Extending Dependencies with Conditions

Loreto Bravo
University of Edinburgh
lbravo@inf.ed.ac.uk

Wenfei Fan
Univ. of Edinburgh & Bell Labs
wenfei@inf.ed.ac.uk

Shuai Ma
University of Edinburgh
sma1@inf.ed.ac.uk

Abstract

This paper introduces a class of conditional inclusion dependencies (CINDs), which extends traditional inclusion dependencies (INDs) by enforcing bindings of semantically related data values. We show that CINDs are useful not only in data cleaning, but are also in contextual schema matching [7]. To make effective use of CINDs in practice, it is often necessary to reason about them. The most important static analysis issue concerns *consistency*, to determine whether or not a given set of CINDs has conflicts. Another issue concerns *implication*, *i.e.,* deciding whether a set of CINDs entails another CIND. We give a full treatment of the static analyses of CINDs, and show that CINDs retain most nice properties of traditional INDs: (a) CINDs are always consistent; (b) CINDs are finitely axiomatizable, *i.e.,* there exists a sound and complete inference system for implication of CINDs; and (c) the implication problem for CINDs has the same complexity as its traditional counterpart, namely, PSPACE-complete, in the absence of attributes with a finite domain; but it is EXPTIME-complete in the general setting. In addition, we investigate the interaction between CINDs and conditional functional dependencies (CFDs), an extension of functional dependencies proposed in [9]. We show that the consistency problem for the combination of CINDs and CFDs becomes undecidable. In light of the undecidability, we provide heuristic algorithms for the consistency analysis of CFDs and CINDs, and experimentally verify the effectiveness and efficiency of our algorithms.

1. Introduction

A class of *conditional functional dependencies* (CFDs) has recently been proposed in [9] as an extension of functional dependencies (FDs). In contrast to traditional FDs, CFDs hold conditionally on a relation, *i.e.,* they apply only to those tuples that satisfy certain data-value patterns, rather than to the entire relation. CFDs have proven useful in data cleaning [9]: inconsistencies and errors in the data may emerge as violations of CFDs, whereas they may not be caught by traditional FDs.

It has been recognized [8] that to clean data, one needs not only FDs but also *inclusion dependencies* (INDs). Furthermore, INDs are commonly used in schema matching systems, *e.g.,* Clio [16]: INDs associate attributes in a source schema with semantically related attributes in a target schema. Both schema matching and data cleaning highlight the need for extending INDs along the same lines as CFDs, as illustrated by the examples below.

	an	cn	ca	cp	at
t_1:	01	J. Smith	NYC, 19087	212-5820844	saving
t_2:	02	G. King	NYC, 19022	212-3963455	checking
t_3:	03	J. Lee	NYC, 02284	212-5679844	checking

(a) account in NYC branch

	an	cn	ca	cp	at
t_4:	01	S. Bundy	EDI, EH8 9LE	131-6516501	saving
t_5:	02	I. Stark	EDI, EH1 4FE	131-6693423	checking

(b) account in EDI branch

	an	cn	ca	cp	ab
t_6:	01	J. Smith	NYC, 19087	212-5820844	NYC
t_7:	01	S. Bundy	EDI, EH8 9LE	131-6516501	EDI

(c) saving

	an	cn	ca	cp	ab
t_8:	02	G. King	NYC, 19022	212-3963455	NYC
t_9:	03	J. Lee	NYC, 02284	212-5679844	NYC
t_{10}:	02	I. Stark	EDI, EH1 4FE	131-6693423	EDI

(d) checking

	ab	ct	at	rt
t_{11}:	EDI	UK	saving	4.5%
t_{12}:	EDI	UK	checking	10.5%
t_{13}:	NYC	US	saving	4%
t_{14}:	NYC	US	checking	1%

(e) interest

Figure 1: Example account**,** saving**,** checking**,** interest **data**

Example 1.1: Consider a bank that has branches in various countries. Each branch B maintains a separate account relation:

source schema: account_B(an, cn, ca, cp, at)

in which each tuple specifies an account: the number (an) and type (at, saving or checking) of the account, along with the name (cn), address (ca) and phone number (cp) of the owner of the account.

The bank needs to integrate the account data from its branches and stores the data in a target database with the following schema:

target schema: saving(an, cn, ca, cp, ab)
checking(an, cn, ca, cp, ab)
interest(ab, ct, at, rt)

where ab is the name of the branch where the account was opened, and an, cn, ca, cp and at are as above. In relation interest, rt indicates the interest rate, and ct is the country where the branch ab is located. Example source (account) and target (saving, checking, interest) data instances are shown in Fig. 1.

A schema matching system might want to match attributes an, cn, ca, cp from source schema account to an, cn, ca, cp in the target schemas saving and checking, and attempt to express the matches in terms of inclusion dependencies from the source to the target, *e.g.,* account_B(an, cn, ca, cp) $\subseteq$ saving(an, cn, ca, cp) and account_B(an, cn, ca, cp) $\subseteq$ checking(an, cn, ca, cp). These traditional INDs, however, do not make sense: an account in a source relation should be stored either in the target saving or checking, but *not* in both. This is where we need contextual schema matching [7]: for any tuple t in an account relation, its attributes an, cn, ca, cp can be mapped to the target saving relation only if t[at] = saving, and to checking only if t[at] = checking.

To capture this, one can use the constraints below (at branch B):

ind$_1$: account_B (an, cn, ca, cp; at ='saving') $\subseteq$
saving (an, cn, ca, cp; ab = 'B')

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

ind_2: $\mathsf{account\_B}$ ($\mathsf{an}, \mathsf{cn}, \mathsf{ca}, \mathsf{cp}$; at =‘checking’) $\subseteq$
$\mathsf{checking}$ ($\mathsf{an}, \mathsf{cn}, \mathsf{ca}, \mathsf{cp}$; ab = ‘B’)

where ind_1 asserts that for each tuple t_1 in the $\mathsf{account}$ relation at branch B, if $t_1[\mathsf{at}]$ = saving, then there must exist a tuple t_2 in saving such that $t_1[\mathsf{an}, \mathsf{cn}, \mathsf{ca}, \mathsf{cp}] = t_2[\mathsf{an}, \mathsf{cn}, \mathsf{ca}, \mathsf{cp}]$, and moreover, $t_2[\mathsf{ab}] = B$. That is, an account in the source is migrated to target relation saving only if the type of the account is saving, and in addition, $t_2[\mathsf{ab}]$ holds the constant B. This constraint is an IND that holds only on the subset of $\mathsf{account}$ tuples that satisfy the pattern at =‘saving’, rather on the entire $\mathsf{account}$ relation; similarly for ind_2. However, these constraints are not considered INDs since they are specified with a *pattern* containing *data values*. □

Example 1.2: Next let us focus on the target database alone and consider data cleaning. It has been recognized that integrity constraints are important in data cleaning [24]. Prior work on constraint-based data cleaning, however, mostly adopts traditional dependencies such as FDs and INDs (*e.g.,* [2, 8, 13, 25]). Traditional FDs and INDs on our example database include:

fd_1: saving ($\mathsf{an}, \mathsf{ab} \rightarrow \mathsf{cn}, \mathsf{ca}, \mathsf{cp}$)
fd_2: $\mathsf{checking}$ ($\mathsf{an}, \mathsf{ab} \rightarrow \mathsf{cn}, \mathsf{ca}, \mathsf{cp}$)
fd_3: $\mathsf{interest}$ ($\mathsf{ct}, \mathsf{at} \rightarrow \mathsf{rt}$)
ind_3: saving (ab) $\subseteq$ $\mathsf{interest}$ (ab)
ind_4: $\mathsf{checking}$ (ab) $\subseteq$ $\mathsf{interest}$ (ab)

These assert that an, ab are a key for saving and $\mathsf{checking}$ (fd_1, fd_2), all the saving (resp. checking) accounts in the same country must have the same interest rate (fd_3), and that any branch in saving and $\mathsf{checking}$ must appear in $\mathsf{interest}$ (ind_3, ind_4).

While the instances of Fig. 1 satisfy these traditional dependencies, the data is not clean. The bank may offer slightly different interest rates for accounts in different countries, *e.g.,* for checking accounts in the UK, the interest rate is 1.5%, whereas it is 1% for the US checking accounts. Tuple t_{12} in Fig. 1(e) indicates that the interest rate for checking accounts in the UK is 10.5% rather than 1.5%. This inconsistency, however, cannot be detected by standard INDs and FDs, which were originally developed for *schema design* rather than *data cleaning*. In contrast, this can be caught by the constraints below, which refine ind_3 and ind_4 by adding patterns:

ind_5: saving (ab = ‘EDI’) $\subseteq$
$\mathsf{interest}$ (ab = ‘EDI’, at = ‘saving’, ct = ‘UK’, rt = 4.5%)
ind_6: $\mathsf{checking}$ (ab = ‘EDI’) $\subseteq$
$\mathsf{interest}$ (ab = ‘EDI’, at = ‘checking’, ct = ‘UK’, rt = 1.5%)
ind_7: saving (ab = ‘NYC’) $\subseteq$
$\mathsf{interest}$ (ab = ‘NYC’, at = ‘saving’, ct = ‘US’, rt = 4%)
ind_8: $\mathsf{checking}$ (ab = ‘NYC’) $\subseteq$
$\mathsf{interest}$ (ab = ‘NYC’, at = ‘checking’, ct = ‘US’, rt = 1%)

ind_6 says that for each Edinburgh checking account, there must exist a tuple t in $\mathsf{interest}$ such that $t[\mathsf{ab}]$ = EDI, $t[\mathsf{at}]$ = checking, $t[\mathsf{ct}]$ = UK and $t[\mathsf{rt}]$ = 1.5%. Thus tuple t_{10} violates ind_6: no $\mathsf{interest}$ tuple matches t_{10} with the correct interest rate 1.5%. This shows that ind_6 catches the error that is not detected by traditional FDs and INDs. In fact, ind_6 and fd_3 together assure that for all Edinburgh checking accounts, 1.5% is the unique interest rate. □

Dependencies such as $\mathsf{ind}_1 - \mathsf{ind}_2$ and $\mathsf{ind}_5 - \mathsf{ind}_8$ apply *conditionally* to relations. Clearly, such constraints are needed for both *schema matching* and *data cleaning*, and hence deserve a full treatment. However, they cannot be expressed as standard INDs.

Contributions. To this end we introduce an extension of INDs and investigate the static analysis of these constraints.

Our first contribution is a notion of *conditional inclusion dependencies* (CINDs). A CIND is defined as a pair consisting of an IND $R_1[X] \subseteq R_2[Y]$ and a *pattern tableau*, where the tableau enforces binding of semantically related data values across relations R_1 and R_2. For example, $\mathsf{ind}_1 - \mathsf{ind}_8$ given above can be expressed as CINDs. In particular, traditional INDs are a *special case* of CINDs. This mild extension of INDs captures a fundamental part of the semantics of data, and suffices to express many applications commonly found in data cleaning and schema matching.

Our second contribution consists of techniques for reasoning about CINDs. Given a set of CINDs, the first thing one wants to do is to determine whether the CINDs are *consistent, i.e.,* whether they have conflicts. This is very important: one does not want to enforce the CINDs on a database at run-time but find, after repeated failures, that the CINDs cannot possibly be satisfied by a nonempty database. Similarly, one does not want to match schema based on CINDs that do not make sense. The consistency analysis help users to develop consistent sets of CINDs for data cleaning and schema matching. For traditional INDs and FDs, consistency is not an issue: one can specify any INDs and FDs without worrying about their consistency. In contrast, it is known that CFDs may have conflicts, and that it is intractable to decide whether or not a set of CFDs is consistent [9]. Another decision problem associated with CINDs is the *implication* problem, which is to decide whether a set of CINDs entails another CIND. For traditional INDs, the implication problem is PSPACE-complete. Furthermore, it is *finitely axiomatizable*: there exists a finite, sound and complete set of axioms. The implication analysis is useful in reducing redundant CINDs, and hence improving performance when detecting CIND violations in a database, and speeding up the derivation of schema mappings from CINDs [16].

We show that although CINDs are more expressive than INDs, they retain most nice properties of their traditional counterpart: (a) CINDs are always consistent; (b) the implication of CINDs is finitely axiomatizable; (c) in the absence of attributes with a finite domain, the implication problem for CINDs is also PSPACE-complete, while in the general setting, it is EXPTIME-complete. Since a problem with a PSPACE lower bound is already beyond reach in practice, the EXPTIME result actually tells us that we do not have to pay too high a price for the increased expressive power of CINDs.

Our third contribution is an investigation of the interaction between CINDs and CFDs. This is necessary: in data cleaning one needs both CFDs and CINDs; so does in schema matching where one needs CINDs and at least conditional keys [16], a special case of CFDs. For traditional FDs and INDs, the interaction is already intriguing: the implication problem for FDs and INDs is undecidable and is not finitely axiomatizable. The interaction between CINDs and CFDs makes our lives even harder: we show that for CINDs and CFDs together, the consistency problem is undecidable.

Our fourth contribution is a set of algorithms for checking the consistency of CFDs and CINDs. In light of the undecidability result mentioned above, any consistency-checking algorithm for CFDs and CINDs that runs in polynomial times is necessarily heuristic. That is, the algorithm is sound on detecting consistent sets of CINDs and CFDs, but not necessarily complete. Our heuristic algorithms are based on a combination of chase techniques, dependency-graph analysis, and bounded-size witness database construction.

Our fifth and final contribution is a preliminary experimental study. We compare the performances of our algorithms in terms of both the accuracy of output and evaluation time. Our experimental results show that our algorithms are effective and efficient.

These results provide not only complexity bounds and an inference system for fundamental problems associated with CINDs (and CFDs), but also efficient algorithms that allow CINDs and CFDs to be used in practice. Our conclusion is that CINDs, together with CFDs, may lead to promising tools for cleaning data and for finding quality schema matches.

We should remark that CINDs do not introduce a new logical formalism. Indeed, in first-order logic, they can be expressed in a form similar to tuple-generating dependencies (TGDs), which have lately generated renewed interests in schema mapping (see [18] for a survey on recent results). However, (a) these simple CINDs suffice to capture data consistency and contextual schema matching commonly found in practice, without incurring the complexity of *full-fledged* TGDs; (b) no prior work has studied the consistency, implication and finite axiomatizability problems for TGDs in the presence of *constants* (data values).

Organization. We define CINDs in Section 2, and investigate their associated consistency and implication problems in Section 3. In Section 4 we study the consistency analysis of CINDs and CFDs, and provide heuristic algorithms in Section 5. Our experimental results are presented in Section 6, followed by related work in Section 7 and conclusion in Section 8.

2. Conditional Inclusion Dependencies

A relational database schema $\mathcal{R}$ is a collection of relation schemas $(R_1, \ldots, R_n)$, where each R_i is defined over a fixed set of attributes $\mathsf{attr}(R)$. Each attribute A_k has an associated domain, $\mathsf{dom}(A_k)$, which is finite or infinite. The set $\mathsf{finattr}(\mathcal{R})$ contains the finite attributes of $\mathcal{R}$. An instance I of R_i is a set of tuples such that for each $t \in I$, $t[A_k] \in \mathsf{dom}(A_k)$ for each attribute $A_k \in \mathsf{attr}(R_i)$. A database instance D of $\mathcal{R}$ is a collection of relations $(I_1, \ldots, I_n)$, where I_i is an instance of R_i for $i \in [1, n]$.

Syntax. A conditional inclusion dependency (CIND) ψ is a pair $(R_1[X; X_p] \subseteq R_2[Y; Y_p], T_p)$, where (1) X, X_p and Y, Y_p are lists of attributes in $\mathsf{attr}(R_1)$ and $\mathsf{attr}(R_2)$, respectively, such that X and X_p (resp. Y and Y_p) are disjoint; (2) $R_1[X] \subseteq R_2[Y]$ is a standard IND, referred to as the IND *embedded in* ψ; and (3) T_p is a tableau, called the *pattern tableau* of ψ; it has all attributes in X, X_p and Y, Y_p, and for each A in X, X_p or Y, Y_p and each tuple $t_p \in T_p$, $t_p[A]$ is either a constant 'a' in $\mathsf{dom}(A)$, or an unnamed variable '_'. Moreover, $t_p[X] = t_p[Y]$.

Abusing set operations, we use $X \cup X_p$ to denote the set of all attributes of X and X_p, and $X - Y$ to denote the list obtained from list X by removing all the elements in list Y. We denote $X \cup X_p$ as $\mathsf{LHS}(\psi)$ and $Y \cup Y_p$ as $\mathsf{RHS}(\psi)$, and separate the LHS and RHS attributes in a pattern tuple with '$\|$'. We use nil to denote *an empty list*. Let $X = [A_1, \ldots, A_m]$ and $Y = [B_1, \ldots, B_m]$. We assume w.l.o.g that $\mathsf{dom}(A_i) \subseteq \mathsf{dom}(B_i)$ for each $i \in [1, m]$.

Example 2.1: Constraints ind_1–ind_8 given in Examples 1.1 and 1.2 can all be expressed as CINDs shown in Fig 2: ψ_1–ψ_4 for ind_1–ind_4, respectively; ψ_5 for both ind_5 and ind_7, *one pattern tuple for each constraint*; and ψ_6 for both ind_6 and ind_8. In ψ_1, for instance, both X and Y are $[\mathsf{an}, \mathsf{cn}, \mathsf{ca}, \mathsf{cp}]$, X_p is $[\mathsf{at}]$ and Y_p is $[\mathsf{ab}]$. In ψ_3, both X and Y are $[\mathsf{ab}]$, while both X_p and Y_p are nil. In ψ_5, both X and Y are nil, while X_p is $[\mathsf{ab}]$ and Y_p is $[\mathsf{ab}, \mathsf{at}, \mathsf{ct}, \mathsf{rt}]$. □

As shown by ψ_3 and ψ_4, a standard IND $R_1[X] \subseteq R_2[Y]$ is a special case of the CIND $(R_1[X; X_p] \subseteq R_2[Y; Y_p],\ T_p)$ in which both X_p and Y_p are nil, and T_p has a single tuple with '_' only.

Semantics. In general the IND embedded in a CIND may not hold on the entire R_1 relation: it applies only to R_1 tuples matching the pattern tuples. More precisely, we define an order $\asymp$ on data values and the unnamed variable '_': $\eta_1 \asymp \eta_2$ if either $\eta_1 = \eta_2$, or η_1 is a data value a and η_2 is '_'. The order $\asymp$ naturally extends to tuples, *e.g.,* (EDI, UK, 1.5%) $\asymp$ (EDI, UK, _) but (EDI, UK, 4.5%) $\not\asymp$ (EDI, UK, 10.5%). We say that a tuple t_1 *matches* t_2 if $t_1 \asymp t_2$.

An instance (I_1, I_2) of (R_1, R_2) *satisfies* the CIND ψ, denoted by $(I_1, I_2) \models \psi$, iff for *each* t_1 in the relation I_1, and for *each* tuple t_p in the pattern tableau T_p, if $t_1[X, X_p] \asymp t_p[X, X_p]$, then *there exists* t_2 in the relation I_2 such that $t_1[X] = t_2[Y] \asymp t_p[Y]$ and moreover, $t_2[Y_p] \asymp t_p[Y_p]$. That is, if $t_1[X, X_p]$ matches the pattern $t_p[X, X_p]$, then the inclusion constraint specified by t_p must apply, which requires the existence of t_2 such that (1) $t_1[X]$ and $t_2[Y]$ are equal as required by the standard IND embedded in ψ, and (2) $t_2[Y_p]$ must match the pattern $t_p[Y_p]$.

ψ_1 = ($\mathsf{account\_B}[\mathsf{an}, \mathsf{cn}, \mathsf{ca}, \mathsf{cp};\ \mathsf{at}] \subseteq \mathsf{saving}[\mathsf{an}, \mathsf{cn}, \mathsf{ca}, \mathsf{cp};\ \mathsf{ab}], T_1$)

T_1:

an	cn	ca	cp	at	‖	an	cn	ca	cp	ab
_	_	_	_	saving	‖	_	_	_	_	B

ψ_2 = ($\mathsf{account\_B}[\mathsf{an}, \mathsf{cn}, \mathsf{ca}, \mathsf{cp}; \mathsf{at}] \subseteq \mathsf{checking}[\mathsf{an}, \mathsf{cn}, \mathsf{ca}, \mathsf{cp};\ \mathsf{ab}], T_2$)

T_2:

an	cn	ca	cp	at	‖	an	cn	ca	cp	ab
_	_	_	_	checking	‖	_	_	_	_	B

ψ_3 = ($\mathsf{saving}[\mathsf{ab}; \mathsf{nil}] \subseteq \mathsf{interest}[\mathsf{ab}; \mathsf{nil}], T_3$)

T_3:

ab	‖	ab
_	‖	_

ψ_4 = ($\mathsf{checking}[\mathsf{ab}; \mathsf{nil}] \subseteq \mathsf{interest}[\mathsf{ab}; \mathsf{nil}], T_4$)

T_4:

ab	‖	ab
_	‖	_

ψ_5 = ($\mathsf{saving}[\mathsf{nil};\ \mathsf{ab}] \subseteq \mathsf{interest}[\mathsf{nil};\ \mathsf{ab}, \mathsf{at}, \mathsf{ct}, \mathsf{rt}], T_5$)

T_5 :

ab	‖	ab	at	ct	rt
EDI	‖	EDI	saving	UK	4.5%
NYC	‖	NYC	saving	US	4%

ψ_6 = ($\mathsf{checking}[\mathsf{nil};\ \mathsf{ab}] \subseteq \mathsf{interest}[\mathsf{nil};\ \mathsf{ab}, \mathsf{at}, \mathsf{ct}, \mathsf{rt}], T_6$)

T_6:

ab	‖	ab	at	ct	rt
EDI	‖	EDI	checking	UK	1.5%
NYC	‖	NYC	checking	US	1%

Figure 2: Example CINDs

The pattern X_p is not part of the embedded IND. Intuitively, it is used to identify the R_1 tuples over which ψ is applied. The pattern Y_p enforces that the matching R_2 tuples must satisfy a certain form. Notice that in real case scenarios it is expected that the pattern tableaux are much smaller than the database.

Example 2.2: The database in Fig. 1 satisfies CFDs ψ_1–ψ_7. Note that although these CINDs are satisfied, their embedded INDs do not necessarily hold. For example, while ψ_1 is satisfied, the IND $\mathsf{account\_edi}[\mathsf{an}, \mathsf{cn}, \mathsf{ca}, \mathsf{cp}] \subseteq \mathsf{saving}[\mathsf{an}, \mathsf{cn}, \mathsf{ca}, \mathsf{cp}]$ is not. The pattern X_p in $\mathsf{LHS}(\psi_1)$ is used to identify the tuples over which ψ has to be enforced, namely, tuples for saving accounts.

On the other hand, ψ_6 is *violated* by the database. Indeed, for tuple t_{10}, there exists a pattern tuple t_p (the first tuple) in T_6 such that $t_{10}[\mathsf{ab}] \asymp t_p[\mathsf{ab}]$ but there is no tuple t in table $\mathsf{interest}$ such that $t[\mathsf{ab}]$ = EDI, $t[\mathsf{at}]$ = checking, $t[\mathsf{cn}]$ = UK and $t[\mathsf{rt}]$ = 1.5%. □

We say that a database D satisfies a set Σ of CINDs, denoted by $D \models \Sigma$, if $D \models \varphi$ for each $\varphi \in \Sigma$.

3. Reasoning about CINDs

With any constraint language L, there are two associated fundamental problems: the consistency problem for determining whether a given set of constraints in L has conflicts, and the implication problem for deriving other constraints from a given set of constraints in L. As remarked in Section 1, for constraints in a language to be effectively used in practice, it is often necessary to be able to answer these two questions at compile time.

One might be tempted to use a constraint language more powerful than CINDs, *e.g.,* full-fledged TGDs extended by allowing constants (data values). The question is whether the language allows us to effectively reason about its constraints. We need a constraint language that is powerful enough to express dependencies commonly found in schema matching and data cleaning, while at the same time well-behaved enough so that its associated decision problems are tractable or, at the very least, decidable [18]. For full-fledged TGDs, it was known 30 years ago that the implication problem is *undecidable* even in the *absence* of data values [5].

As found in most database textbooks, standard INDs have several nice properties. (a) INDs are always consistent. (b) For INDs, the implication problem is decidable (PSPACE-complete). (c) Better still, INDs are finitely axiomatizable, *i.e.,* there exists a finite inference system that is sound and complete for implication of CINDs. The question is: when constants are introduced into INDs as found in CINDs, does the extension of INDs still has these properties?

It was observed in [5] that if TGDs were extended by including data values, their analysis would become more intriguing. Although we are aware of no previous work on the static analyses of TGDs with constants, the study of CFDs [9] tells us that data values in the pattern tableaux of dependencies would make our lives much harder. In particular, in the consistency and implication problems, we have to consider whether or not the domain $\mathsf{dom}(A)$ of each attribute A in a dependency is finite, since a finite domain constrains how we can populate a relation that satisfies the dependencies.

In this section we investigate the consistency and implication problems of CINDs. We show that despite the fact that CINDs contain data values and are more expressive than INDs, they retain most of the nice properties of their standard IND counterpart. That is, CINDs properly balance the expressive power and complexity.

Normal form. To simplify the discussion, we will consider, without loss of generality, CINDs in normal form. A CIND ψ $(R_1[X; X_p] \subseteq R_2[Y; Y_p], T_p)$ is in the *normal form* if T_p consists of a single pattern tuple t_p such that $t_p[A]$ is a *constant* if and only if A is in X_p or Y_p. We write ψ as $(R_1[X; X_p] \subseteq R_2[Y; Y_p], t_p)$.

Two sets Σ_1 and Σ_2 of CINDs are *equivalent*, denoted by $\Sigma_1 \equiv \Sigma_2$, if for any instance D, $D \models \Sigma_1$ iff $D \models \Sigma_2$.

Proposition 3.1: *For a set Σ of* CINDs*, there exists a set Σ' of* CINDs *in the normal form such that $\Sigma \equiv \Sigma'$, and the size of Σ' is linear in the size of Σ.* □

Proposition 3.1 allows us to consider CINDs in the normal form in the sequel. It tells us that every CIND ψ can be rewritten as an equivalent set Σ_ψ of CINDs in the normal form. This can be done as follows: (1) if ψ has more than one pattern tuple, replace it with a set of CINDs, each with only one pattern tuple; (2) for each CIND in the set, remove from the patterns X_p and Y_p those attributes A if $t_p[A] = \_$; note that such pattern attributes pose no constraints; and (3) move to X_p and Y_p any pair (A_i, B_i) such that $A_i \in X$, B_i is the matching attribute of A in Y and $t_p(A_i)$ is a constant.

Example 3.1: CINDs ψ_1–ψ_4 in Fig. 2 are in the normal form, but ψ_5 and ψ_6 are not. We can transform ψ_5 into the normal form by separating it into two CINDs, each carrying only one pattern tuple of ψ_5; similarly for ψ_6. As another example, consider CIND $(R[A, B; C, D] \subseteq S[E, F; G], t_p)$ with $t_p = (\_, h; i, \_||\_, h; o)$. It is not in the normal form, but can be rewritten to $(R[A; B, C] \subseteq S[E; F, G], t'_p)$ with $t'_p = (\_; h, i||\_; h, o)$ in the normal form. □

3.1 Consistency of CINDs

One cannot expect to derive sensible schema matches or clean data from a set of constraints if it is inconsistent itself. Thus before any run-time computation is conducted, we have to make sure that the constraints are consistent, or make sense.

The *consistency problem* for a constraint language L is to determine, given a finite set Σ of constraints in L defined on a database schema $\mathcal{R}$, whether or not there exists a nonempty instance D of $\mathcal{R}$ such that $D \models \Sigma$.

Traditional FDs and INDs do not contain data values, and any set of FDs and INDs is consistent. However, adding data values to constraints may make their consistency analysis much harder. Indeed, CFDs, which extend FDs by adding patterns, may be inconsistent, as illustrated by the following example taken from [9].

Example 3.2: Consider a schema R with $\mathsf{attr}(R) = \{A, B\}$, and the CFDs below on R, refining standard FDs $A \to B$ and $B \to A$:

$$\begin{array}{ll} \phi_1\text{:}\ \ (A = \mathsf{true}) \to (B = b_1), & \phi_2\text{:}\ \ (A = \mathsf{false}) \to (B = b_2), \\ \phi_3\text{:}\ \ (B = b_1) \to (A = \mathsf{false}), & \phi_4\text{:}\ \ (B = b_2) \to (A = \mathsf{true}), \end{array}$$

where $\mathsf{dom}(A)$ is bool, and b_1, b_2 are two distinct constants in $\mathsf{dom}(B)$. CFD ϕ_1 (resp. ϕ_2) asserts that for any R tuple t, if $t[A]$ is true (resp. false), then $t[B]$ must be b_1 (resp. b_2). On the other hand, ϕ_3 (resp. ϕ_4) requires that if $t[B]$ is b_1 (resp. b_2), then $t[A]$ must be false (resp. true). There exists *no* nonempty instance of R satisfying all these CFDs. Indeed, for any R tuple t, no matter what Boolean value $t[A]$ has, these CFDs together force $t[A]$ to take the other value from the finite domain bool.

Note that if $\mathsf{dom}(A)$ and $\mathsf{dom}(B)$ were infinite, we could find a tuple t such that $t[A]$ is neither true nor false, and $t[B]$ is not b_1 or b_2; then the R instance $\{t\}$ satisfies these CFDs. This tells us that attributes with a finite domain may complicate the analysis. □

It was shown in [9] that the consistency problem for CFDs is NP-complete. As opposed to CFDs, we show that for CINDs the consistency analysis is as trivial as their standard counterpart.

Theorem 3.2: *For any set Σ of* CINDs *defined on a schema $\mathcal{R}$, there exists a nonempty instance D of $\mathcal{R}$ such that $D \models \Sigma$.* □

Proof Sketch: Given Σ, one can construct an instance of $\mathcal{R}$ as follows. First define an active domain for each attribute A in $\mathcal{R}$, consisting of the constants appearing in Σ plus at most one distinct value in $\mathsf{dom}(A)$. Then, build an instance of each relation schema in $\mathcal{R}$ as the cross product of the active domains of all attributes in it. This yields a nonempty instance of $\mathcal{R}$ satisfying Σ. □

3.2 Implication and Finite Axiomatization of CINDs

The *implication problem* for CINDs is to determine, given a finite set Σ of CINDs and another CIND ψ defined on a database schema $\mathcal{R}$, whether or not Σ entails ψ, denoted by $\Sigma \models \psi$, *i.e.,* whether or not for all instances D of $\mathcal{R}$, if $D \models \Sigma$ then $D \models \psi$.

Example 3.3: Let Σ be the set of CINDs given in Fig. 2, and assume that $\mathsf{dom}(\mathsf{at}) = \{$saving, checking$\}$. One wants to know whether $\Sigma \models \psi$, where $\psi = (\mathsf{account\_B}[\mathsf{at}; \mathsf{nil}] \subseteq \mathsf{interest}[\mathsf{at}; \mathsf{nil}], (\_||\_))$; *i.e.,* whether or not ψ is derivable from Σ. □

As remarked earlier, for standard INDs the implication problem is not only decidable but also finitely axiomatization. The finite axiomatizability is a property stronger than the decidability since inference rules reveal the essential properties of the constraints.

We now show that CINDs are also finitely axiomatizable. We provide an inference system for CINDs, denoted by $\mathcal{I}$ and shown in Fig. 3. Given a finite set Σ of CINDs and another CIND ψ, we denote by $\Sigma \vdash_{\mathcal{I}} \psi$ that ψ is provable from Σ using $\mathcal{I}$. The rules in $\mathcal{I}$ characterize CIND implication: they are both *sound, i.e.,* if $\Sigma \vdash_{\mathcal{I}} \psi$ then $\Sigma \models \psi$, and *complete, i.e.,* if $\Sigma \models \psi$ then $\Sigma \vdash_{\mathcal{I}} \psi$.

Theorem 3.3: *The inference system $\mathcal{I}$ is sound and complete for implication of* CINDs*.* □

Proof Sketch: The soundness of $\mathcal{I}$ is verified by induction on the length of $\mathcal{I}$-proofs, and its completeness is shown by using a chase technique (see, *e.g.,* [1] for the details of chase). □

Recall that for standard INDs, the inference system proposed in [11] consists of three rules: reflexivity, projection-permutation

CIND1: If X is a sequence of distinct attributes of R, then $(R[X; \mathsf{nil}] \subseteq R[X; \mathsf{nil}], t_p)$, where $t_p[A]$ = '_' for all $A \in X$.

CIND2: If $(R_a[A_1, \ldots, A_m; X_p] \subseteq R_b[B_1, \ldots, B_m; Y_p],\ t_p)$, then $(R_a[A_{i_1}, \ldots, A_{i_k}; X'_p] \subseteq R_b[B_{i_1}, \ldots, B_{i_k}; Y'_p], t'_p)$, where $\{i_1, .., i_k\}$ is a sequence in $\{1, \ldots, m\}$; X'_p and Y'_p are permutations of X_p and Y_p respectively; and $t'_p = t_p[A_{i_1}, \ldots, A_{i_k}; X'_p || B_{i_1}, \ldots, B_{i_k}; Y'_p]$.

CIND3: If $(R_a[X; X_p] \subseteq R_b[Y; Y_p], t_1)$, $(R_b[Y; Y_p] \subseteq R_c[Z; Z_p], t_2)$, and $t_1[Y_p] = t_2[Y_p]$, then $(R_a[X; X_p] \subseteq R_c[Z; Z_p], t_3)$, where $t_3[X; X_p] = t_1[X; X_p]$, and $t_3[Z; Z_p] = t_2[Z; Z_p]$.

CIND4: If $(R_a[X; X_p] \subseteq R_b[Y; Y_p],\ t_p)$, $X = \{A_1,\ \ldots,\ A_m\}$ and $Y = \{B_1, \ldots, B_m\}$, then $(R_a[X - A_j;\ X_p \cup A_j] \subseteq R_b[Y - B_j;\ Y_p \cup B_j],\ t'_p)$, where $A_j \in X$, $t'_p[A_j] \in \mathsf{dom}(A_j)$, $t'_p[A_j] = t'_p[B_j]$, and $t'_p[A] = t_p[A]$ for every $A \in (X, X_p, Y, Y_p) - (A_j, B_j)$.

CIND5: If $(R_a[X; X_p] \subseteq R_b[Y; Y_p], t_p)$, then $(R_a[X; X_p, A] \subseteq R_b[Y; Y_p], t'_p)$, where $A \in \mathsf{attr}(R_a) - (X \cup X_p)$, $t'_p[A] \in \mathsf{dom}(A)$, and $t'_p[X; X_p || Y; Y_p] = t_p$.

CIND6: If $(R_a[X; X_p] \subseteq R_b[Y; Y_p], t_p)$, then $(R_a[X; X_p] \subseteq R_b[Y; Y'_p], t'_p)$, where $Y'_p \subseteq Y_p$, $t'_p = t_p[X; X_p ||\ Y; Y'_p]$.

CIND7: If $(R_a[X; AX_p] \subseteq R_b[Y;\ Y_p],\ t_i)$ for $i \in [1, m]$, $t_1[X_p;\ Y_p] = \ldots = t_n[X_p;\ Y_p]$, $A \in \mathsf{finattr}(\mathcal{R})$, and $\mathsf{dom}(A)$ = $\{t_1[A], \ldots, t_n[A]\}$, then $(R_a[X; X_p] \subseteq R_b[Y; Y_p], t_p)$, where $t_p[X_p || Y_p] = t_1[X_p || Y_p]$.

CIND8: If $(R_a[X; AX_p] \subseteq R_b[Y; BY_p], t_i)$ for $i \in [1, n]$, $t_1[X_p;\ Y_p] = \ldots = t_n[X_p; Y_p]$; $t_i[A] = t_i[B]$ for $i \in [1, n]$, $A \in \mathsf{finattr}(\mathcal{R})$ and $\mathsf{dom}(A) = \{t_{p1}[A], t_{p2}[A], \ldots, t_{pn}[A]\}$, then $(R_a[XA; X_p] \subseteq R_b[YB; Y_p], t_p)$, where $t_p[X_p || Y_p]$ $= t_1[X_p || Y_p]$.

Figure 3: Inference System $\mathcal{I}$ for CINDs

and transitivity. To cope with the richer semantics of CINDs, the inference system $\mathcal{I}$ is more complicated than the one for INDs. Below we briefly illustrate the rules in $\mathcal{I}$.

Rules CIND1–CIND3 correspond to the inference rules for INDs. CIND1 is the reflexivity rule. CIND2 shows that also the pattern portions, *i.e.,* X_p and Y_p, can be permutated. CIND3 enforces that in order for the transitivity rule to be applied, not only the RHS of the first CIND has to be the same as the LHS of the second CIND, but also their respective portion of the tuple patterns. Note that since the CINDs are in the normal form, checking that $t_1[Y; Y_p] = t_2[Y; Y_p]$ is equivalent to checking $t_1[Y_p] = t_2[Y_p]$.

CIND4 allows us to instantiate attributes in X and their corresponding attributes in Y. Given $(R_a[X; X_p] \subseteq R_b[Y; Y_p], t_p)$, we can take attributes from X and the corresponding attributes in Y, replace their values in t_p by constants and move these attributes to the pattern portions of the CIND (X_p and Y_p, respectively).

CIND5 allows one to add extra attributes to X_p. Consider a CIND $(R_a[X; X_p] \subseteq R_b[Y;\ Y_p], t_p)$ and an attribute A of R_a which is not already in X or X_p. If ψ holds for any value of A, then it will also hold for a specific value of A. Thus we can add A to the pattern portion X_p and assign to $t_p[A]$ any constant from $\mathsf{dom}(A)$.

CIND6 removes an attribute from Y_p. If $(R_a[X; X_p] \subseteq R_b[Y; Y_p], t_p)$ holds, then for every tuple in R_a that satisfies the pattern $t_p[X_p]$, there is a match in R_b that satisfies the pattern $t_p[Y_p]$. If attributes are deleted from Y_p, the CIND will clearly still hold.

Finally, CIND7 and CIND8 are only needed when there are finite domains. CIND7 says that if we have a set of CINDs that are the same except for the value $t_p[A]$ of a finite-domain attribute A, and the union of all those $t_p[A]$ values covers the domain of A, then we can replace the set of CINDs by a single CIND in which $t_p[A] =$ _ . Furthermore, since a variable in the pattern portion of the CIND has no effect, we can just delete A from the CIND.

CIND8 is, in a way, the inverse of CIND4. If CIND4 is used over a CIND ψ to instantiate the values in the pattern tuple for attributes A and B when $t_p[A]$ ranges over all the values of $\mathsf{dom}(A)$, then CIND8 can take all those CINDs and restore ψ. In short, CIND8 merges a set of CINDs if (1) they differ only in the value of $t_i[A]$, (2) $t_i[A]$ ranges over all the values in $\mathsf{dom}(A)$, and (3) there is an attribute B in the RHS of each CIND such that $t_i[A] = t_i[B]$.

Example 3.4: Recall Σ and ψ from Example 3.3, where $\mathsf{dom}(\mathsf{at}) = \{$checking,saving$\}$. We show that $\Sigma \vdash_{\mathcal{I}} \psi$; then from Theorem 3.3 it follows that $\Sigma \models \psi$.

(1)	(account_B [nil; at] $\subseteq$ saving[nil; ab], t_1) t_1 =(saving||B)	ψ_1, CIND2
(2)	(account_B [nil; at] $\subseteq$ checking[nil; ab], t_2) t_2=(checking||B))	ψ_2, CIND2
(3)	(saving[nil; ab] $\subseteq$ interest[nil; at], t_3) t_3 =(B||saving)	ψ_5, CIND2
(4)	(checking[nil; ab] $\subseteq$ interest[nil; at], t_4) t_4 =(B||checking)	ψ_6, CIND2
(5)	(account_B [nil; at] $\subseteq$ interest[nil; at], t_5) t_5=(saving||saving)	(1),(3),CIND3
(6)	(account_B [nil; at] $\subseteq$ interest[nil; at], t_6) t_6 =(checking||checking)	(2),(4),CIND3
(7)	(account_B [at; nil] $\subseteq$ interest[at; nil], t_7) t_7 =(_||_))	(5),(6),CIND8

□

It is not surprising that the implication problem of CINDs is harder than standard INDs. The lower bound of the theorem below is verified by reduction from the two-player tiling problem [12].

Theorem 3.4: *The implication problem for* CINDs *is* EXPTIME-*complete.* □

The complication of the implication problem arises from examining attributes with a finite domains. In the absence of such attributes, there is a linear-space non-deterministic algorithm that uses only rules CIND1–CIND6 in $\mathcal{I}$. In this case, the implication problem for CINDs has precisely the same complexity as its IND counterpart, namely, the problem becomes PSPACE-complete.

Theorem 3.5: *For any set $\Sigma \cup \{\psi\}$ of* CINDs *defined on a schema $\mathcal{R}$, it is* PSPACE*-complete to decide whether or not $\Sigma \models \psi$, if neither Σ nor ψ involves $\mathcal{R}$ attributes that have a finite domain. In this setting, the inference rules* CIND1– CIND6 *are sound and complete for implication of* CINDs*.* □

4. Interaction between CINDs and CFDs

We have seen that CINDs do not make the consistency and implication problems much harder than their traditional counterparts. In contrast, we show in this section that when CINDs and CFDs are taken together, the static analysis become far more intriguing. As remarked earlier, in schema matching and data cleaning it is often necessary to use both CINDs and CFDs.

We start with a review of CFDs, which were introduced in [9].

CFDs. A *conditional functional dependency* (CFD) ϕ on a relation R is a pair $(R : X \rightarrow Y,\ T_p)$, where (1) X and Y are subsets of $\mathsf{attr}(R)$; (2) $R : X \rightarrow Y$ is a standard FD, referred to as the FD *embedded in* ϕ; and (3) T_p is a tableau with all attributes in X and Y, referred to as the *pattern tableau* of ϕ, where for each A in X or Y and each tuple $t \in T_p$, $t[A]$ is either a constant $a \in \mathsf{dom}(A)$, or an unnamed variable '_', as defined for CINDs given earlier.

An instance D of R *satisfies* the CFD ϕ, denoted by $D \models \phi$, iff for *each pair* of tuples t_1, t_2 in the relation D, and for *each* tuple t_p in the pattern tableau T_p, if $t_1[X] = t_2[X] \asymp t_p[X]$, then $t_1[Y] = t_2[Y] \asymp t_p[Y]$. That is, if $t_1[X]$ and $t_2[X]$ are equal and match the pattern $t_p[X]$, then $t_1[Y]$ and $t_2[Y]$ must also be equal to each other and match the pattern $t_p[Y]$.

φ_1 = (saving (an, ab $\rightarrow$ cn, ca, cp), T'_1)

T'_1:

an	ab	cn	ca	cp
_	_	_	_	_

φ_2 = (checking (an, ab $\rightarrow$ cn, ca, cp), T'_2)

T'_2:

an	ab	cn	ca	cp
_	_	_	_	_

φ_3 = (interest (ct, at $\rightarrow$ rt), T'_3)

T'_3:

ct	at	rt
_	_	_
UK	saving	4.5%
UK	checking	1.5%
US	saving	4%
US	checking	1%

Figure 4: Example CFDs

Example 4.1: The FDs fd_1-fd_3 given in Example 1.2 can be expressed as CFDs, as shown in Fig. 4. This tells us that standard FDs are a special case of CFDs in which the pattern tableau contains a single tuple that consists of '_' only.

We can refine fd_3 by asserting that when ct is UK (resp. US) and at is saving, rt must be 4.5% (resp. 4%); similarly, if ct is UK (resp. US) and at is checking, rt must be 1.5% (resp. 1%). These are incorporated into φ_3 of Fig. 4 (the last 4 tuples, one per constraint).

While the instance of Fig. 1 satisfies standard FDs fd_1-fd_3 and it satisfies φ_1 and φ_2, it does not satisfy φ_3. Indeed, tuple t_{12} of Fig. 1 *violates* the constraint specified by the third pattern tuple t_p^3 in T'_3: although $t_{12}[\mathsf{ct}, \mathsf{at}] \asymp t_p^3[\mathsf{ct}, \mathsf{at}]$, we can see that $t_{12}[\mathsf{rt}] \not\asymp t_p^3[\mathsf{rt}]$: $t_{12}[\mathsf{rt}]$ is 10.5% but $t_p^3[\mathsf{rt}]$ is 1.5%. From this we can see that while it takes at least two tuples to violate a standard FD, a *single tuple* alone may violate a CFD. Moreover, CFDs can catch inconsistencies that standard FDs cannot detect. □

Along the same lines as CINDs in normal form, we say that a CFD $\phi = (R : X \rightarrow Y,\ T_p)$ is in the *normal form* if T_p consists of a single tuple t_p and Y contains a single attribute A, and we write ϕ as $(R : X \rightarrow A,\ t_p)$. We can always rewrite a CFD into an equivalent set of CFDs in the normal form. In the sequel, we only consider CFDs in the normal form.

For CFDs the following have been established in [9]. (a) The consistency problem for CFDs is NP-complete. (b) The implication problem of CFDs is finitely axiomatizable. (c) The implication problem for CFDs is coNP-complete. (d) The consistency and implication problems are in $O(n^2)$ time, where n is the size of the given CFDs, if the CFDs do not involve attributes with a finite domain.

While CFDs alone already complicate the static analyses, we next show that CFDs and CINDs together make our lives much harder.

Implication analysis. It is not surprising that the implication problem for CINDs and CFDs is undecidable and is not finitely axiomatizable, since the problem has already these characteristics for standard INDs and FDs (see, *e.g.,* [1]), and CINDs and CFDs subsume INDs and FDs, respectively. The result holds if the given constraints do not involve attributes with a finite domain.

Corollary 4.1: *The implication problem for* CINDs *and* CFDs *is undecidable, and is not finitely axiomatizable, even for* CINDs *and* CFDs *that involve only attributes with an infinite domain.* □

Consistency analysis. Even if a set of CFDs and a set of CINDs are separately consistent, when they are put together, there may be conflicts among them, as illustrated below.

Example 4.2: Consider a relation R with $\mathsf{attr}(R) = \{A, B\}$, on which we define a CFD $\phi = (R : A \rightarrow B, (\_ \| a))$ and a CIND $\psi = (R[\mathsf{nil}; B] \subseteq R[\mathsf{nil}; B], (\_ \| b))$, where a and b are distinct constants. Obviously, there exists a nonempty instance of R that satisfies ϕ and there is an instance satisfying ψ. However, there exists *no* nonempty instance of R that satisfies both ψ and ϕ. To see this, assume that such an instance D exists. Then ψ tells us that as long as D is nonempty, there is a tuple t in D such that $t[B] = b$. In contrast, ϕ requires that $t[B] = a$, violating ψ. □

Constraints	Consistency	Implication	Fin. Axiom
CINDs (Th. 3.2, 3.4, 3.3)	$O(1)$	EXPTIME-complete	Yes
CFDs [9]	NP-complete	coNP-complete	Yes
CFDs + CINDs (Th 4.2, 4.1)	undecidable	undecidable	No

Table 1: Complexity in the general setting

Constraints	Consistency	Implication	Fin. Axiom
CINDs (Th. 3.2, 3.5)	$O(1)$	PSPACE-complete	Yes
CFDs [9]	$O(n^2)$	$O(n^2)$	Yes
CFDs + CINDs (Th 4.2, 4.1)	undecidable	undecidable	No

Table 2: Complexity in the absence of finite-domain attributes

While the undecidability of the implication problem for CINDs and CFDs is expected, the following result is a little surprising. The undecidability can be verified by reduction from the implication problem for standard FDs and INDs. The undecidability remains intact in the absence of attributes with a finite domain.

Theorem 4.2: *The consistency problem for* CFDs *and* CINDs *is undecidable, with or without attributes having a finite domain.* □

This tells us that it is necessary to use heuristic methods to solve the consistency and implication problems in practice.

Summary. We summarize the complexity bounds for the consistency and implication problems, as well as for finite axiomatizability (Fin. Axiom) in Tables 1 and 2. Table 1 gives the results in the general setting where attributes of infinite domains and those with finite domains are both present, and Table 2 for constraints involving attributes with an infinite domain only. This gives us a complete picture of the static analyses for CINDs and CFDs, established in this work (for CINDs, and CINDs + CFDs) and in [9] (for CFDs).

5. Algorithms for Consistency Analysis

In light of the undecidability of the consistency problem for CINDs and CFDs, in this section we develop efficient heuristic methods to check the consistency of CINDs and CFDs.

More specifically, given a set Σ of CINDs and CFDs, our algorithms attempt to construct a nonempty *witness database* D such that $D \models \Sigma$. The algorithms conclude that Σ is consistent, and return true, if such a witness can be built. It is guaranteed that if true is returned then Σ is consistent. However, the algorithms might not find a witness database even if Σ is consistent, due to the undecidability of the problem. As will be seen in the next section, the algorithms are able to return accurate answers in most cases.

The algorithms are based on an extension of the chase technique, bounded-size witness databases, and an optimization technique leveraging dependency graphs of CINDs and CFDs. We extend the chase in Section 5.1, present a checking algorithms in Section 5.2 and provide our optimization technique in Section 5.3.

5.1 Chasing with CFDs and CINDs

The chase is an important tool for implication analysis of dependencies and for query optimization (see, *e.g.,* [1] for details about chase). However, even for standard INDs there may be *infinite* chasing sequences, *i.e.,* the chase may not terminate. To cope with this, we present an extension of the chase that, employs tables with bounded-size, therefore, guaranteeing termination. We use this extension of the chase for the *consistency* analysis of CFDs and CINDs.

Consider a database schema $\mathcal{R}$. For each relation schema R in $\mathcal{R}$ and each attribute A in R, we assume a nonempty finite set $\mathsf{var}[A]$ of distinct variables. Intuitively, when chasing with CINDs, we may have to create a new tuple; then we use only the variables in these sets to "populate" the unknown fields in the tuple. All the sets $\mathsf{var}[A]$ have a maximum size of N, which is a predefined parameter. Let Var be the set consisting of all these variable. We assume for convenience a total order $<$ on variables in Var. We also assume that $v < a$ for any $v \in \mathsf{Var}$ and constant a, but do not pose the order on constants. Thus $v \neq a$ and $v \not\asymp a$; but we allow $v \asymp$'_'.

We now define our chase operations for a set Σ of CINDs and CFDs, which transform a database D into a new database D'. To simplify the discussion we denote by R a schema as well as an instance of the schema when it is clear from the context.

For each CIND $\psi = (R_a[A_1, \ldots, A_m; X_p] \subseteq R_b[B_1, \ldots, B_m; Y_p],\ t_p)$ in Σ, we define the chase operation $\mathsf{IND}(\psi)$ as follows. For a tuple $t_a \in R_a$ satisfying $t_a[X_p] = t_p[X_p]$, we add a tuple t_b to R_b such that $t_b[B_i] = t_a[A_i]$ for $i \in [1, m]$, $t_b[Y_p] = t_p[Y_p]$, and $t_b[B]$ takes a random variable from $\mathsf{var}[B]$ for the rest attribute $B \in \mathsf{attr}(R_b) - (\{B_1, \ldots, B_k\} \cup Y_p)$.

For each CFD $\phi = (R : X \rightarrow A, t_p)$ in Σ, we define the chase operation $\mathsf{FD}(\phi)$ as follows. For tuples $t_1, t_2 \in R$ such that $t_1[X] = t_2[X] \asymp t_p[X]$, but either $t_1[A] \neq t_2[A]$ or $t_1[A] = t_2[A] \not\asymp t_p[A]$, we consider the following two cases:

(i) $t_p[A] =$'_': if either $t_1[A]$ or $t_2[A]$ is a variable and $t_1[A] < t_2[A]$ (resp. $t_2[A] < t_1[A]$), we replace $t_1[A]$ with $t_2[A]$ in R (resp. replace $t_2[A]$ with $t_1[A]$). If $t_1[A]$ and $t_2[A]$ are different constants, then the application of $\mathsf{FD}(\phi)$ to D is not defined.

(ii) $t[A] = a$: if either $t_1[A]$ or $t_2[A]$ is a constant distinct from a, then the application of $\mathsf{FD}(\phi)$ is undefined. Otherwise we replace both $t_1[A]$ and $t_2[A]$ with a.

A *chasing sequence* of D *w.r.t.* Σ is a sequence of database templates (with variables) $D_0, D_1, \ldots, D_n$ such that $D_0 = D$ and D_{i+1} is the result of applying a chase operation for a constraint in Σ to D_i. If $\mathsf{IND}(\psi)(D_n) = D_n$ for every CIND $\psi \in \Sigma$ and $\mathsf{FD}(\phi)(D_n) = D_n$ for every CFD $\phi \in \Sigma$, we say that the chase of Σ over D is *terminal* and refer to D_n as the *result* of the chase, denoted by $\mathsf{chase}(D, \Sigma)$. Otherwise, $\mathsf{FD}(\phi)$ must be undefined for some $\phi \in \Sigma$, and in this case we say that $\mathsf{chase}(D, \Sigma)$ is *undefined*. Since the chase takes values from a predefined finite set of variables, it will *always terminate*. Note that for a set of CINDs only, the chase is always defined.

5.2 Heuristic Methods for Consistency Checking

Employing this extension of the chase, we next develop a heuristic method for checking the consistency of CFDs and CINDs.

For any set Σ of CINDs and CFDs defined over $\mathcal{R}$, if Σ does not involve attributes that have finite domains, a possible heuristic to determine if Σ is consistent works as follows: (1) it first constructs a database D that only contains, in a randomly chosen relation $R \in \mathcal{R}$, a tuple $t = (v_1, \ldots, v_n)$ such that $t[A_i] = v_i$ is from $\mathsf{var}[A_i]$; (2) it then checks whether $\mathsf{chase}(D, \Sigma)$ is defined; and (3) it return true if the chase is defined. One can see that if $\mathsf{chase}(D, \Sigma)$ is defined then Σ is consistent, as illustrated by the example below.

Example 5.1: Consider $\mathcal{R} = (R_1, R_2)$, where $\mathsf{attr}(R_1) = \{\mathsf{E}, \mathsf{F}\}$, $\mathsf{attr}(R_2) = \{\mathsf{G}, \mathsf{H}\}$, $\mathsf{finattr}(\mathcal{R}) = \emptyset$, and the domain of all the attributes is string. Also consider $\Sigma = \{\phi_1, \phi_2, \psi_1, \psi_2, \psi_3\}$, where $\phi_1 = (R_1 : \mathsf{E} \rightarrow \mathsf{F}, (\_||\_))$, $\phi_2 = (R_2 : \mathsf{H} \rightarrow \mathsf{G}, (\_||c))$, $\psi_1 = (R_1[\mathsf{E};\ \mathsf{nil}] \subseteq R_2[\mathsf{G};\ \mathsf{nil}], (\_||\_))$, $\psi_2 = (R_2[\mathsf{nil};\ \mathsf{H}] \subseteq R_1[\mathsf{nil}; \mathsf{F}], (0||a))$ and $\psi_3 = (R_2[\mathsf{nil};\ \mathsf{H}] \subseteq R_1[\mathsf{nil}; \mathsf{F}], (1||b))$.

The heuristic mentioned above works as follows. Let $\mathsf{var}[\mathsf{A}] = \{v_{\mathsf{A1}}, v_{\mathsf{A2}}\}$ for $\mathsf{A} \in \{\mathsf{E}, \mathsf{F}, \mathsf{G}, \mathsf{H}\}$. We start with D that contains tuple $(v_{\mathsf{E1}}, v_{\mathsf{E2}})$ in R_1. After applying $\mathsf{IND}(\psi_1)$, tuple $(v_{\mathsf{E1}}, v_{\mathsf{H1}})$ is added to R_2. Then, $\mathsf{FD}(\phi_2)$ makes $v_{\mathsf{E1}} = c$. No chase operation can be applied after that, and $\mathsf{chase}(D, \Sigma)$ is:

R_1:

E	F
c	v_{F1}

R_2:

G	H
c	v_{H1}

```
Algorithm RandomChecking
Input: A set $\Sigma$ of CINDs and CFDs over schema $\mathcal{R} = (R_1, \ldots, R_n)$
Output: true if a database $D$ can be built s.t. $D \models \Sigma$; false otherwise
1. $D :=$ an instance of $\mathcal{R}$ that contains, for a randomly chosen schema
         $R_i \in \mathcal{R}$, a single-tuple instance of fresh variables from Var;
2. $k := 0$;
3. while $\mathcal{V}_{\mathsf{finattr}(\mathcal{R})} \neq \emptyset$ or $k \leq K$ do
4.       randomly choose $\rho \in \mathcal{V}_{\mathsf{finattr}(\mathcal{R})}$;
5.       $\mathcal{V}_{\mathsf{finattr}(\mathcal{R})} := \mathcal{V}_{\mathsf{finattr}(\mathcal{R})} - \{\rho\}$;  $k := k + 1$;
6.       if $\mathsf{chase}_I(\rho(D), \Sigma)$ is defined then
7.         return true;
8. return false;
```

Figure 5: Algorithm RandomChecking

The heuristic concludes that Σ is consistent. Indeed, since the domain of F and H are infinite, it is always possible to find a mapping from the variables to values in the respective domains such that they do not satisfy the left pattern of any CIND and CFD. For example, by mapping $v_{\mathsf{F1}} = d$ and $v_{\mathsf{H1}} = e$, we obtain a database instance of $\mathcal{R}$ that satisfies Σ. □

In contrast, if Σ involves attributes with finite domains, we can no longer use $\mathsf{chase}(D, \Sigma)$ as above, as shown by the next example.

Example 5.2: Consider Σ of Example 5.1. If instead of having an infinite domain for H we had $\mathsf{dom}(\mathsf{H}) = \{0, 1\}$, then it is not always possible to find a valuation for the variables such that the result database of the chase *w.r.t.* the valuation satisfies Σ. For example, for $v_{\mathsf{H1}} = 1$, we could still apply $\mathsf{IND}(\psi_3)$. If, for example, there are also $\psi_4 = (R_1[\mathsf{nil};\ \mathsf{F}] \subseteq R_2[\mathsf{nil}; \mathsf{G}], (a||d))$, and $\psi_5 = (R_1[\mathsf{nil};\ \mathsf{F}] \subseteq R_2[\mathsf{nil}; \mathsf{G}], (b||d))$, then $\mathsf{IND}(\psi_5)$ would now apply, resulting in a database that does not satisfy Σ because of ϕ_2. □

Algorithm RandomChecking**.** To cope with finite domains, we develop an algorithm, called RandomChecking and given in Fig. 5.

While the chase given above always terminates, it may yield a witness database of exponential size. To avoid this, we adopt two further simplifications. (a) When applying $\mathsf{IND}(\psi)$ for a $\psi \in \Sigma$, we need to add a new tuple that might have variables. If this variable is for an attribute with a finite domain, we modify $\mathsf{IND}(\psi)$ in such a way that instead of adding a variable, a constant of the finite domain is used. (b) During the chase, if the number of tuples in any table exceeds a predefined threshold T, we say that the chase is undefined and terminate the process. The chase with these two simplifications is referred to as the *instantiated chase*, and is denoted by $\mathsf{chase}_I(D, \Sigma)$. More specifically, let V be the set of all variables associated with attributes that have finite domains. A *valuation* ρ_V *w.r.t.* V is a mapping from V to constants in the respective domains of the variables. We denote by $\rho(D)$ the database D obtained by applying ρ to D. Note that constants and variables with infinite domains in D remain *unchanged* in $\rho(D)$. The set of all valuations *w.r.t.* V is denoted by $\mathcal{V}_{\mathsf{finattr}(\mathcal{R})}$. If $V = \emptyset$, then we assume that $\mathcal{V}_{\mathsf{finattr}(\mathcal{R})}$ consists of a single empty mapping.

Algorithm RandomChecking starts by creating a database D that, for a randomly chosen relation $R \in \mathcal{R}$, contains a tuple $(v_1, \ldots, v_n)$ such that v_i for attribute A_i is a variable from $\mathsf{var}[A_i]$ (line 1). For a predefined parameter K, it then randomly picks up to K valuations ρ from $\mathcal{V}_{\mathsf{finattr}(\mathcal{R})}$, and checks whether chase$_I(\rho(D), \Sigma)$ is defined (lines 3-5). If it is for any such ρ, then the algorithm immediately returns true (lines 6-7). Otherwise false is returned (line 8). The use of K is to prevent the exponential cost of exploring all possible valuations in $\mathcal{V}_{\mathsf{finattr}(\mathcal{R})}$ in the worst case. However, as will be seen in the next section, in many practical cases K is not necessary because a positive answer can often be found before many valuations are tried out.

Example 5.3: Applying to the constraints Σ of Example 5.1 with $\mathsf{dom(H)} = \{0, 1\}$, Algorithm RandomChecking works as follows. After executing line 1 of the algorithm, D could contain a tuple $(v_{\mathsf{G1}}, v_{\mathsf{H1}})$ in R_2. The only variable with a finite attribute is v_{H1} and its possible mappings are ρ_1 and ρ_2 that maps v_{H1} to 0 and 1, respectively. A sequence of the instantiated chase *w.r.t.* $\rho_1(D)$ is:

	R_1		R_2		
	E	F	G	H	
D_1	v_{E1}	a	v_{G1}	0	IND(ψ_2) applied to $\rho_1(D)$
D_2	v_{E1}	a	c	0	FD(ϕ_2) applied to D_1
D_3	v_{E1}	a	c v_{E1}	0 0	IND(ψ_1) applied to D_2
D_4	c	a	c	0	FD(ϕ_2) applied to D_3

Since chase$_I(D, \Sigma)$ is defined and results in database D_4 (which satisfies the constraints), the algorithm returns true and does not need to check the chase for mapping ρ_2. □

Improvement. While conceptually simple, it may hamper the chance of finding a witness database if we assign a value to *every* variable with a finite domain *before* the chase starts. To rectify this, before applying a valuation ρ from $\mathcal{V}_{\mathsf{finattr}(\mathcal{R})}$, we first chase with CFDs in Σ, which may *instantiate* certain variables by imposing constant bindings in their pattern tuples. This requires a procedure CFD_Checking that, given a database D_i (with variables) in a chase sequence, chases with *only* CFDs in Σ; that is, it applies FD(ϕ) for every CFD ϕ in Σ that is applicable to D_i, instantiating variables in terms of constants in the pattern tuples when possible. The procedure applies ρ from $\mathcal{V}_{\mathsf{finattr}(\mathcal{R})}$ only to the *remaining* variables with a finite domain that have not been assigned a value during the chase. Procedure CFD_Checking returns a database D_{i+1} in which *all* variables with finite domains have constant values, if D_{i+1} is consistent with the CFDs in Σ, and it fails otherwise.

Capitalizing on CFD_Checking, algorithm RandomChecking works as follows. It starts with chase$_I(D, \Sigma)$, and randomly picks a constraint in Σ to chase with. Every time a new tuple is added to the database as a result of some IND(ψ), it invokes procedure CFD_Checking, which instantiates all variables with finite domains as described above. If CFD_Checking fails, chase$_I(D, \Sigma)$ is undefined and the algorithm starts another random run. Eventually either chase$_I$ is defined in some run and thus RandomChecking returns true, or chase$_I(D, \Sigma)$ is undefined for all K runs and the algorithm returns false. This is the algorithm we have implemented.

Procedure CFD_Checking (not shown due to lack of space) can be implemented either as described above, or by leveraging existing tools for known NP problems, since the consistency problem for CFDs is in NP [9]. In the latter case, we reduce it to SAT, a well-known NP-problem, and then check the consistency of the CFDs by using SAT4j [19], a well-developed tool.

5.3 Optimization: Dependency Graph Analysis

To further improve the accuracy and response time of our algorithms, we next present an optimization technique, based on a notion of dependency graphs of CFDs and CINDs. Below we first define dependency graphs. We then present a consistency checking algorithm that benefits from the usage of dependency graphs.

Dependency graph. For a set Σ of CFDs and CINDs defined over a database schema $\mathcal{R}$, the *dependency graph* is defined to be $\mathcal{G}[\Sigma] = (\mathcal{V}, \mathcal{E})$. The set $\mathcal{V}$ contains one vertex per relation R_i in $\mathcal{R}$. Each vertex R_i is associated with the set of CFDs defined on R_i in Σ, denoted by CFD(R_i), and a tuple template τ, denoted by $\tau(R_i)$, which consists of distinct variables in each attribute of R_i. Later, τ will be instantiated to be a tuple that satisfies all the CFDs in CFD(R_i) if CFD(R_i) is consistent. The set $\mathcal{E}$ contains an edge from vertex R_i to R_j if there is at least one CIND from R_i to R_j in Σ. Furthermore, the edge is labeled with the set of all CINDs from R_i to R_j, denoted by CIND(R_i, R_j).

Example 5.4: Consider the following extension of the schema and constraints of Example 5.1: $\mathcal{R} = \{R_1, R_2, R_3, R_4, R_5\}$, $\mathsf{attr}(R_1) = \{E, F\}$, $\mathsf{attr}(R_2) = \{G, H\}$,$\mathsf{attr}(R_3) = \{A, B\}$, $\mathsf{attr}(R_4) = \{C, D\}$, $\mathsf{attr}(R_5) = \{I, J\}$, $\mathsf{finattr}(\mathcal{R}) = \{H\}$ and $\mathsf{dom}(H)$ is bool. Also consider $\Sigma = \{\phi_1, \phi_2, \phi_3, \phi_4, \phi_5, \phi_6, \psi_1, \psi_2, \psi_3, \psi_4, \psi_5\}$, where ϕ_1–ϕ_2 and ψ_1–ψ_3 are those given in Example 5.1, and $\phi_3 = (R_3 : A \to B, (c \| \_))$, $\phi_4 = (R_4 : C \to D, (\_ \| a))$, $\phi_5 = (R_4 : C \to D, (\_ \| b))$, $\phi_6 = (R_5 : I \to J, (\_ \| c))$, $\psi_3 = (R_2[\mathsf{nil}; H] \subseteq R_1[\mathsf{nil}; F], (1 \| b))$, $\psi_4 = (R_3[A; B] \subseteq R_4[C; \mathsf{nil}], (\_; b \| \_))$, and $\psi_5 = (R_5[\mathsf{nil}; J] \subseteq R_2[\mathsf{nil}; G], (c \| d))$. The graph $\mathcal{G}[\Sigma]$ is depicted in Fig. 6. Each node in $\mathcal{G}[\Sigma]$ is associated with a set of CFDs: CFD$(R_1) = \{\phi_1\}$, CFD$(R_2) = \{\phi_2\}$, CFD$(R_3) = \{\phi_3\}$, CFD$(R_4) = \{\phi_4, \phi_5\}$ and CFD$(R_5) = \{\phi_6\}$. □

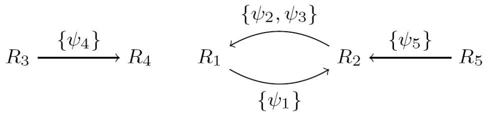


Figure 6: Graph $\mathcal{G}[\Sigma]$

In a nutshell, we want to reduce $\mathcal{G}[\Sigma]$ by removing any node R (and its related edges) for which CFD(R) is *inconsistent* and thus has to be empty in any instance of $\mathcal{R}$ that satisfies Σ. The reduction is conducted with care such that it will not generate impact on the consistency analysis on the remaining graph. When the graph cannot be further reduced, it consists of strongly connected components such that if Σ is consistent, then all relations in some of those components have to be nonempty. Furthermore, for each relation R' in a component, CFD(R') is consistent. This allows us to reduce the consistency analysis on $\mathcal{R}$ to the analysis on a single component. Better still, in some cases the graph reduction tells us whether or not Σ is consistent. For example, if the final $\mathcal{G}[\Sigma]$ is empty then there is no relation R for which CFD(R) is consistent; as a result Σ is inconsistent. On the other hand, we can conclude that Σ is consistent if there is R such that $\tau(R) \models$ CFD(R) and the (instantiated) tuple $\tau(R)$ does not *trigger* any CIND in Σ, *i.e.,* there is no CIND $(R[X; X_p] \subseteq R'[Y; Y_p], t_p)$ in Σ such that $\tau(R)[X_p] \asymp t_p[X_p]$. This is because a consistent instance of $\mathcal{R}$ can be built such that it consists of (a) $\{\tau(R)\}$ as the instance of R, and (b) empty instances for all other relation schemas.

We formalize this idea in algorithm preProcessing, shown in Fig. 7. First, the algorithm performs a topological sort on vertexes in $\mathcal{G}[\Sigma]$ (line 1) such that for any R_i and R_j in $\mathcal{G}[\Sigma]$, (a) if they are on a cycle, then an arbitrary order on R_i and R_j is adopted, and (b) otherwise, if there is edge from R_i to R_j then R_j precedes R_i. The order is stored in a *queue* Q. Second, for each relation R in Q, algorithm CFD_Checking is called to check the consistency of CFD(R) (lines 3-4). After running CFD_Checking, if the set CFD(R) is consistent, $\tau(R)$ becomes a tuple that satisfies CFD(R). Furthermore,

```
Algorithm preProcessing
Input: The dependency graph $\mathcal{G}(\Sigma)$ of a set $\Sigma$ of CINDs and CFDs.
Output: $\mathcal{G}(\Sigma)$ is reduced, containing only strongly connected components;
        1 is returned if a database $D$ such that $D \models \Sigma$ is found, 0 if it
        can conclude that $\Sigma$ is inconsistent, and $-1$ otherwise.
1.  $Q$:= a topological order of nodes in $\mathcal{G}(\Sigma)$;
2.  while $Q$ is not empty do
3.    $R$ := $Q$.dequeue();
4.    if CFD_Checking(CFD($R$), $\tau(R)$) then
5.        if $\tau(R)$ does not trigger any CIND in $\Sigma$ then
6.            return 1;
7.    else
8.        for each $R_j$ such that $(R_j, R) \in \mathcal{E}(\mathcal{G}[\Sigma])$
9.            add CIND$(R_j, R)^{\perp}$ to CFD($R_j$);
10.           if $R_j$ is not in $Q$ then
11.               $Q$.enqueue($R_j$);
12.       Delete node $R$ from $\mathcal{G}[\Sigma]$;
13. Delete all nodes of $\mathcal{G}$ with indegree = 0;
14. if $\mathcal{G}(\Sigma)$ is empty then
15.   return 0;
16. return $-1$;
```

Figure 7: Algorithm preProcessing

if $\tau(R)$ does not *trigger* CIND in Σ, then we can conclude that Σ is consistent, and return 1 (lines 5-6).

Now, if the set CFD(R) is inconsistent, we know that no database that satisfies Σ can have an nonempty R. We can thus delete node R from $\mathcal{G}[\Sigma]$ after adding *non-triggering* CFDs to prevent all the neighboring relations from inserting tuples into R (lines 7-12). More specifically, for each R_j and each CIND $(R_j[X; X_p] \subseteq R[Y; Y_p], t_p)$ in Σ, we add non-triggering CFDs $(R_j : X_p \to A, (t_p[X_p] \mid\mid c_1))$ and $(R_j : X_p \to A, (t_p[X_p] \mid\mid c_2))$, where $A \in \mathsf{attr}(R_j)$ and c_1, c_2 are distinct constants in $\mathsf{dom}(A)$. These two CFDs deny any tuple in R_j that matches the pattern X_p. We use CIND$(R_j, R)^{\perp}$ to denote the set of all such non-triggering CFDs for R_j and its CINDs. If non-triggering CFDs are added to a node R_j for which CFD(R_j) was already checked for consistency, then R_j has to be added back to Q to make sure the updated CFD(R_j) is still consistent (line 11).

After checking the local consistency of CFDs for all nodes in $\mathcal{G}[\Sigma]$, the graph contains only relations for which the set of CFDs is consistent. If there is a node R that has no incoming edges, it can also be deleted (line 13), since we can make R empty without any impact on finding a consistent instance of Σ. If after the process the graph is empty, we can conclude that Σ is inconsistent and return 0 (lines 14-15). Otherwise, whether or not Σ is consistent cannot be decided at this point, and thus -1 is returned.

Example 5.5: Continuing with Example 5.4, let $\mathcal{G}[\Sigma]$ be the graph of Fig. 6. Algorithm preProcessing starts by performing a topological sort. One possible output is $Q = [R_4, R_3, R_1, R_2, R_5]$.

In the first while-iteration $R = R_4$ and $Q = [R_3, R_1, R_2, R_5]$. Procedure CFD_Checking returns false since CFD$(R_4) = \{\phi_4, \phi_5\}$ is inconsistent. Thus R_4 is deleted from $\mathcal{G}[\Sigma]$ after adding CFDs to R_3 in order to ensure that ψ_4 is not triggered. Now CFD$(R_3) = \{\phi_3, (R_3 : B \to A, (b||c_1)), (R_3 : B \to A, (b||c_2))\}$. Since R_4 is deleted from $\mathcal{G}[\Sigma]$, edge (R_3, R_4) no longer exists.

In the next iteration, $R = R_3$ and $Q = [R_1, R_2, R_5]$. Procedure CFD_Checking returns true since CFD(R_3), including the non-triggering constraints added in the previous step, is consistent. In fact, $\tau(R_3)$ could be (v_1, v_2) where v_1 and v_2 are variables. This means that since attributes A and B are infinite, it is always possible find constants in the domains such that the CFDs are satisfied. Better still, since R_3 has no outgoing edges, $\tau(R_3)$ does not trigger any CIND. This implies that $\tau(R_3) \models \Sigma$ and that Σ is consistent. At this point preProcessing returns 1.

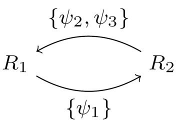


Figure 8: Graph $\mathcal{G}[\Sigma]$ after preProcessing

```
Algorithm Checking
Input: A set $\Sigma$ of CINDs and CFDs over schema $\mathcal{R} = (R_1, \ldots, R_n)$
Output: true if a database $D$ can be built s.t. $D \models \Sigma$; false otherwise
1.  $\mathcal{G}$ := the dependency graph $\mathcal{G}(\Sigma)$ of $\Sigma$;
2.  if preProcessing($\mathcal{G}$) = 1 then
3.    return true;
4.  if preProcessing($\mathcal{G}$) = 0 then
5.    return false;
6.  for each connected component $\mathcal{G}' \in \mathcal{G}$
7.    Let $\Sigma'$ be the CINDs and CFDs defined over $\mathcal{G}'$;
8.    if RandomChecking($\Sigma'$) then
9.        return true;
10. return false;
```

Figure 9: Algorithm Checking

As another example, let us replace ψ_4 in Σ by $\psi_4' = (R_3[A; \mathsf{nil}] \subseteq R_4[C; \mathsf{nil}], (\_||\_))$. In the first while-iteration $R = R_4$ and $Q = [R_3, R_1, R_2, R_5]$. The algorithm CFD_Checking returns false since CFD(R_4) is inconsistent. Thus R_4 is deleted from $\mathcal{G}[\Sigma]$ after adding CFDs to R_3 in order to ensure that ψ_4' is not triggered. Since X_p in ψ_4' is nil, there is no way to avoid triggering it. This implies that R_3 also has to be empty. This is enforced by adding non-triggering CFDs, and now CFD$(R_3) = \{\phi_3, (R_3 : B \to A, (_||c_1)), (R_3 : B \to A, (_||c_2))\}$. These non-triggering CFDs are now inconsistent, and therefore no tuple will be added to R_3.

In the next iteration, $R = R_3$ and $Q = [R_1, R_2, R_5]$. Procedure CFD_Checking returns false since CFD(R_3), including the non-triggering constraints added in the previous step, is inconsistent. Node R_3 is therefore deleted from $\mathcal{G}[\Sigma]$.

Now, $R = R_1$ and $Q = [R_2, R_5]$. Procedure CFD_Checking returns true since CFD(R_1) is consistent. The CIND ψ_1 is triggered by any tuple in R_1 so we need to continue to the next relation. Subsequently, for $R = R_2$ and then for $R = R_5$, procedure CFD_Checking returns true and it is not possible to avoid the triggering of constraints. The queue is now empty and $\mathcal{G}[\Sigma]$ is reduced to relations R_1, R_2 and R_5 and their edges.

The execution of line 13 of the algorithm will delete node R_5, since any database that contains tuples in R_5 and satisfies Σ can be replaced by another database that also satisfies Σ but without R_5.

When preProcessing terminates, $\mathcal{G}[\Sigma]$ is reduced to the graph shown in Fig. 8, and -1 is returned. □

Algorithm Checking**.** We combine algorithm preProcessing with RandomChecking and develop algorithm Checking shown in Fig. 9. Initially, graph $\mathcal{G}[\Sigma]$ is constructed and pre-processed (lines 1-2). If preProcessing returns 1, from the discussion above we know that Σ is consistent and thus Checking returns true (lines 2-3). Similarly, if preProcessing returns 0, Checking returns false (lines 4-5). Otherwise preProcessing does not have an affirmative Boolean answer; it returns $\mathcal{G}'$, a reduced version of $\mathcal{G}[\Sigma]$ that consists of only strongly connected components. Subsequently, Checking takes each connected component of $\mathcal{G}'$ and calls RandomChecking that attempts to find the witness database D that satisfies Σ' (line 6-8). If this database is found, the algorithm returns true (line 9). If for each connected component it cannot find such database D, algorithm Checking returns false (line 10).

Example 5.6: Consider the set Σ given in Example 5.4, with ψ_4' of Example 5.5 in place of ψ_4. If algorithm Checking is run to check the consistency of Σ, it would first call algorithm preProcessing which would return the reduced graph as shown in Fig. 8. The reduced graph has only one connect component with $\mathcal{R} = \{R_1, R_2\}$ and $\Sigma = \{\phi_1, \phi_2, \psi_1, \psi_2, \psi_3\}$. Then, algorithm Checking runs RandomChecking (see Example 5.3). □

It is easy to verify the correctness of our checking algorithms.

Theorem 5.1: *Given a set Σ of* CINDs *and* CFDs*, if either* Checking *or* RandomChecking *returns* true*, then Σ is consistent.* □

For the complexity of the algorithms, given a schema $\mathcal{R}$ and a set Σ of constraints, let n and m be the numbers of CFDs and CINDs in Σ respectively, r be the number of relations, and a be the maximum relation arity. Then we can get the following: (a) RandomChecking is in $O(a \cdot r \cdot (n^2 + m))$, (b) preProcessing is in $O(a \cdot r \cdot (n + m)^2 + r^2)$, and (c) Checking is in $O(a \cdot r \cdot (n + m)^2 + r^2)$. Note that in practice a and r will be much smaller than n and m.

6. Experimental Study

We next present a preliminary experimental study of our heuristic methods for checking the consistency of CINDs and CFDs.

We compare the performance of our algorithms for checking the consistency of (a) CFDs alone, namely, the chase-based method and the method based on reduction to SAT presented in Section 5.2, for implementing CFD_Checking, denoted by Chase and SAT, respectively, and (b) CFDs and CINDs put together, namely, RandomChecking and Checking. As shown by Theorem 3.2, there is no need to consider CINDs alone as they are always consistent.

For these algorithms we investigated their accuracy and scalability when varying both the schema (the number of relations) and the number of constraints. We use F to denote the ratio of finite-domain attributes in the schema.

Experimental setting. We used relational schemas that include up to 100 relations, with F ranging from 0% to 25%. Each finite domain was set to have 2 to 100 elements. The experiments show that N, the maximum size of $\mathsf{var}[A]$, has a negligible impact on the accuracy of the algorithms. This is why we set $N = 2$ in the experiments, which makes the algorithms much more efficient.

We have implemented a generator that, given a schema $\mathcal{R}$, randomly generates sets of Σ consisting of CFDs and CINDs defined over $\mathcal{R}$, with any given cardinality $\mathsf{card}(\Sigma)$ of Σ. More specifically, each set Σ was either consistent or inconsistent. We evaluated the accuracy of the algorithms by applying them on consistent and randomly generated sets of CINDs and CFDs. In order to generate the former, we took care to generate a consistent set Σ of CFDs and CINDs by ensuring that there exists at least one possible value for each attribute so as to make a *witness database* of Σ.

The experiments were run on a machine with an Intel Pentium D 3.00GHz with 1GB of memory. Each experiment was run 6 times and the average is reported here.

Experiments for CFDs only. This experiment aimed at comparing the accuracy and scalability of Chase and SAT. In order to avoid the exponential cost of checking all the valuations of finite attributes in algorithm Chase, no more than K_{CFD} valuations are allowed.

We varied the cardinality of $\mathsf{card}(\Sigma)$ of Σ while fixing the number of relations to 20, and F to 25%. The results, given in Fig. 10(a), show that Chase significantly outperforms SAT in terms of scalability. Indeed, Chase works well even for a large number of CFDs. When the accuracy is concerned, Chase and SAT are comparable and both do very well: the percentage that they reported true when the input Σ was consistent was 100% and only in a few occasions it was 95%. We also experimented with random sets of CFDs. In this case, the accuracy can be determined by running the algorithm with and without a limit K_{CFD}. Fig. 10(b) shows the results obtained for 1000 randomly generated CFDs while varying K_{CFD} from 100 to 16K. In fact even when K_{CFD} reaches $2000K$, our algorithm still runs very fast. Thus we fixed $K_{\mathsf{CFD}} = 2000K$ in the sequel.

Given the advantage of Chase over SAT, we adopted the chase implementation of CFD_Checking in the rest of the experiments.

Experiments for CFDs and CINDs. Our second experiments evaluated the efficiency and accuracy of RandomChecking and Checking. We fixed the following parameters in these experiments:
(1) Schema: $\mathcal{R}$ included 20 relations, with at most 15 attributes in each relation and F ranging from 0% to 20%.
(2) Constraints: Σ consisted of 75% of CFDs and 25% of CINDs.
(3) Other Parameters: K, the number of instantiation of finite domain attributes, is set to 20. T, the maximum number of tuples in each relation of the witness database, ranges between $2K$ and $4K$.

Algorithms RandomChecking and Checking scaled well when the number of constraints was increased for both consistent and random set of constraints (see Fig. 11(b) and 11(c) respectively). Even though the running time of RandomChecking is theoretically better than Checking, in practice, most of the cases are solved in the preProcessing step and therefore Checking shows to be more efficient. Also, as shown in Fig. 11(a), for algorithms Checking the accuracy was almost constantly 100%.The experiments show that the preProcessing not only increases accuracy but it also improves the scalability of the algorithm. The high accuracy can be explained by the difficulty of generating consistent datasets that were complex enough for the algorithm to fail. However, we believe the datasets used in the experiments are already more complex than the ones found in practice.

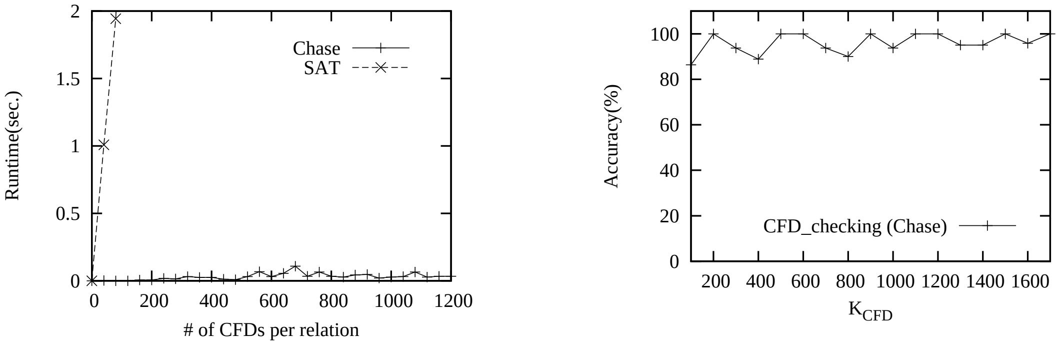
To investigate the impact of the number of relations over the performance, the algorithms were run with different number of relations, but fixing the ratio of $|\Sigma|/|\mathcal{R}| = 1000$. The results of this experiment are given in Fig. 11(d).

Summary. We have presented preliminary results from our experimental study. First, we find that our heuristic methods, in almost all cases, accurately determine the consistency of CFDs and CINDs. Second, all algorithms, except SAT, scale well when the number of constraints or the size of relations increases. Third, we also find that the preProcessing optimization technique not only improves the accuracy, but also reduces the running time.

7. Related work

Closest to our work is the recent study of CFDs [9], which proposed the notion of CFDs, established the intractability of the consistency and implication problems for CFDs, and provided an SQL technique for finding CFD violations. However, neither CINDs nor their static analyses were studied in [9].

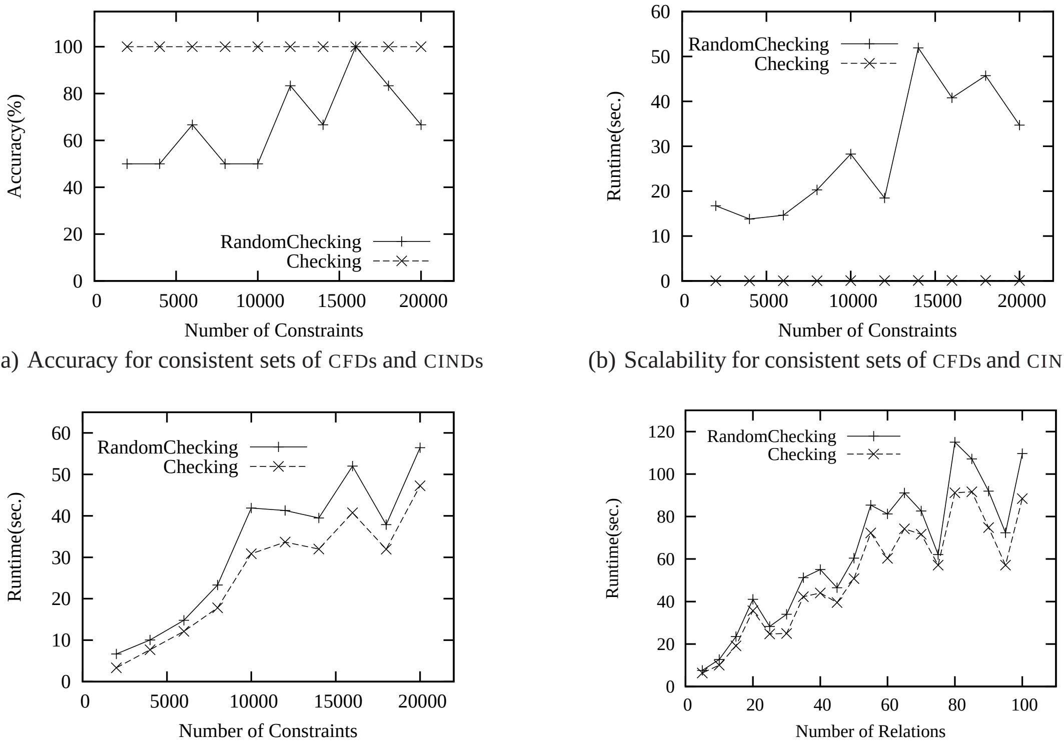
Also relevant are dependencies of [4, 21, 22] developed for constraint databases. Constrained dependencies of [21] are of the form $\xi \rightarrow (Z \rightarrow W)$, where ξ is an arbitrary constraint that is not necessarily an FD. These dependencies apply FD $Z \rightarrow W$ only to the subset of a relation that satisfies ξ. They cannot express CFDs since $Z \rightarrow W$ does not allow patterns with constants as found in CFDs. More expressive are constraint-generating dependencies (CGDs) of [4] and constrained tuple-generating dependencies (CTGDs) of [22], of the form $\forall \bar{x}(R_1(\bar{x}) \wedge \ldots \wedge R_k(\bar{x}) \wedge \xi(\bar{x}) \rightarrow \xi'(\bar{x}))$ and $\forall \bar{x}(R_1(\bar{x}) \wedge \ldots R_k(\bar{x}) \wedge \xi \rightarrow \exists \bar{y}(R_1'(\bar{x}, \bar{y}) \wedge \ldots \wedge R_s'(\bar{x}, \bar{y}) \wedge \xi'(\bar{x}, \bar{y}))$, respectively, where R_i, R_j' are relation symbols, and ξ, ξ' are arbitrary constraints. While both CGDs and CTGDs can express CFDs, and CTGDs can express CINDs, little is known about the complexity of their satisfiability and implication



(a) Performance of CFD_Checking

(b) CFD_Checking accuracy for different K_{CFD}

Figure 10: Scalability and accuracy of consistency checking for CFDs and CINDs



(a) Accuracy for consistent sets of CFDs and CINDs

(b) Scalability for consistent sets of CFDs and CINDs

(c) Scalability for random sets of CFDs and CINDs

(d) Scalability for different number of relations

Figure 11: Scalability and accuracy of consistency checking for CFDs and CINDs

problems, effective algorithms to solve these problems, or their inference systems. Indeed, for CGDs, the complexity of these problems is an open issue in the presence of constants *or* finite-domain attributes, even when ξ and ξ' are $(=, \neq)$ constraints; for CTGDs the satisfiability and implication problems are already undecidable even in the absence of ξ, ξ' and constants. That is, the expressive power of these dependencies comes with the price of high complexity. None of the prior results applies to CFDs or CINDs.

Constraints used in schema matching are typically standard INDs and keys (see, *e.g.,* [16]). Contextual schema matching [7] investigated the applications of contextual foreign keys, a primitive and special case of CINDs, in deriving schema mapping from schema matches. While [7] partly motivated this work, it neither formalized the notion of CINDs nor considered static analyses of CINDs.

Research on constraint-based data cleaning has mostly focused on two topics [2]: *repairing* is to find another database that is consistent and minimally differs from the original database (*e.g.,* [8, 13, 15]); and *consistent query answering* is to find an answer to a given query in every repair of the original database (*e.g.,* [2, 25]). A variety of constraint formalisms have been used in data cleaning, ranging from standard FDs and INDs [2, 8, 13], denial constraints (full dependencies) [20], to logic programs (see [6] for a recent survey). To our knowledge, no prior work has considered pattern tableaux, which, as shown in [9], can be treated as *data tables* in SQL queries and thus allow efficient SQL techniques to detect constraint violations. Moreover, previous work on data cleaning did not study the consistency and implication problems of constraints, which are the focus of this paper.

As remarked earlier, algorithms and inference systems for the implication problems of standard FDs and INDs can be found in most database textbooks, and have also been well studied for a variety of constraints such as TGDs, equality generating dependencies and embedded dependencies (see *e.g.,* [1]). In contrast to CFDs and CINDs, these constraints were studied in the *absence* of constant values (and negation), and thus their consistency analysis is trivial.

The consistency problem, *a.k.a.* the *constraint satisfiability problem*, has been studied for first-order logic constraints, for which heuristic methods have also been developed (see, *e.g.,* [10, 23]). Unfortunately, attributes with finite domains were not considered in that context, and thus those algorithms cannot be applied to CINDs and CFDs. Methods have also been developed for the satisfiability problem for, *e.g.,* description logics (see, *e.g.,* [3]), in which CINDs and CFDs are not expressible.

The chase is widely used in implication analysis and query optimization, and has been studied for a variety of dependencies (see, *e.g.,* [1]). Recently it was extended for query reformulation and schema mapping, and a number of sufficient conditions were identified to guarantee its termination (see [14] for a recent survey). A heuristic method for chasing with FDs and INDs was proposed in [17], with the following simplifications to ensure termination: for a predefined constant n, INDs are applied at most n times and then only one extra variable is allowed to be used to instantiate attributes of the tuples newly inserted when chasing INDs. This is, in spirit, similar to our predefined variable sets.

8. Conclusion

We have proposed CINDs, a mild extension of INDs that is important in both contextual schema matching and data cleaning. We have provided complexity bounds and a sound and complete inference system for consistency and implication problems of CINDs. We also established complexity bounds for reasoning about CINDs together with CFDs. These results settle the fundamental problems associated with conditional dependencies. Even if we consider only finite databases, *i.e.,* databases where each relation has a finite extension, all the obtained complexity bounds still hold. It is left for future work checking if better complexity results can be obtained by considering extra assumptions, such as acyclicity of CINDs or CINDs with only unary relations.

In response to the intractability of the interaction between CFDs and CINDs, we have developed efficient heuristic algorithms for checking the consistency of CINDs and CFDs. As verified by our preliminary experimental results, these algorithms are promising for employing CINDs and CFDs in practical data cleaning and schema matching tools.

There is naturally much more to be done. In practice one often needs to find a minimal cover of a given set Σ of constraints, namely, a set Σ_{mc} that is equivalent to Σ but contains no redundancy. The computation of Σ_{mc} involves implication analysis, which is undecidable for CINDs and CFDs. Thus it is practical to develop heuristic algorithms for checking implication of CFDs and CINDs. Another interesting topic is propagation of CFDs and CINDs through SQL views. This is needed when deriving schema mapping from the constraints [16]. We are also investigating SQL-based techniques for detecting CIND violations in real-life data along the same line as [9] for data cleaning. Finally, effective use of CINDs and CFDs in schema matching and data cleaning requires a full treatment.

Acknowledgments. Wenfei Fan is supported in part by EPSRC GR/S63205/01, GR/T27433/01, EP/E029213/1 and BBSRC BB/D006473/1.

9. References

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

[2] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. *TPLP*, 3(4-5):393–424, 2003.

[3] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook — Theory, Implementation and Applications*. Cambridge University Press, 2003.

[4] M. Baudinet, J. Chomicki, and P. Wolper. Constraint-Generating Dependencies. *JCSS*, 59(1):94–115, 1999.

[5] C. Beeri and M. Vardi. A proof procedure for data dependencies. *JACM*, 31(4):718–741, 1984.

[6] L. Bertossi. Consistent query answering in databases. *SIGMOD Rec.*, 35(2):68–76, 2006.

[7] P. Bohannon, E. Elnahrawy, W. Fan, and M. Flaster. Putting context into schema matching. In *VLDB*, 2006.

[8] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, pages 143–154, 2005.

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

[10] F. Bry, N. Eisinger, H. Schütz, and S. Torge. SIC: Satisfiability checking for integrity constraints. In *DDLP*, pages 25–36, 1998.

[11] M. A. Casanova, R. Fagin, and C. H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. *JCSS*, 28(1):29–59, 1984.

[12] B. S. Chlebus. Domino-tiling games. *JCSS*, 32(3):374–392, 1986.

[13] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197(1-2):90–121, 2005.

[14] A. Deutsch, L. Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1):65–73, 2006.

[15] E. Franconi, A. L. Palma, N. Leone, S. Perri, and F. Scarcello. Census data repair: a challenging application of disjunctive logic programming. In *LPAR*, pages 561–578, 2001.

[16] L. Haas, M. Hernández, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *SIGMOD*, 2005.

[17] D. S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *JCSS*, 28(1):167–189, 1984.

[18] P. G. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS*, 2005.

[19] Lens Computer Science Research Centre. SAT4j home page, 2003. *http://www.sat4j.org/*.

[20] A. Lopatenko and L. Bertossi. Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. In *ICDT*, 2007.

[21] M. J. Maher. Constrained dependencies. *Theoretical Computer Science*, 173(1):113–149, 1997.

[22] M. J. Maher and D. Srivastava. Chasing Constrained Tuple-Generating Dependencies. In *PODS*, 1996.

[23] R. Manthey. Satisfiability of integrity constraints: Reflections on a neglected problem. In *FMLDO*, pages 169–179, 1990.

[24] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[25] J. Wijsen. Database repairing using updates. *TODS*, 30(3):722–768, 2005.