Extending dependencies with conditions for data cleaning

Extending Dependencies with Conditions for Data Cleaning

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

Abstract

Data cleaning aims to effectively detect and repair errors and inconsistencies in real life data. The increasing costs and risks of dirty data highlight the need for data cleaning techniques. This paper provides an overview of recent advances in data cleaning, based on conditional dependencies, an extension of functional and inclusion dependencies for characterizing the consistency of relational data.

1 Introduction

Real life data in all industries worldwide is routinely found dirty, *i.e.,* inconsistent, inaccurate, stale or deliberately falsified. Recent statistics reveals that enterprises typically expect data error rates of approximately 1%–5%. The costs and risks of dirty data are being increasingly recognized. It is reported that dirty data costs US businesses billions of dollars annually (cf. [14]), and that wrong price data in retail databases alone costs US consumers $2.5 billion each year [15]. It is also estimated that data cleaning accounts for 30%-80% of the development time and budget in most data warehouse projects (cf. [24]). While the prevalent use of the Web has made it possible to extract and integrate data from diverse sources, it has also increased the risks, on an unprecedented scale, of creating and propagating dirty data. These highlight the need for data cleaning tools to effectively detect and repair inconsistencies in the data. Indeed, the market for data-cleaning tools is growing at 17%, way above the 7% average forecast for other IT segments, and is projected to pass $677 million by 2011 [19].

One of the central technical questions associated with data cleaning is how to characterize the consistency of data, *i.e.,* how to tell whether the data is clean or dirty? Most data cleaning tools today, including those embedded in commercial ETL (extraction, transformation, loading) tools, heavily rely on manual effort and low-level programs that are difficult to write and maintain [22]. A more systematic approach is *constraint-based* data cleaning, to capture inconsistencies and errors as violations of integrity constraints [3, 4, 5, 8, 9, 11, 12, 21, 25]. Indeed, integrity constraints specify a fundamental part of the semantics of the data, which is critical to data quality [22]. Better still, inference systems, analysis algorithms and profiling methods developed for constraints yield systematic methods to effectively reason about the semantics of the data, and to deduce, discover and apply cleaning rules. However, constraints used for data cleaning are mostly traditional dependencies such as functional and inclusion dependencies. These constraints were developed mainly for schema design; as will be seen shortly, they are not capable of capturing errors and inconsistencies commonly found in real-life data. This calls for new constraint languages designed for data cleaning [22].

In response to the need, an extension of functional and inclusion dependencies, referred to as conditional dependencies, has recently been proposed [16, 7]. In contrast to their traditional counterparts, conditional dependencies specify patterns of semantically related data values. They are capable of capturing many common errors and inconsistencies that traditional dependencies cannot detect.

To use conditional dependencies as rules for data cleaning, one first wants to make sure that the rules are clean themselves. With this comes the need for static analyses of conditional dependencies. There are two important issues associated with conditional dependencies. One concerns consistency analysis, to determine whether or not a given set of conditional dependencies makes sense. The other concerns implication analysis, to decide whether a set of conditional dependencies logically entails another dependency. These decision problems are more intriguing for conditional dependencies than for their traditional counterparts.

We want to detect and repair errors and inconsistencies based on conditional dependencies. Given a set Σ of conditional dependencies (cleaning rules) and a database D, there are SQL techniques to automatically identify tuples in D that violate one or more dependencies in Σ. Furthermore, we want to fix the errors and find candidate repairs by editing D. While the repairing problem is difficult (intractable), it is possible to develop scalable heuristic algorithms for finding database repairs with performance guarantee.

The remainder of the paper is organized as follows. We present conditional dependencies in Section 2, and their reasoning techniques in Section 3, followed by inconsistency detection and repairing techniques in Section 4. Finally, we identify open research issues in Section 5. This paper is by no means a comprehensive survey: a number of related articles are not referenced due to the space constraint.

*Supported in part by EPSRC GR/S63205/01, GR/T27433/01 and EP/E029213/1

2 Extending Dependencies with Conditions

We extend functional and inclusion dependencies with conditions, to characterize the consistency of data.

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 customer tuple specifies a customer's phone number (country code (CC), area code (AC), phone (phn)), name, and address (street, city and zip code). An instance D_0 of the customer schema is shown in Fig. 1.

Traditional functional dependencies (FDs) on customer relations include:

f_1: [CC, AC, phn] → [street, city, zip]
f_2: [CC, AC] → [city]

That is, a customer's phone uniquely determines her address (f_1), and her country code and area code determine her city (f_2). The instance D_0 of Fig. 1 satisfies f_1 and f_2. In other words, when f_1 and f_2 are used to specify the consistency of customer data, no errors or inconsistencies can be detected in D_0 and hence D_0 is considered clean.

A closer examination of D_0, however, reveals that none of the tuples in D_0 is error-free. The inconsistencies are captured by conditional functional dependencies (CFDs):

φ_1: ([CC, zip] → [street], T_1)
φ_2: ([CC, AC, phn] → [street, city, zip], T_2)

where T_1 and T_2 are tableaux shown in Fig. 2. Each tuple in T_1 or T_2 indicates a constraint, in which '_' denotes the wild-card that can be an arbitrary value from the corresponding domain. The CFD φ_1 asserts that for customers in the UK (CC = 44), zip determines street. In other words, φ_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, rather than on the entire customer relation D_0. Tuples t_1 and t_2 in D_0 violate φ_1: they have the same zip but differ in street.

The CFD φ_2 defines three constraints, each by a distinct tuple in the tableau T_2. The first one encodes the standard FD f_1, and the other two refine f_1. More specifically, the second constraint assures that in the UK (CC = 44) and for area code 131, if two tuples have the same phn, then they must have the same street and zip, and moreover, the city *must* be EDI; similarly for the third constraint. While D_0 satisfies f_1, each of t_1 and t_2 in D_0 violates φ_2: CC = 44 and AC = 131, but city ≠ EDI. Similarly, t_3 violates φ_2.

More formally, a CFD φ defined on a relation schema R is a pair $(R : X \to Y, T_p)$, where (1) $X \to Y$ 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 tuple $t_p \in T_p$, $t_p[A]$ is either a constant 'a' in $\mathsf{dom}(A)$, or an unnamed variable '_' that draws values from $\mathsf{dom}(A)$. We write φ as $(X \to Y,\ T_p)$ when R is clear from the context.

	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

(a) Tableau T_1 of φ_1 = ([CC, zip] → [street], T_1)

CC	zip	street
44	_	_

(b) Tableau T_2 of φ_2 = ([CC, AC, phn] → [street, city, zip], T_2)

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

Figure 2. Example CFDs

To give the semantics of CFDs, we define a *match* operator $\asymp$ on data values and '_': $\eta_1 \asymp \eta_2$ if either $\eta_1 = \eta_2$, or η_1 is a constant 'a' and η_2 is '_'. The operator $\asymp$ extends to tuples, *e.g.,* (Mayfield, EDI) $\asymp$ (_, EDI) but (Mayfield, EDI) $\not\asymp$ (_, NYC). We say that a tuple t_1 *matches* t_2 if $t_1 \asymp t_2$.

An instance D of R *satisfies* the CFD φ, denoted by $D \models \varphi$, if for *each pair* of tuples t_1, t_2 in D, and for *each* tuple t_p in the pattern tableau T_p of φ, if $t_1[X] = t_2[X] \asymp t_p[X]$, then $t_1[Y] = t_2[Y] \asymp t_p[Y]$. Intuitively, each tuple t_p in the pattern tableau T_p of φ is a constraint defined on the set $D_{t_p} = \{t \mid t \in D, t[X] \asymp t_p[X]\}$ 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 embedded FD, and (b) assures the binding between *constants* in $t_p[Y]$ and *constants* in $t_1[Y]$. Note that this constraint is defined on the subset D_{t_p} of D identified by $t_p[X]$, rather than on the entire D.

Conditional inclusion dependencies. Next consider two schemas, referred to as source and target, respectively:

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, specifying customer orders of books and CDs. Example source and target databases are shown in Fig. 3.

To find schema mapping from source to target (*e.g.,* [20]), or to detect errors across these databases (*e.g.,* [5]), one may want to specify inclusion dependencies (INDs) such as order(title, price) ⊆ book(title, price), and order(title, price) ⊆ 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 matching CD tuple; similarly for CDs in the order table.

In contrast, one can specify the following conditional inclusion dependencies (CINDs), an extension of INDs:

	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_3: (order(title, price; type) $\subseteq$ book(title, price), T_3)
φ_4: (order(title, price; type) $\subseteq$ CD(album, price), T_4)
φ_5: (CD(album, price; genre)) $\subseteq$ book(title, price; format), T_5)

where T_3–T_5 are pattern tableaux shown in Fig. 4. The CIND φ_3 asserts 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 attribute; similarly for φ_4. The CIND φ_5 states 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' match the album and price of t, and moreover, the format of t' must be "audio".

While the databases of Fig 3 satisfy φ_3 and φ_4, they violate φ_5. Indeed, tuple t_9 in the CD table has an "a-book" genre, but it cannot find a match in the book table. Note that while t_9 and t_7 in the book table agree on their album (title) and price, the format of t_7 is "paper cover" rather than "audio" as required by the pattern given in tableau T_5.

Formally, a CIND ψ defined on 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, 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 the *pattern tableau* of ψ with attributes in X, X_p and Y, Y_p, such that for each tuple $t_p \in T_p$, (a) $t_p[X] = t_p[Y]$, consisting of unnamed variable '_'; and (b) for each A in X_p or Y_p, $t_p[A]$ is a constant 'a'.

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* t_1 in the relation D_1, and for *each* tuple t_p in the pattern tableau T_p, 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, t_p is a constraint defined on $D_{(1,t_p)} = \{t_1 \mid t_1[X_p] = t_p[X_p]\}$, such that (a) the IND $R_1[X] \subseteq R_2[Y]$ embedded 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 match the pattern $t_p[Y_p]$. Intuitively, X_p is used to identify the R_1 tuples on which ψ is defined, and Y_p enforces the matching R_2 tuples to satisfy a certain form.

From these examples one can see that in contrast to FDs and INDs, CFDs and CINDs specify patterns of semantically related constants, and are capable of capturing more errors and inconsistencies than their traditional counterparts can catch. In practice dependencies that hold conditionally may arise in a number of domains. In particular, when integrating data, dependencies that hold only in a subset of sources will hold only conditionally in the integrated data.

Traditional FDs and INDs are special cases of CFDs and CINDs, respectively, in which the pattern tableau consists of a single tuple, containing unnamed variable '_' only.

(a) T_3 in φ_3 = (order(title, price; type) $\subseteq$ book(title, price), T_3)

title	price	type	title	price
_	_	book	_	_

(b) T_4 in φ_4 = (order(title, price; type) $\subseteq$ CD(album, price), T_4)

title	price	type	album	price
_	_	CD	_	_

(c) T_5 in φ_5=(CD(album, price; genre) $\subseteq$ book(title, price; format), T_5)

album	price	genre	title	price	format
_	_	a-book	_	_	audio

Figure 4. Example CINDs

As remarked earlier, dependencies considered for data cleaning so far include traditional FDs, INDs as well as a form of full dependencies, referred to as denial constraints (see [12] for a recent survey). There have also been extensions of CFDs, by supporting inequality and disjunctions, without incurring extra complexity [6]. Data cleaning tools based on CFDs and CINDs are also being developed.

3 Reasoning about Dependencies

To use CFDs and CINDs to detect and repair errors and inconsistencies, a number of fundamental questions associated with these conditional dependencies have to be settled. In this section we address three central technical problems, namely, consistency, implication and axiomatizability.

Consistency. Given a set Σ of CFDs (resp. CINDs), can one tell whether the dependencies in Σ are dirty themselves? If the input set Σ is found inconsistent, then there is *no need* to check the cleaning rules against the data at all. Further, the analysis helps the user discover errors in the 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 D, we write $D \models \Sigma$ if $D \models \varphi$ for *all* $\varphi \in \Sigma$. The *consistency problem* is to determine, given Σ defined on a relational schema $\mathcal{R}$, whether or not there exists a nonempty instance D of $\mathcal{R}$ such that $D \models \Sigma$.

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

Example 3.1: 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 patterns (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, 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. □

It turns out that while for CINDs the consistency problem is not an issue, for CFDs it is nontrivial. Worse, when CFDs and CINDs are put together, the problem becomes undecidable, as opposed to their trivial traditional counterpart.

Theorem 3.1 [16, 7]: *The consistency problem is*

- NP-*complete for* CFD*s,*
- *trivially decidable for* CIND*s,* i.e., *for any set* Σ *of* CIND*s defined on a schema* $\mathcal{R}$*, there always exists a nonempty instance* D *of* $\mathcal{R}$ *such that* $D \models \Sigma$*, and*
- *undecidable for* CFD*s and* CIND*s taken together.* □

Fortunately, there are effective approximate and heuristic algorithms to check consistency for CFDs and for CFDs and CINDs taken together, respectively (see [16, 6] for details).

Implication. Another central technical problem is the *implication problem*: given a set Σ of CFDs (resp. CINDs) and a single CFD (resp. CIND) φ defined on a relational schema $\mathcal{R}$, it is to determine whether or not Σ entails φ, denoted by $\Sigma \models \varphi$, *i.e.,* whether or not for all instances D of $\mathcal{R}$, if $D \models \Sigma$ then $D \models \varphi$. Effective implication analysis allows us to deduce new cleaning rules and to remove redundancies from a given set of rules, among other things.

It is known that for FDs, the implication problem is decidable in linear time, while for INDs, it is PSPACE-complete. It becomes more intriguing for CFDs and CINDs.

Theorem 3.2 [16, 7]: *The implication problem is*

- *co*NP-*complete for* CFD*s,*
- EXPTIME-*complete for* CIND*s, and*
- *undecidable for* CFD*s and* CIND*s taken together.* □

The undecidability result is not surprising: the problem is already undecidable for FDs and INDs put together.

In certain practical cases the consistency and implication analyses for CFDs and CINDs have complexity comparable to their traditional counterparts, as stated below. For data cleaning in practice, the relational schema is often fixed, and only dependencies vary and are treated as the input.

Theorem 3.3 [16, 7]: *For* CFD*s and* CIND*s defined on a relational schema* $\mathcal{R}$*, if either* $\mathcal{R}$ *is predefined, or no attributes in the given dependencies have a finite domain, then*

- *the consistency and implication problems are both decidable in quadratic time for* CFD*s; and*
- *the implication problem is* PSPACE-*complete for* CIND*s, the same as for standard* IND*s.* □

Axiomatizability. Armstrong's Axioms for FDs are found in almost every database textbook, and are fundamental to the implication analysis of FDs. Similarly, there exists a finite set of inference rules for INDs. For conditional dependencies the finite axiomatizability is also important, as it reveals insight of the implication analysis and helps us understand how cleaning rules interact with each other.

Dependencies	Consistency	Implication	Fin. Axiom
CFDs	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
with predefined schema or in the absence of finite domain			
CFDs	$O(n^2)$	$O(n^2)$	Yes
CINDs	$O(1)$	PSPACE-complete	Yes

Table 1. Complexity and finite axiomatizability

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

The good news is that when CFDs and CINDs are taken separately, they are finitely axiomatizable. However, just like their traditional counterparts, when CFDs and CINDs are taken together, they are not finitely axiomatizable.

Theorem 3.4 [16, 7]: *There exist finite inference systems that are sound and complete for* CFD*s and* CIND*s taken separately. When* CFD*s and* CIND*s are taken together, they are not finitely axiomatizable.* □

Table 1 compares the complexity bounds for static analyses as well as the finite axiomatizability of CFDs and CINDs with their traditional counterparts.

4 Detecting and Repairing Inconsistencies

Given a set Σ of conditional dependencies defined on a schema $\mathcal{R}$ and an instance D of $\mathcal{R}$, we want to effectively detect inconsistencies in D that emerge as violations of Σ, and moreover, if D is dirty, to find candidate repairs of D.

Detecting inconsistencies. Given Σ and D, one needs an automated method to find all the *inconsistent tuples* in D *w.r.t.* Σ, *i.e.,* the tuples that (perhaps together with other D tuples) violate some dependencies in Σ.

In contrast to traditional FDs, a CFD $\varphi = (X \rightarrow Y,\ T_p)$ carries a possibly large pattern tableau T_p. Nevertheless, one can use a single pair of SQL queries $(Q^C_\varphi, Q^V_\varphi)$ to find all tuples in D that violate φ. In a nutshell, Q^C_φ detects *single-tuple* violations, *i.e.,* the tuples t in D that match some pattern tuple $t_p \in T_p$ on the X attributes, but t does not match t_p on the Y attributes. On the other hand, query Q^V_φ finds *multi-tuple* violations, *i.e.,* tuples that match $t_p[X]$ for some $t_p \in T_p$ but violate the standard FD embedded in φ. The size of the SQL queries is independent of the size of T_p.

Example 4.1: When evaluated against a customer relation D_0, the two SQL queries given in Fig. 5 find all tuples in D_0 that violate CFD φ_2 of Fig. 2. Here $t[A] \asymp t_p[A]$ denotes the SQL expression ($t[A] = t_p[A]$ OR $t_p[A] =$ '_'), while $t[B] \not\asymp t_p[B]$ denotes ($t[B] \neq t_p[B]$ AND $t_p[B] \neq$ '_'). □

$Q^C_{\varphi_2}$ **select** t **from** customer $t, T_2\ t_p$
where $t[\mathsf{CC}] \asymp t_p[\mathsf{CC}]$ AND $t[\mathsf{AC}] \asymp t_p[\mathsf{AC}]$ AND $t[\mathsf{phn}] \asymp t_p[\mathsf{phn}]$
AND ($t[\mathsf{street}] \not\asymp t_p[\mathsf{street}]$ OR $t[\mathsf{city}] \not\asymp t_p[\mathsf{city}]$ OR
$t[\mathsf{zip}] \not\asymp t_p[\mathsf{zip}]$)

$Q^V_{\varphi_2}$ **select distinct** $t[\mathsf{CC}], t[\mathsf{AC}], t[\mathsf{phn}]$ **from** customer $t, T_2\ t_p$
where $t[\mathsf{CC}] \asymp t_p[\mathsf{CC}]$ AND $t[\mathsf{AC}] \asymp t_p[\mathsf{AC}]$ AND $t[\mathsf{phn}] \asymp t_p[\mathsf{phn}]$
group by $t[\mathsf{CC}]$, $t[\mathsf{AC}]$, $t[\mathsf{phn}]$
having count(**distinct** $t[\mathsf{street}]$, $t[\mathsf{city}]$, $t[\mathsf{zip}]$)> 1

Figure 5. SQL queries for checking CFD φ_2

This method can be extended to a set Σ of CFDs (resp. CINDs): one can find a single pair of SQL queries to find all inconsistent tuples in D *w.r.t.* Σ, such that the size of the queries depends on neither the number of CFDs (resp. CINDs) in Σ nor the size of pattern tableau in each dependency in Σ (see [16, 6] for details).

Finding candidate repairs. Given Σ and possibly dirty D, we want to find a candidate *repair* of D, *i.e.,* an instance D' of $\mathcal{R}$ that is *consistent*, *i.e.,* $D' \models \Sigma$, and moreover, D' *minimally differs* from the original database D. That is, we edit D to fix the errors and to make the data consistent. This is the data cleaning approach that US national statistical agencies, among others, has been practicing for decades [17].

The effectiveness and complexity of data repair methods depend on what repair model is used. One model allows tuple deletions only [11], assuming that the information in D is inconsistent but complete. Here a repair D' is a maximal subset of D such that $D' \models \Sigma$. Another model allows both tuple deletions and insertions [3], assuming that D is neither consistent nor complete. Here a repair D' is an instance of $\mathcal{R}$ such that $(D \setminus D') \cup (D' \setminus D)$ is minimal when D' ranges over all instances of $\mathcal{R}$ that satisfy Σ. A more practical model is based on updates, *i.e.,* attribute value modifications. It is common that in an inconsistent tuple, only some fields contain errors. One should fix these fields rather than remove the entire tuple, to avoid loss of correct information. This is the model adopted by US national statistical agencies [17] and recently revisited by [25, 5].

An immediate question about the update model concerns what values should be changed and what values should be chosen to replace the old values. One should make the decisions based on both the accuracy of the attribute values to be modified, and the "closeness" of the new value to the original value. Following the practice of US national statistical agencies [17], one can define a cost metric as follows. Assuming that a *weight* in the range $[0, 1]$ 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 weight can be used instead). The weight reflects the confidence of the *accuracy* placed by the user in the attribute $t[A]$, and can be propagated via data provenance analysis in data transformations. For two values v, v' in the same domain, assume that a *distance function* $\mathsf{dis}(v, v')$ is in place, with lower values indicating greater similarity. One way to define the cost of changing the value of an attribute $t[A]$ from v to v' is:

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

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 this change. The cost of changing the value of a tuple t to t' is the sum of $\mathsf{cost}(t[A], t'[A])$ when A ranges over all attributes in t for 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. A repair of D in the update model is an instance D' of $\mathcal{R}$ such that $\mathsf{cost}(D, D')$ is minimal when D' ranges over all instances of $\mathcal{R}$ that satisfy Σ. This allows us to reduce repairing problem to an optimization problem. In practice, we want to pick new values v' from a reference database or from the active domain of the database based on certain statistical analysis.

The accuracy of a repair can be measured by precision and recall metrics, which are 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.

It is prohibitively expensive to find a repair by manual effort. The objective of data cleaning is to develop effective methods that automatically find candidate repairs of D, which are subject to inspection and changes by human experts. It is, however, nontrivial to find a candidate repair.

Theorem 4.1 [5]: *Given a set Σ of dependencies and a database D, the problem of determining whether there exists a repair D' with minimal $\mathsf{cost}(D, D')$ is* NP*-complete, when Σ is either a fixed set of* FD*s or a fixed set of* IND*s.* □

To cope with the tractability, several heuristic algorithms have been developed (*e.g.,* [5, 13]). A central idea is to separate the decision of which attribute values should be made equal from the decision of what value should be assigned to these attributes. Delaying value assignment allows a poor local decision to be improved in a later stage of the repairing process, and also allows a user to inspect and modify a repair. To this end an equivalence class $\mathsf{eq}(t, A)$ can be associated with each tuple t in the dirty database D and each attribute A in t. The repairing is conducted by merging and modifying the equivalence classes of attributes in D. For example, if tuples t_1, t_2 in D violate an FD $X \rightarrow A$, one can fix the inconsistency by merging $\mathsf{eq}(t_1, A)$ and $\mathsf{eq}(t_2, A)$ into one, *i.e.,* by forcing t_1 and t_2 to agree on their A attributes. If a tuple t_1 violates an IND $R_1[X] \subseteq R_2[Y]$, one can resolve the conflict by picking a tuple t_2 in the R_2 relation that is close to t_1, or inserting a new tuple t_2 into the R_2 table, such that for each corresponding attribute pair (A, B) in $[X]$ and $[Y]$, $t_1[A] = t_2[B]$ by merging $\mathsf{eq}(t_1, A)$ and $\mathsf{eq}(t_2, B)$ into one. A target value is picked and assigned to each equivalence class when no more merging is possible.

Based on this idea, heuristic algorithms have been developed for repairing databases using FDs and INDs [5]. The

algorithms modify tuple attributes in the right-hand side of an FD or an IND in the presence of a violation. This strategy, however, no longer works for CFDs: the process may not even terminate if only tuple attributes in the right-hand side of a CFD can be modified. Heuristic algorithms for repairing CFDs have been developed [13], which may modify tuple attributes in either the left-hand side or right-hand side of a CFD. Together with a statistical method, this approach guarantees that the accuracy of the candidate repairs found is above a predefined bound with a high confidence.

5 Concluding Remarks

The primary goal of this paper is to provide an overview of recent advances in conditional dependencies for data cleaning. There is much more to be done. One topic for future research is to find heuristic methods, with performance guarantees, for reasoning about CFDs and CINDs taken together. Another topic is data profiling, to develop effective methods to discover useful CFDs and CINDs from sample data. While there has been work on discovering FDs and INDs, we are not aware of any solid technique for discovering CFDs and CINDs, which is more involved than their traditional counterpart. A more challenging topic is to develop scalable algorithms for finding repairs based on both CFDs and CINDs, with performance guarantee.

The notion of constraint-based repairs is introduced in [3]. Also proposed in [3] is the notion of *consistent query answers*, which, given a query Q posed on an inconsistent database D, is to find tuples that are in the answer of Q over every repair of D [3, 9, 11, 21, 25] (see [10, 11] for surveys). Another alternative to finding database repairs is by developing finite and succinct representations of all possible repairs [25, 1, 2]. Data cleaning systems reported in the literature include AJAX [18], which provides users with a declarative language for specifying cleaning programs, and Potter's Wheel [23] that extracts structure for attribute values and uses these to flag discrepancies in the data. Most commercial ETL tools have little built-in cleaning capability, covering mainly data transformation needs such as type conversions, string functions, etc (see [22] for a survey). While a constraint repair facility will logically become part of the cleaning process, we are not aware of analogous functionality currently in any of the systems. It is interesting to extend these systems by supporting CFDs and CINDs.

References

[1] L. Antova, C. Koch, and D. Olteanu. 10^{10^6} worlds and beyond: Efficient representation and processing of incomplete information. In *ICDE*, 2007.

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

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

[4] L. Bertossi and J. Chomicki. Query answering in inconsistent databases. In J. Chomicki, R. van der Meyden, and G. Saake, editors, *Logics for Emerging Applications of Databases*. Springer, 2003.

[5] 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.

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

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

[8] R. Bruni and A. Sassano. Errors detection and correction in large scale data collecting. In *IDA*, 2001.

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

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

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

[12] J. Chomicki and J. Marcinkowski. On the computational complexity of minimal-change integrity maintenance in relational databases. In L. Bertossi, A. Hunter, and T. Schaub, editors, *Integrity Tolerance*, pages 119–150. Springer, 2005.

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

[14] W. W. Eckerson. Data Quality and the Bottom Line: Achieving Business Success through a Commitment to High Quality Data. Technical report, The Data Warehousing Institute, 2002. *http://www.tdwi.org/research/display.aspx?ID=6064.*

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

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

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

[18] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Declarative data cleaning: Language, model and algorithms. In *VLDB*, 2001.

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

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

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

[22] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 23(4), 2000.

[23] V. Raman and J. M. Hellerstein. "Potter's Wheel: An Interactive Data Cleaning System". In *VLDB*, 2001.

[24] C. C. Shilakes and J. Tylman. Enterprise information portals, Nov. 1998.

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