THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Dependencies revisited for improving data quality

Citation for published version:
Fan, W 2008, Dependencies revisited for improving data quality. in Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada. ACM, pp. 159-170. DOI: 10.1145/1376916.1376940

Digital Object Identifier (DOI):
10.1145/1376916.1376940

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Early version, also known as pre-print

Published In:
Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 17. Aug. 2017

Dependencies Revisited for Improving Data Quality

Wenfei Fan
University of Edinburgh & Bell Laboratories
wenfei@{inf.ed.ac.uk, research.bell-labs.com}

Abstract

Dependency theory is almost as old as relational databases themselves, and has traditionally been used to improve *the quality of schema*, among other things. Recently there has been renewed interest in dependencies for improving *the quality of data*. The increasing demand for data quality technology has also motivated revisions of classical dependencies, to capture more inconsistencies in real-life data, and to match, repair and query the inconsistent data. This paper aims to provide an overview of recent advances in revising classical dependencies for improving data quality.

Categories and Subject Descriptors: H.2.0 [**Database Management**]: General – *Security, integrity, and protection*

General Terms: Languages, Theory, Design, Reliability.

1. Introduction

Data dependencies, *a.k.a.* integrity constraints, have been well studied for relational databases. Since Codd introduced functional dependencies in 1972 [27], a variety of dependency languages, defined as certain classes of first-order (FO) logic sentences, have been proposed to specify the semantics of relational data. Fundamental questions associated with these dependencies, such as implication analysis and finite axiomatizability, had already been settled in the 1970s and the 1980s. Along with the theory of query languages, dependency theory constitutes a major part of database theory, and is covered by most database texts. Dependencies have traditionally been used to optimize queries, prevent invalid updates, and above all, to improve *the quality of schema* via normalization.

Recently there has been renewed interest in dependencies, for improving *the quality of data*. Dependencies are being used to repair and help query inconsistent data; furthermore, classical dependencies are revised and extended in order to capture more errors found in real-life data. This paper aims to provide an overview of recent advances in the study of dependencies for improving data quality, highlighting revisions of classical dependencies.

The need for revisiting dependencies is motivated by data quality issues. Real-world data is typically dirty, *i.e.,* containing inconsistencies, conflicts, and errors. Recent statistics reveals that enterprises typically expect data error rates of approximately 1%–5% [65]. The costs and risks of dirty data are being increasingly recognized by all industries worldwide. It is reported that dirty data costs US businesses 600 billion dollars annually [31], and that erroneously priced data in retail databases alone costs US consumers $2.5 billion each year [33]. It is also estimated that data cleaning accounts for 30%-80% of the development time and budget in most data warehouse projects [66]. While the prevalent use of the Web has made it possible to extract and integrate data from diverse sources on an unprecedented scale, it has also increased the risks of creating and propagating dirty data.

Data quality issues have been recognized and addressed in several disciplines, *e.g.,* statistics, management and computer science. A variety of approaches have been proposed for improving data quality: probabilistic, empirical, knowledge-based and logic-based approaches (see [10] for a survey). There has also been increasing demand in industries for developing data-quality tools, aiming to effectively detect and repair errors in the data, and thus to add accuracy and value to business processes. The market for data-quality tools is growing at 17%, way above the 7% average forecast for other IT segments. These tools are also a critical part of master data management (MDM) [30, 62], one of the fastest growing software markets. Most commercial data quality and ETL (extraction, transformation, loading) tools, however, heavily rely on manual effort and low-level programs that are difficult to write and maintain [64].

There are good reasons to believe that dependencies should play an important role in data-quality tools. One can specify the semantics of data with dependencies, in a declarative way, and catch inconsistencies and errors that emerge as violations of the dependencies. Furthermore, inference systems, analysis algorithms and profiling methods for dependencies have shown promise as a systematic method for reasoning about the semantics of the data, and for deducing and discovering rules for cleaning the data, among other things. Indeed, there has been a host of work on querying and repairing inconsistent data based on dependencies (*e.g.,* [7, 6, 8, 3, 13, 18, 14, 16, 23, 25, 41, 42, 43]).

Initial work on dependency-based data quality methods focused on traditional dependencies that were mainly developed for schema design, such as functional and inclusion dependencies (FDs and INDs), along with a class of universally quantified FO sentences, referred to as denial constraints. Their limited expressiveness often does not allow us to state inconsistencies commonly found in real-life data as violations of these dependencies, or to specify rules for identifying objects from multiple unreliable data sources.

These limitations highlight the need for extending classical dependencies. On the other hand, it is clear that in order to have some meaningful data cleaning algorithms, the dependencies to be used have to be reasonably limited. It is unlikely that any useful automated cleaning algorithm could be constructed for arbitrary dependencies – or even for general SQL boolean queries. Indeed, it is well-known that one cannot determine whether there is a database satisfying a given set of boolean SQL queries. These require a balance between the expressive power needed to deal with important data quality issues, and the restrictions needed to ensure decidable analysis of dependencies and effective cleaning algorithms.

We present two attempts towards achieving this balance.

Conditional dependencies. We begin with an extension of traditional FDs and INDs that capture more of the inconsistencies in real-life data. Consider, for example, a relation consisting of records of customers in the US and UK. While in the UK, zip code determines street, it is not the case in the US; thus one cannot detect errors in the UK records by enforcing zip $\rightarrow$ street as an FD on the entire customer relation. To remedy the limitations, extensions of functional and inclusion dependencies have been introduced [36, 20], referred to as *conditional functional dependencies* and *conditional inclusion dependencies* (CFDs and CINDs), respectively. Conditional dependencies add to their traditional counterparts a specification of patterns of data values and variables. The semantics is obtained by restricting the traditional semantics to only those tuples that match the patterns, rather than on the entire relation(s). These dependencies make a *weaker assertion* than traditional FDs and INDs, and hence are more widely applicable.

Matching dependencies. Another longstanding line of research associated with data quality is object identification, *a.k.a.* data deduplication, record linkage, merge-purge and record matching. Given one or more relations, we want to identify tuples from those relations that refer to the same real-world object. This is essential to, among other things, data cleaning, data integration, and credit-card fraud detection. Prior approaches to object identification are often seen as orthogonal to dependency-based ones. Central to those approaches is to determine comparison vectors and matching rules, *i.e.,* what attributes should be selected and how they should be compared in order to identify tuples; these rules are typically given in a procedural way, and heavily rely on domain-specific heuristics (see [32] for a recent survey on object identification).

We show that matching rules can be incorporated into the framework of dependencies, by introducing *matching dependencies*, an extension of FDs, defined across multiple relations and by incorporating domain-specific similarity and matching operators [38]. For example, a matching rule (taken from [48]) can be expressed as a matching dependency to assure that if two customer tuples have the same address and last name, and moreover, their first names are similar (but may not be identical), then the two tuples refer to the same person. These rules could then be combined with other dependencies (traditional or conditional) for data cleaning. This allows us to study the interaction between matching and cleaning rules in a uniform framework, and automatically deduce new matching rules via implication analysis of the dependencies.

Static analyses. These extensions also call for a revision of static analyses of classical dependencies. As mentioned above, static analyses of dependencies play a crucial role in their application to data quality. The data repairing problem [7] itself is a generalization of satisfiability analysis for sets of dependencies, since it requires to find a satisfying instance with additional properties (*e.g.,* proximity to the original data). Implication analysis for dependencies is critical to their use in cleaning as well. In particular, note that for *detecting* constraint violations, adding derived dependencies is pointless: elementary propositional logic tells us that any instance that fails the derived dependency must already have failed one of the given dependencies from which it was derived. However, when considered as matching rules ("if ϕ holds then identify x and y"), derived dependencies can indeed add value [40].

The increased expressive power of conditional dependencies and matching dependencies comes at a price for static analysis. A set of conditional functional dependencies (CFDs), for example, may have conflicts themselves. Thus it is necessary to conduct consistency analysis, to determine whether a given set of conditional dependencies is dirty itself. The consistency problem is nontrivial: it is already NP-complete for CFDs alone, and is undecidable for CFDs and CINDs taken together. In contrast, this is a non-issue for collections of FDs and INDs, which are known to be always satisfiable. Furthermore, for conditional dependencies, the implication analysis, finite axiomatization and the computation of view dependencies [52, 53] also become more intriguing than their traditional counterparts. This calls for a full treatment of the classical decision problems for these extensions of traditional dependencies.

Improving data quality based on dependencies. The ultimate goal for revisiting dependencies is to handle inconsistencies in the data. We provide a brief overview of dependency-based methods for dealing with inconsistencies. There are at least three approaches. The first two were formally introduced in [7]: repairing is to find another database that is consistent and minimally differs from the original database; and consistent query answering is to find an answer to a given query in every repair of the original database, without editing the data. The third approach is to find a condensed representation of all repairs of the inconsistent database, in terms of tableaux [68] or answer sets of logic programs [6, 47]. There has been considerable work on repairing [7, 16, 28, 40, 58, 69], consistent query answering [7, 8, 13, 23, 25, 42, 43, 57, 68], and on condensed representations of repairs [6, 47, 68]. Most of the work focused on traditional dependencies.

Open research issues. The study of data quality based on dependencies has raised as many questions as it has answered. Moreover, the study is closely related to, *e.g.,* incomplete information [46, 50, 61], probabilistic data [29], data exchange [54], integration [55, 56], and Web data management [9, 39, 41]. The connections also give rise to a host of open questions. We explore these connections.

Organization. In Section 2 we present conditional dependencies for characterizing the consistency of data, followed by matching dependencies for object identification in Section 3. In Section 4 we give an account of results on reasoning about these revisions. Section 5 presents an overview of the three approaches for handling inconsistencies, followed by open research issues in Section 6.

Our focus in this article is on revisions of classical dependencies to improve data quality. A survey of constraint-based data-quality methods is beyond the scope of this paper, and a number of related papers are not referenced due to space constraints. We refer the reader to [10, 64] for general issues in connection with data quality, and to [12, 14, 24, 26] for constraint-based methods. Formal presentations on dependency theory are provided in [1, 35, 51].

2. Extending Dependencies with Conditions

We first present conditional dependencies [36, 20, 19], extensions of functional and inclusion dependencies with patterns.

2.1 Conditional Functional Dependencies

To illustrate conditional functional dependencies, let us consider the following relational schema for customer data:

customer (CC: int, AC: int, phn: int, name: string, street: string, city: string, zip: string)

where each tuple specifies a customer's phone number (country code CC, area code AC, phone phn), name and address (street, city, zip code); we defer the discussion of domains to Section 4. An instance D_0 of the customer schema is shown in Fig. 1.

Functional dependencies (FDs) on customer relations include:

$$f_1\colon [\mathsf{CC}, \mathsf{AC}, \mathsf{phn}] \rightarrow [\mathsf{street}, \mathsf{city}, \mathsf{zip}], \quad f_2\colon [\mathsf{CC}, \mathsf{AC}] \rightarrow [\mathsf{city}].$$

That is, a customer's phone uniquely determines her address (f_1), and the country code and area code determine the city (f_2). The

	CC	AC	phn	name	street	city	zip
t_1:	44	131	1234567	Mike	Mayfield	NYC	EH4 8LE
t_2:	44	131	3456789	Rick	Crichton	NYC	EH4 8LE
t_3:	01	908	3456789	Joe	Mtn Ave	NYC	07974

Figure 1: An instance of customer relation

instance D_0 of Fig. 1 satisfies f_1 and f_2. In other words, if we use f_1 and f_2 to specify the consistency of customer data, *i.e.,* to characterize errors as violations of these dependencies, then no errors or inconsistencies are found in D_0, and D_0 is regarded clean.

A closer examination of D_0, however, reveals that none of the tuples in D_0 is error-free. Indeed, the inconsistencies become obvious when the following constraints are considered, which intend to capture the semantics of real-world customer data:

cfd_1: ([CC = 44, zip] $\rightarrow$ [street])
cfd_2: ([CC = 44, AC = 131, phn] $\rightarrow$ [street, city = 'EDI', zip])
cfd_3: ([CC = 01, AC = 908, phn] $\rightarrow$ [street, city = 'MH', zip])

Here cfd_1 asserts that for customers in the UK (CC = 44), zip code uniquely determines street. In other words, cfd_1 is an "FD" that is to hold on the subset of tuples that satisfies the pattern "CC = 44", *e.g.,* $\{t_1, t_2\}$ in D_0. It is not a traditional FD since it is defined with constants, and it is not required to hold on the entire customer relation D_0 (in the US, for example, zip code does not determine street). The last two constraints refine the FD f_1 given earlier: cfd_2 states that for any two UK customer tuples, if they have area code 131 and have the same phn, then they must have the same street and zip, and moreover, the city *must* be EDI; similarly for cfd_3.

Tuples t_1 and t_2 in D_0 violate cfd_1: they refer to customers in the UK and have identical zip, but they differ in street. Further, while D_0 satisfies f_1, each of t_1 and t_2 in D_0 violates cfd_2: CC = 44 and AC = 131, but city $\neq$ EDI. Similarly, t_3 violates cfd_3.

Syntax. We now give the formal definition of conditional functional dependencies (CFDs). Consider a relation schema R defined over a set of attributes, denoted by $\mathsf{attr}(R)$. For each attribute $A \in \mathsf{attr}(R)$, its domain is specified in R, denoted by $\mathsf{dom}(A)$. For an instance D of R and a tuple $t \in D$, we use $t[A]$ to denote the projection of t onto A; similarly, for a sequence X of attributes in $\mathsf{attr}(R)$, $t[X]$ denotes the projection of t onto X.

A CFD φ defined on R is a pair $R(X \rightarrow Y, T_p)$, where (1) $X \rightarrow Y$ is a standard FD, referred to as the FD *embedded in* φ; and (2) T_p is a tableau with attributes in X and Y, referred to as the *pattern tableau* of φ, where for each A in $X \cup Y$ and each pattern tuple $t_p \in T_p$, $t_p[A]$ is either a constant 'a' in $\mathsf{dom}(A)$, or an unnamed (yet marked) variable '_' that draws values from $\mathsf{dom}(A)$.

If A occurs in both X and Y, we use $t[A_L]$ and $t[A_R]$ to indicate the occurrence of A in X and Y, respectively. We separate the X and Y attributes in a pattern tuple with '$\|$'. We write φ as $(X \rightarrow Y,\ T_p)$ when R is clear from the context.

Example 2.1: All the constraints we have encountered so far can be expressed as the CFDs shown in Fig. 2 (φ_1 for cfd_1, φ_2 for f_1, cfd_2 and cfd_3, and φ_3 for f_2). Note that each tuple in a pattern tableau indicates a constraint, *e.g.,* φ_2 defines three constraints. □

Note that traditional FDs are a special case of CFDs, in which the pattern tableau consists of a single tuple, containing '_' only.

Semantics. We define an operator $\asymp$ on constants and '_': $\eta_1 \asymp \eta_2$ if either $\eta_1 = \eta_2$, or one of η_1, η_2 is '_'. The operator $\asymp$ naturally extends to tuples, *e.g.,* (Mayfield, EDI) $\asymp$ (_, EDI) but (Mayfield, EDI) $\not\asymp$ (_, NYC).

An instance D of R *satisfies* the CFD φ, denoted by $D \models \varphi$, if for *each* tuple t_p in the pattern tableau T_p of φ, and for *each pair* of tuples t_1, t_2 in D, if $t_1[X] = t_2[X] \asymp t_p[X]$, then $t_1[Y] = t_2[Y] \asymp t_p[Y]$.

(a) φ_1 = ([CC, zip] $\rightarrow$ [street], T_1), where T_1 is

CC	zip	street
44	-	-

(b) φ_2 = ([CC, AC, phn] $\rightarrow$ [street, city, zip], T_2), where T_2 is

CC	AC	phn	street	city	zip
-	-	-	-	-	-
44	131	-	-	EDI	-
01	908	-	-	MH	-

(c) φ_3 = ([CC, AC] $\rightarrow$ [city], T_3), where T_3 is

CC	AC	city
-	-	-

Figure 2: Example CFDs

Intuitively, each tuple t_p in the pattern tableau T_p of φ is a constraint defined on $D_{t_p} = \{t \mid t \in D, t[X] \asymp t_p[X]\}$ such that for any $t_1, t_2 \in D_{t_p}$, if $t_1[X] = t_2[X]$, then (a) $t_1[Y] = t_2[Y]$, and (b) $t_1[Y] \asymp t_p[Y]$. Here (a) enforces the semantics of the FD embedded in φ, and (b) assures the binding between *constants* in $t_p[Y]$ and *constants* in $t_1[Y]$. This constraint is defined on the subset D_{t_p} of D identified by $t_p[X]$, rather than on the entire D.

For example, while the instance D_0 of Fig. 1 satisfies the CFD φ_3 given in Fig. 2, it satisfies neither φ_1 nor φ_2.

2.2 Conditional Inclusion Dependencies

We next present a revision of inclusion dependencies (INDs). Consider the two schemas below, referred to as source and target:

Source: order (asin: string, title: string, type: string, price: real)
Target: book (isbn: string, title: string, price: real, format: string)
CD (id: string, album: string, price: real, genre: string)

The source database contains a single relation order, specifying items of various types such as books, CDs, DVDs, ordered by customers. The target database has two relations, namely, book and CD, specifying customer orders of books and CDs, respectively. Example source and target instances D_1 are shown in Fig. 3.

To find schema mappings from source to target, or detect errors across these databases, one might be tempted to use INDs such as:

order(title, price) $\subseteq$ book(title, price),
order(title, price) $\subseteq$ CD(album, price).

These INDs, however, do not make sense: one cannot expect the title and price of a book item in the order table to find a corresponding CD tuple to match; similarly for CDs in the order table.

There are indeed inclusion dependencies from the source to the target, as well as on the target, but only under certain conditions:

cind_1: (order(title, price, type = 'book') $\subseteq$ book(title, price))
cind_2: (order(title, price, type = 'CD') $\subseteq$ CD(album, price))
cind_3: (CD(album, price, genre = 'a-book')
$\subseteq$ book(title, price, format = 'audio'))

Here cind_1 states that for each order tuple t, if its type is 'book', then there must exist a book tuple t' such that t and t' agree on their title and price attributes; similarly for cind_2. Constraint cind_3 asserts that for each CD tuple t, if its genre is 'a-book' (audio book), then there must be a book tuple t' such that the title and price of t' are identical to the album and price of t, and moreover, the format of t' must be 'audio'. Like CFDs, these constraints are required to hold only on a subset of tuples satisfying certain patterns. They are specified with constants, and cannot be expressed as standard INDs.

While D_1 of Fig 3 satisfies cind_1 and cind_2, it violates cind_3. Indeed, tuple t_9 in the CD table has an 'a-book' genre, but it cannot find a match in the book table with 'audio' format. Note that the book tuple t_7 is not a match for t_9: although t_9 and t_7 agree on

	asin	title	type	price
t_4:	a23	Snow White	CD	7.99
t_5:	a12	Harry Potter	book	17.99

(a) Example order data

	isbn	title	price	format
t_6:	b32	Harry Potter	17.99	hard-cover
t_7:	b65	Snow White	7.99	paper-cover

(b) Example book data

	id	album	price	genre
t_8:	c12	J. Denver	7.94	country
t_9:	c58	Snow White	7.99	a-book

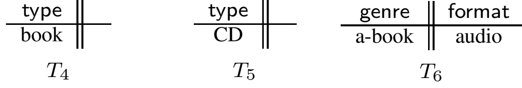
(c) Example CD data

Figure 3: Example order, book and CD data

φ_4 = (order(title, price; type) $\subseteq$ book(title, price), T_4)

φ_5 = (order(title, price; type) $\subseteq$ CD(album, price), T_5)

φ_6=(CD(album, price; genre) $\subseteq$ book(title, price; format), T_6)



type ‖
book ‖

T_4

type ‖
CD ‖

T_5

genre ‖	format
a-book ‖	audio

T_6

Figure 4: Example CINDs

their album (title) and price attributes, the format of t_7 is 'paper-cover' rather than 'audio' as required by cind$_3$.

Syntax. We now give the formal definition of CINDs.

A CIND ψ defined on relation schemas R_1 and R_2 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 of R_1 and R_2, respectively; (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 the *pattern tableau* of ψ with attributes in X_p and Y_p, such that for each pattern tuple $t_p \in T_p$ and each attribute B in X_p (or Y_p), $t_p[B]$ is a constant in $\mathsf{dom}(B)$. If A occurs in both X_p and Y_p, we use $t[A_L]$ and $t[A_R]$ to indicate the occurrence of A in X_p and Y_p, respectively. We separate X_p and Y_p in a pattern tuple with '‖'.

Example 2.2: Constraints cind$_1$, cind$_2$ and cind$_3$ can be expressed as CINDs φ_4, φ_5 and φ_6 shown in Fig. 4, respectively. □

Semantics. An instance (D_1, D_2) of (R_1, R_2) *satisfies* the CIND ψ, denoted by $(D_1, D_2) \models \psi$, iff for *each* tuple t_p in the pattern tableau T_p and for *each* t_1 in the relation D_1, if $t_1[X_p] = t_p[X_p]$, then *there must exist* t_2 in D_2 such that $t_1[X] = t_2[Y]$ and moreover, $t_2[Y_p] = t_p[Y_p]$.

That is, each pattern tuple t_p in T_p is a constraint defined on $D_{(1,t_p)} = \{t_1 \mid t_1 \in D_1, t_1[X_p] = t_p[X_p]\}$, such that (a) the IND $R_1[X] \subseteq R_2[Y]$ in ψ is defined on $D_{(1,t_p)}$ rather than the entire D_1; (b) for each $t_1 \in D_{(1,t_p)}$, there exists a tuple t_2 in D_2 such that $t_1[X] = t_2[Y]$ as required by the standard IND and moreover, $t_2[Y_p]$ must be the same as the pattern $t_p[Y_p]$. Intuitively, X_p identifies those R_1 tuples on which ψ is defined, and Y_p enforces the corresponding R_2 tuples to have a certain constant pattern.

Traditional INDs are a special case of CINDs, in which X_p and Y_p are empty lists.

2.3 Further Extensions

One natural extension to consider is to add disjunction and inequality to CFDs. Consider, for example, customers in New York State. A cursory examination of New York area codes reveals that most cities (CT) in the state have a *unique* area code, except NYC and LI (Long Island). Further, area codes for New York City are limited to 212, 718, 646, 347, *or* 917. One can express these as:

ecfd$_1$: CT $\notin$ {NYC, LI} $\rightarrow$ AC
ecfd$_2$: CT $\in$ {NYC} $\rightarrow$ AC $\in$ {212, 718, 646, 347, 917}

where ecfd$_1$ asserts that the FD CT $\rightarrow$ AC holds if CT is *not in* the set {NYC, LI}; and ecfd$_2$ is defined with disjunction: it states that when CT is NYC, AC must be one of 212, 718, 646, 347, *or* 917.

An extension of CFDs by supporting disjunction and inequality was studied in [19], referred to as eCFDs. As will be seen in Section 4, the increased expressive power does not make our lives harder when it comes to reasoning about these dependencies.

It is natural to consider extensions of constraint languages beyond FDs and INDs with conditions. Most dependencies studied for relational databases can be expressed as FO sentences of the following form (cf. [1, 35]), simply referred to as *dependencies*:

$$\forall x_1 \ldots x_m\, (\phi(x_1, \ldots, x_m) \rightarrow \exists y_1 \ldots y_n\, \psi(z_1, \ldots, z_k)),$$

where (a) $\{y_1, \ldots, y_n\} = \{z_1, \ldots, z_k\} - \{x_1, \ldots, x_m\}$; (b) ϕ is a conjunction of (at least one) relation atoms of the form $R(w_1, \ldots, w_l)$, where w_i is a variable for each $i \in [1, l]$, and ϕ uses all of the variables in $\{x_1, \ldots, x_m\}$; (c) ψ is a conjunction of either relation atoms or equality atoms $w = w'$, where w, w' are variables, and ψ uses all of the variables in $\{z_1, \ldots, z_k\}$; and (d) no equality atoms in ψ use existentially quantified variables.

Dependencies are often classified as follows.

(a) *Full dependencies*: universally quantified dependencies.

(b) *Tuple generating dependencies* (TGDs): dependencies in which the right-hand side (RHS) ψ is a relation atom. A TGD says that if a certain pattern of entries appears then another pattern must appear.

(c) *Equality generating dependencies* (EGDs): full dependencies in which the RHS ψ is an equality atom. An EGD says that if a certain pattern of entries appears then a certain equality must hold.

These dependencies are defined in terms of relation atoms, variables and equality, *without constants*. To capture more errors in real-life data one might want to revise these full-fledged constraint languages by incorporating data-value patterns. However, this may not be very practical: the implication problem is already EXPTIME-complete for full dependencies, and is undecidable for TGDs (cf. [35]). To balance the tradeoff between expressive power and complexity, it is often more realistic to consider revisions of fragments of these constraint languages for data quality tools.

A variety of extensions of classical dependencies have been proposed, for specifying constraint databases [11, 17, 59, 60]. Constraints of [17], also called conditional functional dependencies, are of the form $(X \rightarrow Y) \rightarrow (Z \rightarrow W)$, where $X \rightarrow Y$ and $Z \rightarrow W$ are standard FDs. Constrained dependencies of [59] extend [17] by allowing $\xi \rightarrow (Z \rightarrow W)$, where ξ is an arbitrary constraint that is not necessarily an FD. These dependencies cannot express CFDs since $Z \rightarrow W$ does not allow constants. More expressive are constraint-generating dependencies (CGDs) of [11] and constrained tuple-generating dependencies (CTGDs) of [60], both subsuming CFDs. A CGD is of the form $\forall \bar{x}(R_1(\bar{x}) \wedge \ldots \wedge R_k(\bar{x}) \wedge \xi(\bar{x}) \rightarrow \xi'(\bar{x}))$, where R_i's are relation atoms, and ξ, ξ' are arbitrary constraints that may carry constants. A CTGD is of the form $\forall \bar{x}(R_1(\bar{x}) \wedge \ldots \wedge R_k(\bar{x}) \wedge \xi \rightarrow \exists \bar{y}(R'_1(\bar{x}, \bar{y}) \wedge \ldots \wedge R'_s(\bar{x}, \bar{y}) \wedge \xi'(\bar{x}, \bar{y})))$, subsuming both CINDs and TGDs. The increased expressive power of CGDs and CTGDs comes at the price of a higher complexity for reasoning about these dependencies. No previous work has studied these extensions for data cleaning.

Besides CFDs and CINDs, non-traditional dependencies studied for data cleaning include denial constraints [7, 8, 13, 25, 57, 58, 68], which are universally quantified FO sentences of the form:

$$\forall \bar{x}_1 \ldots \bar{x}_m\, \neg(R_1(\bar{x}_1) \wedge \ldots \wedge R_m(\bar{x}_m) \wedge \varphi(\bar{x}_1, \ldots, \bar{x}_m)),$$

where R_i is a relation atom for $i \in [1, m]$, and φ is a conjunction of built-in predicates such as $=, \neq, <, >, \leq, \geq$. Note that FDs are a special case of denial constraints. While some denial constraints in the literature (*e.g.,* [13]) allow constants, numerical values and aggregate functions, the implication and consistency problems and finite axiomatizability for these constraints are yet to be settled [12].

3. Incorporating Domain Specific Operators

The dependencies we have seen so far are in pure first-order logic. Nonetheless, data quality techniques often rely on domain-specific tools: they may match tuples and compare values using particular metrics. While these domain-specific operations may not be themselves expressible in any reasonable declarative formalism, it is still possible to integrate them into the framework of dependencies. We explain this by presenting another extension of functional dependencies [38], defined across different relations and in terms of domain-specific similarity and matching operators. This extension is proposed for specifying matching rules for object identification.

3.1 Object Identification

We first illustrate object identification (see [32] for a survey). Consider two data sources, specified by the following schemas:

card (c#, SSN, FN, LN, addr, tel, email, type),
billing (c#, FN, SN, post, phn, email, item, price).

Here a card tuple specifies a credit card (number c# and type) issued to a card holder identified by SSN, FN (first name), LN (last name), addr (address), tel (phone) and email. A billing tuple indicates that the price of a purchased item is paid by a credit card of number c#, issued to a holder that is specified in terms of forename FN, surname SN, postal address post, phone phn and email.

Given an instance (D_c, D_b) of (card,billing), for fraud detection, one has to ensure that for any tuple $t \in D_c$ and $t' \in D_b$, if $t[\mathsf{c\#}] = t'[\mathsf{c\#}]$, then $t[Y_c]$ and $t'[Y_b]$ refer to the same holder, where

Y_c = [FN, LN, addr, tel, email], Y_b = [FN, SN, post, phn, email].

The difficulty posed by this seemingly simple problem is that data may not have a uniform representation for the same object in different sources (*e.g.,* a person's name may appear as "John Smith" and "J. Smith" in D_c and D_b, respectively). As a result, it is quite likely that $t[Y_c]$ and $t'[Y_b]$ are not identical, although they indeed refer to the same person.

This example is an instance of the object identification problem. More specifically, let R, R' be two relation schemas, and Y, Y' be lists of attributes in $\mathsf{attr}(R)$ and $\mathsf{attr}(R')$, respectively. The *object identification problem* is to determine, given an instance (D, D') of (R, R'), for any $t \in D$ and $t' \in D'$, whether $t[Y]$ and $t'[Y']$ refer to the same real-world object. Here "referring to the same real-world object" is the conclusion drawn by a domain-specific operation, referred to as the *matching* operation.

Object identification is essential to not only data quality, but also data integration, for which it is often necessary to correlate information about an object from multiple data sources.

A central issue for object identification concerns how to determine comparison vectors: what alternative attributes X and X' in $\mathsf{attr}(R)$ and $\mathsf{attr}(R')$ should be considered, and how should $t[X]$ and $t'[X']$ be compared, in order to conclude that $t[Y]$ and $t'[Y']$ match. The comparisons may also include domain-specific operators, for example, similarity relations on attributes.

Returning to our example for fraud detection, the following "matching rules" are used in practice, either specified by human experts or discovered via learning [48]. (a) If t[tel] and t'[phn] are identical, then t[addr] and t'[post] should match (*i.e.,* referring to the same address, even if t[addr] and t'[post] might not be identical). (b) If t[email] and t'[email] match, then so do t[FN, LN] and t'[FN, SN]. (c) If t[LN, addr] and t'[SN, post] match, and if t[FN] and t'[FN] either match or are *similar w.r.t.* a similarity operator $\approx_d$ based on edit distance, then $t[Y_c]$ and $t'[Y_b]$ match, *i.e.,* they are the same person. Here (c) is a comparison vector for (Y_c, Y_b): to match $t[Y_c]$ and $t'[Y_b]$, we only need to consider ([LN, addr, FN], [SN, post, FN]) and compare them in terms of matching and similarity operators.

One can draw an analogy between comparison vectors and the familiar notion of keys: both notions attempt to provide an invariant connection between tuples and the real-world entities they represent. However, we note a difference both in the hypothesis (the use of similarity and matching) and in the conclusion (referring to matching). Still, we hope to conduct generic reasoning with these rules, to derive comparison vectors. An example of derived rules is: if t[LN, tel] and t'[SN, phn] equal, and if t[FN] and t'[FN] are similar, then $t[Y_c]$ and $t'[Y_b]$ match. Used as matching rules, the *derived* comparison vectors can *improve match quality*: true matches may not be found by given matching rules, but they may still be identified by derived rules. For example, when t and t' radically differ in some pairs of attributes, *e.g.,* ([addr], [post]), $t[Y_c]$ and $t'[Y_b]$ may not be matched by the rules (a)-(c) given. In contrast, they may still be identified by the derived comparison vector ([LN, tel, FN], [SN, phn, FN]). The need for such automated reasoning has long been recognized for census data cleaning, where deriving implicit edits from explicit edits is a routine practice [40, 69].

3.2 Matching Dependencies

To effectively reason about matching rules, we give an extension of functional dependencies, referred to as matching dependencies (MDs), to express those rules. We focus on the definition of MDs below, and defer their use in object identification to Section 3.3.

To define MDs we first present some domain-specific operators.

Similarity operators. Assume a fixed set Θ of domain-specific similarity relations. For each $\approx$ in Θ, and (lists of) values x, y in the specific domains in which $\approx$ is defined, we write $x \approx y$ if (x, y) is in $\approx$, and refer to $\approx$ as a similarity operator. Each $\approx$ is reflexive, *i.e.,* $x \approx x$, symmetric, *i.e.,* if $x \approx y$ then $y \approx x$, and it subsumes equality, *i.e.,* if $x = y$ then $x \approx y$. The equality relation $=$ is in Θ.

Matching operator. A particular operator $\rightleftharpoons$ is in Θ, referred to as the *matching operator* (*match relation*), defined on value lists. It is transitive: if $L_1 \rightleftharpoons L_2$ and $L_2 \rightleftharpoons L_3$ then $L_1 \rightleftharpoons L_3$. In addition, for any $L = [L_1, \ldots, L_k]$ and $L = [L'_1, \ldots, L'_k]$, if $L_i \rightleftharpoons L'_i$ for $i \in [1, k]$, then $L \rightleftharpoons L'$, and vice versa. That is, for any partition of L and L', all parts of L and L' pairwise match iff $L \rightleftharpoons L'$.

Matching dependencies (MDs). For a list L of length k and each $j \in [1, k]$, denote by $L[j]$ the j-th element of L. Consider a pair of relation schemas (R_1, R_2), and a pair of lists (X_1, X_2) of length k, such that for all $j \in [1, k]$, $X_1[j] \in \mathsf{attr}(R_1)$ and $X_2[j] \in \mathsf{attr}(R_2)$. We say that (X_1, X_2) is *compatible* if $\mathsf{dom}(X_1[j])$ and $\mathsf{dom}(X_2[j])$ are compatible for each $j \in [1, k]$. We refer to $(X_1[j], X_2[j])$ as a *pair* of attributes in (X_1, X_2).

We now give the definition of *matching dependencies* (MDs).

An MD ϕ defined on schemas (R_1, R_2) is an expression of the form $\bigwedge_{j \in [1,k]} (R_1[X_1[j]] \approx_j R_2[X_2[j]]) \rightarrow R_1[Z_1] \approx R_2[Z_2]$, where (X_1, X_2) and (Z_1, Z_2) are compatible attribute lists over (R_1, R_2), X_1 and X_2 have the same length k, and $\approx$ and $\approx_j$ are similarity operators in Θ, for all $j \in [1, k]$.

The MD ϕ *holds* on an instance (D_1, D_2) of (R_1, R_2), denoted by $(D_1, D_2) \models \phi$, if for any tuples $t_1 \in D_1$ and $t_2 \in D_2$, if $\bigwedge_{j \in [1,k]} t_1[X_1[j]] \approx_j t_2[X_2[j]]$, then $t_1[Z_1] \approx t_2[Z_2]$.

As will be seen in Section 3.3, similarity and matching operators in an MD play different roles in object identification.

Example 3.1: The rules (a)-(c) for identifying card and billing tuples given earlier can be expressed as MDs, as follows.

ϕ_1: card[tel] = billing[phn] $\rightarrow$ card[addr] $\rightleftharpoons$ billing[post]

ϕ_2: card[email] $\rightleftharpoons$ billing[email] $\rightarrow$ card[FN, LN] $\rightleftharpoons$ billing[FN, SN]

ϕ_3: card[LN] $\rightleftharpoons$ billing[SN] $\wedge$ card[addr] $\rightleftharpoons$ billing[post] $\wedge$
card[FN] $\rightleftharpoons$ billing[FN] $\rightarrow$ card[Y_c] $\rightleftharpoons$ billing[Y_b]

ϕ_4: card[LN] $\rightleftharpoons$ billing[SN] $\wedge$ card[addr] $\rightleftharpoons$ billing[post] $\wedge$
card[FN] $\approx_d$ billing[FN] $\rightarrow$ card[Y_c] $\rightleftharpoons$ billing[Y_b]

where $\approx_d$ is the similarity operator based on edit distance. □

Relative keys. Along the same lines as that traditional keys are a special case of FDs, we next define a notion of *relative keys*.

A *key* ψ *relative to* compatible attribute lists (Y_1, Y_2) of (R_1, R_2) is an MD $\bigwedge_{j\in[1,k]}(R_1[X_1[j]] \approx_j R_2[X_2[j]]) \rightarrow R_1[Y_1] \rightleftharpoons R_2[Y_2]$, where no $\approx_j$ is $\rightleftharpoons$. That is, while the matching operator is in the conclusion, it is not allowed in the hypothesis. We refer to k as the *length* of ψ. When (Y_1, Y_2) is clear from the context, we write ψ as (X_1, X_2, C), where $C = [\approx_1, \ldots, \approx_k]$.

The relative key ψ states that if $t_1[X_1]$ and $t_2[X_2]$ match *w.r.t.* the similarity operators C, then $t_1[Y_1]$ and $t_2[Y_2]$ must match *w.r.t.* the matching operator $\rightleftharpoons$.

Example 3.2: Below are keys relative to (Y_c, Y_b) of (card, billing).

rck$_1$: ([email, addr], [email, post] ‖ [=, =])
rck$_2$: ([LN, tel, FN], [SN, phn, FN] ‖ [=, =, $\approx_d$])
rck$_3$: ([LN, addr, FN], [SN, post, FN] ‖ [=, =, $\approx_d$])

Here the key rck$_2$ states that for any card tuple t and billing tuple t', if t[LN, tel] = t'[SN, phn] and t[FN] $\approx_d$ t'[FN], then $t[Y_c]$ and $t'[Y_b]$ match; similarly for rck$_1$ and rck$_3$. □

3.3 Known vs. Unknown Operators in Object Identification

We have seen that MDs are defined *w.r.t.* a collection of similarity operators. We now elaborate on which domain-specific operators in an MD are considered to be fixed in object identification.

(a) Similarity operators. Except $\rightleftharpoons$, all similarity operators in Θ tend to *compare data values* in unreliable sources, based on similarity metrics used in object identification, *e.g.,* edit distance, q-grams, Jaro distance (see [32]), or their boolean combinations. For example, a similarity operator may be parameterized with a threshold θ, denoted by '$\approx_\theta$', such that $x \approx_\theta y$ if the distance between x and y is no larger than θ. In data quality tools, such metrics are given, and are total mappings defined on specific domains.

(b) Matching operator. In contrast, $\rightleftharpoons$ is typically either not given or partially defined; it is to be "inferred" via generic reasoning about matching rules. Here "$t_1[Z_1] \rightleftharpoons t_2[Z_2]$" indicates that $t_1[Z_1]$ and $t_2[Z_2]$ *refer to the same object*. Note that $t_1[Z_1]$ and $t_2[Z_2]$, as they are in the data sources, may be *radically different* and cannot be directly matched using any similarity metric that is known in advance. Hence what we want is to reason generically about the matching operator, via implication analysis of MDs.

(c) Matching dependencies. While the semantics of MDs is defined relative to the notions of similarity and matching operators, MDs are essentially used to *infer* the match relation $\rightleftharpoons$. Currently, we can only reason generically about similarity operators (see the definition of implication for MDs in Section 4.2), but in ongoing work we are looking at how to reason *w.r.t.* fixed similarity operators.

On the other hand, a key (X_1, X_2, C) relative to (Y_1, Y_2) is to be used as a matching rule to determine whether $t_1[Y_1] \rightleftharpoons t_2[Y_2]$, by comparing $t_1[X_1]$ and $t_2[X_2]$ based solely on the *similarity metrics* C given on the source data. That is, if $t_1[X_1]$ and $t_2[X_2]$ are similar based on the given metrics, then $t_1[Y_1]$ and $t_2[Y_2]$ match, even if they may be radically different. For example, for any card tuple t and billing tuple t', to decide whether $t[Y_c]$ and $t'[Y_b]$ match, one can inspect the attributes in rck$_1$–rck$_3$ given in Example 3.2 *w.r.t.* the similarity operators specified, instead of comparing the entire Y_c and Y_b lists of t and t'. If t and t' satisfy any of rck$_1$–rck$_3$, it follows that $t[Y_c]$ and $t'[Y_b]$ refer to the same card holder.

To use relative keys as matching rules, we want the keys to be "minimal". This motivates us to define the following notion.

Relative candidate keys (RCKs). For two keys $\psi = (X_1, X_2, C)$ and $\psi' = (Z_1, Z_2, C')$ relative to (Y_1, Y_2), we denote by $\psi \leq \psi'$ if (a) $k \leq k'$, where k, k' are the lengths of ψ and ψ', respectively; (b) for each $j \in [1, k]$, $(X_1[j], X_2[j])$ is a pair $(Z_1[i], Z_2[i])$ of attributes in (Z_1, Z_2) for some i, and the similarity relation $\approx'_i$ is contained in $\approx_j$, where $C[j]$ is $\approx_j$ and $C'[i]$ is $\approx'_i$. We say that $\psi < \psi'$ if $\psi \leq \psi'$ but $\psi' \not\leq \psi$.

A key ψ is called a *relative candidate key* (RCK) for (Y_1, Y_2) if there exists no key ψ' relative to (Y_1, Y_2) such that $\psi' < \psi$.

What we want is to derive RCKs, as matching rules, via inference from given MDs. This will be discussed in Section 4.2.

Several subtleties distinguish MDs from traditional FDs. First, MDs bring domain-specific operators into the play. Second, MDs are defined across different relations; in contrast, the study of dependencies has mostly focused on the uni-relational setting (except INDs) [1, 35], where a single relation is involved. Third, MDs aim to derive matching rules on unreliable data, a departure from familiar terrain of classical dependency theory.

A notion of fuzzy functional dependencies (FFDs) has been studied (*e.g.,* [63]), which also uses similarity but differs from MDs in several aspects; in particular, as Section 4.2 will show, the implication problem for MDs is quite different from its FFD counterpart.

4. Static Analyses: New Challenges

We have introduced conditional dependencies and matching dependencies, as revisions of classical dependencies. To improve data quality using these dependencies, several fundamental questions have to be settled. In this section we provide an account of results on classical decision problems associated with these revisions. We show that these revised dependencies introduce new challenges. and make our lives harder when reasoning about them.

We focus on finite database instances. In particular, by implication we mean finite implication, although most of the results of this section remain intact for unrestricted implication (see [1, 35] for formal presentations on finite and unrestricted implication).

4.1 Reasoning about Conditional Dependencies

We begin with results on consistency, implication, finite axiomatization and propagation analyses of conditional dependencies.

Consistency. To use conditional dependencies to detect inconsistencies in real-world data, the first question we have to answer concerns whether a given set of CFDs (resp. CINDs) has conflicts and inconsistencies, *i.e.,* whether the dependencies are dirty themselves. If the dependencies are inconsistent, then there is *no need* to validate them against the data at all. Furthermore, the consistency analysis can help the users discover errors in their cleaning rules.

Formally, this can be stated as the consistency problem for conditional dependencies. For a set Σ of CFDs (resp. CINDs) and a database instance D, we write $D \models \Sigma$ if $D \models \varphi$ for *all* $\varphi \in \Sigma$.

The *consistency problem* is to determine, given a set Σ of CFDs (resp. CINDs) defined on a relational schema $\mathcal{R}$, whether there exists a nonempty database instance D of $\mathcal{R}$ such that $D \models \Sigma$.

One can specify arbitrary FDs and INDs without worrying about their consistency. This is no longer the case for CFDs.

Example 4.1: Consider two CFDs $\psi_1 = ([A] \rightarrow [B], T_1)$ and $\psi_2 = ([B] \rightarrow [A], T_2)$, where dom$(A)$ is bool, T_1 has two pattern tuples (true ‖b_1), (false ‖b_2), and T_2 contains (b_1‖false) and (b_2 ‖ true). Then there exists no nonempty instance D such that $D \models \{\psi_1, \psi_2\}$. Indeed, assume that there were such a D. Then for any tuple t in D, no matter what value $t[A]$ has, ψ_1 and ψ_2 together force $t[A]$ to take the other value from the finite domain bool. □

Recall that the domains of attributes involved in dependencies

are typically not considered in dependency theory. In contrast, the example above tells us that for consistency analysis one may have to consider whether finite-domain attributes are present. Because CFDs and CINDs are defined with constants drawn from certain domains, they may interact with domain constraints (this is why we explicitly include domains in schema specifications in Section 2).

It turns out that the consistency problem for CFDs is nontrivial. Worse, when CFDs and CINDs are taken together, the problem is undecidable, as opposed to their trivial traditional counterpart.

Theorem 4.1 [36, 20]: The consistency problem is

- NP-complete for CFDs;
- $O(1)$ for CINDs, *i.e.,* for any set Σ of CINDs defined on a schema $\mathcal{R}$, there always exists a nonempty instance D of $\mathcal{R}$ such that $D \models \Sigma$; and
- undecidable for CFDs and CINDs taken together. □

Approximation algorithms to check consistency for CFDs and eCFDs, and heuristic algorithms for checking consistency of CFDs and CINDs taken together, can be found in [36, 20].

Implication. Another central technical problem is the *implication problem*: it is to determine, given a set Σ of CFDs (resp. CINDs) and a single CFD (resp. CIND) φ defined on a relational schema $\mathcal{R}$, whether or not Σ entails φ, denoted by $\Sigma \models \varphi$, *i.e.,* whether for all instances D of $\mathcal{R}$, if $D \models \Sigma$ then $D \models \varphi$. Effective implication analysis allows us to remove redundancies from a given set of rules by finding a minimal cover of the set. Since CFDs (CINDs) tend to be larger than their traditional counterparts (due to pattern tableaux), the impact of redundancies is more evident on the performance of inconsistency detecting and repairing processes.

Recall that for FDs, the implication problem is in linear time, while for INDs, it is PSPACE-complete [1, 35]. For their conditional counterparts, the implication analyses become more intriguing.

Theorem 4.2 [36, 20]: The implication problem is

- coNP-complete for CFDs,
- EXPTIME-complete for CINDs, and
- undecidable for CFDs and CINDs taken together. □

The undecidability result is immediate from the fact that the problem is already undecidable for FDs and INDs put together.

Recall eCFDs, which extend CFDs by adding disjunction and inequality (Section 2.3). It is known that the increased expressive power of eCFDs does not incur extra complexity [19]: the consistency and implication problems for eCFDs remain NP-complete and coNP-complete, respectively, the same as their CFD counterparts.

Special cases. The absence of finite-domain attributes simplifies the consistency and implication analyses of CFDs and CINDs: their complexity bounds are comparable to the bounds for FDs and INDs.

Theorem 4.3 [36, 20]: For CFDs and CINDs that do not involve attributes with a finite domain,

- the consistency and implication problems are both decidable in quadratic time for CFDs; and
- the implication problem is PSPACE-complete for CINDs. □

The absence of finite-domain attributes, however, does not make our lives easier when it comes to eCFDs. This is because one can enforce, via eCFDs, an attribute A to draw values from a finite set only, regardless of whether $\mathsf{dom}(A)$ is infinite or not. When it comes to CFDs and CINDs taken together, their static analyses are still beyond reach in practice even without finite-domain attributes.

Theorem 4.4 [19, 20]: In the absence of finite-domain attributes, the consistency and implication problems remain

Dependencies	Consistency	Implication	Fin. Axiom
CFDs	NP-complete	coNP-complete	Yes
eCFDs	NP-complete	coNP-complete	Yes
FDs	$O(1)$	$O(n)$	Yes
CINDs	$O(1)$	EXPTIME-complete	Yes
INDs	$O(1)$	PSPACE-complete	Yes
CFDs + CINDs	undecidable	undecidable	No
FDs + INDs	$O(1)$	undecidable	No
in the absence of finite-domain attributes			
CFDs	$O(n^2)$	$O(n^2)$	Yes
CINDs	$O(1)$	PSPACE-complete	Yes
eCFDs	NP-complete	coNP-complete	Yes
CFDs + CINDs	undecidable	undecidable	No

Table 1: Complexity and finite axiomatizability

- NP-complete and coNP-complete for eCFDs, respectively;
- undecidable for CFDs and CINDs taken together. □

In practice, the relational schema is often fixed, and only dependencies vary and are treated as the input.

Theorem 4.5 [36, 20]: For CFDs and CINDs defined on a predefined, fixed relational schema,

- the consistency and implication problems are in quadratic time for CFDs; and
- the implication problem is PSPACE-complete for CINDs. □

Axiomatizability. Armstrong's Axioms for FDs can be found in almost every database textbook, and are fundamental to the implication analysis of FDs. For CFDs and CINDs the finite axiomatizability is also important, as it reveals insight into implication analysis and helps us understand how cleaning rules interact with each other.

This suggests that we find a finite set $\mathcal{I}$ (resp. $\mathcal{I}'$) of inference rules that is *sound and complete* for implication analysis of CFDs (resp. CINDs), *i.e.,* for any set Σ of CFDs (resp. CINDs) and a CFD (resp. CIND) φ, $\Sigma \models \varphi$ iff φ is provable from Σ using $\mathcal{I}$ (resp. $\mathcal{I}'$).

For CFDs and CINDs taken separately, they are finitely axiomatizable. However, just like their traditional counterparts, when CFDs and CINDs are taken together, there exists no finite axiomatization.

Theorem 4.6 [36, 20]: (a) There exist sound and complete finite inference systems for CFDs and CINDs taken separately. (b) CFDs and CINDs taken together are not finitely axiomatizable. □

We compare the complexity bounds for static analyses of CFDs, eCFDs and CINDs with their traditional counterparts in Table 1, where n denotes the size of input dependencies.

Propagation. Another important issue concerns dependency propagation (*a.k.a.* view dependencies [1]). Consider two classes of dependencies, referred to as *source dependencies* and *view dependencies*, respectively. The *dependency propagation problem* is to decide, given a view σ defined on relational sources $\mathcal{R}$ and a set Σ of source dependencies on $\mathcal{R}$, whether or not a view dependency φ is *propagated* from Σ via σ, denoted by $\Sigma \models_\sigma \varphi$, *i.e.,* whether for any instance D of $\mathcal{R}$ that satisfies the given source dependencies Σ, the view $\sigma(D)$ is guaranteed to satisfy the view dependency φ.

The need for dependency propagation analysis has long been recognized [52, 53]. For conditional dependencies, the propagation analysis is particularly important for data exchange, integration and cleaning. Indeed, dependencies on data sources often only hold *conditionally* on the view, as illustrated by the example below.

Example 4.2: Consider three data sources R_1, R_2 and R_3, containing information about customers in the UK, US and Netherlands, respectively. The following source FDs are defined on the sources:

$$f_3\colon R_1\colon [\mathsf{zip}] \rightarrow [\mathsf{street}], \quad f_{3+i}\colon R_i\colon [\mathsf{AC}] \rightarrow [\mathsf{city}],$$

where AC indicates area code, and $i \in [1, 3]$.

Denote by Σ_0 the set consisting of these FDs. Define a view σ_0 that integrates data from the sources, in terms of a union of conjunctive queries. The view schema, denoted by R, has attributes $\mathsf{zip}, \mathsf{street}, \mathsf{AC}, \mathsf{city}$ and in addition, it carries a country code (CC) attribute. Then one can expect neither $\Sigma_0 \models_{\sigma_0} f_3$ nor $\Sigma_0 \models_{\sigma_0} f_{3+i}$. Indeed, f_3 does not hold on, *e.g.,* the data from the R_2 source. Moreover, although f_{3+i} holds on each individual source R_i, it may not hold on the view because, *e.g.,* 20 is an area code in both the UK and Netherlands, for London and Amsterdam, respectively.

In contrast, $\Sigma_0 \models_{\sigma_0} \varphi_7$ and $\Sigma_0 \models_{\sigma_0} \varphi_8$ for CFDs φ_7 and φ_8:

$$\varphi_7\colon R([\mathsf{CC}, \mathsf{zip}] \to [\mathsf{street}], T_7),\ \ \varphi_8\colon R([\mathsf{CC}, \mathsf{AC}] \to [\mathsf{city}], T_8),$$

where T_7 contains a pattern tuple (44, _‖_), *i.e.,* in the UK, zip code uniquely determines street, and T_8 consists of pattern tuples (c, _‖_), when c ranges over 44, 31 (Netherlands) and 01 (the US). In other words, f_3 and f_{3+i} hold *conditionally* on the view. □

Propagation from FDs to FDs has been studied long ago (*e.g.,* [52, 53]). It is known that for views expressed in relational algebra, the problem is undecidable [52]. It is generally believed that for source FDs, view FDs, and views defined as an SPCU query (union of conjunctive queries), the problem is in PTIME (cf. [1]). The results below, taken from [37], give complexity bounds for propagation from CFDs to CFDs. In particular, it shows that in the general setting, *i.e.,* when finite-domain attributes may be present, the problem already becomes coNP-complete for source FDs, view FDs, and views defined as an SC query (with selection and Cartesian product operators); in other words, the PTIME result cited above for FD propagation only holds in the absence of finite-domain attributes.

Theorem 4.7 [37]: The dependency propagation problem is

- in PTIME for SPCU views, source CFDs and view CFDs, in the absence of finite-domain attributes;
- coNP-complete in the general setting, for
  - source FDs, view FDs, and SC views;
  - source CFDs, view CFDs, and SPCU views, and views defined with a single S, C or P (projection) operator. □

4.2 Reasoning about Matching Dependencies

We next study implication analysis of MDs. Recall that the semantics of MDs is defined *w.r.t.* domain-specific similarity and matching operators. The implication analysis of MDs aims to deduce MDs that are logical consequences of a given set of MDs, independent of any particular similarity and matching operators used.

For a set Σ of MDs and another MD ϕ defined on a relational schema $\mathcal{R}$, we say that Σ *entails* ϕ, denoted by $\Sigma \models_m \phi$, if for any instance D of $\mathcal{R}$, and *w.r.t. all* similarity and matching operators satisfying their generic axioms given in Section 3.2, if $D \models \Sigma$, then $D \models \phi$. That is, no matter how matching rules are interpreted, if Σ is enforced, then so must be ϕ. In other words, our implication analysis represents *generic reasoning* about the domain-specific matching operator. The *implication problem* for MDs is to determine, given any Σ and ϕ, whether or not $\Sigma \models_m \phi$.

Example 4.3: Consider a set Σ_1 consisting of MDs ϕ_1, ϕ_2, ϕ_3 and ϕ_4 specified in Example 3.1, and relative keys rck_1, rck_2 and rck_3 given in Example 3.2. Then $\Sigma_1 \models_m \mathsf{rck}_\mathsf{i}$ for each $i \in [1, 3]$. □

To get algorithmic insight into reasoning about MDs, a finite set $\mathcal{I}_m$ of inference rules has been proposed in [38], which is *sound and complete*, *i.e.,* for any set Σ of MDs and another MD ϕ, $\Sigma \models_m \phi$ iff ϕ is provable from Σ using $\mathcal{I}_m$. Based on the inference system a PTIME algorithm for deducing MDs has also been developed [38].

Theorem 4.8 [38]: (a) There exists a finite inference system that is sound and complete for MDs. (b) The implication problem for MDs is in PTIME. □

Based on the inference system, an effective method can be developed for deducing RCKs from a set of known MDs [38], assuming that the containment relationship of similarity relations in Θ is known (excluding $\rightleftharpoons$; see Section 3.2). Preliminary experimental results [38] tell us that derived RCKs indeed improve the quality and efficiency of various object identification methods.

Generic reasoning over similarity and matching operators may miss implications that hold for a particular application. A topic for future work is to explore possible axiomatization of practical classes of similarity and matching operators.

5. Improving Data Quality with Dependencies

Capturing inconsistencies is just a first step toward improving data quality. The next question concerns how to deal with inconsistencies and errors that emerge as violations of dependencies.

Three approaches have been put forward to handling inconsistent data: data repairing, consistent query answering, and finding condensed representations of all repairs. The first two methods were formally introduced in [7], and the third one was first studied in [6, 47, 68]. Most work on these three topics focused on traditional FDs, INDs, full dependencies, universal constraints, as well as denial constraints (see Section 2.3). Universal constraints are an extension of full dependencies, of the form: $\forall x_1 \ldots x_m\ (\phi(x_1, \ldots, x_m) \to \psi(x_1, \ldots, x_m))$; while ϕ is a conjunction of relation atoms, ψ is a *disjunction* of relation atoms and built-in predicates. For conditional dependencies only data repairing is studied [28], using CFDs.

In this section we present an overview of these approaches. We focus on main results and refer the reader to recent surveys [12, 14, 24, 26] for formal and comprehensive presentations of these topics.

5.1 Data Repairing

Given a set Σ of dependencies and an instance D of a schema $\mathcal{R}$, *data repairing* is to find a candidate *repair* of D *w.r.t.* Σ, *i.e.,* an instance D' of $\mathcal{R}$ such that $D' \models \Sigma$ and moreover, D' *minimally differs* from the original database D. That is, we edit D to fix the errors and make the data consistent. Data repairing, *a.k.a.* data reconciliation, imputation and cleaning, is the method that US national statistical agencies have been practicing for decades [40, 69].

The formal statement and complexity of data repairing highly depend on what repair model and dependencies are used.

Repair models. Repair models studied in the literature include:

X-repair [25]: a *maximal* subset $D' \subseteq D$ such that $D' \models \Sigma$. Assuming that the information in D is inconsistent but complete, this model allows tuple deletions only.

S-repair [7]: a database D' such that $D' \models \Sigma$ and $(D \setminus D') \cup (D' \setminus D)$ is *minimal*. Assuming that D is neither consistent nor complete, this model allows both tuple deletions and insertions.

U-repair [68, 16, 57]: a database D' such that $D' \models \Sigma$ and moreover, for a fixed numerical aggregation function cost over distances and accuracy of attribute values in D and D', $\mathsf{cost}(D, D')$ is *minimal*. This model supports attribute value modifications.

Observe that when only denial constraints are involved, X-repair and S-repair coincide, since tuple insertions do not help when it comes to resolving violations of denial constraints.

Several other repair models can be found in [57, 23].

Repair checking. No matter what repair semantics is considered, data repairing is essentially an optimization problem. Its associated

decision problem is to determine, given Σ, D and an instance D' of $\mathcal{R}$, whether or not D' is a repair of D *w.r.t.* Σ. This problem is referred to as the *repair checking* problem [26].

The analysis of repair checking is nontrivial, as shown by the following data complexity results.

Theorem 5.1 [25, 16, 67]: The repair checking problem is

- in PTIME for full dependencies and S-repair [67];
- coNP-hard for universal constraints [67], and in coNP for any FO constraints (cf. [26]), for S-repair;
- in PTIME for FDs and acyclic INDs taken together, for X-repair [25];
- in PTIME for INDs and X-repair [25];
- coNP-complete for one FD and one IND taken together, for X-repair [25];
- NP-complete for either a fixed set of INDs or a fixed set of FDs, for U-repair with the cost function given below [16]. □

Repairing algorithms. To clean a large dataset D, manually editing the data is unrealistic. Indeed, manually cleaning a sample of census data could easily take months by dozens of clerks [69]. This highlights the need for (semi-)automated repairing tools.

It is nontrivial to develop an algorithm that always efficiently finds accurate repairs. The example below, taken from [8], shows that even for a single key, there may be exponentially many repairs (see, *e.g.,* [1] for the definition of keys and primary keys).

Example 5.1: Consider a relation schema $R(A, B)$, a family of instances D_n of R such that $D_n = \{(a_i, b), (a_i, b') \mid i \in [1, n]\}$, and a key $A \to B$, where b and b' are distinct. Then each D_n has $2n$ tuples and 2^n repairs, for S-repair and X-repair alike. □

Among the repair models reviewed above, U-repair is often used in practice (*e.g.,* [40, 69]). Indeed, in real-life data one often finds that in an inconsistent tuple, only some fields contain errors. It is more reasonable to fix these fields rather than remove the entire tuple, to avoid loss of information.

An immediate question concerns what values should be changed and to what values they should be changed. The decision should be based on both the accuracy of the attribute values to be modified, and the "closeness" of the new value to the original value.

Below we present a simple cost metric, motivated by an approach proposed for use in US national statistical agencies [40, 69]. Assume that a *weight* is associated with each attribute A of each tuple t in D, denoted by $w(t, A)$ (if $w(t, A)$ is not available, a default value is used). The weight indicates the confidence in the *accuracy* placed by the user in $t[A]$, and can be propagated via transformations (provenance analysis [22]). Assume a *distance function* $\mathsf{dis}(v, v')$ for values v, v' in the same domain, with lower values indicating greater similarity. The cost of changing the value of an attribute $t[A]$ from v to v' can be defined as:

$$\mathsf{cost}(v, v') = w(t, A) \cdot \mathsf{dis}(v, v').$$

Intuitively, the more accurate the original $t[A]$ value v is and more distant the new value v' is from v, the higher the cost of the change is. The cost of changing the value of a tuple t to t' is the sum of $\mathsf{cost}(t[A], t'[A])$ for A ranging over all attributes in t in which the value of $t[A]$ is modified. The cost of changing D to D', denoted by $\mathsf{cost}(D, D')$, is the sum of the costs of modifying tuples in D.

Based on this metric, heuristic repairing algorithms have been developed for traditional FDs and INDs taken together [16], and for CFDs alone [28]. For a restricted (local) class of denial constraints, an approximation algorithm is also in place [58]. Another metric was proposed in [13] for repairing numerical values.

Remark. The cost metric given above is rather primitive. In particular, for practitioners it does not provide any guidance for where one should draw new values from. One may use values in the active domain of the given dataset D, guided by statistical analysis. A more reasonable way is to conduct repairing based on master data (reference data) [30, 62], *i.e.,* a collection D_r of data for an enterprise that has been cleaned, whenever available. This is, however, nontrivial. At the very least this involves object identification to match tuples in D_r and those in D that refer to the same object. Add to this structural heterogeneity, when D_r and D have different schemas. To this end, matching dependencies and relative candidate keys may help us conduct data repairing and object identification in a uniform dependency-based framework.

As observed by [14, 25], data repairing is related to belief revision, for the computation of the models of a revised theory [70]. It is also related to the study of satisfaction families (*e.g.,* [45, 49]), in particular for extending a partially-specified table to a completely-specified table in order to satisfy a given set of FDs [45].

5.2 Consistent Query Answering

Given a set Σ of dependencies, an instance D of a schema $\mathcal{R}$, and a query Q, consistent query answering aims to return *certain answers* to Q in D *w.r.t.* Σ, *i.e.,* tuples that are in the answers to Q in *each* repair of D *w.r.t.* Σ, without editing D. The *consistent query answering problem* is to determine, given Σ, D, Q and a tuple t, whether or not t is in the certain answers to Q in D *w.r.t.* Σ.

One can draw analogies between consistent query answers and certain answers studied for incomplete information (*e.g.,* [50, 46]; see [61] for a survey), and certain query answers in data integration [56]. What distinguishes the notion of consistent query answers from certain answers is its focus on repair minimality: only answers in repairs are returned, where repairs are required to minimally differ from the original D, regardless of what repair model is used. In contrast, certain answers on incomplete information are answers in all possible worlds, and certain answers in data integration (GAV) are answers over all source database instances that are consistent with a view, without any minimality requirement.

As pointed out by [7], another notion of consistent query answers was proposed in [21], in terms of minimal (propositional) logic that precludes refutation proofs, without requiring repair minimality.

Complexity bounds. The complexity of the consistent query answering problem is determined by the repair model, constraint language and query language involved. A number of data complexity bounds have been established in various settings.

We start with results established for X-repair and conjunctive queries with built-in predicates $=, \neq, <, >, \leq, \geq$.

Theorem 5.2 [25, 43, 67]: For X-repair and conjunctive queries, the consistent query answering problem is

- in PTIME for denial constraints and quantifier-free conjunctive queries [67];
- in PTIME for primary keys and a restricted class $\mathsf{C}_{\mathsf{tree}}$ of conjunctive queries [43];
- coNP-complete for denial constraints, and is already coNP-hard for a single primary key, for boolean conjunctive queries (in the absence of free variables) [25];
- in PTIME for INDs alone [25];
- Π_2^p-complete for FDs and INDs taken together [25]. □

Here $\mathsf{C}_{\mathsf{tree}}$ is defined in terms of the notion of the join graph of a query [43]. A conjunctive query is in $\mathsf{C}_{\mathsf{tree}}$ if (a) it has neither built-in predicates nor repeated relation atoms; (b) its join graph is a forest, and (c) every non-key to key join is full, *i.e.,* the join attributes cover the entire key (see [43] for the detailed definition).

The PTIME bounds given above are mostly developed by following a query rewriting approach proposed in [7]. Given a set Σ of

dependencies and an FO query Q, the idea of [7] is to rewrite Q into an FO query Q_Σ such that for every instance D, the set of answers to Q_Σ in D is precisely the set of consistent answers to Q *w.r.t.* Σ.

The results of Theorem 5.2 on denial constraints (including keys) carry over to S-repair. For S-repair, consistent query answering has been studied for queries beyond conjunctive queries. We include below complexity bounds claimed in [24], for various fragments of relational algebra. We denote a fragment $\mathcal{C}$ by listing the operators supported by $\mathcal{C}$: the presence or absence of σ (selection), π (projection), $\times$ (Cartesian product), $\cup$ (union) and $-$ (set difference).

Theorem 5.3 [24, 25, 67]: For S-repair, fragments of relational algebra and classes of universal constraints, the data complexity bounds on the consistent query answering problem include:

- $\mathcal{C}(\sigma, \times, -)$: Π_2^p-complete for universal constraints [67];
- $\mathcal{C}(\sigma, \times, -, \cup)$: in PTIME for denial constraints [25], and Π_2^p-complete for universal constraints;
- $\mathcal{C}(\sigma, \pi)$: in PTIME for primary keys [25], coNP-complete for denial constraints, and Π_2^p-complete for universal constraints [67];
- $\mathcal{C}(\sigma, \pi, \times)$: coNP-complete for primary keys [25], and Π_2^p-complete for universal constraints;
- $\mathcal{C}(\sigma, \pi, \times, -, \cup)$: coNP-complete for primary keys, and Π_2^p-complete for universal constraints. □

In the presence of INDs, the analysis of consistent query answering becomes more intriguing. In particular, the problem becomes undecidable for FDs and INDs taken together.

Theorem 5.4 [23]: For S-repair and union of conjunctive queries, the consistent query answering problem is

- Π_2^p-complete for keys and non-key conflicting INDs taken together; and
- undecidable for keys and INDs taken together. □

Here non-key conflicting INDs are those INDs that have limited interaction with keys (see [23] for the definition).

Remark. Consistent query answering has also been studied for aggregate queries and FDs [6, 42], for aggregate queries and denial constraints [13], as well as for variants of S-repair models [23, 13, 57]. Detailed discussions can be found in [12, 24, 26].

The study of consistent query answering has focused on traditional FDs, INDs and denial constraints. As we have seen earlier, CFDs and CINDs are often able to capture more inconsistencies and errors than their traditional counterparts. It is interesting to extend consistent query answering to conditional dependencies.

5.3 Condensed Representations of All Repairs

Consistent query answering is somewhat conservative since it only provides a "lower bound" on the information contained in a database [24]. Worse still, its high complexity bounds hamper its practical use. When it comes to data repairing, it is unlikely to find repairing algorithms with guaranteed precision and recall (*i.e.,* the ratio of the number of errors correctly fixed to the total number of changes made, and the ratio of the number of errors correctly fixed to the total number of errors in the database, respectively).

The limitations of data repairing and consistent query answering methods suggest that we explore alternative approaches to handling inconsistent data. Below we briefly present an approach, by developing finite, succinct representations of all repairs of a database. The idea of condensed representations is closely related to the notion of strong dependency systems studied for incomplete information, which aims to represent, in a single table, all possible worlds that satisfy a given set of constraints, such that queries in practical languages can be answered using the table (see, *e.g.,* [61]).

A notion of *nuclei* is proposed in [68]: given a database D and a set Σ of dependencies, a nucleus represents all U-repairs of D *w.r.t.* Σ in terms of a single tableau with variables. More specifically, the minimality of U-repairs is captured by means of the subsumption of tableaux. It is shown [68] that for any satisfiable set Σ of full dependencies (EGDs or TGDs) and any database D, a nucleus G can be computed such that for any conjunctive query Q, the consistent answers to Q in D *w.r.t.* Σ can be obtained by evaluating Q on G. Moreover, the nucleus G is homomorphic to all U-repairs; and for any other tableau that is homomorphic to all repairs, it is also homomorphic to G. In this sense one can draw the analogy of nuclei to the notion of cores studied for data exchange [54]. Unfortunately, for a fixed set of full dependencies, the nucleus of a database D can be exponentially large, in the size of D [68]. The space complexity hampers the applicability of nuclei.

Another approach to representing repairs is by means of answer sets of disjunctive logic programs with strong negation [6, 47], along the same lines as logical databases studied for incomplete information [61]. This approach is capable of dealing with FO queries and full dependencies, at the price of high complexity.

As another revision of techniques for incomplete information, a notion of world-set decompositions (WSDs) has recently been proposed to represent *finite sets* of possible worlds, by means of the product of decomposed relations [4, 5]. It is shown [5] that an extension of WSDs is as expressive as conditional tables, and is exponentially more succinct than unions of v-tables (see [46, 50] for the definitions of conditional tables and v-tables). They yield a strong representation system for relational algebra (see, *e.g.,* [1, 61] for strong representation systems). Although WSDs do not involve dependencies, query constructs are proposed for specifying repairs *w.r.t.* keys as WSDs [4]. However, it remains to be explored whether WSDs can be extended to yield a condensed representation of repairs, because tuples in decomposed relations are assumed to be independent across relations, which is not the case for repairs.

6. Open Research Issues

We argue that dependencies should logically become an essential part of data quality technology. Nonetheless, to find practical use of dependencies in data-quality tools, classical dependencies often need to be revised and extended. The area of dependency-based data-quality techniques is a rich source of questions and vitality. Indeed, the study of "right" application-oriented constraint languages for improving data quality is still in its infancy. Furthermore, to resolve inconsistencies that emerge as violations of dependencies, effective methods have to be developed, such as efficient repairing algorithms with performance guarantee, practical consistent query answering algorithms, and space-efficient representation systems for repairs. In particular, data repairing and object identification interact with each other, and the two processes should be combined. This highlights the need for, *e.g.,* the analyses of conditional dependencies and matching dependencies taken together.

The study of dependencies for improving data quality interacts with several other popular lines of research. The connections also give rise to a number of interesting and practical research issues.

Incomplete information. There is an intimate connection between the study of inconsistent data and incomplete information (see [1, 61] for surveys on incomplete information). As remarked in Section 5, techniques for handling inconsistent data by and large originated from the study of incomplete information. This connection deserves a full treatment. In particular, representation systems [46, 50] developed for incomplete information may shed light on the study of "strong dependency systems" [61] for representing repairs.

Another interesting aspect is the completeness of information *w.r.t. queries*: whether a query can be answered given the information available. As an example, consider a query to find the medical family history for a group of patients. To obtain accurate analytical results, the query needs to return the medical history of the last three generations of each patient in the group. Given the query, one wants to find out whether the necessary information is available for each patient. Interesting results in this direction are reported in [2], for XML data with partial information. We envisage that extensions of INDs and TGDs, reinforced with appropriate conditions (*e.g.,* for specifying the group of patients), may help in checking and assuring the completeness of information *w.r.t.* queries.

Probabilistic data management. Probabilistic databases also aim to manage imprecise and uncertain data, by associating probabilities with each tuple (see [29] for a recent survey). Inconsistent data can be recorded in a probabilistic database, as possible worlds, upon the availability of meaningful probabilities. Furthermore, probabilistic databases may yield a representation system for possible repairs, under certain extensions, which may also allow us to "rank" various repairs. Based on probabilistic databases, consistent query answering has been studied for primary keys and a class of conjunctive queries [3]. The connection between probabilistic databases and inconsistent data needs further investigation. In particular, a probabilistic database is often assumed to be disjoint independent, *i.e.,* any set of possible tuples with distinct keys is independent [29]. This is, however, often not the case for inconsistent data in the presence of INDs, let alone CINDs.

Data exchange and integration. Recently dependencies have also enjoyed a revival for data exchange [54], which specifies schema mappings in terms of TGDs, defined on a target schema or from a source schema to a target schema. Similarly, mappings in a data integration system can also be specified with TGDs (see [56] for a comprehensive survey). As shown in Section 2.2, it is often more reasonable to specify correspondences between source and target schemas in terms of dependencies augmented with conditions. Indeed, an extension of TGDs with data values has recently been studied for data exchange [34]. On the other hand, it is also interesting to explore this extension of TGDs for improving data quality.

As already pointed out by [56], source data in real life is often inconsistent or incomplete. It is important and practical to deal with dirty data in data exchange and integration. First attempts to tackle this issue are reported in, *e.g.,* [18, 55]. Another way around this is by means of "strong dependency systems" for representing all source repairs, upon their availability. That is, instead of dealing with an original inconsistent data source directly, one may want to treat a condensed representation of all repairs as the source.

Provenance. To trace the origin of errors, determine the confidence in the accuracy of data elements, and to ensure that corrections and annotations added to data by experts are properly propagated during data transformations, we need the analysis of data provenance (*a.k.a.* data lineage; see [22] for a recent survey). In particular, to decide whether a data element should be changed for repairing, one may want to know where the data came from and why it got there. While these issues have been studied for transformations defined in terms of simple conjunctive queries, we need to explore these further, for transformations specified in terms of dependencies equipped with conditions.

Web data management. The study of data quality has mostly focused on relational data. There is no reason to believe that the scale of the quality problem is any better for data on the Web. While data repairing, consistent query answering and strong dependency systems are already hard for relational data, these issues become more challenging for Web data and XML. Some preliminary results on handling inconsistent XML data are reported in [41], as adaptations of relational FDs, repairs and consistent query answering to XML.

It is known that to cope with the hierarchical structure of XML data, traditional dependencies have to be revised, *e.g.,* one needs not only absolute constraints that hold on the entire document, but also *relative* ones that hold only on certain sub-documents; moreover, there are intricate interactions between XML constraints and "type" specifications, *e.g.,* DTDs [9, 39]. To handle inconsistent XML data, we need to further extend these XML constraints with patterns, and investigate their interactions with DTDs.

Acknowledgments. I thank Philip Bohannon, Floris Geerts, Xibei Jia, Anastasios Kementsietsidis and Shuai Ma, for working with me on data quality. I am grateful to Michael Benedikt, Jan Chomicki, Floris Geerts and Jie Liu for their valuable comments, and to Richard Hull for his unfailing support. The work is supported in part by EPSRC GR/S63205/01, GR/T27433/01 and EP/E029213/1.

7. References

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

[2] S. Abiteboul, L. Segoufin, and V. Vianu. Representing and querying XML with incomplete information. *TODS* 31(1): 208-254, 2006.

[3] P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases: A probabilistic approach. In *ICDE*, 2006.

[4] L. Antova, C. Koch and D. Olteanu. From complete to incomplete information and back. In *SIGMOD*, 2007.

[5] L. Antova, C. Koch and D. Olteanu. World-set decompositions: expressiveness and efficient algorithms. In *ICDT*, 2007.

[6] M. Arenas, L. E. Bertossi, and J. Chomicki. Answer sets for consistent query answering in inconsistent databases. *TPLP* 3(4-5): 393-424, 2003.

[7] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, 1999.

[8] M. Arenas, L. E. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. Spinrad. Scalar aggregation in inconsistent databases. *TCS* 296(3): 405-434, 2003.

[9] M. Arenas, W. Fan, and L. Libkin. On the complexity of verifying consistency of XML specifications. *SICOMP*, to appear.

[10] C. Batini and M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Springer, 2006.

[11] M. Baudinet, J. Chomicki, and P. Wolper. Constraint-generating dependencies. *JCSS* 59(1): 94-115, 1999.

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

[13] L. E. Bertossi, L. Bravo, E. Franconi, and A. Lopatenko. Complexity and approximation of fixing numerical attributes in databases under integrity constraints. In *DBPL*, 2005.

[14] L. Bertossi and J. Chomicki. Query answering in inconsistent databases. *Logics for Emerging Applications of Databases*, 2003.

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

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

[17] P. D. Bra and J. Paredaens. Conditional dependencies for horizontal decompositions. In *ICALP*, 1983.

[18] L. Bravo and L. E. Bertossi. Consistent query answers in virtual data integration systems. *Inconsistency Tolerance*, 2005.

[19] L. Bravo, W. Fan, F. Geerts, and S. Ma. Increasing the expressivity of conditional functional dependencies without extra charge for complexity. In *ICDE*, 2008.

[20] L. Bravo, W. Fan, and S. Ma. Extending dependencies with conditions. In *VLDB*, 2007.

[21] F. Bry. Query answering in information systems with integrity constraints. In *IICIS*, 1996.

[22] P. Buneman, J. Cheney, W. Tan, and S. Vansummeren. Curated databases. In *PODS*, 2008.

[23] A. Calì, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *PODS*, 2003.

[24] J. Chomicki. Consistent query answering: Five easy pieces. In *ICDT*, 2007.

[25] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.* 197(1-2):90-121, 2005.

[26] J. Chomicki and J. Marcinkowski. On the computational complexity of minimal-change integrity maintenance in relational databases. *Inconsistency Tolerance*:119-150, 2005.

[27] E. F. Codd. Relational completeness of data base sublanguages. In R. Rustin (ed.): *Database Systems*: 65-98, Prentice Hall and IBM Research Report RJ 987, 1972.

[28] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, 2007.

[29] N. N. Dalvi and D. Suciu. Management of probabilistic data: Foundations and challenges. In *PODS*, 2007.

[30] A. Dreibelbis, E. Hechler, B. Mathews, M. Oberhofer, and G. Sauter. Master Data Management architecture patterns. IBM, Mar. 2007.

[31] W. W. Eckerson. Data quality and the bottom line: Achieving business success through a commitment to high quality data. The Data Warehousing Institute, 2002.

[32] A. K. Elmagarmid, P. G. Ipeirotis and V. S. Verykios. Duplicate record detection: A survey. *TKDE* 19(1): 1-16, 1007.

[33] L. English. Plain English on data quality: Information quality management: The next frontier. *DM Review Magazine*, 2000.

[34] R. Fagin. Inverting schema mappings. in *PODS*, 2007.

[35] R. Fagin and M. Y. Vardi. The theory of data dependencies - An overview. In *ICALP*, 1984.

[36] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *TODS*, to appear.

[37] W. Fan, Y. Hu, J. Liu, S. Ma, and Y. Wu. Computing view dependencies with conditions. Unpublished manuscript.

[38] W. Fan, X. Jia, and S. Ma. Object identification based on dependencies. Unpublished manuscript.

[39] W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. *J. ACM* 49(3):368-406, 2002.

[40] I. Fellegi and D. Holt. A systematic approach to automatic edit and imputation. *J. American Statistical Association* 71(353):17–35, 1976.

[41] S. Flesca, F. Furfaro, S. Greco, and E. Zumpano. Querying and repairing inconsistent XML data. In *WISE* 2005.

[42] A. Fuxman, E. Fazli, and R. J. Miller. ConQuer: Efficient management of inconsistent databases. In *SIGMOD* 2005.

[43] A. Fuxman and R. J. Miller. First-order query rewriting for inconsistent databases. *JCSS* 73(4): 610-635, 2007.

[44] Gartner. Forecast: Data quality tools, worldwide, 2006-2011. 2007.

[45] S. Ginsburg and E. H. Spanier. On completing tables to satisfy functional dependencies. *TCS* 39: 309-317, 1985.

[46] G. Grahne. *The Problem of Incomplete Information in Relational Databases*. Springer, 1991.

[47] G. Greco, S. Greco, and E. Zumpano. A logical framework for querying and repairing inconsistent databases. *TKDE* 15(6): 1389-1408, 2003.

[48] M. A. Hernandez and S. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.* 2(1): 9-37, 1998.

[49] R. Hull. Specifiable implicational dependency families. *J. ACM* 31(2): 210-226, 1984.

[50] T. Imieliński and W. Lipski Jr. Incomplete information in relational databases. *J. ACM* 31(4): 761-791, 1984.

[51] P. C. Kanellakis. Elements of relational database theory. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*: 1073-1156, 1990.

[52] A. C. Klug. Calculating constraints on relational expressions. *TODS* 5(3):260-290, 1980.

[53] A. C. Klug and R. Price. Determining view dependencies using tableaux. *TODS* 7(3):361-380, 1982.

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

[55] D. Lembo, M. Lenzerini, and R. Rosati. Source inconsistency and incompleteness in data integration. In *KRDB*, 2002.

[56] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, 2002.

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

[58] A. Lopatenko and L. Bravo. Efficient approximation algorithms for repairing inconsistent databases. In *ICDE*, 2007.

[59] M. J. Maher. Constrained dependencies. *TCS* 173(1): 113-149, 1997.

[60] M. J. Maher and D. Srivastava. Chasing constrained tuple-generating dependencies. In *PODS*, 1996.

[61] R. van der Meyden. Logical approaches to incomplete information: A survey. In J. Chomicki and G. Saake (eds.): *Logics for Databases and Information Systems*: 307-356, 1998.

[62] J. Radcliffe and A. White. Key issues for Master Data Management. Gartner, Jan. 2008.

[63] K. V. S. V. N. Raju and A. K. Majumdar. Fuzzy functional dependencies and lossless join decomposition of fuzzy relational database systems. *TODS* 13(2): 129-166, 1988.

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

[65] T. Redman. The impact of poor data quality on the typical enterprise. *Commun. ACM* 41(2): 79-82, 1998.

[66] C. C. Shilakes and J. Tylman. Enterprise information portals. Merrill Lynch, 1998.

[67] S. Staworko. Declarative inconsistency handling in relational and semi-structured databases. PhD thesis, the State University of New York at Buffalo, 2007, UB CSE TR 2008-03.

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

[69] W. E. Winkler. Methods for evaluating and creating data quality. *Inf. Syst.* 29(7): 531-550, 2004.

[70] M. Winslett. Reasoning about action using a possible models approach. In *AAAI*, 1988.