarrow \mathbf{X} - P$

EXAMPLE 9. *Consider Evi_{Emp} for the table in Example 7. We compute the cover for each predicate, such as $Cov(P_2, Evi_{Emp})$ = 3, $Cov(P_8, Evi_{Emp})$ = 2, $Cov(P_9, Evi_{Emp})$ = 1, etc. The initial ordering for the predicates according to Evi_{Emp} is $>_{init} = P_2 > P_3 > P_6 > P_8 > P_{10} > P_{12} > P_{14} > P_5 > P_7 > P_9 > P_{11} > P_{13}$.*

Opt2: Branch Pruning. The purpose of performing dynamic ordering of candidate predicates is to get covers as early as possible so that those covers can be used to prune unnecessary branches of the search tree. We list three pruning strategies.

(i) Lines(2-4) describe the first pruning strategy. This branch would eventually result in a DC of the form $\varphi : \neg(\overline{\mathbf{X} - P} \wedge \overline{P} \wedge \mathbf{W})$, where P is the most recent predicate added to this branch and $\mathbf{W}$ other predicates if we traverse this branch. If $\exists Q \in \overline{\mathbf{X} - P}$, s.t. $P \in Imp(Q)$, then φ is trivial according to Axiom Triviality.

(ii) Lines(5-6) describe the second branch pruning strategy, which is based on ***MC***. If $\mathbf{Y}$ is in the cover, then $\neg(\overline{\mathbf{Y}})$ is a valid DC. Any branch containing $\mathbf{X}$ would result in a DC of the form $\neg(\overline{\mathbf{X}} \wedge \mathbf{W})$, which is implied by $\neg(\overline{\mathbf{Y}})$ based on Axiom Augmentation, since $\overline{\mathbf{Y}} \subseteq \overline{\mathbf{X}}$.

(iii) Lines(7-8) describe the third branching pruning strategy, which is also based on ***MC***. If $\mathbf{Y}$ is in the cover, then $\neg(\overline{\mathbf{Y}}_{-i} \wedge \overline{Y_i})$ is

a valid DC. Any branch containing $\mathbf{X} \supseteq \mathbf{Y}_{-i} \cup \overline{Q}$ would result in a DC of the form $\neg(\overline{\mathbf{Y}}_{-i} \wedge Q \wedge \mathbf{W})$. Since $Q \in Imp(Y_i)$, by applying Axiom Transitive on these two DCs, we would get that $\neg(\overline{\mathbf{Y}}_{-i} \wedge \mathbf{W})$ is also a valid DC, which would imply $\neg(\overline{\mathbf{Y}}_{-i} \wedge Q \wedge \mathbf{W})$ based on Axiom Augmentation. Thus this branch can be pruned.

5.4 Dividing the Space of DCs

Instead of searching for all minimal DCs at once, we divide the space into subspaces, based on whether a DC contains a specific predicate P_1, which can be further divided according to whether a DC contains another specific predicate P_2. We start by defining *evidence set modulo a predicate* P, i.e., Evi_I^P, and we give a theorem that reduces the problem of discovering all minimal DCs to the one of finding all minimal set covers of Evi_I^P for each $P \in \mathbf{P}$.

DEFINITION 3. *Given a $P \in \boldsymbol{P}$, the* evidence set *of I* modulo *P is, $Evi_I^P = \{E - \{P\} | E \in Evi_I, P \in E\}$.*

THEOREM 3. *$\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n} \wedge P)$ is a valid minimal DC, that contains predicate P, if and only if $\boldsymbol{X} = \{X_1, \ldots, X_n\}$ is a* minimal set cover *for Evi_I^P.*

EXAMPLE 10. *Consider Evi_{Emp} for the table in Example 7, $Evi_{Emp}^{P_1} = \emptyset$, $Evi_{Emp}^{P_{13}} = \{\{P_2, P_3, P_6, P_7, P_{10}, P_{11}\}\}$. Thus $\neg(P_1)$ is a valid DC because there is nothing in the cover for $Evi_{Emp}^{P_1}$, and $\neg(P_{13} \wedge \overline{P_{10}})$ is a valid DC as $\{P_{10}\}$ is a cover for $Evi_{Emp}^{P_{13}}$. It is evident that Evi_{Emp}^{P} is much smaller than Evi_{Emp}.*

However, care must be taken before we start to search for minimal covers for Evi_I^P due to the following two problems.

First, a minimal DC containing a certain predicate P is not necessarily a global minimal DC. For instance, assume that $\neg(P, Q)$ is a minimal DC containing P because $\{\overline{Q}\}$ is a minimal cover for Evi_I^P. However, it might not be a minimal DC because it is possible that $\neg(Q)$, which is actually smaller than $\neg(P, Q)$, is also a valid DC. We call such $\neg(P, Q)$ a *local minimal DC w.r.t.* P, and $\neg(Q)$ a *global minimal DC*, or a minimal DC. It is obvious that a global minimal DC is always a local minimal DC w.r.t. each predicate in the DC. Our goal is to generate all globally minimal DCs.

Second, assume that $\neg(P, Q)$ is a global minimal DC. It is an local minimal DC w.r.t. P and Q, thus would appear in subspaces Evi_I^P and Evi_I^Q. In fact, a minimal DC φ would then appear in $|\varphi.Pres|$ subspaces, causing a large amount of repeated work.

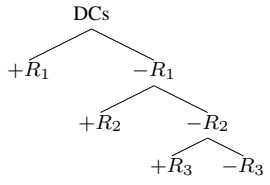


Figure 2: Taxonomy Tree.

We solve the second problem first, then the solution for the first problem comes naturally. We divide the DCs space and order all searches in a way, such that we ensure the output of a locally minimal DC is indeed global minimal, and a previously generated minimal DC will never appear again in latter searches. Consider a predicate space $\mathbf{P}$ that has only 3 predicates R_1 to R_3 as in Figure 2, which presents a taxonomy of all DCs. In the first level, all DCs can be divided into DCs containing R_1, denoted as $+R_1$, and DCs not containing R_1, denoted as $-R_1$. Since we know how to search for local minimal DCs containing R_1, we only need to further process DCs not containing R_1, which can be divided based on containing R_2 or not, i.e., $+R_2$ and $-R_2$. We will divide $-R_2$ as in Figure 2. We can enforce searching for DCs not containing R_i by disallowing $\overline{R_i}$ in the initial ordering of candidate predicates for minimal cover. Since this is a taxonomy of all DCs, no minimal DCs can be generated more than once.

We solve the first problem by performing DFS according to the taxonomy tree in a bottom-up fashion. We start by search for DCs containing R_3, not containing R_1, R_2. Then we search for DCs, containing R_2, not containing R_1, and we verify the resulting DC is global minimal by checking if the reverse of the minimal cover is a super set of DCs discovered from $Evi_I^{R_3}$. The process goes on until we reach the root of the taxonomy, thus ensuring that the results are both globally minimal and complete.

Dividing the space enables more optimization opportunities:

1. Reduction of Number of Searches. If $\exists P \in \mathbf{P}$, such that $Evi_I^P = \emptyset$, we identify two scenarios for Q, where DFS for Evi_I^Q can be eliminated.

(i) $\forall Q \in Imp(\overline{P})$, if $Evi_I^P = \emptyset$, then $\neg(P)$ is a valid DC. The search for Evi_I^Q would result in a DC of the form $\neg(Q \wedge \mathbf{W})$, where $\mathbf{W}$ represents any other set of predicates. Since $Q \in Imp(\overline{P})$, applying Axiom Transitivity, we would have that $\neg(\mathbf{W})$ is a valid DC, which implies $\neg(Q \wedge \mathbf{W})$ based on Axiom Augmentation.

(ii) $\forall \overline{Q} \in Imp(\overline{P})$, since $\overline{Q} \in Imp(\overline{P})$, then $Q \models P$. It follows that $Q \wedge \mathbf{W} \models P$ and therefore $\neg(P) \models \neg(Q \wedge \mathbf{W})$ holds.

EXAMPLE 11. *Consider Evi_{Emp} for the table in Example 7, since $Evi_{Emp}^{P_1} = \emptyset$ and $Evi_{Emp}^{P_4} = \emptyset$, then $\boldsymbol{Q} = \{P_1, P_2, P_3, P_4\}$. Thus we perform $|\boldsymbol{P}| - |\boldsymbol{Q}| = 10$ searches instead of $|\boldsymbol{P}| = 14$.*

2. Additional Branch Pruning. Since we perform DFS according to the taxonomy tree in a bottom-up fashion, DCs discovered from previous searches are used to prune branches in current DFS described by Lines(9-10) of Algorithm 4.

Since Algorithm 4 is an exhaustive search for all minimal covers for Evi_I, Algorithm 3 produces all minimal DCs.

THEOREM 4. *Algorithm 3 produces all non-trivial minimal DCs holding on input database I.*

Complexity Analysis of FASTDC. The initialization of evidence sets takes $O(|\mathbf{P}| * n^2)$. The time for each DFS search to find all minimal covers for Evi_I^P is $O((1 + w_P) * K_P)$, with w_P being the extra effort due to imperfect search of Evi_I^P, and K_P being the number of minimal DCs containing predicate P. Altogether, our FASTDC algorithm has worst time complexity of $O(|\mathbf{P}| * n^2 + |\mathbf{P}| * (1 + w_P) * K_P)$.

5.5 Approximate DCs: A-FASTDC

Algorithm FASTDC consumes the whole input data set and requires no violations for a DC to be declared valid. In real scenarios, there are multiple reasons why this request may need to be relaxed: (1) overfitting: data is dynamic and as more data becomes available, overfitting constraints on current data set can be problematic; (2) data errors: while in general learning from unclean data is a challenge, the common belief is that errors constitute small percentage of data, thus discovering constraints that hold for most of the dataset is a common workaround [8, 15, 17].

We therefore modify the discovery statement as follows: given a relational schema R and instance I, the *approximate DCs discovery problem* for DCs is to find all valid DCs, where a DC φ is valid if the percentage of violations of φ on I, i.e., number of violations of φ on I divided by total number of tuple pairs $|I|(|I| - 1)$, is within threshold ϵ. For this new problem, we introduce A-FASTDC.

Different tuple pairs might have the same satisfied predicate set. For every element E in Evi_I, we denote by $count(E)$ the number of tuple pairs $\langle t_x, t_y\rangle$ such that $E = SAT(\langle t_x, t_y\rangle)$. For example, $count(\{P_2, P_3, P_6, P_8, P_9, P_{12}, P_{14}\}) = 2$ for the table in Example 7 since $SAT(\langle t_{10}, t_9\rangle) = SAT(\langle t_{11}, t_9\rangle) = \{P_2, P_3, P_6, P_8, P_9, P_{12}, P_{14}\}$.

DEFINITION 4. *A set of predicates $\boldsymbol{X} \subseteq \boldsymbol{P}$ is an* ϵ-minimal cover *for Evi_I if $Sum(count(E)) \leq \epsilon|I|(|I|-1)$, where $E \in Evi_I, \boldsymbol{X} \cap E = \emptyset$, and no subset of $\boldsymbol{X}$ has such property.*

Theorem 5 transforms approximate DCs discovery problem into the problem of searching for ϵ-minimal covers for Evi_I.

THEOREM 5. *$\neg(\overline{X_1} \wedge \ldots \wedge \overline{X_n})$ is a valid approximate minimal DC if and only if $\boldsymbol{X}$=$\{X_1, \ldots, X_n\}$ is a* ϵ-minimal cover *for Evi_I.*

There are two modifications for Algorithm 4 to search for ϵ-minimal covers for Evi_I: (1) the dynamic ordering of predicates is based on $Cov(P, Evi) = \sum_{E \in \{E \in Evi, P \in E\}} count(E)$; and (2) the base cases (Lines 12-17) are either when the number of violations of the corresponding DC drops below $\epsilon|I|(|I|-1)$, or the number of violation is still above $\epsilon|I|(|I|-1)$ but there are no more candidate predicates to include. Due to space limitations, we present in [9] the detailed modifications for Algorithm 4 to search for ϵ-minimal covers for Evi_I.

5.6 Constant DCs: C-FASTDC

FASTDC discovers DCs without constant predicates. However, just like FDs may not hold on the entire dataset, thus CFDs are more useful, we are also interested in discovering constant DCs (CDCs). Algorithm 5 describes the procedure for CDCs discovery. The first step is to build a constant predicate space **Q** (Lines 1-6)[3]. After that, one direct way to discover CDCs is to include **Q** in the predicate space **P**, and follow the same procedure in Section 5.3. However, the number of constant predicates is linear w.r.t. the number of constants in the active domain, which is usually very large. Therefore, we follow the approach of [15] and focus on discovering τ-frequent CDCs. The support for a set of constant predicates $\mathbf{X}$ on I, denoted by $sup(\mathbf{X}, I)$, is defined to be the set of tuples that satisfy all constant predicates in $\mathbf{X}$. A set of predicates is said to be τ-frequent if $\frac{|sup(\mathbf{X},I)|}{|I|} \geq \tau$. A CDC φ consisting of only constant predicates is said to be τ-frequent if all strict subsets of $\varphi.Pres$ are τ-frequent. A CDC φ consisting of constant and variable predicates is said to be k-frequent if all subsets of φ's constant predicates are τ-frequent.

EXAMPLE 12. *Consider c_3 in Example 1, $sup(\{t_\alpha.CT =$ 'Denver'$\}, I) = \{t_2, t_6\}$, $sup(\{t_\alpha.ST \neq$ 'CO'$\}, I) = \{t_1, t_3, t_4, t_5, t_7, t_8\}$, and $sup(\{c3.Pres\}, I) = \emptyset$. Therefore, c_3 is a τ-frequent CDC, with $\frac{2}{8} \geq \tau$.*

We follow an "Apriori" approach to discover τ-frequent constant predicate sets. We first identify frequent constant predicate sets of length L_1 from **Q** (Lines 7-15). We then generate candidate frequent constant predicate sets of length m from length $m-1$ (Lines 22-28), and we scan the database I to get their support (Line 24). If the support of the candidate c is 0, we have a valid CDC with only constant predicates (Lines 12-13 and 25-26); if the support of the candidate c is greater than τ, we call FASTDC to get the variable DCs (VDCs) that hold on $sup(c, I)$, and we construct CDCs by combining the τ-frequent constant predicate sets and the variable predicates of VDCs (Lines 18-21).

[3]We focus on two tuple CDCs with the same constant predicates on each tuple, i.e., if $t_\alpha.A\theta c$ is present in a two tuple CDC, $t_\beta.A\theta c$ is enforced by the algorithm. Therefore, we only add $t_\alpha.A\theta c$ to **Q**.

Algorithm 5 C-FASTDC

Input: Instance I, schema R, minimal frequency requirement τ
Output: Constant DCs Γ

```
1: Let Q ← ∅ be the constant predicate space
2: for all A ∈ R do
3:    for all c ∈ ADom(A) do
4:       Q ← Q + t_α.Aθc, where θ ∈ {=, ≠}
5:       if A is numerical type then
6:          Q ← Q + t_α.Aθc, where θ ∈ {>, ≤, <, ≥}
7: for all t ∈ I do
8:    if t satisfies Q then
9:       sup(Q, I) ← sup(Q, I) + t
10: Let L_1 be the set of frequent predicates
11: for all Q ∈ Q do
12:    if |sup(Q, I)| = 0 then
13:       Γ ← Γ+¬(Q)
14:    else if |sup(Q,I)|/|I| ≥ τ then
15:       L_1 ← L_1 + {Q}
16: m ← 2
17: while L_{m-1} ≠ ∅ do
18:    for all c ∈ L_{m-1} do
19:       Σ ← FASTDC(sup(c, I),R)
20:       for all φ ∈ Σ do
21:          Γ ← Γ + φ, φ's predicates comes from c and φ
22:    C_m = {c|c = a ∪ b ∧ a ∈ L_{m-1} ∧ b ∈ ⋃ L_{k-1} ∧ b ∉ a}
23:    for all c ∈ C_m do
24:       scan the database to get the support of c, sup(c, I)
25:       if |sup(c, I)| = 0 then
26:          Γ ← Γ + φ, φ's predicates consist of predicates in c
27:       else if |sup(c,I)|/|I| ≥ τ then
28:          L_m ← L_m + c
29:    m ← m + 1
```

6. RANKING DCS

Though our FASTDC (C-FASTDC) is able to prune trivial, non-minimal, and implied DCs, the number of DCs returned can still be too large. To tackle this problem, we propose a scoring function to rank DCs based on their size and their support from the data. Given a DC φ, we denote by $Inter(\varphi)$ its *interestingness* score.

We recognize two different dimensions that influence $Inter(\varphi)$: *succinctness* and *coverage* of φ, which are both defined on a scale between 0 and 1. Each of the two scores represents a different yet important intuitive dimension to rank discovered DCs.

Succinctness is motivated by the Occam's razor principle. This principle suggests that among competing hypotheses, the one that makes fewer assumptions is preferred. It is also recognized that overfitting occurs when a model is excessively complex [6].

Coverage is also a general principle in data mining to rank results [2]. They design scoring functions that measure the statistical significance of the mining targets in the input data.

Given a DC φ, we define the interestingness score as a linear weighted combination of the two dimensions: $Inter(\varphi) = a \times Coverage(\varphi) + (1-a) \times Succ(\varphi)$.

6.1 Succinctness

Minimum description length (MDL), which measures the code length needed to compress the data [6], is a formalism to realize the Occam's razor principle. Inspired by MDL, we measure the length of a DC $Len(\varphi)$, and we define the *succinctness* of a DC φ, i.e., $Succ(\varphi)$, as the minimal possible length of a DC divided by $Len(\varphi)$ thus ensuring the scale of $Succ(\varphi)$ is between 0 and 1.

$$Succ(\varphi) = \frac{\text{Min}(\{Len(\phi)|\forall\phi\})}{Len(\varphi)}$$

One simple heuristic for $Len(\varphi)$ is to use the number of predicates in φ, i.e., $|\varphi.Pres|$. Our proposed function computes the length of a DC with a finer granularity than a simple counting of the predicates. To compute it, we identify the alphabet from which DCs are formed as $\mathbb{A} = \{t_\alpha, t_\beta, \mathbb{U}, \mathbb{B}, Cons\}$, where $\mathbb{U}$ is the set of all attributes, $\mathbb{B}$ is the set of all operators, and $Cons$ are constants. The length of φ is the number of symbols in $\mathbb{A}$ that appear in φ: $Len(\varphi) = |\{a|a \in \mathbb{A}, a \in \varphi\}|$. The shortest possible DC is of length 4, such as c_5, c_9, and $\neg(t_\alpha.SAL \leq 5000)$.

EXAMPLE 13. *Consider a database schema R with two columns A, B, with 3 DCs as follows:*
$c_{14}: \neg(t_\alpha.A = t_\beta.A)$, $c_{15}: \neg(t_\alpha.A = t_\beta.B)$,
$c_{16}: \neg(t_\alpha.A = t_\beta.A \wedge t_\alpha.B \neq t_\beta.B)$
Len(c_{14}) = 4 < Len(c_{15}) = 5 < Len(c_{16}) = 6. Succ(c_{14}) = 1, Succ(c_{15}) = 0.8, and Succ(c_{16})=0.67. However, if we use $|\varphi.Pres|$ as Len(φ), Len(c_{14}) = 1 < Len(c_{15}) = 1 < Len(c_{16}) = 2, and Succ(c_{14})=1, Succ(c_{15})=1, and Succ(c_{16})=0.5.

6.2 Coverage

Frequent itemset mining recognizes the importance of measuring statistical significance of the mining targets [2]. In this case, the support of an itemset is defined as the proportion of transactions in the data that contain the itemset. Only if the support of an itemset is above a threshold, it is considered to be frequent. CFDs discovery also adopts such principle. A CFD is considered to be interesting only if their support in the data is above a certain threshold, where support is in general defined as the percentage of single tuples that match the constants in the patten tableaux of the CFDs [8, 15].

However, the above statistical significance measures requires the presence of constants in the mining targets. For example, the frequent itemsets are a set of items, which are constants. In CFDs discovery, a tuple is considered to support a CFD if that tuple matches the constants in the CFD. Our target DCs may lack constants, and so do FDs. Therefore, we need a novel measure for statistical significance of discovered DCs on I that extends previous approaches.

EXAMPLE 14. *Consider c_2, which is a FD, in Example 1. If we look at single tuples, just as the statistical measure for CFDs, every tuple matches c_2 since it does not have constants. However, it is obvious that the tuple pair $\langle t_4, t_7\rangle$ gives more support than the tuple pair $\langle t_2, t_6\rangle$ because $\langle t_4, t_7\rangle$ matches the left hand side of c_2.*

Being a more general form than CFDs, DCs have more kinds of evidence that we exploit in order to give an accurate measure of the statistical significance of a DC on I. An *evidence* of a DC φ is a pair of tuples that does not violate φ: there exists at least one predicate in φ that is not satisfied by the tuple pair. Depending on the number of satisfied predicates, different evidences give different support to the statistical significance score of a DC. The larger the number of satisfied predicates is in a piece of evidence, the more support it gives to the interestingness score of φ. A pair of tuples satisfying k predicates is a *k-evidence* (kE). As we want to give higher score to high values of k, we need a weight to reflect this intuition in the scoring function. We introduce $w(k)$ for kE, which is from 0 to 1, and increases with k. In the best case, the maximum k for a DC φ is equal to $|\varphi.Pres| - 1$, otherwise the tuple pair violates φ.

DEFINITION 5. *Given a DC φ:*
A k-evidence (kE) for φ w.r.t. a relational instance I is a tuple pair $\langle t_x, t_y\rangle$, where k is the number of predicates in φ that are satisfied by $\langle t_x, t_y\rangle$ and $k \leq |\varphi.Pres| - 1$.
The weight for a kE ($w(k)$) for φ is $w(k) = \frac{(k+1)}{|\varphi.Pres|}$.

EXAMPLE 15. *Consider c_7 in Example 3, which has 2 predicates. There are two types of evidences, i.e., 0E and 1E.*
$\langle t_1, t_2\rangle$ is a 0E since $t_1.FN \neq t_2.FN$ and $t_1.GD = t_2.GD$.
$\langle t_1, t_3\rangle$ is a 1E since $t_1.FN \neq t_3.FN$ and $t_1.GD \neq t_3.GD$.
$\langle t_1, t_6\rangle$ is a 1E since $t_1.FN = t_6.FN$ and $t_1.GD = t_6.GD$.
Clearly, $\langle t_1, t_3\rangle$ and $\langle t_1, t_6\rangle$ have higher weight than $\langle t_1, t_2\rangle$.

Given such evidence, we define $Coverage(\varphi)$ as follows:

$$Coverage(\varphi) = \frac{\sum_{k=0}^{|\varphi.Pres|-1} |kE| * w(k)}{\sum_{k=0}^{|\varphi.Pres|-1} |kE|}$$

The enumerator of $Coverage(\varphi)$ counts the number of different evidences weighted by their respective weights, which is divided by the total number of evidences. $Coverage(\varphi)$ gives a score between 0 and 1, with higher score indicating higher statistical significance.

EXAMPLE 16. *Given 8 tuples in Table 1, we have 8*7=56 evidences. $Coverage(c_7) = 0.80357$, $Coverage(c_2) = 0.9821$. It can be seen that coverage score is more confident about c_2, thus reflecting our intuitive comparison between c_2 and c_7 in Section 1.*
Coverage for CDC is calculated using the same formula, such as $Coverage(c_3) = 1.0$.

6.3 Rank-aware Pruning in DFS Tree

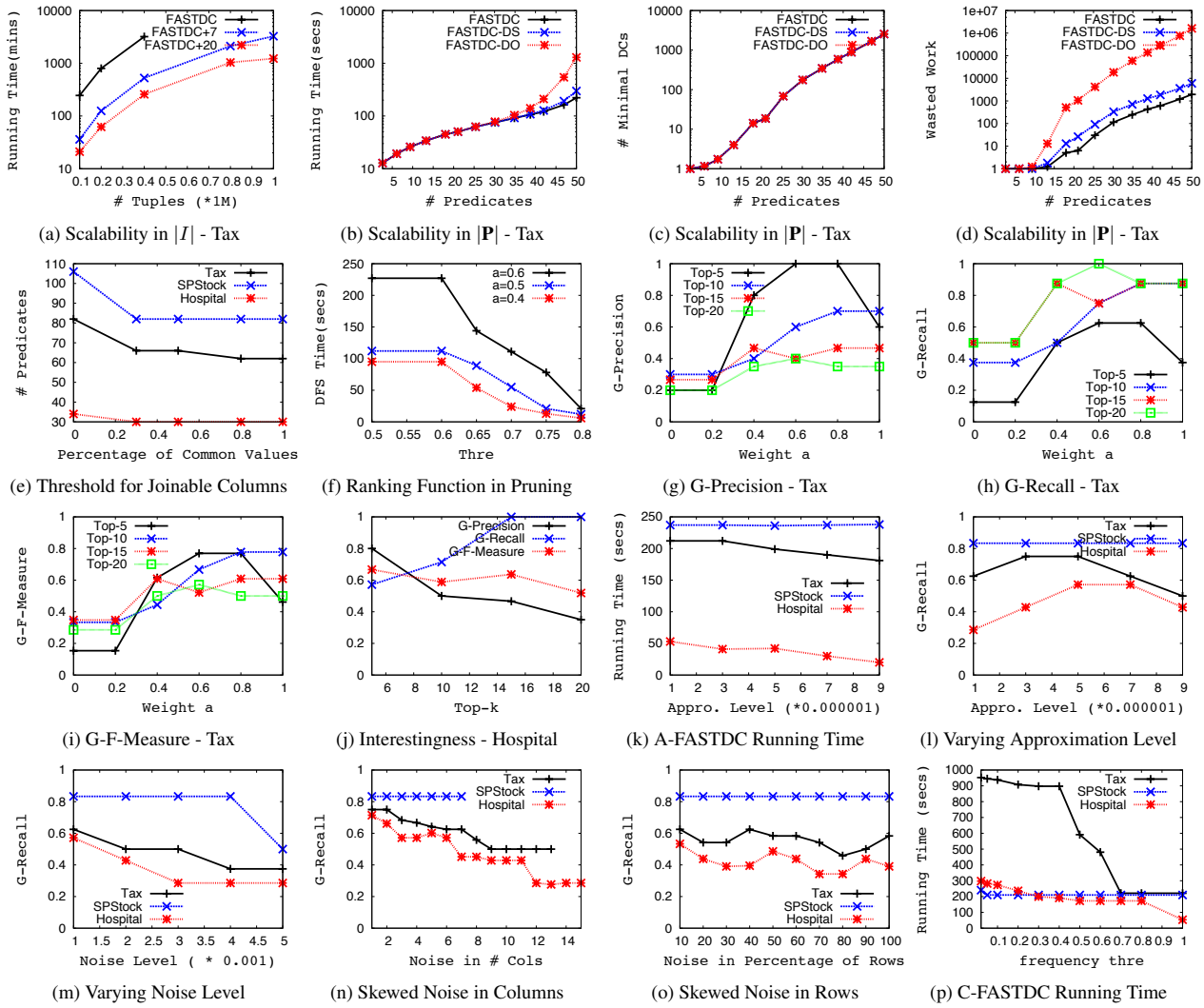
Having defined $Inter$, we can use it to prune branches in the DFS tree when searching for minimal covers in Algorithm 4. We can prune any branch in the DFS tree, if we can upper bound the $Inter$ score of any possible DC resulting from that branch, and the upper bound is either (i) less than a minimal $Inter$ threshold, or (ii) less than the minimal $Inter$ score of the Top-k DCs we have already discovered. We use this pruning in Algorithm 4 (Lines 11-12), a branch with the current path $\mathbf{X}$ will result in a DC φ: $\neg(\overline{\mathbf{X}} \wedge \mathbf{W})$, with $\mathbf{X}$ known and $\mathbf{W}$ unknown.

$Succ$ score is an anti-monotonic function: adding more predicates increases the length of a DC, thus decreases the $Succ$ of a DC. Therefore we bound $Succ(\varphi)$ by $Succ(\varphi) \leq Succ(\neg(\overline{\mathbf{X}}))$. However, as $Coverage(\varphi)$ is not anti-monotonic, we cannot use $\neg(\overline{\mathbf{X}})$ to get an upper bound for it. A direct upper bound, but not useful bound is 1.0, so we improve it as follows. Each evidence E or tuple pair is contributing $w(k) = \frac{(k+1)}{|\varphi.Pres|}$ to $Coverage(\varphi)$ with k being the number of predicates in φ that E satisfies. $w(k)$ can be rewritten as $w(k) = 1 - \frac{l}{|\varphi.Pres|}$ with l being the number of predicates in φ that E does not satisfy. In addition, we know l is greater than or equal to the number of predicates in $\overline{\mathbf{X}}$ that E does not satisfy; and we know that $|\varphi.Pres|$ must be less than the $\frac{|\mathbf{P}|}{2}$. Therefore, we get an upper bound for $w(k)$ for each evidence. The average of the upper bounds for all evidences is a valid upper bound for $Coverage(\varphi)$. However, to calculate this bound, we need to iterate over all the evidences, which can be expensive because we need to do that for every branch in the DFS tree. Therefore, to get a tighter bound than 1.0, we only upper bound the $w(k)$ for a small number of evidences[4], and for the rest we set $w(k) \leq 1$. We show in the experiments how different combinations of the upper bounds of $Succ(\varphi)$ and of $Coverage(\varphi)$ affect the results.

7. EXPERIMENTAL STUDY

We experimentally evaluate FASTDC, $Inter$ function, A-FASTDC, and C-FASTDC. Experiments are performed on a Win7 machine with QuadCore 3.4GHz cpu and 4GB RAM. The scalability experiment runs on a cluster consisting of machines with the same configuration. We use one synthetic and two real datasets.

[4]We experimentally identified that 1000 samples improve the upper bound without affecting execution times.



(a) Scalability in $|I|$ - Tax (b) Scalability in $|\mathbf{P}|$ - Tax (c) Scalability in $|\mathbf{P}|$ - Tax (d) Scalability in $|\mathbf{P}|$ - Tax

(e) Threshold for Joinable Columns (f) Ranking Function in Pruning (g) G-Precision - Tax (h) G-Recall - Tax

(i) G-F-Measure - Tax (j) Interestingness - Hospital (k) A-FASTDC Running Time (l) Varying Approximation Level

(m) Varying Noise Level (n) Skewed Noise in Columns (o) Skewed Noise in Rows (p) C-FASTDC Running Time

Figure 3: FASTDC scalability (a-f), ranking functions w.r.t. Σ_g (g-j), A-FASTDC scalability (k) and quality (l-o), C-FASTDC scalability (p).

Synthetic. We use the *Tax* data generator from [7]. Each record represents an individual's address and tax information, as in Table 1. The address information is populated using real semantic relationship. Furthermore, salary is synthetic, while tax rates and tax exemptions (based on salary, state, marital status and number of children) correspond to real life scenarios.

Real-world. We use two datasets from different Web sources[5].

Hospital data is from the US government. There are 17 string attributes, including Provider # (PN), measure code (MC) and name (MN), phone (PHO), emergency service (ES) and has 115k tuples.

SP Stock data is extracted from historical S&P 500 Stocks. Each record is arranged into fields representing Date, Ticker, Open Price, High, Low, Close, and Volume of the day. There are 123k tuples.

7.1 Scalability Evaluation

We mainly use the Tax dataset to evaluate the running time of FASTDC by varying the number of tuples $|I|$, and the number of predicates $|\mathbf{P}|$. We also report running time for the Hospital and the SP Stock datasets. We show that our implication testing algorithm, though incomplete, is able to prune a huge number of implied DCs.

[5] http://data.medicare.gov, http://pages.swcp.com/stocks

Algorithms. We implemented FASTDC in Java, and we test various optimizations techniques. We use FASTDC+M to represent running FASTDC on a cluster consisting of M machines. We use FASTDC-DS to denote running FASTDC without dividing the space of DCs as in Section 5.4. We use FASTDC-DO to denote running FASTDC without dynamic ordering of predicates in the search tree as in Section 5.3.

Exp-1: Scalability in $|I|$. We measure the running time in minutes on all 13 attributes, by varying the number of tuples (up to 1 million tuples), as reported in Figure 3a. The size of the predicate space $|\mathbf{P}|$ is 50. The Y axis of Figure 3a is in log scale. We compare the running time of FASTDC and FASTDC+M with number of blocks B=2M to achieve load balancing. Figure 3a shows a quadratic trend as the computation is dominated by the tuple pairwise comparison for building the evidence set. In addition, Figure 3a shows that we achieve almost linear improvement w.r.t the number of machines on a cluster; for example, for 1M tuples, it took 3257 minutes on 7 machines, but 1228 minutes on 20 machines. Running FASTDC on a cluster is a viable approach if the number of tuples is too large to run on a single machine.

Exp-2: Scalability in $|\mathbf{P}|$. We measure the running time in seconds using 10k tuples, by varying the number of predicates through

including different number of attributes in the Tax dataset, as in Figure 3b. We compare the running time of FASTDC, FASTDC-DS, and FASTDC-DO. The ordering of adding more attributes is randomly chosen, and we report the average running time over 20 executions. The Y axes of Figures 3b, 3c and 3d are in log scale. Figure 3b shows that the running time increases exponentially w.r.t. the number of predicates. This is not surprising because the number of minimal DCs, as well as the amount of wasted work, increases exponentially w.r.t. the number of predicates, as shown in Figures 3c and 3d. The amount of wasted work is measured by the number of times Line 15 of Algorithm 4 is hit. We estimate the wasted DFS time as a percentage of the running time by *wasted work / (wasted work + number of minimal DCs)*, and it is less than 50% for all points of FASTDC in Figure3d. The number of minimal DCs discovered is the same for FASTDC, FASTDC-DS, and FASTDC-DO as optimizations do not alter the discovered DCs.

Hospital has 34 predicates and it took 118 minutes to run on a single machine using all tuples. Stock has 82 predicates and it took 593 minutes to run on a single machine using all tuples.

Exp-3: Joinable Column Analysis. Figure 3e shows the number of predicates by varying the % of common values required to declare joinable two columns. Smaller values lead to a larger predicate space and higher execution times. Larger values lead to faster execution but some DCs involving joinable columns may be missed. The number of predicates gets stable with low percentage of common values, and with our datasets the quality of the output is not affected when at least 30% common values are required.

Exp-4: Ranking Function in Pruning. Figure 3f shows the DFS time taken for the Tax dataset varying the minimum $Inter$ score required for a DC to be in the output. The threshold has to exceed 0.6 to have pruning power. The higher the threshold, the more aggressive the pruning. In addition, a bigger weight for $Succ$ score (indicated by smaller a in Figure 3f) has more pruning power. Although in our experiment golden DCs are not dropped by this pruning, in general it is possible that the upper bound of $Inter$ for interesting DCs falls under the threshold, thus this pruning may lead to losing interesting DCs. The other use of ranking function for pruning is omitted since it has little gain.

Dataset	# DCs Before	# DCs After	% Reduction
Tax	1964	741	61%
Hospital	157	42	73%
SP Stock	829	621	25%

Table 3: # DCs before and after reduction through implication.

Exp-5: Implication Reduction. The number of DCs returned by FASTDC can be large, and many of them are implied by others. Table 3 reports the number of DCs we have before and after implication testing for datasets with 10k tuples. To prevent interesting DCs from being discarded, we rank them according to their $Inter$ function. A DC is discarded if it is implied by DCs with higher $Inter$ scores. It can be seen that our implication testing algorithm, though incomplete, is able to prune a large amount of implied DCs.

7.2 Qualitative Analysis

Table 4 reports some discovered DCs, with their semantics explained in English[6]. We denote by Σ_g the golden VDCs that have been designed by domain experts on the datasets. Specifically, Σ_g for Tax dataset has 8 DCs; Σ_g for Hospital is retrieved from [10] and has 7 DCs; and Σ_g for SP Stock has 6 DCs. DCs that are implied by Σ_g are also golden DCs. We denote by Σ_s the DCs returned by FASTDC. We define *G-Precision* as the percentage of DCs in Σ_s that are implied by Σ_g, *G-Recall* as the number of DCs in Σ_s that are implied by Σ_g over the total number of golden DCs, and *G-F-Measure* as the harmonic mean of *G-Precision* and *G-Recall*. In order to show the effectiveness of our ranking function, we use the golden VDCs to evaluate the two dimensions of $Inter$ function in Exp-6, the performance of A-FASTDC in Exp-7. We evaluate C-FASTDC in Exp-8. However, domain experts might not be exhaustive in designing all interesting DCs. In particular, humans have difficulties designing DCs involving constants. We show with $U\text{-}Precision(\Sigma_s)$ the percentage of DCs in Σ_s that are verified by experts to be interesting, and we report the result in Exp-9. All experiments in this section are done on 10k tuples.

Exp-6: Evaluation of $Inter$ score. We report in Figures 3g– 3i G-Precision, G-Recall, and G-F-Measure for Tax, with Σ_s being the Top-k DCs according to $Inter$ by varying the weight a from 0 to 1. Every line is at its peak value when a is between 0.5 and 0.8. Moreover, Figure 3h shows that $Inter$ score with $a = 0.6$ for Top-20 DCs has perfect recall; while it is not the case for using $Succ$ alone ($a = 0$), or using $Coverage$ alone ($a = 1$). This is due to two reasons. First, $Succ$ might promote shorter DCs that are not true in general, such as c_7 in Example 3. Second, $Coverage$ might promote longer DCs that have higher coverage than shorter ones, however, those shorter DCs might be in Σ_g; for example, the first entry in Table 4 has higher coverage than $\neg(t_\alpha.AC = t_\beta.AC \wedge t_\alpha.PH = t_\beta.PH)$, which is actually in Σ_g. For Hospital, $Inter$ and $Coverage$ give the same results as in Figures 3j, which are better than $Succ$ because golden DCs for Hospital are all FDs with two predicates, therefore $Succ$ has no effect on the interestingness. For Stock, all scoring functions give the same results because its golden DCs are simple DCs, such as $\neg(t_\alpha.Low > t_\alpha.High)$.

This experiment shows that both succinctness and coverage are useful in identifying interesting DCs. We combine both dimensions into $Inter$ with $a = 0.5$ in our experiments. Interesting DCs usually have $Coverage$ and $Succ$ greater than 0.5.

Exp-7: A-FASTDC. In this experiment, we test A-FASTDC on noisy datasets. A noise level of α means that each cell has α probability of being changed, with 50% chance of being changed to a new value from the active domain and the other 50% of being changed to a typo. For a fixed noise level $\alpha = 0.001$, which will introduce hundreds of violating tuple pairs for golden DCs, Figure 3l plots the G-Recall for Top-60 DCs varying the approximation level ϵ. A-FASTDC discovers an increasing number of correct DCs as we increase ϵ, but, as it further increases, G-Recall drops because when ϵ is too high, a DC whose predicates are a subset of a correct DC might get discovered, thus the correct DC will not appear. For example, the fifth entry in Table 4 is a correct DC; however, if ϵ is set too high, $\neg(t_\alpha.PN = t_\beta.PN)$ would be in the output. G-Recall for SPStock data is stable and higher than the other two datasets because most golden DCs for SPStock data are one tuple DCs, which are easier to discover. Finally, we examine Top-60 DCs to discover golden DCs, which is larger than Top-20 DCs in clean datasets. However, since there are thousands of DCs in the output, our ranking function is still saving a lot of manual verification.

Figure 3m shows that for a fixed approximate level $\epsilon = 4 \times 10^{-6}$, as we increase the amount of noise in the data, the G-Recall for Top-60 DCs shows a small drop. This is expected because the nosier gets the data, the harder it is to get correct DCs. However, A-FASTDC is still able to discover golden DCs.

Figure 3n and 3o show how A-FASTDC performs when the noise is skewed. We fix 0.002 noise level, and instead of randomly distributing them over the entire dataset, we distribute them over a certain region. Figure 3n shows that, as we distribute the noise over

[6]All datasets, as well as their golden and discovered DCs are available at "`http://da.qcri.org/dc/`".

	Dataset	DC Discovered	Semantics
1	Tax	$\neg(t_\alpha.ST = t_\beta.ST \wedge t_\alpha.SAL < t_\beta.SAL \wedge t_\alpha.TR > t_\beta.TR)$	There cannot exist two persons who live in the same state, but one person earns less salary and has higher tax rate at the same time.
2	Tax	$\neg(t_\alpha.CH \neq t_\beta.CH \wedge t_\alpha.STX < t_\alpha.CTX \wedge t_\beta.STX < t_\beta.CTX)$	There cannot exist two persons with both having CTX higher than STX, but different CH. *If a person has CTX, she must have children.*
3	Tax	$\neg(t_\alpha.MS \neq t_\beta.MS \wedge t_\alpha.STX = t_\beta.STX) \wedge t_\alpha.STX > t_\alpha.CTX)$	There cannot exist two persons with same STX, one person has higher STX than CTX and they have different MS. *If a person has STX, she must be single.*
4	Hospital	$\neg(t_\alpha.MC = t_\beta.MC \wedge t_\alpha.MN \neq t_\beta.MN)$	Measure code determines Measure name.
5	Hospital	$\neg(t_\alpha.PN = t_\beta.PN \wedge t_\alpha.PHO \neq t_\beta.PHO)$	Provider number determines Phone number.
6	SP Stock	$\neg(t_\alpha.Open > t_\alpha.High)$	The open price of any stock should not be greater than its high of the day.
7	SP Stock	$\neg(t_\alpha.Date = t_\beta.Date \wedge t_\alpha.Ticker = t_\beta.Ticker)$	Ticker and Date is a composite key.
8	Tax	$\neg(t_\alpha.ST =$ 'FL' $\wedge\, t_\alpha.ZIP < 30397)$	State Florida's ZIP code cannot be lower than 30397.
9	Tax	$\neg(t_\alpha.ST =$ 'FL' $\wedge\, t_\alpha.ZIP \geq 35363)$	State Florida's ZIP code cannot be higher than 35363.
10	Tax	$\neg(t_\alpha.MS \neq$ 'S' $\wedge\, t_\alpha.STX \neq 0)$	One has to be single to have any single tax exemption.
11	Hospital	$\neg(t_\alpha.ES \neq$ 'Yes' $\wedge\, t_\alpha.ES \neq$ 'No')	The domain value of emergency service is yes or no.

Table 4: Sample DCs discovered in the datasets.

a larger number of columns, the G-Recall drops because noise in more columns affect the discovery of more golden DCs. Figure 3o shows G-Recall as we distribute the noise over a certain percentage of rows; G-Recall is quite stable in this case.

Exp-8: C-FASTDC. Figure 3p reports the running time of C-FASTDC varying minimal frequent threshold τ from 0.02 to 1.0. When $\tau = 1.0$, C-FASTDC falls back to FASTDC. The smaller the τ, the more the frequent constant predicate sets, the bigger the running time. For the SP Stock dataset, there is no constant predicate set, so it is a straight line. For the Tax data, $\tau = 0.02$ results in many frequent constant predicate sets. Since it is not reasonable for experts to design a set of golden CDCs, we only report U-Precision.

	FASTDC			C-FASTDC		
Dataset	k=10	k=15	k=20	k=50	k=100	k=150
Tax	1.0	0.93	0.75	1.0	1.0	1.0
Hospital	1.0	0.93	0.7	1.0	1.0	1.0
SP Stock	1.0	1.0	1.0	0	0	0
Tax-Noise	0.5	0.53	0.5	1.0	1.0	1.0
Hosp.-Noise	0.9	0.8	0.7	1.0	1.0	1.0
Stock-Noise	0.9	0.93	0.95	0	0	0

Table 5: U-Precision.

Exp-9: U-Precision. We report in Table 5 the U-Precision for all datasets using 10k tuples, and the Top-k DCs as Σ_s. We run FASTDC and C-FASTDC on clean data, as well as noisy data. For noisy data, we insert 0.001 noise level, and we report the best result of A-FASTDC using different approximate levels. For FASTDC on clean data, Top-10 DCs have U-precision 1.0. In fact in Figure 3g, Top-10 DCs never achieve perfect G-precision because FASTDC discovers VDCs that are correct, but not easily designed by humans, such as the second and third entry in Table 4. For FASTDC on noisy data, though the results degrade w.r.t. clean data, at least half of the DCs in Top-20 are correct. For C-FASTDC on either clean or noisy data, we achieve perfect U-Precision for the Tax and the Hospital datasets up to hundreds of DCs. SP Stock data has no CDCs. This is because C-FASTDC is able to discover many business rules such as entries 8-10 in Table 4, domain constraints such as entry 11 in Table 4, and CFDs such as c_3 in Example 1.

8. CONCLUSION AND FUTURE WORK

Denial Constraints are a useful language to detect violations and enforce the correct application semantics. We have presented static analysis for DCs, including three sound axioms, and a linear implication testing algorithm. We also developed a DCs discovery algorithm (FASTDC), as well as A-FASTDC and C-FASTDC. In addition, experiments shown that our interestingness score is effective in identifying meaningful DCs. In the future, we want to investigate sampling techniques to alleviate the quadratic complexity of computing the evidence set.

9. ACKNOWLEDGMENTS

The authors thank the reviewers for their useful comments.

10. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, 1993.

[3] M. Baudinet, J. Chomicki, and P. Wolper. Constraint-generating dependencies. *J. Comput. Syst. Sci.*, 59(1):94–115, 1999.

[4] O. Benjelloun, H. Garcia-Molina, H. Gong, H. Kawai, T. E. Larson, D. Menestrina, and S. Thavisomboon. D-swoosh: A family of algorithms for generic, distributed entity resolution. In *ICDCS*, 2007.

[5] L. E. Bertossi. *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers, 2011.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, 2006.

[7] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. *ICDE*, 2007.

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

[9] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. Tech. Report QCRI2013-1 at `http://da.qcri.org/dc/`.

[10] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, 2013.

[11] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *SIGMOD*, pages 240–251, 2002.

[12] D. Deroos, C. Eaton, G. Lapis, P. Zikopoulos, and T. Deutsch. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill, 2011.

[13] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Morgan & Claypool Publishers, 2012.

[14] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2), 2008.

[15] W. Fan, F. Geerts, J. Li, and M. Xiong. Discovering conditional functional dependencies. *IEEE TKDE*, 23(5):683–698, 2011.

[16] L. Golab, H. J. Karloff, F. Korn, D. Srivastava, and B. Yu. On generating near-optimal tableaux for conditional functional dependencies. *PVLDB*, 1(1):376–390, 2008.

[17] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

[18] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. In *SIGMOD*, pages 647–658, 2004.

[19] C. M. Wyss, C. Giannella, and E. L. Robertson. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *DaWaK*, 2001.