Pattern functional dependencies for data cleaning

Pattern Functional Dependencies for Data Cleaning

Abdulhakim Qahtan[★*] Nan Tang[♠] Mourad Ouzzani[♠] Yang Cao[♣] Michael Stonebraker[♦]
[★]Utrecht University [♠]Qatar Computing Research Institute [♣]University of Edinburgh [♦]MIT CSAIL
a.a.a.qahtan@uu.nl {ntang, mouzzani}@hbku.edu.qa
yang.cao@ed.ac.uk stonebraker@csail.mit.edu

ABSTRACT

Patterns (or regex-based expressions) are widely used to constrain the format of a domain (or a column), *e.g.,* a Year column should contain only four digits, and thus a value like "1980-" might be a typo. Moreover, integrity constraints (ICs) defined over multiple columns, such as (conditional) functional dependencies and denial constraints, *e.g.,* a ZIP code uniquely determines a city in the UK, have been widely used in data cleaning. However, a promising, but not yet explored, direction is to combine regex- and IC-based theories to capture data dependencies involving partial attribute values. For example, in an employee ID such as"F-9-107", "F" is sufficient to determine the finance department.

Inspired by the above observation, we propose a novel class of ICs, called *pattern functional dependencies* (PFDs), to model fine-grained data dependencies gleaned from partial attribute values. These dependencies cannot be modeled using traditional ICs, such as (conditional) functional dependencies, which work on entire attribute values. We also present a set of axioms for the inference of PFDs, analogous to Armstrong's axioms for FDs, and study the complexity of consistency and implication analysis of PFDs. Moreover, we devise an effective algorithm to automatically discover PFDs even in the presence of errors in the data. Our extensive experiments on 15 real-world datasets show that our approach can effectively discover valid and useful PFDs over dirty data, which can then be used to detect data errors that are hard to capture by other types of ICs.

PVLDB Reference Format:
Abdulhakim Qahtan, Nan Tang, Mourad Ouzzani, Yang Cao, Michael Stonebraker. Pattern Functional Dependencies for Data Cleaning. *PVLDB*, 13(5): 684 - 697, 2020.
DOI: https://doi.org/10.14778/3377369.3377377

1. INTRODUCTION

Functional dependencies (FDs) [4] and their different variants, *e.g.,* conditional functional dependencies (CFDs) [12], have been widely used in data cleaning and other data management tasks such as query optimization and data modeling. In addition, *patterns* (or regex-based expressions) are widely used to specify the format of a set of values in a given domain, *e.g.,* a Year column should contain *only* four digits. Nevertheless, all previous *integrity constraints* (ICs), including FDs and CFDs, are limited to work on the entire attribute values and do not exploit the intrinsic knowledge carried out by partial attribute values in the form of patterns.

We introduce ***pattern functional dependencies*** (PFDs), a new type of ICs that combines dependency- and regex-based theories.

1.1 Error Detection with Traditional ICs

Next, we discuss how traditional ICs, in particular FDs and CFDs, are used for detecting data errors.

Example 1: Consider two tables: D_1 with the schema (name, gender) in Table 1, and D_2 over the schema (zip, city) in Table 2, respectively. Erroneous cells, r_4[gender] in D_1 and s_4[city] in D_2, are annotated in pink. Their correct values, F and Los Angeles, are shown and highlighted in green.

[**FDs.**] Suppose the following FDs are defined on these tables:

φ_1 : **Name** ([name] → [gender])
φ_2 : **Zip** ([zip] → [city])
FDs

where φ_1 states that name uniquely determines gender in table Name, and φ_2 says that zip uniquely determines city in table Zip.

Clearly, φ_1 cannot detect the error r_4[gender] in D_1, because there is no other tuple r : (Susan Boyle, F) in D_1 – an FD requires two tuples to cause a violation [4]. Similarly, φ_2 cannot detect the error s_4[city] in D_2.

[**CFDs.**] One possible, but very expensive, way to detect errors in D_1 and D_2 is by using *many* constant CFDs, as shown below:

ϕ_1 : **Name** ([name = John Charles] → [gender = M])
ϕ_2 : **Name** ([name = John Bosco] → [gender = M])
ϕ_3 : **Name** ([name = Susan Orlean] → [gender = F])
ϕ_4 : **Name** ([name = Susan Boyle] → [gender = F])
ϕ_5 : **Zip** ([zip = 90001] → [city = Los Angeles])
ϕ_6 : **Zip** ([zip = 90002] → [city = Los Angeles])
ϕ_7 : **Zip** ([zip = 90003] → [city = Los Angeles])
ϕ_8 : **Zip** ([zip = 90004] → [city = Los Angeles])
CFDs

where ϕ_1 means that in table Name, if someone's name is John Charles, then his gender value should be M. The other

*Work done while at QCRI.

Proceedings of the VLDB Endowment, Vol. 13, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3377369.3377377

Table 1: D_1: Name

	name	gender
r_1:	John Charles	M
r_2:	John Bosco	M
r_3:	Susan Orlean	F
r_4:	Susan Boyle	M
		F

Table 2: D_2: Zip

	zip	city
s_1:	90001	Los Angeles
s_2:	90002	Los Angeles
s_3:	90003	Los Angeles
s_4:	90004	New York
		Los Angeles

constant CFDs (ϕ_2–ϕ_8) can be interpreted similarly. This method is *impractical* because it would amount to knowing the entire *ground truth* for all tuples. □

1.2 Key Observation

One *fundamental limitation* of previous ICs (such as FDs and CFDs), which will be addressed in this paper, is that they enforce data dependencies using the entire attribute values. Consequently, they cannot specify the fine-grained semantics found in partial attribute values. For example, given a column of full names, sometimes first names are actually sufficient, and more meaningful than full names, to determine the gender.

Our *key observation* is that by relaxing the limitation of previous FDs of operating on entire attribute values, we are able to introduce a new type of dependencies that can capture partial attribute values that follow some regex-like patterns. For example, in D_1, the first name is enough to determine gender, *e.g.,* John is a male and Susan is a female; and in D_2, the first three digits of zip, *e.g.,* 900, are sufficient to determine the city Los Angeles.

Example 2: Let us now consider a new type of pattern-based constraints:

λ_1 : **Name** ([name = John\␣\A*] → [gender = M])
λ_2 : **Name** ([name = Susan\␣\A*] → [gender = F])
λ_3 : **Zip** ([zip = 900\D{2}] → [city = Los Angeles])
PFDs

where λ_1/λ_2 states that if someone's first name is John/Susan, then the gender should be M/F (\A* matches any string, which will be defined later; and λ_3 says that if a five-digit zip code starts by 900, then the city is Los Angeles (\D{2} matches any two consecutive digits). Clearly, λ_2 can detect error r_4[gender] in D_1 and λ_3 can detect error s_4[city] in D_2.

Alternatively, consider two other constraints as follows:

λ_4 : **Name** ([name = $\overline{\text{\LU\LL*\␣}}$ \A*] → [gender])
λ_5 : **Zip** ([zip = $\overline{\text{\D{3}}}$ \D{2}] → [city])
PFDs

where λ_4 states that one's first name uniquely determines one's gender for table Name (assuming that name is written as first name followed by last name) (\LU matches any upper case letter and \LL* matches any consecutive lower case letters); and λ_5 states that the first 3 digits of a 5-digit zip code determines the city for table Zip. These two PFDs (λ_4 and λ_5) are defined over a pair of tuples, *e.g.,* two tuples match as specified by the left hand side (LHS) of λ_4 if they both satisfy the pattern \LU\LL*\␣\A*, and their first names are the same, which is enforced by $\overline{\text{\LU\LL*\␣}}$.

λ_4 can detect error r_4[gender] by comparing r_3 and r_4: they have the same first name Susan but different gender, which identifies a violation consisting of four cells (r_3[name], r_3[gender], r_4[name], r_4[gender]). Similarly, λ_5 can detect error s_4[city] by comparing s_4 with s_1, s_2, or s_3. □

Table 3: Real-world PFDs and Errors

Dependendcy	Pattern Tableau	Errors
Phone Number → State	$\overline{850}$\D{7} → FL	8505467600 — CA
	$\overline{607}$\D{7} → NY	6073771300 — PA
	$\overline{404}$\D{7} → GA	4048481918 — OK
	$\overline{217}$\D{7} → IL	2176163297 — TX
	$\overline{860}$\D{7} → CT	8602713444 — SC
Full Name → Gender	\A*,\␣ $\overline{\text{Donald}}$\A*→M	Holloway, Donald E. — F
	\A*,\␣ $\overline{\text{Stacey}}$\A* → F	Jones, Stacey R. — M
	\A*,\␣ $\overline{\text{David}}$ → M	Kimbell, David — F
	\A*,\␣ $\overline{\text{Jerry}}$\A* → M	Mallack, Jerry L. — F
	\A*,\␣ $\overline{\text{Alan}}$\A* → M	Otillio, Alan P. — F
ZIP → CITY	$\overline{6060}$\D → Chicago	60601 — Chicag 60603-6263 — C 60601 — Chciago
ZIP → STATE	$\overline{60}$\D{3} → IL	60603 — lL
	$\overline{95}$\D{3} → CA	95603 — MI

Remark. Specialized PFDs such as λ_1–λ_3 are more conservative, and more general PFDs such as λ_4–λ_5 are less conservative, potentially leading to false positives (*e.g.,* a unisex name cannot determine the gender). Also, and not surprisingly, real-world data is not homogeneous. Taking Boston as an example, the first three digits of a zip code in Boston could be either 201, 202, 203, or 204, not unique as in the case of Los Angeles.

1.3 Contributions

We summarize our notable contributions below.

1. We introduce PFDs, a new type of ICs, based on our key observation that data dependencies can be captured by partial attribute values (see real-world examples in Table 3). We also describe their semantics. (Section 2)
2. We provide an *inference system* for PFDs, similar to Armstrong's axioms for FDs, and provide an analysis over PFDs in terms of consistency and implication. (Section 3)
3. We devise an effective and efficient algorithm to automatically *discover PFDs from dirty datasets.* Note that, although profiling ICs from clean data [13, 3] has been widely studied, discovering them from dirty data is known to be much harder [5, 8]. (Section 4)
4. We conduct *extensive experiments* on 15 real-world datasets. The results show that our discovery algorithm can effectively find valid dependencies with an average precision and recall of 78% and 93%, respectively. Furthermore, it can detect errors that cannot be found by other FDs with an average precision of 65%. Table 3 shows sample PFDs (the dependencies and some of the tuples in their tableaux), as well as the errors that these PFDs were able to uncover. (Section 5)

Moreover, Section 6 discusses related work. Section 7 provides all proofs for Section 3. Section 8 closes this paper by providing concluding remarks and discussing future work.

2. PATTERN FUNCTIONAL DEPENDENCIES

2.1 Syntax

We first discuss (regex-like) patterns that we use for modeling partial attribute values. Intuitively, the class of general regular expressions could be used. However, this class is too large for our purpose. In addition, it complicates the problems

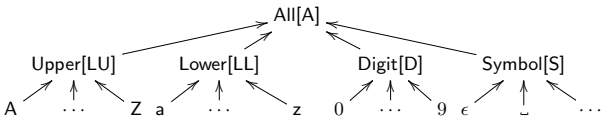

Figure 1: A Generalization Tree

(*i.e.,* high time complexity) of discovering and applying PFDs, *e.g.,* checking the equivalence of two regular expressions is PSPACE-complete [38]. Fortunately, for finding data-driven patterns, simple patterns are typically sufficient, as it has been shown in [18, 31].

Generalization Tree. A *generalization tree* is a tree defined over an alphabet Σ, where each leaf node is a character in Σ and each intermediate node is a generalization of its child nodes. The generalization tree used in this paper (Figure 1) contains upper case letters [A-Z], lower case letters [a-z], digits [0-9], and other symbols. Here, ϵ represents the empty string.

Patterns. A *pattern* P is a sequence of characters defined over the generalization tree. For strings α and β, $\alpha\{N\}$ means N repetitions of α, α & β is the logical **and** of α and β, $\alpha+$ means one-or-more repetitions, and the Kleene star operator $\alpha*$ denotes zero-or-more repetitions. We do not consider *recursive patterns* such as $(\alpha+)*$.

The benefits of employing a simple definition of patterns, in contrast to complicated regular expressions, are manifold: (1) they are easy to specify, (2) they are easy to discover, (3) they are easy to apply, (4) they are easy to reason about, and (5) most importantly, they are enough to detect most errors that more general regular expressions can find in practice. It will still be safe to use regular expressions to replace the patterns defined above; the semantics of PFDs and the axioms for PFD inference (Section 3.1) will remain the same. However, the complexity of reasoning, discovering, and applying PFDs will be much higher.

Pattern Matching. A string s *matches* a pattern P, denoted by $s \mapsto P$, if s is evaluated to be *true* by P; the value satisfies the pattern definition. For example, $90001 \mapsto \backslash\mathsf{D}\{5\}$. Also, it is easy to see that the patterns used in this paper can be converted to non-deterministic finite automata (NFAs) in polynomial time. Obviously, checking whether a string is accepted by a pattern, two patterns are equivalent, or whether one pattern is contained by another can be done in PTIME [9]. Only if a string is accepted by the NFA of a pattern, it is considered to match the pattern.

Constrained Patterns. Let R be a relation and $P = \backslash A * Q \backslash A*$ be a regular expression that generates a subset of the values in attribute $B \in R$. The pattern Q is called a *constrained pattern* and denoted by $\overline{Q}$ if $\forall s, s' \in B$, the portions of s and s' that match $\overline{Q}$ should be exactly the same. The main purpose of introducing constrained patterns is to model the *equivalence* of two strings when their constrained substrings are the same. For instance, the last two constraints in the motivating examples in the introduction define two constraints where the first name are the same and the first 3-digit are equal, respectively. More specifically, given two strings s and s', s and s' are *equivalent w.r.t.* a constrained pattern $\overline{Q}$, denotes by $s \equiv_{\overline{Q}} s'$, if $s(\overline{Q}) = s'(\overline{Q})$, where $s(\overline{Q})$ represents the portion of s that matches the expression $\overline{Q}$. It is worth noting that the constrained patterns provide better equivalence between strings than the simple approximate string matching.

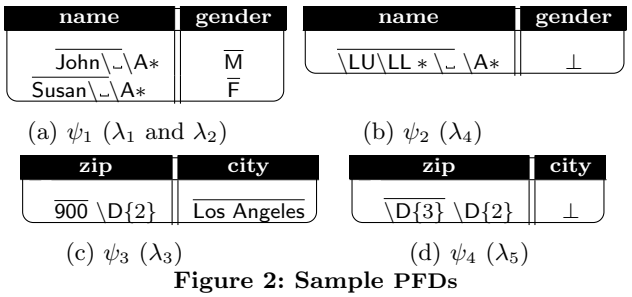

Figure 2: Sample PFDs

Example 3: [***Constrained Patterns.***] One sample constrained pattern is $\overline{Q} = \overline{\backslash\mathsf{LU}\backslash\mathsf{LL}*\backslash\␣}\ \backslash\mathsf{A}*$ from the constraint λ_4 presented in the introduction. It is used on the name attribute to enforce the matching over the first name. Another constrained pattern example is $\overline{Q'} = \overline{\backslash\mathsf{LU}\backslash\mathsf{LL}*\backslash\␣}\ \backslash\mathsf{A}*\overline{␣\backslash\mathsf{LU}\backslash\mathsf{LL}*}$, which can be used to enforce the matching over both the first name and the last name, but with an arbitrary number of middle names.

The embedded patterns of $\overline{Q}$ and $\overline{Q'}$ are $\backslash\mathsf{LU}\backslash\mathsf{LL}*\backslash␣\backslash\mathsf{A}*$ and $\backslash\mathsf{LU}\backslash\mathsf{LL}*\backslash␣\backslash\mathsf{A}*␣\backslash\mathsf{LU}\backslash\mathsf{LL}*$, respectively. Obviously, $Q' \subseteq Q$, *i.e.,* pattern Q' is contained by Q, and $\overline{Q'} \subseteq \overline{Q}$, *i.e.,* $\overline{Q'}$ is a restricted constrained pattern of $\overline{Q}$.

Consider two names in Table 1, $r_1[\mathsf{name}] = \mathsf{John\ Charles}$ and $r_2[\mathsf{name}] = \mathsf{John\ Bosco}$. We have $r_1[\mathsf{name}] \mapsto \overline{Q}$, $r_2[\mathsf{name}] \mapsto \overline{Q}$. Moreover, we have $r_1[\mathsf{name}] \equiv_{\overline{Q}} r_2[\mathsf{name}]$, because $r_1[\mathsf{name}](\overline{Q}) = \{\mathsf{John}\}$, $r_2[\mathsf{name}](\overline{Q}) = \{\mathsf{John}\}$, and $r_1[\mathsf{name}](\overline{Q}) \cap r_2[\mathsf{name}](\overline{Q}) = \{\mathsf{John}\} \neq \emptyset$. □

Restricting and Generalizing the Patterns. Given two constrained patterns $\overline{Q}$ and $\overline{Q'}$, we say that $\overline{Q}$ is a *restricted* pattern of $\overline{Q'}$, denoted by $\overline{Q} \subseteq \overline{Q'}$, if for any two strings s, s', $s \equiv_{\overline{Q}} s'$ implies $s \equiv_{\overline{Q'}} s'$. The pattern $\overline{Q'}$ is said to be a *generalized* pattern of $\overline{Q}$.

Example 4: [***Restricted Patterns.***] Consider zip code 90001 and two patterns $Q = \backslash\mathsf{D}\{5\}$ and $Q' = \backslash\mathsf{D}*$. We have $90001 \mapsto Q$, $90001 \mapsto Q'$, and $Q \subseteq Q'$. □

A special case of the string equivalence $s \equiv_{\overline{Q}} s'$ is when Q is a constant and $\overline{Q}$ is constraining this entire constant (*e.g.,* $\overline{\mathsf{M}}$ for the gender male). In such a case, strings s_1 and s_2 must be exactly the same (*e.g.,* M for male).

In the rest of the paper, we limit our discussions to the case of $\overline{Q}$ with *only* one constrained part (*e.g.,* $\overline{Q}$ but not $\overline{Q'}$, both from Example 3). This is observed based on our empirical study over many real-world datasets – more than one constrained part is not common in practice.

Pattern Functional Dependencies (PFDs). A PFD ψ defined over schema R is a pair $R(X \to Y, T_p)$, where:

1. X and Y are sets of attributes from R,
2. $X \to Y$ is a standard FD, called an *embedded* FD, and
3. T_p is a tableau with all attributes in X and Y, where for attribute A in X or Y and each tuple $t_p \in T_p$, $t_p[A]$ is either a constrained pattern that matches values in $\mathsf{dom}(A)$, or an unnamed variable '$\perp$' that is used as a wildcard.

Along the same notation convention of CFDs [12], we separate attributes X and Y of a tuple in T_p with '$\|$'. If $X \cap Y \neq \emptyset$, for each attribute $A \in X \cap Y$, we use A_L for the attribute A in X indicating the LHS, and A_R for the attribute A in Y indicating the right hand side (RHS, for short). For any PFD

to make sense when $X \cap Y \neq \emptyset$, we have $t_p[A_L] \subseteq t_p[A_R]$ for each $t_p \in T_p$. Also, we simply write ψ as $(X \to Y, T_p)$ when R is clear from the context.

Example 5: [***PFDs.***] The constraints λ_1–λ_5 given in the introduction can be expressed as PFDs (Figure 2). □

Remark. Data types can be classified into *quantitative values* that typically have meaning as a measurement (for example, a person's height) or a count (for example, a person's stock shares), and *qualitative values* that are non-statistical and hence cannot be aggregated (for example, one cannot compute the average value of two zip codes). Semantically and intuitively, (functional) data dependencies (*e.g.,* FDs [4], CFDs [12], fixing rules [41], and Sherlock rules [20]) apply only on qualitative values, not on quantitative values. In contrast, constraints on quantitative values (*e.g.,* metric dependencies [23], differential dependencies [37]) are not functional. PFDs are defined on qualitatively values.

2.2 Semantics

Intuitively, the tableau T_p of $\psi : R(X \to Y, T_p)$ constrains the FD $R(X \to Y)$ embedded in ψ by specifying data dependencies on partial attribute values in X and Y. In order for a PFD to hold, the constrained patterns in $t_p \in T_p$ should be matched by a predefined number of tuples that is called the support of the PFD. A data tuple $t \in T$ *matches* a tuple $t_p \in T_p$, denoted by $t \mapsto t_p$, if for each attribute $A \in X \cup Y$, $t[A]$ can be generated by the regular expression in $t_p[A]$.

A table T *satisfies* a PFD $\psi(X \to Y, T_p)$, denoted by $T \vDash \psi$, when: $\forall t_p \in T_p$, if $\exists (t_1, t_2) \in T$ such that:

$$t_1[A] \mapsto t_p[A] \wedge t_2[A] \mapsto t_p[A] \wedge t_1[A] \equiv_{t_p[A]} t_2[A], \quad \forall A \in X$$

then:

$$\forall B \in Y : t_1[B] \mapsto t_p[B] \wedge t_2[B] \mapsto t_p[B] \wedge t_1[B] \equiv_{t_p[B]} t_2[B].$$

That is, if the LHS values, $t_1[X]$ and $t_2[X]$, are equivalent as constrained by $t_p[X]$, then their RHS values, $t_1[Y]$ and $t_2[Y]$, should also be equivalent as specified by $t_p[Y]$. If for a tuple t_p in the tableau T_p, the constrained parts only contain constants (such as $\overline{\mathsf{John}\backslash\_}\backslash\mathsf{A}*$), a PFD can be applied on a single tuple. That is, for each tuple t in T and for each attribute $A \in X$, if $t[A] \mapsto t_p[A]$, then for each attribute $B \in Y$, $t[B] \mapsto t_p[B]$. Moreover, if Ψ is a set of PFDs, we write $T \vDash \Psi$ if $T \vDash \psi$ for each PFD $\psi \in \Psi$.

Example 6: Consider two tables Name (Table 1) and Zip (Table 2), and the PFDs given in Figure 2.

[***Violations of a Single Tuple.***] Tuple r_1 in Table 1 satisfies PFD ψ_1 ($r_1 \vDash \psi_1$), but r_4 violates ψ_1 ($r_4 \nvDash \psi_1$) because r_4's first name is Susan but r_4[gender] is M while it should be F.

[***Violations of Two Tuples.***] Tuples (r_1, r_2) satisfy PFD ψ_2, but tuples (r_3, r_4) violates ψ_2 (they have the same first name Susan but different gender, F for r_3 and M for r_4, which causes a violation). Similarly, tuples (s_1, s_2) in Table 2 satisfy the PFD ψ_4 in Figure 2, but (s_1, s_4), (s_2, s_4) and (s_3, s_4) violate ψ_4. □

As shown in Example 6, both ψ_1 and ψ_2 can detect the erroneous cell r_4[gender]. Seemingly, ψ_2 is more general than ψ_1. However, there are two notable cases. First, if r_3 is not present in Table 1, ψ_1 can still detect r_4[gender] as an error, but ψ_2 cannot, because there is no enough redundancy in the data. Second, ψ_2 may detect violations if Table 1 has many unisex name such as Kim, which are actually not errors. Hence, the generalization of patterns is a double-edged sword and should be used carefully – an expert could validate a generalized PFD *only* if she can ensure that there are no exceptional cases.

For brevity, we also introduce a normal form for PFDs where the RHS consists of a single attribute, as $R(X \to A, T_p)$, where X is a set of attributes and A is a single attribute, which are also commonly used by FDs and CFDs.

3. FUNDAMENTAL PROBLEMS

3.1 Inference Axioms for PFDs

We develop axioms for PFDs, analogous to Armstrong's Axioms for FDs [4] and the inference system for CFDs [12].

Let X, Y, Z be subsets of attributes from relation R, and A, B be attributes of R. For simplicity, we write XY for the union of two sets of attributes X and Y instead of $X \cup Y$, and XA for the union of a set of attribute X and one attribute A instead of $X \cup \{A\}$. Also, given a PFD $\psi : R(X \to Y, T_p)$, since tuples in T_p are independent from each other, it is sufficient to reason about $R(X \to Y, t_p)$ for each $t_p \in T_p$.

We are now ready to present the axioms for PFD inference, which are shown in Figure 3.

Reflexivity. If $A \in X$, then $\mathrm{R}(X \to A, t_p)$ where the constrained pattern for A in the LHS of the PFD (or A_L for short) is a restricted form of the constrained pattern for A in the RHS of the PFD (or A_R for short), *i.e.,* $t_p[A_L] \subseteq t_p[A_R]$.

Consider the Name table (Table 1), we have a PFD $\mathsf{Name}(\mathsf{name} \to \mathsf{name}, (\overline{\mathsf{John}}\backslash\mathsf{A}* \parallel \overline{\backslash\mathsf{LU}\backslash\mathsf{LL}*\backslash\_}\backslash\mathsf{A}*))$, meaning that two persons both having the first name John determines that they have the same first name.

Inconsistency-EFQ. Let S_B be a subset of the domain $\mathsf{dom}(B)$ of B. For a set Ψ of PFDs, we say that $B \in \mathsf{S}_B$ is *consistent w.r.t.* Ψ if for any value $c \in \mathsf{S}_B$, there exists an instance T with a tuple $t \in T$ such that $t[B] = c$ and $T \models \Psi$. $B \in \mathsf{S}_B$ is not consistent if for each B-value in S_B, such an instance T does not exist. We will study the consistency of PFDs in more detail in Section 3.2. The Inconsistency-EFQ axiom derives $R(B \to Y, t_p)$ for arbitrary attributes Y and their patterns if $t_p[B]$ is contained in S_B and $B \in \mathsf{S}_B$ is not consistent with the current set Ψ of PFDs. This is similar to the principle of explosion (EFQ) in the natural deduction system [6].

Augmentation. If $\mathrm{R}(X \to Y, t_p)$ and $A \in R$ and $A \notin XY$ then $\mathrm{R}(XA \to YA, t'_p)$, where the pattern does not change for X and Y, *i.e.,* $t'_p[XY] = t_p[XY]$, and the pattern for A in the LHS is the same as the pattern for A in the RHS, *i.e.,* $t'_p[A_L] = t'_p[A_R]$. The axiom states that adding a new attribute A (*i.e.,* $t'_p[A_L] = t'_p[A_R]$) does not change the basic dependency between X and Y.

Transitivity. Given $R(X \to Y, t_p)$ and $R(Y \to Z, t'_p)$, if $t_p[A] \subseteq t'_p[A]$ for each $A \in Y$, then $R(X \to Z, t''_p)$ with $t''_p[X] = t_p[X]$ and $t''_p[Z] = t'_p[Z]$. Note that transitivity applies only when patterns in t'_p for Y subsume patterns in t_p for Y.

Reduction. This axiom is carried over from CFDs [12], it is needed to remove useless attributes from the LHS of a PFD. Intuitively, given a PFD $R(XB \to A, t_p)$, if B is a wildcard and A is a constant, the constrained patterns on A must come from X and not from B. Hence, it is safe to drop attribute B from the LHS of such PFDs.

$$\textbf{Reflexivity}\ \frac{A \in X}{R(X \to A, t_p),\ \textbf{where}\ \ t_p[A_L] \subseteq t_p[A_R]}$$

$$\textbf{Inconsistency-EFQ}\ \frac{B \in \mathsf{S}_B \text{ is not consistent}}{R(B \to Y, t_p),\ \textbf{where}\ \ t_p[B] \subseteq \mathsf{S}_B}$$

$$\textbf{Augmentation}\ \frac{R(X \to Y, t_p) \quad A \notin XY}{R(XA \to YA, t'_p),\ \textbf{where}\ \ t'_p[XY] = t_p[XY]\ \textbf{and}\ \ t'_p[A_L] = t'_p[A_R]}$$

$$\textbf{Transitivity}\ \frac{R(X \to Y, t_p) \quad R(Y \to Z, t'_p) \quad t_p[A] \subseteq t'_p[A]\ \textbf{for}\ \ \textit{all}\ A \in Y}{R(X \to Z, t''_p),\ \textbf{where}\ \ t''_p[X] = t_p[X]\ \textbf{and}\ \ t''_p[Z] = t'_p[Z].}$$

$$\textbf{Reduction}\ \frac{R(XB \to A, t_p) \quad t_p[B] = \perp \quad t_p[A] \text{ is a constant}}{R(X \to A, t'_p),\ \textbf{where}\ \ t'_p[XA] = t_p[XA]}$$

$$\textbf{LHS-Generalization}\ \frac{R(XB \to Y, t_p) \quad R(XB \to Y, t'_p) \quad t_p[XY] = t'_p[XY]}{R(XB \to Y, t''_p),\ \textbf{where}\ \ t''_p[XY] = t[XY]\ \textbf{and}\ \ t''_p[B] = t_p[B] \cup t'_p[B]}$$

Figure 3: Inference Axioms for PFDs

LHS-Generalization. This axiom generalizes the patterns on the LHS of PFDs. More specifically, given two PFDs $R(XB \to Y, t_p)$ and $R(XB \to Y, t'_p)$ such that the patterns for XY in t_p and t'_p are identical, then it derives $R(XB \to Y, t''_p)$ that combines the patterns for B in t_p and t'_p, *i.e.,* $t''_p[B] = t_p[B] \cup t'_p[B]$. In other words, for any value s over B, $s \mapsto t''_p[B]$ if and only if $s \mapsto t_p[B]$ or $s \mapsto t'_p[B]$.

Remark. In contrast to prior work on inference axioms for FDs [4] and CFDs [12], Inconsistency-EFQ and LHS-generalization are new axioms dictated by the patterns introduced by PFDs. This justifies the novelty and fundamental difference between PFDs and prior work. The axioms of reflexivity, augmentation, and transitivity extend standard Armstrong's axioms for FDs, the axiom of reduction is extended from CFDs.

Implication. Given a set Ψ of PFDs and a PFD ψ, the implication problem for PFDs is to determine whether Ψ implies ψ, denoted by $\Psi \models \psi$, *i.e.,* whether for all instances T of R, if $T \models \Psi$ then $T \models \psi$.

Finite axiomatizability. PFDs are finitely axiomatizable. Indeed, the axioms in Figure 3 provide an inference system that is *sound* and *complete* for logical implication of PFDs.

Below we first formalize the notion of finite axiomatizability of PFDs, *i.e.,* soundness and completeness for logical implication of PFDs. We then prove that the inference axioms do provide a finite axiomatization of PFDs.

Over attributes U, let Ψ be a set of PFDs and ψ another PFD, A *proof* of ψ from Ψ using set $\mathcal{I}$ of axioms is a sequence of PFDs $\psi_1, \ldots, \psi_n = \psi (n \geq 1)$ such that for each $i \in [1, n]$, either (a) $\psi_i \in \Psi$, or (b) there exists a substitution for some rule $\rho \in \mathcal{I}$ such that ψ_i corresponds to the consequent of ρ such that for each PFD in the antecedent of ρ the corresponding PFD is in the set $\{\psi_j \mid 1 \leq j < i\}$.

The PFD ψ is *provable* from Ψ using $\mathcal{I}$ given U, denoted by $\Psi \vdash^{\mathcal{I}} \psi$, if there exists a proof of ψ from Ψ using $\mathcal{I}$.

$\mathcal{I}$ is *sound* for PFDs implication if $\Psi \vdash^{\mathcal{I}} \psi$ implies $\Psi \models \psi$. $\mathcal{I}$ is *complete* for PFD implication if $\Psi \models \psi$ implies $\Psi \vdash^{\mathcal{I}} \psi$.

Here $\mathcal{I}$ refers to the inference axioms in Fig. 3. We write $\Psi \vdash \psi$ instead of $\Psi \vdash^{\mathcal{I}} \psi$ when $\mathcal{I}$ is clear from the context.

Theorem 1: *The inference system $\mathcal{I}$ is sound and complete for logical implication of PFDs.* □

Please see Section 7.1 for the proof.

Table 4: Name Patterns

	$\widetilde{\text{name}}$	$\widetilde{\text{gender}}$
r'_1:	John	M
r'_2:	Susan	F

Table 5: Zip Patterns

	$\widetilde{\text{zip}}$	$\widetilde{\text{city}}$
s'_1:	900	Los Angeles

Based on Theorem 1, we give the complexity of logical implication of PFDs (please see Section 7.2 for a proof).

Theorem 2: *The implication of PFDs is coNP-complete.* □

3.2 Consistency

The *consistency* problem is to check whether there is a conflict given a set of ICs. As studied in CFDs [12], although a set of FDs is always consistent, a set of CFDs may be inconsistent. Similarly, PFDs may also be inconsistent, *e.g.,* any inconsistent set of CFDs is also a set of inconsistent PFDs. The *consistency problem* for PFDs is to determine, given a set Ψ of PFDs defined over a relational schema R, whether there exists a nonempty instance T of R such that $T \models \Psi$.

Theorem 3: *(a) The consistency of PFDs is NP-complete. (b) It remains NP-hard even if all domains are infinite.* □

Please see Section 7.3 for the proof.

Theorem 3 tells us that, in contrast to CFDs whose consistency is PTIME decidable when all domains are infinite [12], the consistency analysis for PFDs remains NP-hard even over infinite domains. This further highlights the fundamental difference between PFD and CFD.

4. DISCOVERING PFDS

Essentially, PFDs represent some latent *knowledge* that captures the dependencies between partial attribute values in a table. Consider Tables 1 and 2, the knowledge we want to discover is shown in Tables 4 and 5, respectively. Here, "$\widetilde{\ }$" denotes that the values in this column are substrings of the original column, *e.g.,* "$\widetilde{\text{name}}$" means that values in this column are substrings of the name values. This observation will serve as a guide for discovering PFDs.

4.1 The Brute-Force Solution

A simple method for discovering PFDs, from A to B, is to enumerate all combinations of substrings of $t[A]$ and $t[B]$ for each tuple t, group substrings of all A-values based on exact

string matching, and make a decision based on a function f using the information on the corresponding substrings in B.

Example 7: [***The Brute-force Solution.***] Assume that we want to find PFDs from name to gender (Table 1). A brute-force solution works as follows:

Step 1. Enumerate substrings. Enumerate all combinations of substrings of name and gender for each tuple. Considering r_1 : (John Charles, M), we have: (J, M), (o, M), ... (s, M) for the substrings of name of length 1, (Jo, M), ... (es, M) for the substrings of length 2, and so on, until (John Charles, M) for the entire string of name.

Step 2. Group common substrings on the LHS. Group the common substrings of attribute name, and record the corresponding RHS values using a bag semantics. We get (J, {M, M}) where the two gender values M are from r_1 and r_2, ..., (John, {M, M}), (Susan, {M, F}), and so on.

Step 3. Decide on PFDs. Let the function f be "If the number of distinct values on the RHS is at most three, and the majority value is at least 50%, then it forms a partial value dependency". This would produce good information such as (John, {M, M}) and (Susan, {M, F}) (*i.e.,* true positives), as well as bad information such as (J, {M, M}) and (e, {M, M, M}) (*i.e.,* false positives). □

Clearly, the brute-force approach does not work in practice due to the following challenges:

(**C1**.) *Huge number of attribute combinations.* The number of attribute combinations for $X \rightarrow Y$ is exponential for sets of attributes X and Y, and quadratic for single attributes $A \rightarrow B$, *w.r.t.* the number of attributes in relation R.

(**C2**.) *Huge number of substrings.* Given a string s_1, the number of its substrings is $|s_1|(|s_1|+1)/2$ where $|s_1|$ is the length of s_1. Given two strings s_1 and s_2, the number of substring combinations is $|s_1||s_2|(|s_1|+1)(|s_2|+1)/4$, for a single attribute PFD $A \rightarrow B$.

(**C3**.) *High recall but low precision.* While such an approach may find all correct partial dependencies, it may unavoidably include meaningless partial value dependencies as well.

4.2 Restrictions from Practical Perspectives

Before optimizing the aforementioned brute-force solution, we discuss some restrictions based on the insights we identified from the real-world datasets we have worked with.

(i) String Tokenization. Special characters, such as "–" in F-9-107 and "\␣" in John\␣ Charles, often provide strong signals to extract meaningful substrings. Hence, when these special characters (or signals) are present, we should leverage them to tokenize a string.

(ii) Report Dependencies with Minimum Coverage. The coverage of a PFD is the number of records that contain its patterns. Without any restrictions, we may always be able to find at least one PFD between any two attributes. Hence, we report a dependency between A and B only if the PFDs in the tableau accumulate a coverage above a set threshold. This restriction is based on the intuition that PFDs with high coverage give a stronger signal about the dependency.

(iii) Report PFDs with Large Support and Minimal Noise. We define two important parameters to reduce false positives: (a) the minimum support K, which represents the minimum number of records that the pattern should appear in to report the PFD that includes the pattern as a valid PFD and (b) δ, the ratio of allowed violations, which represents the ratio of the patterns which are different from the main pattern that may appear in the dependent attribute values. For example, if pattern p_1 appears in n records in the LHS and pattern p_2 appeared in more than $n-(\delta*100)$ in the RHS, we declare $p_1 \rightarrow p_2$ as a valid PFD for the embedded dependency.

(iv) Avoid Unnecessary Checks. A PFD $\psi : (X \rightarrow Y, T_p)$ can be decomposed to $\psi_i : (X \rightarrow B_i, T_{p_i}), \forall B \in Y$, restricting the right hand side to single attribute only so as to avoid unnecessary attribute combinations in the RHS. Moreover, PFDs of the type $\psi : (X \rightarrow A, T_p)$ when $A \in X$ are considered trivial dependencies. We ignore trivial PFDs in this work.

For generalizable PFDs, we used the attribute-set lattice from [19] to mine the PFDs at level $n+1$ of the lattice after pruning the sets of attributes based on the discovered dependencies in level n. The level number n represents the number of attributes that should be in the LHS of the PFD. In case of constant PFDs discovery, we ignore testing dependencies when the coverage of the frequent patterns in the combination of the attributes cannot be greater than the minimum coverage.

The above restrictions suggest the followings. Restriction (i) can significantly reduce the number of substrings to be considered, and thus addressing Challenge C2, and restrictions (ii, iii) reduces the number of false positives significantly. A positive side effect of the above restrictions is that it will increase the precision, without reducing the recall, by discarding many meaningless substrings, *i.e.,* and thus addressing Challenge C3.

Restriction (iv) tells us that we do not need to traverse the full lattice to check for dependencies, allowing us to avoid a reasonable number of unnecessary tests. Hence, restriction (iv) is used to tackle Challenge C1.

4.3 An Efficient Algorithm

The algorithm to discover PFDs by leveraging the above practical restrictions is shown in Figure 4. Given a table and a function to decide whether a set of values forms a PFD as input, it outputs a set of PFDs. The algorithm first profiles the data to prune attributes for which PFDs cannot be found (line 1). For example, we drop all columns with pure numerical (*i.e.,* quantitative) values. We then treat all remaining combinations of columns as potential dependencies for PFDs. Thus, the algorithm will be able to detect PFDs with multiple attributes on the LHS. The profiling process also decides whether to **Tokenize** or to use **NGrams** for each attribute to extract partial patterns (lines 2-3). Then we create an index (hash-based inverted list) for the patterns in the different attributes of the table (lines 5–11). The index stores the pattern and its position in the value as a key and the tuple ids in which the pattern appeared in that position. The patterns are extracted either using **Tokenize** or **NGrams** based on the decision made by the function "Tokenize_or_NGrams". **Tokenize** is based on restriction (i), mentioned earlier whereas **NGrams** takes an attribute value as input and outputs all the n-grams up to the length of the largest value in the column.

After that, for each candidate dependency, the algorithm checks whether there are patterns that can be used to form a PFD (lines 13–28). We pick the attribute A with the largest number of frequent patterns from attributes in the LHS of the candidate PFD to be our starting attribute. For each token/n-gram of $h[A]$ (line 16), we check all the tokens/n-grams in the

Algorithm Discover PFDs

Input: a relational table T,
a function f to make PFD decisions
a minimum coverage threshold γ

Output: a set Ψ of PFDs

```
/* Profile and prune T to obtain candidate dependencies in the
   form (X → B) in a lattice representation. */
1. Φ := CandidateDependecies(T)
2. for each A ∈ attr(T) do
3.    Tokenize_or_NGrams(A)
4. Ψ := ∅            /* the set of discovered PFDs */
5. H := ∅          /* a hash-based inverted list */
6. for each tuple t ∈ T do
7.   for each A ∈ attr(T) do
8.     for each u ∈ Tokenize(t[A])|NGrams(t[A]) do
9.       if (u, pos_u) ∉ H[A] then
10.        H[A].insert((u, pos_u), id_list[id(t)])
11.      else
12.        H[A].update((u, pos_u), id_list.append[id(t)])
13. for each φ : (X → B) ∈ Φ do
14.  T_p = ∅ for a new PFD ψ : (X → B, T_p)
15.  sort attributes of X according to the number of patterns
16.  for each entry h ∈ H[A] do
17.    ids = h[id_list]
18.    S_X = frequent itemset of patterns in X[ids]
19.    S_B = frequent itemset of patterns in B[ids]
20.    if f(S_X, S_B) is true then
21.          add a tuple t_p to T_p, w.r.t. entry S
22.  if coverage(T_p) ≥ γ then
23.    ψ_g :=generalize(ψ)
24.    if (ψ_g) is not null then
25.          remove the children of X in the lattice
26.          Ψ := Ψ ∪ {ψ_g}
27.    else
28.          Ψ := Ψ ∪ {ψ}
29. return Ψ
```

Figure 4: Algorithm for Discovering PFDs

remaining attributes of the LHS. The set of frequent patterns in the LHS attributes is represented by S_X in the algorithm. The algorithm also searches for the most frequent pattern in attribute B (the RHS) that appears in the same tuples as the pattern $h[A](u, pos_u)$. Let S_B represent the set of tuple ids in which the frequent pattern appears. The function $f(S_X, S_B)$ returns True if the length of $|S_X| = n$ is greater than the minimum support K and the length of S_B is greater than $n-(\delta*100)$ records. The PFDs for each candidate dependency ψ with a coverage greater than the minimum coverage will be checked for generalization and either the generalized PFD ψ_g or the constant PFD ψ will be added to the result (lines 22–28). The final result will be returned (line 29).

Since some of the constant PFDs discovered so far may be generalizable to variable PFDs, such as from Figure 2 (a) to Figure 2 (b), we further process ψ to check if a variable PFD ψ_g can be discovered. More specifically, for each attribute $A \in X$, `Generalize`(ψ) (line 23) checks all the constrained patterns in the PFDs from A and tries to find a general form that can represent all of them. If the general constrained pattern can be found, it is applied on all the values of the attribute even those in which the pattern frequency is less than the minimum support. The same process is applied on the dependent attribute B (the RHS) of ψ. If the general PFD is satisfied with a set of violations less than a given threshold, it is reported to replace the constant PFDs. The PFDs λ_4 and λ_5 in the introduction are examples of general (variable PFDs).

Example 8: Consider Table 6, the algorithm decides

Table 6: A Running Example for PFD Discovery

	name	country	gender
r_1:	Tayseer Fahmi	Egypt	F
r_2:	Tayseer Qasem	Yemen	M
r_3:	Tayseer Salem	Egypt	F
r_4:	Tayseer Saeed	Yemen	M
r_5:	Noor Wagdi	Egypt	M
r_6:	Noor Shadi	Yemen	F
r_7:	Noor Hisham	Egypt	M
r_8:	Noor Hashim	Yemen	F
r_9:	Esmat Qadhi	Yemen	M
r_{10}:	Esmat Farahat	Egypt	F

to use tokenization on the name attribute and n-grams on the country and gender attributes. Recall that the patterns are in the form $((u, pos_u), id\_list[\mathtt{id}(t)])$. Thus, $((\text{'Tayseer'}, 0), [r_1, r_2, r_3, r_4])$ and $((\text{'Fahmi'}, 2), [r_1])$ will be added to $\mathcal{H}$[name].In $\mathcal{H}$[country], all the patterns (extracted using n-grams) are reduced to only two patterns. This is because the patterns will be substrings of the full value and they appear exactly in the same set of records as the full value. For example, the entry $((\text{'Egy'}, 0), [r_1, r_3, r_5, r_7, r_{10}])$ is similar to $((\text{'Egypt'}, 0), [r_1, r_3, r_5, r_7, r_{10}])$ and the latter will be preferable as it is more expressive. So $\mathcal{H}$[country] will have two entries only $((\text{'Egypt'}, 0), [r_1, r_3, r_5, r_7, r_{10}])$ and $((\text{'Yemen'}, 0), [r_2, r_4, r_6, r_8, r_9])$. For $\mathcal{H}$[gender], it has only two entries, $((\text{'M'}, 0), [r_2, r_4, r_5, r_7, r_9])$ and $((\text{'F'}, 0), [r_1, r_3, r_6, r_8, r_{10}])$. Assuming $K = 2$ and $\delta = 5\%$, the algorithm will not be able to detect any single LHS PFDs. Thus, it tries to find multi-attribute LHS PFDs. For example, checking if [name, country] → gender, we find that country has a higher frequency in its patterns. Thus, we will start by the sub-table that contains 'Egypt' in the country and combine that with all the patterns in the name attribute. In the sub-table $[r_1, r_3, r_5, r_7, r_{10}]$, we will be able to discover two PFDs: λ_1 : ([name = Tayseer\␣\A∗, country = Egypt] → [gender = F]) and λ_2 : ([name = Noor\␣\A∗, country = Egypt] → [gender = M]). Similarly, for the sub-table $[r_2, r_4, r_5, r_7, r_9]$, the algorithm discovers two PFDs: λ_3 : ([name = Tayseer\␣\A∗, country = Yemen] → [gender = M]) and λ_4 : ([name = Noor\␣\A∗, country = Yemen] → [gender = F]).

After discovering λ_i ($i \in [1, 4]$), we find that these PFDs cover 80% of the records. The algorithm then looks for a general form of the discovered PFDs, which would be in this case λ : ([name = $\overline{\text{LU\LL∗\␣}}$ \A∗, country] → [gender]). Here the algorithm discovered that the full values in the country attribute and the first token of the name attribute, which are represented as a restricted pattern of a single uppercase letter followed by any number of lowercase letters, should be considered. It then applies the rule on the rest of the records which are r_9 and r_{10}. Since these records do not violate the general PFD λ, the algorithm reports the general PFD λ instead of the constant PFDs λ_i ($i \in [1, 4]$). □

Time Complexity. The algorithm is in $O(2^{|R|}n)$-time. The first loop (lines 4–10) runs $n|R|$ times where n is the number of tuples and $|R|$ is the number of attributes. The second loop (lines 11–25) iterates at most $O(2^{|R|}n)$ times to traverse the lattice and test the patterns in each attribute assuming the the max length of the values in a given attribute is constant.

4.4 Optimizations

Substring Pruning. Note that in the hash table, some entries could be substrings of other entries. Moreover, if they

also have the same set of tuples, then we do not need to keep all of them. Take Table 2 for example, the values 900, 9000, and 90000 are associated with tuples $\{s_1, s_2, s_3, s_4\}$. In this case, we pick the most specific one, *i.e.,* 90000.

Single Semantics. Given attributes A and B and assuming that values in A are homogeneous, if substrings of A-values can determine B-values, then most likely these substrings have one semantic explanation, *e.g.,* the first name in the first token determines gender, or the first digit of a zip code determines a city. This single semantics is often reflected by the *positional* information of the tokens. Hence, we can group many tokens from the same attribute based on their positions and pick the group with the largest size.

4.5 Selecting PFDs

The discovery of ICs is a data-driven mining approach with no guarantee that the discovered ICs are genuine. The basic reason is that the function f to decide whether a dependency is a PFD is syntactic, not semantic, and the result is thus data dependent. Obviously, our algorithm will produce both true positives and false positives, so do all other IC discovery algorithms [8]. However, these algorithms, including ours, are very valuable in practice, for several reasons:

(1) *Human Effort.* Compared with asking a human to manually provide PFDs, discovering candidate PFDs and then involving a human to select genuine ones is more practical in terms of the required human effort.

(2) *Automatic and Explainable Repairs.* Automatic data repairing is hard to operate in real applications if the repair algorithm is a *black-box* which cannot be explained. In case of wrong repairs, there is no explanation about why the mistake happened. Automatic and explainable repairs are widely used in industry, such as ETL rules, which are usually manually coded. Moreover, the explainability enables interactive debugging of the results [26], maintenance [7], and explicit specification of domain knowledge [16].

5. EXPERIMENTS

Datasets. We used 15 tables (five datasets from each repository) from **data.gov (GOV)** (an open data repository from the US government), **ChEMBL (CHE** (a public chemical database), and **University Data Warehouse (UDW)** (a private repository from the administration of a large university). Details are shown in the top part of Table 7. These tables have less than 10 attributes, with the primary purpose of making their manual annotation feasible.

Baselines. We consider two state-of-the-art algorithms for discovering FDs and CFDs, namely FDep [14] and CFDFinder [12, 13], respectively. We use the implementations provided by Metanome [28]. We use the default parameter setting, except for the confidence value, which was set to 0.995 instead of 1 to allow CFDFinder to discover CFDs over dirty data. All datasets and code are available at `https://github.com/daqcri/PFD_Experiments`. The demo delivered at SIGMOD [33] is available at `https://github.com/daqcri/PFD_Demo`.

5.1 PFD vs. CFD Discovery

In this experiment, we evaluate how good our method is in discovering dependencies that cannot be discovered by existing methods. We compared our method with the FDep [14] and CFDFinder [12, 13]. We manually verified the discovered dependencies. When attribute values are considered, we manually checked through external websites, *e.g.,* for names and genders we went to `https://gender-api.com/`, for zip code and city we used `https://pypi.org/project/uszipcode/`, and so forth We have also manually verified the discovered PFDs with 2-3 attributes on the LHS. Unfortunately, PFDs with multiple-attribute LHS is a very rare case. Hence, we focus on single LHS attribute PFDs in our experimental evaluation.

We fixed the minimum coverage to report a dependency to 10%, the allowed noise to 5%, and the minimum number of records that contain the pattern in each reported PFD to 5. These parameters are set empirically with the goal of allowing a trade-off between precision and recall. For example, from our experiments, the minimum support value $K \geq 4$ will result in almost 100% precision but a low recall.

We show in Table 7 the results in terms of the number of dependencies, precision, and recall for the three strategies (rows 1-3, 5-7, 9, 11-12). Note that we are counting the embedded dependencies, not the number of FDs, CFDs, or PFDs. We consistently discover more valid dependencies than FDep and CFDFinder with only two exceptions (T2 and T9). In 8 tables out of 15, all uncovered embedded dependencies are correct ($P = 100\%$), and $P \geq 64\%$ for all but two tables. In 9 tables, we were able to uncover all embedded dependencies ($R = 100\%$), and $R \geq 80\%$ for all but two tables. These dependencies (Table 3 shows some of them) can only be captured if one takes into account partial attribute values, which in our case, are expressed through the constrained patterns.

A set of the dependencies can also be expressed using variable PFDs. We should note that we were able to generalize a reasonable number of dependencies to variable PFDs This is due to the strong connection between the attributes such as Year → Date. The number of dependencies that are represented by variable PFDs are shown in Table 7 (row 10).

It should be noted that even though FDep and CFDFinder are revealing dependencies in the data, PFDs are showing the patterns in the data that is responsible for the dependency. For example, FDep reports (Full Name → Gender) because full name is almost a key which is reported to determine all the other attributes in the table. Our PFD method shows that the tokens that represent the first name are responsible for this dependency.

5.2 PFD Validation

In this experiment, we selected three embedded dependencies: $\{Full\ Name \rightarrow Gender\}$, $\{Fax \rightarrow State\}$ *and* $\{Zip \rightarrow City\}$, which we manually validated by consulting different web services. To this end, we extracted the pattern in each PFD and checked if that pattern actually determines the dependent value (we consider here only constant PFDs). More concretely, validating (Full Name → Gender) was performed by checking the gender associated with each first name using an API that retrieves the gender of each name (*e.g.,* from `https://gender-api.com/get?name=David`). For validating (Fax → State), we collected the first three digits of the Fax numbers in each state and matched the first three digits and the state in each PFD with those in the real-world. Finally, we validated (Zip → City) using the "uszipcode" package at `https://pypi.org/project/uszipcode/`.

Table shows the precision and coverage of our method. A few PFDs in the "Full Name → Gender" were reported as errors because we considered the names which might be

Table 7: PFD vs CFD Discovery: Precision, Recall, Runtime, and Error Detection Accuracy

	Row id	Metrics	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
Size		# Columns	9	9	7	6	9	5	5	5	7	7	7	8	7	9	7
		# Rows	6,704	1,077	306	920	9,101	2,409	812	9,536	1,200	858	33,727	42,715	105,748	22,485	42,226
FDep	1	# Dependencies	12	13	9	5	5	8	4	5	10	15	6	2	3	5	9
	2	Precision (P)	66.7%	38.46%	66.7%	80%	60%	50%	0%	20%	0%	20%	100%	50%	66.7%	100%	100%
	3	Recall (R)	42.1%	45.5%	60%	36.4%	60%	80%	0%	20%	0%	50%	42.9%	9.1%	18.2%	17.2%	50%
	4	Runtime (secs)	5.4	0.33	0.14	0.24	10.7	0.37	0.13	5.16	0.29	0.29	96.7	205.8	805.4	62.8	124.2
CFDFinder	5	# Dependencies	0	18	3	4	5	0	1	3	6	3	4	0	6	4	1
	6	Precision (P)	–	61.1%	0%	100%	0%	–	100%	100%	16.7%	37.8%	100%	–	85.7%	80%	100%
	7	Recall (R)	–	55%	0%	33.3%	0%	–	100%	60%	100%	60%	28.6	–	54.5%	13.8%	5.5%
	8	Runtime (secs)	89.5	8	0.5	0.6	154.4	0.8	0.4	12.3	1.3	1.6	291	2,529	1,277	2,236	580
PFD	9	# Dependencies	16	16	8	10	15	6	1	5	1	8	14	17	11	38	31
	10	Variable PFDs	8	12	8	6	1	2	0	2	0	1	6	4	3	8	8
	11	Precision (P)	100%	68.8%	100%	90%	33.3%	83.3%	100%	100%	100%	100%	100%	64.7%	100%	76.3%	51.6%
	12	Recall (R)	84.2%	100%	80%	81.8%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	88.9 %
	13	Runtime (secs)	125.6	11.4	2.39	8.05	27.17	4.3	0.26	32.2	0.58	4.78	155.7	598.7	224.8	263.8	374.9
Multi-LHS	14	Runtime (secs)	3276	348	36.1	15.1	689	4.3	0.26	91	0.58	5.15	2284	4729	1973	2773	6121
PFD	15	# Errors	0	8	0	13	18	0	2	5	0	31	0	6	20	43	8
	16	Precision (P)	–	37.5%	–	77%	77.7%	–	100%	40%	–	58.1%	–	100%	40%	86%	50%

Table 8: Precision and Coverage of Discovered PFDs

Dependency	# PFDs	Precision	Coverage
Full Name → Gender	401	97.1%	54.9%
Fax → State	176	98.3%	46%
Zip → City	26	100%	78.3%

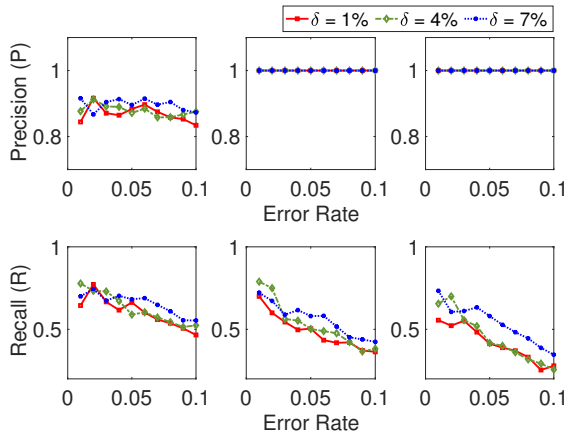

Figure 5: Effectiveness by Varying Error Rates {Zip →State}. Subfigures represent different values for the minimum support $K = 2$ (left), 4 (middle) and 6 (right). (Note: Injected errors are not from the same attribute's active domain.)

unisex names as errors even though the data shows that they are associated with a specific gender. Also, for (Fax → State), we noticed that some companies record the fax of their main branch for branches in other states which misled our PFD discovery algorithm. Overall, we achieved a very high precision (> 97%) for discovered PFDs.

We also used CFDFinder to discover dependencies between attributes using partial values (patterns) instead of full values. However, CFDFinder was unable to discover the dependencies until we lowered the confidence parameter to 0.85, in which case, it reported dependencies between each pair of attributes, leading to a large number of false positives.

5.3 Error detection

Errors in Real-world Data. In this experiment, we show how good the discovered and validated PFDs are at finding errors. Given a table R and a PFD $R(X \to Y, t_p)$, for each tuple t in R, if $t[A] \mapsto t_p[A]$ and $t[B] \neq t_p[B]$, then there is a violation of the PFD. When there is a violation of a PFD *w.r.t.* tuple t, the PFD will change $t[B]$ according to the PFD, which is then compared with the ground truth. Note that if $t[A]$ (*i.e.,* the LHS) is erroneous, the precision will be lowered. Because of the high accuracy of our methods in discovering the correct PFDs as shown in Table 7, we manually validated the dependencies and used the PFDs of each validated dependency to detect errors. Since we do not know all the errors that are present in each dataset, we only report precision. Again, we used the 15 datasets and run error detection using the manually validated dependencies.

Note that the university data has been manually curated multiple times and the ChEMBL database has already 24 versions and thus has gone through multiple curation steps. Discovering errors in such datasets is quite challenging. However, our PFD method was able to discover a set of errors that could not have been discovered otherwise. The results are shown in Table 7 (rows 14-15). We should note that we limited ourselves to PFDs for which we could decide, based on the knowledge available to us, whether the matching tuples contain an error. For example, $T10$ (pref_name → protein_class_desc, "Nicotinic acetylcholine receptor $\backslash A*$ → ion channel lgic ach chrn $\backslash A*$") is a valid PFD but we do not have access to the appropriate resources to check the reported errors. For the 10 tables on which we could report precision, we achieved 100% in 2 tables, then more than 77% in 3 tables, and more than 50% in 2 tables. The last 3 tables had a precision of less than 50%. This shows that in most cases, we are able to discover errors with a good precision. Examples of the discovered errors are shown in Table 3.

A Controlled Evaluation. In this experiment, we evaluate how good is our approach at discovering valid PFDs that can then be effective in discovering injected errors and how robust it is in the presence of dirty data. To this end, we selected the $\{Zip \to State\}$ dependency from one the datasets. We manually cleaned the errors by deleting the twelve records (tuples) that contain erroneous values out of a total of 924 records. Afterwards, we injected errors at varying rates from $1\%, 2\%, \ldots, 10\%$, to the "State" attribute. The injected values belong to the same attribute domain with changes that make them different from the original values. Since the attribute "State" contains values of 27 states, we considered two cases of noise: (i) outside the active domain: randomly select a value from the remaining 23 states, and (ii) from the active domain: select the value from the 26 states' abbreviations

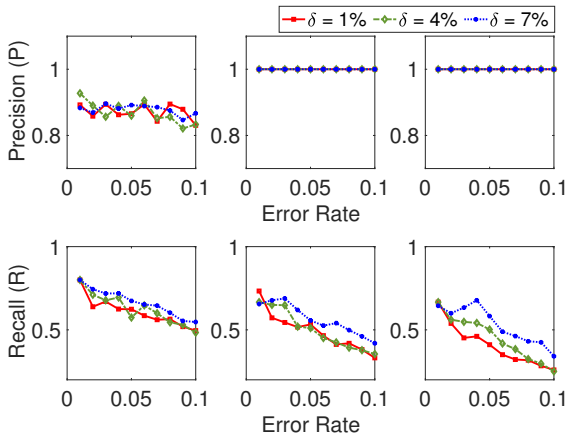

Figure 6: Effectiveness by Varying Error Rates {Zip →State}. Sub-figures represent the different values for the minimum support $K = 2$ (left), 4 (middle) and 6 (right). (Note: Injected errors are from the active domain of the same attribute.)

included in the attribute that differ from the current value. The second case is expected to confuse the PFD discovery algorithm. We then run our PFD discovery on the dirty data and use the discovered PFDs to detect the injected errors. In this experiment, we also take into account the effects of the parameters of our PFD discovery algorithm, namely the minimum support K and the ratio of allowed violations δ.

The results are shown in Figures 5 and 6, for errors injected from outside the active domain and from the active domain, respectively. We used three values for K: 2, 4, and 6. We make the following observations: (i) As K increases, the precision increases but the recall decreases. In fact, when $K = 6$ the precision reaches the highest value for both cases and irrespective of the injected error rate as we require a high support value to declare the presence of a PFD. However, the recall suffers significantly as many valid PFDs are discarded. (ii) Overall, large values for δ lead to better recall but lower precision except for the larger value of K. Increasing δ allows for the discovery of more PFDs and thus the ability to discover more violations with the side-effect of more false positives and thus lower precision. A large value of K compensates for the latter effect. The different values of δ are shown by the different curves in each sub-figure. (iii) Usually, selecting the injected errors from the active domain would make it conceptually harder to discover errors than when the injected values are not from the active domain. However, our method is robust enough so it is not affected much by the selection of the error source. (iv) As expected, increasing the error rate confuses the algorithm as the noise increases and many PFDs are ignored because the number of violations becomes greater than the allowed threshold. This reduces the recall significantly where we can see that the discovered errors becomes less than 30% when the error rate reaches 10%.

A question that might be asked here is: what would happen if we inject errors on the LHS attributes? In this case, the injected erroneous values might produce new PFDs or would reduce the frequency of the existing patterns to be bellow the minimum support K such that the algorithm will not be able to discover the PFD in the first place and thus will reduce the recall in discovering the errors. We did not include the results for injecting errors in the LHS due to space limitations.

5.4 Efficiency

As discussed in Section 4.3, our method requires extra time compared with other methods to deal with partial values. However, we reduced the required time by deploying multiple indexes where the first index stores the patterns in each attribute with the records' ids that include that pattern. The other index stores the records' ids and the attributes' ids together with all the patterns that appeared in the cell of that attribute and record. This allows for fast retrieval of the patterns and hence a shorter running time. Since our method utilizes patterns in discovering the PFDs, we ignore all the attributes with numerical values except the attributes with integer values that represent codes such as phone numbers, IDs and Zip code. The number of different lengths of the numerical values in attributes that represent code is significantly small and in most case values have the same length. For example, a zip code could have 5 digits or 9 digits and phone numbers have 10 digits. This heuristic allows us to avoid a large number of unnecessary checks.

Table 7 (rows 4,8, 13 and 14) reports the running time for FDep, CFDFinder and our PFD (both single and multiple LHS attributes) discovery. In general, FDep runs faster than CFDFinder, and CFDFinder runs faster than PFD, as expected; the discovered constraints are more complicated, as we move fromFDs to CFDs and then to PFDs. Our goal is to show that these algorithms can run in reasonable time, not to compare their runtimes, because they are used to discover different ICs.

6. RELATED WORK

Integrity Constraints. ICs have been widely studied for detecting data errors, such as FDs [4], CFDs [12], and denial constraints (DCs) [8]. PFDs differ from existing ICs in that PFDs are based on the key observation that partial attribute values, not necessarily entire values, are sufficient to determine values on correlated attributes. This fundamentally distinguishes PFDs from other ICs. Indeed, over infinite domains the consistency of PFDs is NP-complete while it is PTIME for CFDs and is always guaranteed for FDs.

Pattern-based Error Detection. In practice, many systems (for example, Trifacta [40] and NADEEF [10]) have predefined domain specific declarative rules to detect errors of wrong formats, such as phone numbers or email addresses. Instead of asking users to pre-define such rules, an interesting and practical idea is to automatically detect them. FAHES [31, 32] uses a similar idea that combines automatically discovered frequent patterns within a column and other statistics to detect disguised missing values. PFDs bridge ICs and pattern-based error detection methods for a single column, by allowing the use of patterns to enforce conditions across attributes, which were not previously used for ICs.

Other Error Detection Methods. There are also other types of tools for error detection: data transformation tools (*e.g.,* Data Wrangler [21] and BlinkFill [35]) that wrangle values within a column, quantitative error detection tools [42, 30] that expose outliers and glitches in the data, and also entity resolution for detecting duplicate data records [11, 36, 22, 27].

Despite all these efforts, detecting data errors with high accuracy is far from automatic [2, 17], and almost all practical tools (heavily) involve users to properly tune the parameters

and provide feedback. PFDs shine some light on automatically detecting data errors by carefully examining the correlation of partial values across different attributes.

Integrity Constraints Discovery. Due to the importance of ICs, many techniques and systems have been proposed for discovering different kinds of ICs, namely, FDs [25, 29, 43], temporal FDs [1], differential dependencies [37], conditional differential dependencies [24], CFDs [13, 34], and order dependencies [39]. These techniques work on entire values and adapting them to PFD discovery is quite challenging; we have to carefully combine pattern discovery and data dependency discovery, to effectively and efficiently discover PFDs.

7. PROOFS

7.1 Proof of Theorem 1: Inference System

We first show $\mathcal{I}$ is sound, followed by its completeness. We only consider consistent PFDs Ψ, *i.e.,* there exists an instance T such that $I \models \Psi$, since otherwise $\Psi \models \psi$ trivially holds.

(I) $\mathcal{I}$ is sound. The soundness of $\mathcal{I}$ is a direct consequence of the definition of the rules and satisfaction of PFDs. More specifically, it can be proved by an induction on the length $|\ell|$ of the proof $\ell = \psi_1 \stackrel{r_1}{\longmapsto} \ldots, \stackrel{r_{n-1}}{\longmapsto} \psi_n$ from Ψ for ψ, where each r_i is one of the inference rules in Fig. 3. The induction hypothesis $\mathcal{H}(k)$ is: If $|\ell| = k (k \geq 1)$, then from that $\ell = \psi_1, \ldots, \psi_k = \psi$ is a proof for ψ from Ψ we have that $\Psi \models \psi$. The induction step can be readily verified on the last proof step of ℓ and is omitted here due to space limit.

(II) $\mathcal{I}$ is complete. We prove that $\mathcal{I}$ is complete for the logical implication of PFDs in two steps as follows:

We first give an algorithm that computes, given a set Ψ of PFDs and a single PFD $\psi = R(X \to Y, t_p)$ over relation schema R, a set W of attributes of R and associated patterns $t_W[A]$ for all $A \in W$, such that (i) $\psi_W = R(X \to W, t_W)$ can be implied from Ψ, and (ii) for any PFD $\psi' = R(X \to Y, t'_p)$ that can be implied from Ψ, $Y \subseteq W$ and $t_W[A] \subseteq t'_p[A]$ or $A \in t_W[A] \setminus t'_p[A]$ is not consistent *w.r.t.* Ψ. Intuitively, $Y \subseteq W$ and $t_W[A] \subseteq t_p[A]$ or $B \in t_W[A] \setminus t_p[A]$ is not consistent *w.r.t.* Ψ for each $A \in Y$ if $\Psi \models \psi$. We denote by $(X, t_p[X])^\Psi$ the set $\{(A, t_W[A]) \mid A \in W\}$ and refer to it as the *PFD-closure* of $(X, t_p[X])$ under Ψ. It is easy to verify that for any $\psi = R(X \to Y, t_p)$ and set Ψ of PFDs, there exists a unique PFD-closure of $(X, t_p[X])$ under Ψ.

In the second step, we constructively prove that if (W, t_W) is the PFD-closure of $(X, t_p[X])$ under ψ, then there must exist a proof of ψ_W from Ψ.

Below we present these two steps in more details.

(I) From logical implication to PFD-closure. As shown in Fig. 7, the algorithm takes as input a set Ψ of PFDs and a single PFD $\varphi = R(X \to Y, t_p)$ over relation schema R and returns $(X, t_p[X])^\Psi$. Analogous to the closure set of standard FDs [4] and CFDs [12], the set $(X, t_p)^\Psi$ satisfies the following property: $(A, t_W[A]) \in (X, t_p)^\Psi$ if and only if $\Sigma \models R(X \to A, t_p)$ with $t_p[A] = t_w[A]$.

While the algorithm is similar to the one for computing the closure of standard FDs [4], it differs along four points:

(1) It takes into account the constrained pattern t_p for X;

(2) It returns, instead of attributes, a set of pairs $(A, t_W[A])$, where A is an attribute and $t_W[A]$ is a pattern for A;

(3) It extends *closure* differently: it could either add $(A, t_p[A])$ into *closure* when A does not appear in *closure* (lines 8-9) or update an existing pair $(A, t_W[A])$ in *closure* if $t_p[A]$ is tighter than $t_W[A]$ (line 10-11); and most importantly

(4) It uses a different condition to check whether the *closure* set could be extended with new PFDs in Ψ (line 6): an PFD $R(Y \to A, t_p)$ can possibly trigger an extension of *closure* only if attributes in Y all appear in *closure* and their patterns can subsume the patterns in *closure* for Y (condition (a.i)), or their patterns can extend those patterns in *closure* with inconsistent values (condition (a.ii)), or $t_p[A]$ is a constant while all attributes in Y that do not appear in *closure* have wildcard patterns $\perp$ (condition (b)).

One can readily verify that the set *closure* returned by the algorithm is the PFD-closure of ψ under Ψ.

(II) From PFD-closure to inference proof. We next inductively construct, from the trace of computing $(X, t_p[X])^\Psi$, a proof of $R(X \to W, t_W)$ from Ψ using $\mathcal{I}$, where $t_W[A] = t_p[A]$ for each $A \in X$ and $(B, t_W[B]) \in (X, t_p[X])^\Psi$ for each $B \in W$.

For the base case, *i.e.,* $(A, t_w[A])$ added by line 2-3, the proof consists of one step with the Reflexivity rule.

When condition (a.i) is triggered, the interpreted proof is (denote by W the set of attributes in *closure* initially):

(1) $R(X \to W, t_W)$ (induction hypothesis);
(2) $R(X \to Y, t_W[XY])$ (by (1) and the Reflexivity axiom);
(3) $R(Y \to A, t_p)$ (in Ψ);
(4) $R(X \to A, t_W[XA])$ with $t_W[A] = t_p[A]$ (by (2), (3) and the Transitivity axiom).

For the case when condition (a.ii) is triggered (assuming *w.l.o.g.* only $B \in Y$ triggers (a.ii)), the interpreted proof consists of the following steps (denote by $Y' = Y \setminus \{B_0\}$):

(1) $R(B \to B, t_B)$, where $t_B[B_\mathsf{L}] = t_W[B] \setminus t_p[B]$ and $t_B[B_\mathsf{R}] = t_p[B]$ (by the Inconsistency-EFQ axiom);
(2) $R(Y'B \to Y'B, t_1)$, where $t_1[Y'_\mathsf{L}] = t_1[Y'_\mathsf{R}] = t_p[Y']$, $t_1[B_\mathsf{L}] = t_B[B_\mathsf{L}]$, $t_1[B_\mathsf{R}] = t_B[B_\mathsf{R}] = t_p[B]$ (by (1) and the Augmentation axiom);
(3) $R(Y'B \to Y'B, t_2)$, where $t_2[Y'_\mathsf{L}] = t_2[Y'_\mathsf{R}] = t_p[Y']$, $t_2[B_\mathsf{L}] = t_2[B_\mathsf{R}] = t_p[B]$ (by the Reflexivity and Augmentation axioms);
(4) $R(Y'B \to Y'B, t_3)$, where $t_3[Y'_\mathsf{L}] = t_3[Y'_\mathsf{R}] = t_p[X] = t_W[X]$, $t_3[B_\mathsf{L}] = t_3[B] \cup t_2[B_\mathsf{L}] = t_W[B]$, $t_3[B_\mathsf{R}] = t_p[B]$ (by (2), (3) and LHS-Generalization). That is, $R(Y \to Y, t_4)$, where $t_4[Y_\mathsf{L}] = t_W[Y]$ and $t_4[Y_\mathsf{R}] = t_p[Y]$;
(5) $R(Y \to A, t_p)$ (in Ψ);
(6) $R(Y \to A, t_5)$, where $t_5[Y] = t_W[Y]$ and $t_5[A] = t_p[A]$ (by (4), (5) and the Transitivity axiom);
(7) $R(X \to W, t_W)$ (induction hypothesis);
(8) $R(W \to Y, t_W)$ (by (7) and the Reflexivity axiom);
(9) $R(X \to A, t_W)$ (by (7), (8), (6) and Transitivity).

For the case when condition (b) is triggered, one can similarly construct a proof using the Reduction axiom (omitted)

7.2 Proof of Theorem 2: Implication

The coNP-hard lower bound follows from the coNP-hardness of the implication of CFDs [12], as the CFDs are a special case of PFDs. Below we focus on the upper bound and show that the implication checking of PFDs remains in coNP.

To show the coNP upper bound, we give an NP algorithm for the complement of the implication problem. The algorithm decides, given a set Ψ of PFDs over relation R and another PFD ψ over R, whether $\Psi \not\models \psi$ holds. To do this, the algorithm checks whether there exists an instance T of

Algorithm From logical implication to PFD-closure

Input: Relation schema R, set Ψ of PFDs and PFD $\psi = R(X \to Y, t_p)$.
Output: The PFD-closure $(X, t_p[X])^{\Psi}$.

1. *unused* := $\emptyset$;
2. **for each** $R(X \to Y, t_p) \in \Psi$ **do**
3. *unused* := *unused* $\cup \{R(X \to A, t_p[XA]) \mid$ for each $A \in Y\}$;
4. *closure* := $\{(A, t_p[A]) \mid A \in X\}$;
5. **repeat until** no further change:
6. **if** $R(Y \to A, t_p) \in$ *unused*,
   (a) attributes in Y all appear in *closure* and $\forall B \in Y$
   (i) there is $(B, t_W[B]) \in$ *closure* s.t. $t_W[B] \subseteq t_p[B]$, or
   (ii) there is $(B, t_W[B]) \in$ *closure* s.t. $B \in t_W[B] \setminus t_p[B]$ is not consistent *w.r.t.* Ψ; or
   (b) Y contains attributes not appeared in *closure*, $t_p[A]$ is a constant and $t_p[B] = \perp$ for all $B \in Y$ that does not appear in *closure*, **then**
7. *unused* := *unused* $-\{R(Y \to A, t_p)\}$;
8. **if** A is not in *closure* **then**
9. *closure* := *closure* $\cup\{(A, t_p[A])\}$;
10. **else if** $(A, t_w[A]) \in$ *closure* and $t_p[A] \subseteq t_W[A]$ **then**
11. *closure* := *closure* $\cup\{(A, t_p[A])\}$;
12. **return** *closure*;

Figure 7: Algorithm for the Proof of Theorem 1

R such that $I \models \Psi$ but $I \not\models \psi$. This is carried out by using a small model property, given as follows.

Consider a set Ψ of PFDs over relation schema R and PFD $\psi = R(X \to Y, t_p)$. If there exists a nonempty instance T of R such that $I \models \Psi$ but $I \not\models \psi$, then there must exist two tuples $t, t' \in I$ such that: (a) $I_s = \{t, t'\} \models \Psi$, (b) $t[X] = t'[X]$, and (c) either $t[Y] \neq t'[Y]$ or $t[Y] \not\asymp t_p[Y]$ (resp. $t'[Y] \not\asymp t_p[Y]$); moreover, for each A of R, $t[A]$ (resp. $t'[A]$) is of length no longer than $\sum_{\psi \in \Psi} |t_\psi[A]|$, where $t_\psi[A]$ is the pattern of ψ on A and $|t_\psi[A]|$ is the length of the pattern.

Based on the small model property, we give an NP algorithm that checks whether $\Psi \not\models \psi$ as follows:

(1) guess two tuples t and t' such that for each attribute A, both $t[A]$ and $t'[A]$ draw characters from the generalization tree and are of length bounded by $\sum_{\psi \in \Psi} |t_\psi[A]|$;
(2) check whether $I_s = \{t, t'\} \models I$ and $I_s \not\models \psi$; return "Yes" if so and go to step (1) otherwise.

By the small model property, the algorithm correctly decides whether $\Psi \not\models \psi$. It is in NP since (a) there are at most exponentially in $|R|$ and $|\Psi|$ (the total length of PFDs in Ψ) many guesses for step (1) and (b) step (2) is in PTIME in $|R|$ and $|\Psi|$. Therefore, the implication of PFDs is in coNP.

7.3 Proof of Theorem 3: Consistency

We only need to show that the consistency of PFDs is in NP and it is NP-hard when all domains are infinite. Note that the NP-hardness of the consistency checking of CFDs [12] does not carry over here since it only works for finite domains.

Upper bound. We show that the consistency checking of PFDs is in NP, by giving an NP algorithm. It is based on the following *small model property.*

Small model property. Consider a set Ψ of PFDs over relation schema R. If there exists a nonempty instance T of R such that $I \models \Psi$, then (a) for any $t \in I$, $I_t = \{t\}$ is an instance of R and $I_t \models \Psi$; and (b) value of t on attribute A is of length bounded by $\sum_{\psi \in \Psi} |t_\psi[A]|$, where $t_\psi[A]$ is the pattern of ψ on attribute A and $|t_\psi[A]|$ is the length of $t_\psi[A]$ (assuming numbers are stored as unaries)

An NP algorithm. Using this property, we have an NP algorithm for consistency checking of PFDs working as follows:

(1) guess a tuple t with each value $t[A]$ of attribute A draws characters from the generalization tree and is of length bounded by $\sum_{\psi \in \Psi} |t_\psi[A]|$;
(2) check whether $I_t = \{t\} \models \Psi$; return "Yes" if so and go to step (1) otherwise.

The algorithm is correct by the small model property. It is in NP since (a) there are at most $2^{O(|\sum_{\psi \in \Psi} |\psi| + |R|)}$ guesses for step (1) and (b) step (2) decidable in PTIME in $|\sum_{\psi \in \Psi} |\psi| + |R|$, where $|\psi|$ is the length of PFD ψ and so is $|R|$.

Lower bound. We show that the consistency of PFDs is NP-hard over infinite domains by reduction from the nontautology problem, similar to the proof of consistency of CFDs [12]. More specifically, an instance of the nontautology problem is a DNF Boolean formula $\phi = C_1 \vee \cdots \vee C_n$, where (a) variables in ϕ are $x_1, \ldots, x_m$ and (b) C_j is of the form $\ell_1^j \wedge \ell_2^j \wedge \ell_3^j$, in which $\ell_i^j (i \in [1, 3])$ is either x_k or $\bar{x}_k$ for some $k \in [1, m]$. The problem is to decide whether there is a truth assignment such that ϕ is false. The problem is NP-complete [15].

Given an instance ϕ of the nontautology problem, we define an instance of the PFD consistency problem, namely, a relation schema R over *infinite domains* and a set Ψ of PFDs on R such that ϕ is not a tautology if and only if Ψ is consistent.

(1) R is defined to be $(X_1, \ldots, X_m, C)$, where all attributes $X_i (i \in [1, m])$ and C are over infinite domains of strings consisting of lower case letters (LU) and digits (D). As will be shown later, for each tuple t in an instance T of R, $t[X_1, \ldots, X_m]$ encodes a truth assignment μ of variables $x_1, \ldots, x_m$: $\mu(x_i) = \mathit{true}$ if $t[X_i]$ is a string starts with digits and $\mu(x_i) = \mathit{false}$ if $t[X_i]$ starts with lower case letters.

(2) The set Ψ consists of $n+1$ PFDs. More specifically, for each clause $C_j (j \in [1, n])$ in ϕ, Ψ includes $\psi_j = R(X_1 \ldots X_m \to C, t_j)$ encoding C_j such that (i) $t_j[C] = \mathsf{D}^+\mathsf{LU}^*$, (ii) $t_j[X_i] = \mathsf{D}^+\mathsf{LU}^*$ if x_i appears in C_j, and (iii) $t_j[X_i] = \mathsf{LU}^+\mathsf{D}^*$ if $\bar{x}_i$ appears in C_j. In addition, Ψ includes $\psi_{n+1} = R(C \to C, t_{n+1})$, where $t_{n+1}[C_\mathsf{L}] = \mathsf{D}^+\mathsf{LU}^*$ and $t_{n+1}[C_\mathsf{R}] = \mathsf{LU}^+\mathsf{D}^*$.

Intuitively, any tuple t in an instance T of R that satisfies PFDs $\psi_1 - \psi_n$ must have $t[C]$ as a string started with a digit if the truth assignment encoded by $t[X_1, \ldots, X_m]$ makes ϕ *true*; moreover, if this the case then T does not satisfy ψ_{n+1}.

One can verify that Ψ is consistent iff there exists a truth assignment that makes ϕ *false* (omitted due to space limit).

8. CONCLUSION AND FUTURE WORK

We have introduced PFDs, a new class of ICs that can capture dependencies between partial attribute values, in contrast to previous ICs that consider the entire attribute values. We have provided a sound and complete set of inference axioms for PFDs. Moreover, we have proposed an effective and efficient algorithm to discover PFDs from the data even if is dirty, instead of asking domain experts to provide them manually. Most importantly, we have applied our solutions on many real-world datasets and found many "new" data errors that cannot be found by existing automatic error detection solutions. One followup work is to study effective pruning strategies for reducing the candidate attribute combinations for PFD discovery. Another future work is to test on more real-world datasets to examine PFDs with multiple LHS attributes.

9. REFERENCES

[1] Z. Abedjan, C. G. Akcora, M. Ouzzani, P. Papotti, and M. Stonebraker. Temporal rules discovery for web data cleaning. *PVLDB*, 9(4):336–347, 2015.

[2] Z. Abedjan, X. Chu, D. Deng, R. C. Fernandez, I. F. Ilyas, M. Ouzzani, P. Papotti, M. Stonebraker, and N. Tang. Detecting data errors: Where are we and what needs to be done? *PVLDB*, 9(12):993–1004, 2016.

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

[4] S. Abiteboul, R. Hull, and V. Vianu. Foundations of databases. 1995.

[5] L. Berti-Équille, H. Harmouch, F. Naumann, N. Novelli, and S. Thirumuruganathan. Discovery of genuine functional dependencies from relational data with missing values. *PVLDB*, 11(8):880–892, 2018.

[6] W. A. Carnielli and J. Marcos. Ex contradictione non sequitur quodlibet. *Bulletin of Advanced Reasoning and Knowledge*, 1:89–109, 2001.

[7] L. Chiticariu, Y. Li, and F. R. Reiss. Rule-based information extraction is dead! long live rule-based information extraction systems! In *EMNLP*, pages 827–832, 2013.

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

[9] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[10] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. NADEEF: a commodity data cleaning system. In *SIGMOD*, pages 541–552, 2013.

[11] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1):1–16, 2007.

[12] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, 33(2), 2008.

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

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

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[16] P. S. G.C., C. Sun, K. G. Kuchimanchi, H. Zhang, F. Yang, N. Rampalli, S. Prasad, E. Arcaute, G. Krishnan, R. Deep, V. Raghavendra, and A. Doan. Why big data industrial systems need rules and what we can do about it. In *SIGMOD*, pages 265–276, 2015.

[17] J. He, E. Veltri, D. Santoro, G. Li, G. Mecca, P. Papotti, and N. Tang. Interactive and deterministic data cleaning. In *SIGMOD*, pages 893–907, 2016.

[18] Z. Huang and Y. He. Auto-detect: Data-driven error detection in tables. In *SIGMOD*, pages 1377–1392, 2018.

[19] 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.

[20] M. Interlandi and N. Tang. Proof positive and negative in data cleaning. In *ICDE*, pages 18–29, 2015.

[21] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *CHI*, pages 3363–3372, 2011.

[22] P. Konda, S. Das, P. S. G.C., A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton, S. Prasad, G. Krishnan, R. Deep, and V. Raghavendra. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12):1197–1208, 2016.

[23] N. Koudas, A. Saha, D. Srivastava, and S. Venkatasubramanian. Metric functional dependencies. In *ICDE*, pages 1275–1278, 2009.

[24] S. Kwashie, J. Liu, J. Li, and F. Ye. Conditional differential dependencies (cdds). In *Advances in Databases and Information Systems (ADBIS)*, pages 3–17, 2015.

[25] P. Mandros, M. Boley, and J. Vreeken. Discovering reliable approximate functional dependencies. In *KDD*, pages 355–363, 2017.

[26] F. Panahi, W. Wu, A. Doan, and J. F. Naughton. Towards interactive debugging of rule-based entity matching. In *EDBT*, pages 354–365, 2017.

[27] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, and M. Koubarakis. The return of JedAI: End-to-end entity resolution for structured and semi-structured data. *PVLDB*, 11(12):1950–1953, 2018.

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

[29] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *PVLDB*, 8(10):1082–1093, 2015.

[30] C. Pit-Claudel, Z. Mariet, R. Harding, and S. Madden. Outlier detection in heterogeneous datasets using automatic tuple expansion. In *Technical Report*.

[31] A. A. Qahtan, A. K. Elmagarmid, R. C. Fernandez, M. Ouzzani, and N. Tang. FAHES: A robust disguised missing values detector. In *KDD*, pages 2100–2109, 2018.

[32] A. A. Qahtan, A. K. Elmagarmid, M. Ouzzani, and N. Tang. FAHES: detecting disguised missing values. In *ICDE*, pages 1609–1612, 2018.

[33] A. A. Qahtan, N. Tang, M. Ouzzani, Y. Cao, and M. Stonebraker. ANMAT: automatic knowledge discovery and error detection through pattern functional dependencies. In *SIMOD*, pages 1977–1980, 2019.

[34] J. Rammelaere and F. Geerts. Revisiting conditional functional dependency discovery: Splitting the "C" from the "FD". In *ECML-PKDD*, pages 552–568, 2018.

[35] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *PVLDB*, 9(10):816–827, 2016.

[36] R. Singh, V. V. Meduri, A. K. Elmagarmid, S. Madden,

P. Papotti, J. Quiané-Ruiz, A. Solar-Lezama, and N. Tang. Synthesizing entity matching rules by examples. *PVLDB*, 11(2):189–202, 2017.

[37] S. Song and L. Chen. Differential dependencies: Reasoning and discovery. *ACM Trans. Database Syst.*, 36(3):16:1–16:41, 2011.

[38] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In *STOC*, pages 1–9, 1973.

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

[40] Trifacta Documentation. Trifacta built-in data types. `https://docs.trifacta.com/display/PE/Supported+Data+Types`.

[41] J. Wang and N. Tang. Towards dependable data repairing with fixing rules. In *SIGMOD*, pages 457–468, 2014.

[42] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.

[43] G. Zhu, Q. Wang, Q. Tang, R. Gu, C. Yuan, and Y. Huang. Efficient and scalable functional dependency discovery on distributed data-parallel platforms. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2663–2676, 2019.