Increasing the Expressivity of Conditional Functional Dependencies without Extra Complexity

Loreto Bravo [1], Wenfei Fan [1,2], Floris Geerts [1], Shuai Ma [1]

[1]*School of Informatics, University of Edinburgh, UK*
[2]*Bell Laboratories, USA*

{lbravo,wenfei,fgeerts,sma1}@inf.ed.ac.uk

***Abstract*— The paper proposes an extension of CFDs [1], referred to as *extended Conditional Functional Dependencies* (eCFDs). In contrast to CFDs, eCFDs specify patterns of semantically related values in terms of disjunction and inequality, and are capable of catching inconsistencies that arise in practice but cannot be detected by CFDs. The increase in expressive power does not incur extra complexity: we show that the satisfiability and implication analyses of eCFDs remain NP -complete and coNP -complete, respectively, *the same as* their CFDs counterparts. In light of the intractability, we present an algorithm that *approximates* the maximum number of eCFDs that are satisfiable. In addition, we revise SQL techniques for detecting CFD violations, and show that violations of multiple eCFDs can be captured via a single pair of SQL queries. We also introduce an incremental SQL technique for detecting eCFD violations in response to database updates. We experimentally verify the effectiveness and efficiency of our SQL -based detection methods.**

I. Introduction

Conditional functional dependencies (CFDs) have recently been introduced in [1] for data cleaning. CFDs extend functional dependencies (FDs) by enforcing patterns of semantically related values, and have proved more effective in catching data inconsistencies than FDs, which were currently the basis of many data-cleaning tools [2], [3], [4], [5], [6].

To capture inconsistencies that commonly arise in real-life data, however, one often needs to use more expressive constraints, as illustrated by the following example.

Example 1.1: Let us consider a schema similar to the one used in [1]: cust(AC, PN, NM, STR, CT, ZIP). It specifies a customer in New York State in terms of the customer's phone (area code (AC), phone number (PN)), name (NM), and address (street (STR), city (CT), zip code (ZIP)). An instance D_0 of cust is shown in Fig. 1.

One may want to specify the CFD below on cust:

ϕ_1: (CT $\rightarrow$ AC, {(Albany $\|$ 518), (Troy $\|$ 518), (Colonie $\|$ 518)})

The CFD is a pair consisting of an embedded standard FD and a pattern tableau. It states that the FD CT $\rightarrow$ AC (city uniquely determines area code) holds if CT is Albany, Troy or Colonie; and in addition, the pattern tableau refines the FD by enforcing bindings between cities and area codes: if CT is one of these cities, then AC must be 518. This CFD identifies tuple t_1 in Fig. 1 as an error: CT is Albany but AC is not 518. This error cannot be caught by traditional FDs.

	AC	PN	NM	STR	CT	ZIP
t_1:	718	1111111	Mike	Tree Ave.	Albany	12238
t_2:	518	2222222	Joe	Elm Str.	Colonie	12205
t_3:	518	2222222	Jim	Oak Ave.	Troy	12181
t_4:	100	1111111	Rick	8th Ave.	NYC	10001
t_5:	212	3333333	Ben	5th Ave.	NYC	10016
t_6:	646	4444444	Ian	High St.	NYC	10011

Fig. 1. Instance D_0 of the cust relation

A cursory examination of New York area codes reveals that most cities in the state have a *unique* area code, except NYC and LI (Long Island). Such situations commonly arise in practice, and it is useful to capture this as constraints when checking inconsistencies of the data. Unfortunately, it cannot be defined as a standard FD or even a CFD.

This, however, can be expressed as the constraint below:

ϕ_2: CT $\notin$ {NYC, LI} $\rightarrow$ AC

which assures that the FD CT $\rightarrow$ AC holds if CT is *not in* the set {NYC, LI}, instead of on the entire cust database.

For NYC, one can write the following constraint:

ϕ_3: CT $\in$ {NYC} $\rightarrow \emptyset$ with AC $\in$ {212, 718, 646, 347, 917}

This asserts that when CT is NYC, AC must be *either* 212, 718, 646, 347, *or* 917. That is, here CT is associated with *a disjunction of options* rather than with a *single* value, and chooses one from the *multiple* choices. Again this is common in practice. With ϕ_3 we can identify tuple t_4 of Fig. 1 as an error: 100 is *not* one of the area codes associated with NYC. Similarly one can specify the area codes for LI.

However, these constraints cannot be defined as CFDs: they are specified with inequality (ϕ_2) and disjunction (ϕ_3), which are beyond the expressive power of CFDs. □

A more powerful language is necessary to express these constraints. However, increased expressive power often comes with the extra charge for complexity.

Contribution. The first contribution of the paper is a CFD extension, well-balanced between expressiveness and complexity, referred to *extended Conditional Functional Dependencies* (eCFDs). eCFDs support disjunction and inequality, and can

express all the constraints we have encountered so far. They include CFDs as a special case, and can catch inconsistencies, as violations of eCFDs, that commonly arise in real-life data but cannot be detected by CFDs.

Our second contribution consists of complexity bounds of two central technical problems associated with eCFDs: *the satisfiability problem* is to determine whether or not an input set of eCFDs makes sense, *i.e.,* whether there exists a nonempty database that satisfies the eCFDs; and the *implication* problem is to determine whether or not an eCFD is entailed by a given set of eCFDs. These are important in validation and optimization of data-cleaning processes in practice. We show that despite the increased expressive power, eCFDs do not make our lives harder: for eCFDs these problems remain NP-complete and coNP-complete, respectively, *the same as* their CFD counterparts.

Our third contribution is an approximation-preserving reduction from the maximum satisfiability problem of eCFDs to the maximum generalized satisfiability problem (MAXGSAT) [7]. As a result we can apply existing approximation algorithms for MAXGSAT to find an approximation of the largest possible subset of satisfiable eCFDs in a given input set Σ. This not only allows to efficiently detect the unsatisfiability of Σ with some certainty, but also provides the user with a set of satisfiable eCFDs that can serve as a starting point to inspect the remaining eCFDs in Σ.

Our fourth contribution consists of two algorithms for detecting data inconsistencies. The first one is a batch algorithm that, given a dataset D and a set Σ of eCFDs, finds all tuples that violate some eCFDs in Σ. We revise the SQL detection technique developed in [1] and show that a *single pair* of SQL queries suffices to find all violations of Σ despite that Σ may consist of *multiple* eCFDs. The second one is a novel *incremental* algorithm: in response to updates ΔD to the database D, the algorithm efficiently finds all violations of Σ in the updated dataset $D \oplus \Delta D$, by updating violations of Σ in D and minimizing unnecessary recomputation. This again uses SQL queries only.

Our fifth contribution is an experimental study of the performance of our batch and incremental detection methods. We find that our methods scale well when the data size gets large and eCFDs (pattern tuples in the eCFDs) get complicated; in addition, the incremental method outperforms the batch one in response to reasonably-sized database updates.

Organization. Section II formally defines eCFDs. The satisfiability and implication problems of eCFDs are studied in Section III, followed by our approximation analysis of the satisfiabilty problem in Section IV. Next, Section V presents the batch and incremental detection techniques, followed by their experimental study in Section VI. Related work is discussed in Section VII and Section VIII presents some concluding remarks.

ψ_1 = (cust: [CT] $\rightarrow$ [AC], $\emptyset$, T_1), where the pattern tableau T_1 is

CT	AC	$\emptyset$
$\overline{\{\text{NYC, LI}\}}$	_	
{Albany, Troy, Colonie}	{518}	

ψ_2 = (cust: [CT] $\rightarrow \emptyset$, AC, T_2), where the pattern tableau T_2 is

CT	$\emptyset$	AC
{NYC}		{212, 718, 646, 347, 917}

Fig. 2. Example eCFDs

II. Extended CFDs

We now define extended Conditional Functional Dependencies (eCFD). Consider a relation schema R defined over a finite set of attributes, denoted by $\mathsf{attr}(R)$. For each A in $\mathsf{attr}(R)$ we denote by $\mathsf{dom}(A)$ the domain of attribute A, which can be infinite or finite (with at least two elements).

Syntax. An eCFD φ is a triple $(R : X \rightarrow Y,\ Y_p,\ T_p)$, where (1) $X, Y, Y_p \subseteq \mathsf{attr}(R)$, and $Y \cap Y_p = \emptyset$; (2) $X \rightarrow Y$ is a standard FD, referred to as the *embedded functional dependency* of φ; and (3) T_p is a *pattern tableau* consisting of a finite number of pattern tuples over the attributes in $X \cup Y \cup Y_p$, such that for any tuple $t_p \in T_p$ and for each attribute A in $X \cup Y \cup Y_p$, $t_p[A]$ is either an unnamed variable '_', a set S or a complement set $\overline{S}$, where S is a finite subset of $\mathsf{dom}(A)$. If A appears in both X and $Y \cup Y_p$, we use $t_p[A_L]$ and $t_p[A_R]$ to indicate the A field of t_p corresponding to A in X and $Y \cup Y_p$, respectively. We denote X by $\mathsf{LHS}(\varphi)$ and $Y \cup Y_p$ by $\mathsf{RHS}(\varphi)$.

Example 2.1: Constraints $\phi_1 - \phi_3$ of Example 1.1 can be expressed as the eCFDs shown in Fig. 2. In ψ_1, $X = \{\mathsf{CT}\}$, $Y = \{\mathsf{AC}\}$ and $Y_p = \emptyset$. In ψ_2, $X = \{\mathsf{CT}\}$, $Y = \emptyset$ and $Y_p = \{\mathsf{AC}\}$. Here ψ_1 expresses ϕ_1 and ϕ_2, while ψ_2 represents ϕ_3. We use $\|$ to separate X, Y and Y_p. □

Observe that *each pattern tuple* is actually a constraint that enforces binding of semantically related values, and is referred to as a *pattern* constraint in the eCFD.

Semantics. Let us consider $Z \subseteq X \cup Y \cup Y_p$, a tuple t in an instance I of R, and a *pattern tuple* $t_p \in T_p$. We say that the *data* tuple $t[Z]$ *matches* the pattern tuple $t_p[Z]$, denoted by $t[Z] \asymp t_p[Z]$, if for each $A \in Z$, (1) if $t_p[A] =$ '_', then $t[A] \in \mathsf{dom}(A)$, *i.e.,* an arbitrary value; (2) if $t_p[A] = S$, then $t[A] \in S$; and (3) if $t_p[A] = \overline{S}$, then $t[A] \notin S$.

For example, consider t_1, t_4 of Fig. 1 and the first pattern tuple t_p of ψ_1 in Fig. 2. Then $t_1[\mathsf{CT}, \mathsf{AC}] \asymp t_p[\mathsf{CT}, \mathsf{AC}]$ since $t_1[\mathsf{CT}] \notin \{\text{NYC, LI}\}$, and $t_1[\mathsf{AC}] \asymp$ '_'. However, $t_4[\mathsf{CT}, \mathsf{AC}] \not\asymp t_p[\mathsf{CT}, \mathsf{AC}]$ since $t_4[\mathsf{CT}] \in \{\text{NYC, LI}\}$.

A relation I of R *satisfies* eCFD φ, denoted by $I \models \varphi$, if for *each* pattern tuple $t_p \in T_p$, the following holds. Let $I(t_p) = \{t \in I \mid t[X] \asymp t_p[X]\}$, which is the set of tuples t in I such that $t[X]$ matches $t_p[X]$. Then (1) $I(t_p)$ must satisfy the embedded FD $X \rightarrow Y$, *i.e.,* for any two tuples t_1, t_2 in

$I(t_p)$, if $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$; and furthermore, (2) for *each* tuple $t \in I(t_p)$, $t[Y, Y_p] \asymp t_p[Y, Y_p]$, *i.e.,* all tuples t in $I(t_p)$ must match the pattern $t_p[Y, Y_p]$.

Intuitively, $I(t_p)$ identifies the set of tuples on which the constraint t_p is defined, *i.e.,* the constraint only applies to the tuples in I that match the pattern $t_p[X]$. Both, the embedded FD $X \rightarrow Y$ and the pattern $t_p[Y, Y_p]$, are enforced on the tuples in $I(t_p)$.

Example 2.2: Consider the database D_0 of Fig. 1 and the first pattern tuple t_p in ψ_1. Here $D_0(t_p) = \{t_1, t_2, t_3\}$, *i.e.,* the tuples whose CT attribute is neither NYC nor LI. In other words, the constraint specified by t_p does *not* apply to the *entire* D_0; it holds *conditionally* on D_0, *i.e.,* only on $D_0(t_p)$.

The database D_0 satisfies neither ψ_1 nor ψ_2. Even though $t_1[\mathsf{CT}] \asymp t'_p[\mathsf{CT}]$ and t_1 does not violate the FD $\mathsf{CT} \rightarrow \mathsf{AC}$, where t'_p is the second pattern tuple of ψ_1, t_1 violates ψ_1 since $t_1[\mathsf{AC}] \not\asymp t'_p[\mathsf{AC}]$. The tuple t_4 violates ψ_2 since $t_1[\mathsf{AC}] \not\asymp t''_p[\mathsf{AC}]$ although $t_1[\mathsf{CT}] \asymp t''_p[\mathsf{CT}]$, where t''_p is the pattern tuple of ψ_2 (here CT is the Y_p attribute of ψ_2). These tell us that a *single* tuple may violate an eCFD while it takes two tuples to violate a standard FD. □

An instance I of R *satisfies* a set Σ of eCFDs, denoted by $I \models \Sigma$, if $I \models \varphi$ for each $\varphi \in \Sigma$.

Remarks. (1) eCFDs support *inequality* ($\overline{S}$, *e.g.,* the first pattern tuple of ψ_1) and *disjunction* (S, *e.g.,* ψ_2 in which the area code for NYC is specified as *either* 212, 718, 646, 347 *or* 917). (2) Conditional functional dependencies (CFDs) introduced in [1] are a special case of eCFDs. Recall that a CFD is of the form $(R : X \rightarrow Y,\ T_p)$, in which each pattern tuple consists of either '_' or a *single* constant value. Hence, a CFD can be written as an eCFD $\varphi = (R : X \rightarrow Y,\ \emptyset,\ T'_p)$, where T'_p is identical to T_p except that each constant a in T_p is replaced with $\{a\}$ in T'_p. That is, a CFD is an eCFD with neither inequality nor disjunction. Since CFDs extend standard FDs, so do eCFDs.

III. Satisfiability and Implication of *e*CFDs

In this section we investigate the satisfiability and implication analyses of eCFDs. These are classical decision problems associated with any constraint language.

The *satisfiability problem* for eCFDs is to decide, given a set Σ of eCFDs on a relation schema R, whether or not there exists a nonempty instance I of R such that $I \models \Sigma$.

The *implication problem* for eCFDs is to determine, given a set Σ of eCFDs and another eCFD φ defined on the same relation schema R, whether or not $\Sigma \models \varphi$, *i.e.,* whether for every instance I of R, if $I \models \Sigma$ then also $I \models \varphi$.

The main result of this section is that despite the increased expressive power of eCFDs, they retain the same complexity bounds for these static analyses as CFDs.

Satisfiability. As shown in [1], CFDs may not be satisfiable. It is thus not surprising that the same holds for eCFDs.

Example 3.1: Consider an eCFD ψ_3 on cust databases: (cust: [CT] → [CT], ∅, {({NYC} ‖ {NYC} ‖ ∅), ({NYC} ‖ {LI} ‖ ∅)}). This eCFD is not satisfiable. Indeed, for any cust instance I and any tuple t in I, if $t[\mathsf{CT}]$ = NYC, then ψ_3 requires it to be LI; but ψ_3 forces it to be NYC again. □

This highlights the need for the satisfiability analysis of eCFDs: it is necessary to determine whether or not the given eCFDs are not dirty themselves before one uses the eCFDs to detect inconsistencies in a database, which is typically much larger than the set of constraints.

It is known that the satisfiability problem for CFDs is NP-complete [1]. The result below tells us that eCFDs do not make the satisfiability analysis more complicated.

Proposition 3.1: *The satisfiability problem for* eCFD*s is* NP*-complete.*

Proof: The lower bound follows from the NP hardness of the CFD counterpart, since CFDs are a special case of eCFDs. For the upper bound, the satisfiability problem has the following *small model property*: For any given set Σ of eCFDs, if Σ is satisfiable, then there exists a database consisting of a *single* tuple that satisfies Σ. Thus an NP algorithm for checking eCFD satisfiability (i) guesses a database I with a single tuple and (ii) then checks whether $I \models \Sigma$; the latter can be done in PTIME. ■

Implication. Due to the presence of pattern tuples in eCFDs, one expects the number of eCFDs to be larger than their FD counterparts. A natural optimization strategy for cleaning data with eCFDs is by removing redundancies in a given set of eCFDs, *i.e.,* by removing eCFDs and pattern tuples that are entailed by other eCFDs. This calls for the implication analysis of eCFDs.

The implication problem is coNP-complete for CFDs [1]. The complexity remains unchanged for eCFDs:

Proposition 3.2:. *The implication problem for* eCFD*s is co*NP*-complete.*

Proof: The coNP-hardness follows from the lower bound for CFD implication. For the upper bound, the complement of the implication problem has the following *small model property*: for any given set Σ of eCFDs and a single eCFD φ on a schema R, $\Sigma \not\models \varphi$ iff there exists an instance I of R such that $I \models \Sigma$ but $I \not\models \varphi$, and moreover, I consists of at *most two tuples*. Thus an NP algorithm for checking the complement of eCFD implication (i) guesses a database I with two tuples and (ii) checks whether $I \models \Sigma$ and $I \not\models \varphi$. The latter can be done in PTIME. ■

Special case. A tractable special case is identified in [1]: if the given CFDs involve no attributes that have a *finite* domain, then the satisfiability and implication analyses are in PTIME. This is not longer the case for eCFDs, since we can enforce, via eCFDs, an attribute A to draw values from a finite set only, no matter whether $\mathsf{dom}(A)$ is infinite or not.

Proposition 3.3: *The satisfiability and implication problems for* eCFD*s remain* NP*-complete and co*NP*-complete, respectively, in the absence of finite-domain attributes.*

Proof: For the satisfiability problem, the NP-hardness is by reduction from CFD satisfiability with finite-domain attributes. Given any CFDs Σ on schema R, we define another schema R' with only infinite-domain attributes: for each $A \in \mathsf{attr}(R)$ with a finite domain we replace A with A' of an infinite domain; in addition, we define Σ' to be the set consisting of CFDs in Σ and an eCFD $\psi_A = (R' : [A'] \rightarrow \emptyset,\ A',\ \{(\_ \parallel \emptyset \parallel \mathsf{dom}(A))\})$ for each such attribute A. The eCFD ψ_A forces A' to take value only from $\mathsf{dom}(A)$. Clearly, Σ' is satisfiable iff Σ is. Similarly we can show the coNP-hardness of the implication of eCFDs with infinite-domain attributes only. ■

IV. Approximation Algorithm for Satisfiability

In light of the intractability of the eCFD satisfiability analysis, it is beyond reach to find an efficient algorithm that, given a set Σ of eCFDs, returns true iff Σ is satisfiable.

This motivates us to consider the *maximum satisfiable subset* problem (MAXSS): given a set Σ of eCFDs, it is to find a maximum subset of Σ that is satisfiable. Although this problem is also intractable, we develop an approximation factor preserving reduction to a well-studied NP-complete problem, called Maximum Generalized Satisfiability (MAXGSAT), for which a number of approximation algorithms are already in place (see, *e.g.,* [7]). The MAXGSAT problem is, given a set $\Phi = \{\phi_1, \ldots, \phi_n\}$ of Boolean expressions, to find a truth assignment that satisfies the maximum number of expressions in Φ.

Given this reduction and an ϵ-approximation algorithm for MAXGSAT, we obtain an ϵ-approximation algorithm for MAXSS that, given Σ, finds a satisfiable subset Σ_m of Σ such that $\mathsf{card}(\Sigma_m) \geq (1-\epsilon) \cdot \mathsf{card}(\mathsf{OPT}_{maxss}(\Sigma))$, where $\mathsf{OPT}_{maxss}(\Sigma)$ denotes a maximum satisfiable subset of Σ, and $\mathsf{card}(S)$ denotes the cardinality of S.

Clearly, if $|\Sigma_m| = |\Sigma|$ then we may conclude that Σ is satisfiable. Moreover, if $|\Sigma_m| < (1-\epsilon)|\Sigma|$ then Σ is unsatisfiable. Therefore, only when $|\Sigma| > |\Sigma_m| \geq (1-\epsilon)|\Sigma|$ we cannot conclude whether Σ is satisfiable or not.

Approximation factor preserving reduction. An *approximation factor preserving reduction* from MAXSS to MAXGSAT consists of two PTIME functions f and g such that for any set Σ of eCFDs,

1) $f(\Sigma)$ is an MAXGSAT instance Φ_Σ, and $g(\Phi_m)$ is a satisfiable set in Σ if Φ_m is a set of satisfied formulas in Φ_Σ;
2) $\mathsf{card}(\mathsf{OPT}_{maxgsat}(f(\Sigma))) \geq \mathsf{card}(\mathsf{OPT}_{maxss}(\Sigma))$;
3) $\mathsf{card}(g(\Phi_m)) \geq \mathsf{card}(\Phi_m)$, where $\mathsf{OPT}_{maxgsat}(f(\Sigma))$ is the maximum set of satisfied expressions in Φ_Σ. Thus g is guaranteed to return a feasible MAXSS solution for Σ.

Such a reduction ensures that MAXSS has an ϵ-factor approximation algorithm if MAXGSAT has one. Indeed, if $\mathsf{card}(\Phi_m) \geq (1-\epsilon) \cdot \mathsf{card}(\mathsf{OPT}_{maxgsat}(f(\Sigma)))$, then from (2) and (3) above, it follows that $\mathsf{card}(g(\Phi_m)) \geq (1-\epsilon) \cdot \mathsf{card}(\mathsf{OPT}_{maxss}(\Sigma))$. In particular, one can verify that if Φ_m is the optimal solution of MAXGSAT for $f(\Sigma)$, then $g(\Phi_m)$ is the optimal solution of MAXSS for Σ.

Reduction. We give a reduction by leveraging the small model property shown in the proof of Proposition 3.1. We define an R-tuple template t such that Σ is satisfiable iff there is a valuation ρ for variables in t with $\{\rho(t)\} \models \Sigma$.

We first introduce variables used in the reduction. Consider a set Σ of eCFDs defined on a relation schema R, where $\mathsf{attr}(R) = \{A_1, \ldots, A_n\}$. For each $i \in [1, n]$, we define the active domain $\mathsf{adom}(A_i)$ of A_i to be the set consisting of (1) all the constants appearing in $t_p[A_i]$ for some pattern tuple t_p in Σ; (2) a value in $\mathsf{dom}(A_i)$ that is not yet in $\mathsf{dom}(A_i)$ if there exists any (if $\mathsf{dom}(A_i)$ is a finite domain, there may not exist such a value). Let k be the size of Σ. Then $\mathsf{adom}(A_i)$ has at most $k+1$ values. For each $i \in [1, n]$ and $a \in \mathsf{adom}(A_i)$, we introduce a Boolean variable $x(i, a)$ such that $x(i, a) = \mathsf{true}$ iff $t[A_i] = a$. We use the Boolean expression below to ensure that $t[A_i]$ has a unique value:

$$\varphi_i = \bigvee_{a \in \mathsf{adom}(A_i)} x(i,a) \ \wedge \bigwedge_{a,b \in \mathsf{adom}(A_i) \wedge b \neq a} ((x(i,a) \rightarrow \overline{x(i,b)})$$

Let Φ_R be the conjunction of all φ_i for $i \in [1, n]$. Then Φ_R guarantees that t is characterized by these variables.

We define the reduction function f to characterize eCFDs. For each $\psi = (R : X \rightarrow Y,\ Y_p,\ T_p) \in \Sigma$ and each $t_p \in T_p$, we define a Boolean expression $\xi(\psi, t_p)$:

$$\bigvee_{B \in X} t[B] \not\asymp t_p[B] \vee (\bigwedge_{A \in Y} t[A] \asymp t_p[A] \wedge \bigwedge_{A \in Y_p} t[A] \asymp t_p[A])$$

where $t[B] \asymp t_p[B]$ can be written as

1) the disjunction of $x(i, a)$ for all $a \in S$ if $t_p[B] = S$;
2) the conjunction of $x(i, a)$ all $a \in S$ if $t_p[B] = \overline{S}$; and
3) true if $t_p[B]$ = '_'. Similarly $t[B] \not\asymp t_p[B]$ can be expressed as a Boolean expression with variables given above. We define $f(\Sigma) = \Phi_\Sigma = \{\xi(\psi, t_p) \wedge \Phi_R \mid \psi \in \Sigma, t_p \in \psi\}$, with $|\Phi_\Sigma|$ in $O(k)$.

Finally we define the reduction function g. For a truth assignment ρ for Φ_Σ, let Φ_m be the set of expressions in Φ_Σ satisfied by ρ. We instantiate t based on ρ as follows: $t[A_i] = a$ if and only if $\rho(x(i, a)) = \mathsf{true}$. Then $g(\Phi_m)$ is defined to be the set of eCFDs satisfied by t. It is easy to verify that $\mathsf{card}(\Phi_m) = \mathsf{card}(g(\Phi_m))$.

Proposition 4.1: *The reduction given above is approximation factor preserving from* MAXSS *to* MAXGSAT.

Proof: First, functions f and g can be computed in PTIME in k and n. Second, $\mathsf{card}(\mathsf{OPT}_{maxgsat}(f(\Sigma))) = \mathsf{card}(\mathsf{OPT}_{maxss}(\Sigma))$. Third, for any truth assignment ρ for Φ_Σ, if Φ_m is the set of formulas in Φ_Σ satisfied by ρ, then $\mathsf{card}(\Phi_m) = \mathsf{card}(g(\Phi_m))$. Taken together, the reduction is indeed approximation factor preserving. ■

V. Detecting *e*CFD Violations

In this section, we develop techniques for detecting violations in a database D *w.r.t.* a given set Σ of eCFDs. We consider *static* and *dynamic* settings, stated as follows:

Given a database D and a set Σ of eCFDs, a *batch detection algorithm* is to find the *violation set* $\mathsf{vio}(D)$ *w.r.t.* Σ, *i.e.,* the set of all tuples in D that violate some eCFDs in Σ.

Given D, Σ, the violation set $\mathsf{vio}(D)$ of D *w.r.t.* Σ, and updates ΔD to the database D, an *incremental detection algorithm* is to find the set $\Delta\mathsf{vio}(D)$ such that $\Delta\mathsf{vio}(D) \oplus \mathsf{vio}(D)$ is the violation set $\mathsf{vio}(\Delta D \oplus D)$ *w.r.t.* Σ, where $\Delta S \oplus S$ denotes applying the updates ΔS to the set S. Here the updates ΔD can be either a set of tuple *insertions* or *deletions*, denoted by ΔD^+ and ΔD^-, respectively.

In Sections V-A and V-B, we develop a batch and an incremental detection algorithm, referred to as BatchDetect and IncDetect, respectively. Both algorithms only generate SQL queries to detect violations. This is important since eCFD violation detection can then be directly implemented on top of RDBMS, and we can therefore benefit from existing optimization techniques of RDBMS. Better still, in both settings, only a *fixed* number of SQL queries are needed, no matter how many eCFDs are in Σ, how many pattern tuples are in the eCFDs, and how large the sets are in each pattern-tuple attribute. The key idea is to treat pattern tableaux in Σ as *data tables*, rather than as *meta-data*.

Before we present our detection algorithms we decide on a uniform way of representing the set of violations. Instead of simply returning the tuples in D that violate some eCFD in Σ, we *explicitly store* whether a tuple in D is a violation or not. More precisely, we extend the schema R of D with two Boolean attributes: SV (for "Single tuple Violation") and MV (for "Multiple tuple Violation"). That is, $t.\mathsf{SV} = 1$ if t violates an eCFD in Σ all by itself; and $t.\mathsf{SV} = 0$ otherwise. Similarly, $t.\mathsf{MV} = 1$ if t violates an embedded FD for an eCFD in Σ; and $t.\mathsf{MV} = 0$ otherwise. Hence, $t \in \mathsf{vio}(D)$ if either $t.\mathsf{SV} = 1$ or $t.\mathsf{MV} = 1$.

A. A Batch Algorithm

We first consider the static case and outline algorithm BatchDetect. Here we extend the approach proposed in [1] and generate a pair of SQL queries for violation detection and corresponding update statements:

1) Query Q_{sv} finds all single-tuple violations due to violations of the pattern constraints enforced by eCFDs in Σ;
2) Query Q_{mv} identifies multiple-tuple violations caused by a violation of an FD embedded in some eCFD in Σ.
3) Given these two queries, BatchDetect performs update statements to D and sets the SV (resp. MV) attribute to 1 for those tuples returned by Q_s (resp. Q_v).

Encoding of eCFDs. To achieve this, we encode Σ with several auxiliary relations. This encoding is also used in the incremental detection algorithm and therefore we explain it in detail. We start by encoding the attribute structure of the eCFDs with a relation enc, and the patterns in the eCFDs in terms of separate relations, as will be explained below. The relation enc is a relation of arity $2|\mathsf{attr}(R)| + 1$ consisting of (i) an attribute CID that stores an identifier of the eCFDs in Σ; and (ii) attributes A_L (for "left") and A_R (for "right") for each attribute in $A \in \mathsf{attr}(R)$. We may assume that the eCFDs in Σ all contain a single pattern tuple only. Indeed, we can always split an eCFD with multiple patterns into a set of eCFDs with only a single pattern tuple. Consider a single eCFD $\varphi = (R : X \to Y, T_p = \{t_p\})$ in Σ. We now encode whether an attribute is part of X, Y or Y_p and whether the pattern tuple for that attribute is of the form S, $\overline{S}$ or '_', using integers in $\{-3, -2, -1, 0, 1, 2, 3\}$ as follows: we add a tuple t_φ to enc such that $t_\varphi[\mathsf{CID}]$ is an identifier for φ; the other attributes of t_φ are defined as follows: $t_\varphi[A_L] = 0$ (resp. $t_\varphi[A_R] = 0$) in case that A does not appear on the $\mathsf{LHS}(\varphi)$ (resp. $\mathsf{RHS}(\varphi)$); $t_\varphi[A_L] = 1$ (resp. $t_\varphi[A_R] = 1$) in case that A does appear in X (resp. Y) and moreover $t_p[A] = S$. Similarly, $t_\varphi[A_L] = 2$ (resp. $t_\varphi[A_R] = 2$) in case that A does appear in X (resp. Y) and $t_p[A] = \overline{S}$. Moreover, $t_\varphi[A_L] = 3$ (resp. $t_\varphi[A_R] = 3$) in case that A does appear in X (resp. Y) and $t_p[A] =$'_'. Finally, for attributes A_R that appear in Y_p, we use a similar encoding but use the negative integers -1, -2, and -3 instead. We illustrate the relation enc encoding of ψ_1 and ψ_2 of Example 2.1 in Fig. 3 (top). We only show some attributes relevant for these eCFDs. The tuples have zeroes for all other attributes. Note that enc encodes *all* eCFDs in Σ uniformly in the single relation, one tuple for each pattern tuple in Σ.

enc

CID	CT_L	AC_L	CT_R	AC_R
1	2	0	0	3
2	1	0	0	2
3	1	0	0	-2

T_{CT_L}

CID	CT_L
1	NYC
1	LI
2	Albany
2	Troy
2	Colony
3	NYC

T_{AC_R}

CID	AC_R
2	518
3	212
3	718
3	646
3	347
3	917

Fig. 3. Encoding of eCFDs.

We also need to store the tuples in the pattern tableaux of the eCFDs in Σ. For this, we create $2|\mathsf{attr}(R)|$ binary relations as follows: For each attribute $A \in \mathsf{attr}(R)$, we define T_{A_L} (resp. T_{A_R}) as the relation consisting pairs (cid, a) where cid is an identifier of an eCFD in Σ and $a \in S$ or $a \in \overline{S}$, where S is such that $t_p[A_L] = S$ or $t_p[A_L] = \overline{S}$ (resp. $t_p[A_R] = S$ or $t_p[A_R] = \overline{S}$). Figure 3 (bottom) shows these relations for the eCFDs of Example 2.1.

Algorithm BatchDetect. The following SQL queries are employed by BatchDetect.

(1) We first detect single-tuple violations that are caused by

pattern constraints, *i.e.,* tuples in D that satisfy the pattern constraints of the LHS of an eCFD in Σ but do not satisfy those of its RHS. The encoding is similar to the one for CFDs presented in [1], by literally expressing pattern-constraint violation in SQL. In contrast to CFDs, patterns are now sets (or the complement thereof). For this, we need to express the fact that an element is in (resp. not in) a set by means of EXISTS (resp. NOT EXISTS). Figure 4 (top) shows the query Q_{sv} for the example eCFDs given in Fig. 2 In the queries in Fig. 4, μ stands for a mapping defined by $\mu(A_L) = A$ and $\mu(A_R) = A$ for all $A \in \mathsf{attr}(R)$. We omit parts of the queries addressing irrelevant attributes.

(2) We next detect the multiple-tuple violations that are caused by violations of the embedded FDs in the eCFDs in Σ. Similar to [1], detection of such violations can be readily expressed using GROUP BY in SQL. However, we have to group by different attributes depending on the eCFD under consideration. This can be achieved by blanking out (using a constant "@" not appearing in any database) those attributes that are not relevant. Attributes irrelevant to the embedded FD have non-positive entries in the relation enc. We use the CASE construct in the SELECT statement to replace the attributes values of tuples in D by '@' if the attribute is irrelevant to the embedded FD; otherwise we return the attribute value of the tuple instead. We provide an example query Q_{mv} in Fig. 4 (bottom). Note that Q_{mv} returns tuples of the form (cid, p), where cid is an identifier for an eCFD (as given by enc) and p is a tuple consisting of constant values and "@"s. Intuitively, if a tuple $t \in D$ matches p for some $(cid, p) \in Q_{mv}(D)$ then it violates the embedded FD of the eCFD identified by cid.

(3) We set the SV attribute to "1" for tuples returned by Q_{sv}. For the MV-attribute, note that a tuple t in D is involved in a multiple tuple violation iff there exists a $(cid, p) \in Q_{mv}(D)$ such that t matches p. An additional SQL query identifies these tuples and updates their MV-attribute to "1".

Putting these together, given schema R and set Σ of eCFDs defined on R, algorithm BATCHDETECT generates SQL queries and update statements for detecting pattern-constraint violations and embedded FD violations, respectively, by capitalizing on the encoding given above.

Remarks. (1) The schema of the encoding relations, namely, enc and the binary relations T_A, is determined by the schema R rather than Σ. (2) The entire encoding relations are *linear* in the size of the input eCFDs Σ. (3) The detection SQL queries conduct two passes of the database D, regardless of the number of eCFDs and the size of pattern tuples in Σ. That is, they have *the same data complexity as* detection queries for CFDs [1]. Note that these queries necessarily use EXISTS and NOT EXISTS; but these operations are only applied to auxiliary relations that encode the sets of constants mentioned in the eCFD patterns, rather than to the underlying database. Indeed, for each data tuple Q_{sv} conducts a linear scan of Σ, *the same as* its CFD counterpart; similarly for Q_{mv}. It is also worth remarking that the coding of eCFDs for algorithm BATCHDETECT is more involved than that of [1],

Q_{sv} =(SELECT t.AC, t.PN, t.NM, t.STR, t.CT, t.ZIP
FROM cust t, enc c
WHERE ($c.\mathsf{CT}_L \neq 1$ OR ((EXISTS Q^{CT_L}) AND $c.\mathsf{CT}_L = 1$))
AND ($c.\mathsf{CT}_L \neq 2$ OR ((NOT EXISTS Q^{CT_L}) AND $c.\mathsf{CT}_L = 2$))
AND (((NOT EXISTS Q^{AC_R}) AND ABS($c.\mathsf{AC}_R$) = 1)
OR ((EXISTS Q^{AC_R}) AND ABS($c.\mathsf{AC}_R$) = 2)))
where for any attribute A, Q^A stands for
(SELECT $T_A.A$ FROM T_A
WHERE $c.\mathsf{CID} = T_A.\mathsf{CID}$ AND $t.\mu(A) = T_A.A$).

Q_{mv} =(SELECT $m.\mathsf{CID}, m.\mathsf{CT}_L$, COUNT(*) FROM macro m
GROUP BY $m.\mathsf{CID}, m.\mathsf{CT}_L$
HAVING COUNT(*) > 1)
where macro stands for:
(SELECT DISTINCT $c.\mathsf{CID}$, (CASE $c.\mathsf{CT}_L$ WHEN > 0 THEN
t.CT ELSE @ END) AS CT_L ···
(CASE $c.\mathsf{AC}_R$ WHEN > 0 THEN
t.AC ELSE @ END) AS AC_R
FROM cust t, enc c
WHERE ($c.\mathsf{CT}_L \neq 1$ OR ((EXISTS Q^{CT_L}) AND $c.\mathsf{CT}_L = 1$))
AND ($c.\mathsf{CT}_L \neq 2$ OR ((NOT EXISTS Q^{CT_L}) AND $c.\mathsf{CT}_L = 2$)))

Fig. 4. Batch detection of violations.

in order to cope with the *set* elements in pattern tuples. A direct extension of the technique of [1] may lead to excessive space overhead, as opposed to the linear space taken by BATCHDETECT.

B. An Incremental Algorithm

We next present incremental algorithm INCDETECT in response to database updates ΔD. Of course, BATCHDETECT can be directly applied to the new database obtained by updating the database D with ΔD. We want to *incrementally* detect violations because the deletion or insertion of a small number of tuples only affects a small part of D and as a result, one only needs to identify violations in the affected part rather than inspect the *entire* database. Algorithm INCDETECT aims to minimize unnecessary recomputation conducted for finding violations.

Like BATCHDETECT, Algorithm INCDETECT also generates SQL queries to identify changes to the violations of pattern constraints and changes to the violations of embedded FDs in Σ. In addition, it maintains an auxiliary relation in order to reuse previous computations. Observe that tuple deletions ΔD^- may remove violations from D but do not introduce new violations; on the other hand, tuples insertions ΔD^+ may add new violations introduced by inserted tuples alone or together with tuples in D.

Auxiliary relation. We maintain an auxiliary relation $\mathsf{Aux}(D)$, initialized by storing the query result of Q_{mv} from BATCHDETECT on D. Recall that $\mathsf{Aux}(D)$ consists of tuples of the form (cid, p) where cid is an eCFD identifier and p is a tuple consisting of constants and "@". As noted above, each (cid, p) corresponds to the set of tuples that are involved in a multiple-tuple violation of the eCFD identified by cid and that match p. We next describe how $\mathsf{Aux}(D)$ is maintained during updates on D and how it can be used to incrementally compute the updated set of violations.

Algorithm IncDetect. Due to space limitations, we only provide a high-level description of IncDetect. Initially, we are given (i) D in which the SV and MV attributes correctly indicate the violations of Σ (this can be obtained by running algorithm BatchDetect); (ii) the set of updates ΔD; and (iii) the auxiliary relation $\mathsf{Aux}(D)$ (initialized as described above).

Algorithm IncDetect needs to perform several tasks: it needs to compute $D \oplus \Delta D$, correctly update the SV and MV attributes for the tuples in $D \oplus \Delta D$, and update $\mathsf{Aux}(D)$ to $\mathsf{Aux}(D \oplus \Delta D)$. Moreover, IncDetect performs these tasks using SQL statements only. Since deletions and insertions are dealt with in different ways, we treat them separately. We first consider the case of deletions.

Tuple deletions. Let ΔD^- be the set of tuples that are to be deleted from D. We first explain how $\mathsf{Aux}(D)$ is updated and then show how it is used to correctly update the multiple violation attribute MV in D. Because deletions do not eliminate single tuple violations (except for those that are in ΔD^-), we do not need to update the SV attribute.

(1) To update $\mathsf{Aux}(D)$, observe the following: a tuple (cid, p) can be removed from $\mathsf{Aux}(D)$ if it either does not match any tuple in $D \oplus \Delta D^-$, or all matching tuples in $D \oplus \Delta D^-$ do not violate the embedded FD of the eCFD identified by cid. It suffices to only consider (cid, p)'s that are *potentially* affected by the update ΔD^-, *i.e.,* those (cid, p)'s that match a tuple in ΔD^-, and thus avoid unnecessary computations After removing these (cid, p)'s from $\mathsf{Aux}(D)$, we obtain the updated $\mathsf{Aux}(D \oplus D^-)$.

(2) In order to update the MV attribute, we first observe that it is sufficient to only consider tuples t in D with $t.\mathsf{MV} = 1$. For each such t, we check whether it does not match any p in $(cid, p) \in \mathsf{Aux}(D \oplus \Delta D^-)$, and if so, update $t.\mathsf{MV}$ to 0.

Tuple insertions. Let $\Delta\mathsf{D}^+$ be the set of tuples to be inserted into D. We perform the following steps:

(1) We first detect the single-tuple violations in ΔD^+. That is, we apply Q_{sv} of BatchDetect on ΔD^+ and update the SV-attribute in ΔD^+ accordingly.

(2) Next, we identify new multiple-tuple violations in $D \oplus \Delta D^+$ by performing the following steps:

(2.a) Update the MV attribute of tuples in ΔD^+ that violate an eCFD together with a tuple in D. These can be easily found by matching tuples in $\mathsf{Aux}(D)$ with tuples in ΔD^+.

(2.b) Update $\mathsf{Aux}(D)$. Denote by D_{clean} the set of tuples in D satisfying Σ, which can be easily identified using the MV attribute. We insert tuples (cid, p) into $\mathsf{Aux}(D)$ that correspond to violations between (previously clean) tuples in D and tuples in ΔD^+.

(2.c) We then update the MV attribute for tuples in $D_{\mathsf{clean}} \oplus \Delta D^+$ that match some tuple (cid, p) in $\mathsf{Aux}(D)$.

(2.d) To account for multiple-tuple violations caused by tuples in ΔD^+ alone, we have to update $\mathsf{Aux}(D)$ again. For this, we run Q_{mv} on ΔD^+ and insert the result tuples into $\mathsf{Aux}(D)$. After this step, $\mathsf{Aux}(D)$ becomes $\mathsf{Aux}(D \oplus \Delta D^+)$.

(2.e) Finally, we add ΔD^+ to D and update the MV-attribute of tuples in ΔD^+ that match a (cid, p) tuple in $\mathsf{Aux}(D \oplus \Delta D^+)$.

It is easily verified that the above steps correctly maintain both the auxiliary relation and violation set for both tuple deletions and insertions. Moreover, they can all be performed using SQL statements.

Remarks. (1) Algorithm IncDetect uniformly employs an auxiliary relation and SQL queries to handle *multiple* tuple deletions and insertions, for the *entire* set Σ of eCFDs. This is the first SQL-based technique for incrementally detecting violations of *multiple* eCFDs. (2) Recomputation is avoided by only considering relevant tuples in D using both the auxiliary relation and the update set.

VI. Experimental Study

Our experimental study focuses on the SQL-based algorithms BatchDetect and IncDetect for detecting data inconsistencies. We evaluate (1) the scalability of BatchDetect and IncDetect *w.r.t.* the size of databases, the complexity of eCFDs and the error rate in the databases, and (2) the performance of IncDetect versus BatchDetect in response to database updates.

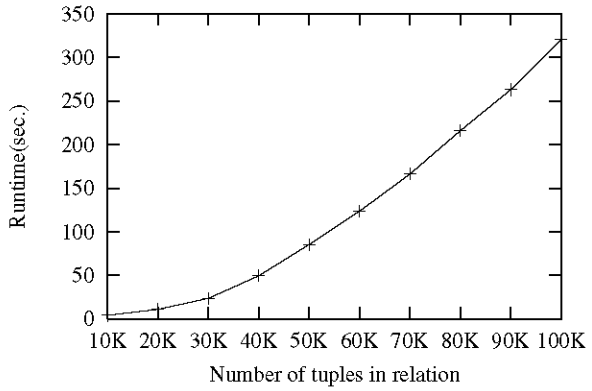
Experimental setting. Our experiments are based on an extension of the cust relation shown in Fig. 1, that adds information about items bought by different customers. We scraped real-life $\mathsf{CT}, \mathsf{AC}, \mathsf{ZIP}$ data for cities and towns in the US and different items, such as books, CDs and DVDs, from online stores. Using this, we wrote a program to generate synthetic datasets, denoted by D. We considered two parameters of the datasets D: $|\mathsf{D}|$ for the number of tuples in D, ranging from 10k to 100k, and noise% for the percentage of tuples in D that were modified to violate an eCFD, ranging from 0% to 9%. The modification consists of changing tuples in D in attributes in the right-hand side of some eCFDs from a correct to an incorrect value.

We used a set Σ consisting of 10 eCFDs to express real-life semantics of the real-life data, including the two eCFDs of Fig. 2. We measured the complexity of the eCFDs in terms of the $|\mathsf{Tp}|$, *i.e.,* the number of tuples in the pattern tableaux T_p, ranging from 10 to 500 pattern tuples. Note that each tuple itself is a constraint. The number of wildcards ('_'), positive domain constraints (S) and negative domain constraints ($\overline{S}$) in the pattern tuples are uniformly distributed.

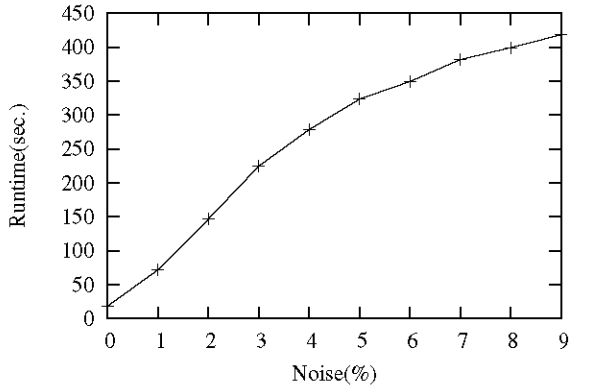
Our experiments were conducted on an Apple Xserve with 2.3GHz PowerPC dual CPU and 4GB of memory, and with a commercial DBMS installed. Each experiment was run five times and the average is reported here.

Experiment 1: Scalability. In the first set of experiments we evaluated the scalability of BatchDetect.

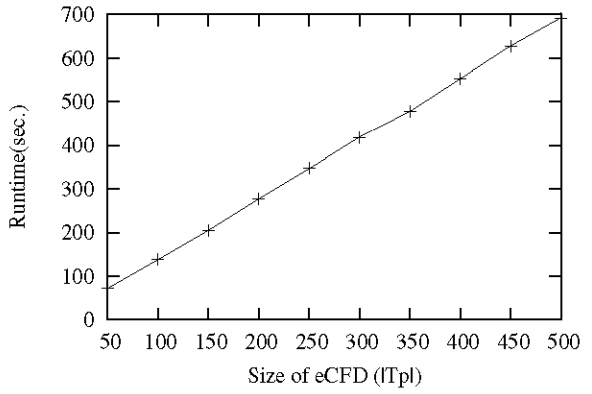
We first set $|\mathsf{Tp}| = 10$ and investigated the effect of varying $|\mathsf{D}|$ and noise% on the performance of BatchDetect. Fixing



(a) Scalability in |D|

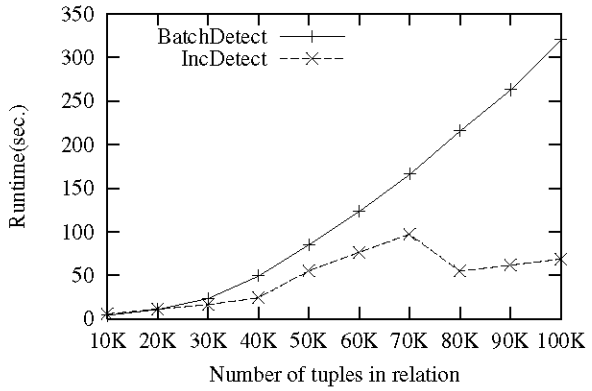


(b) Scalability in noise

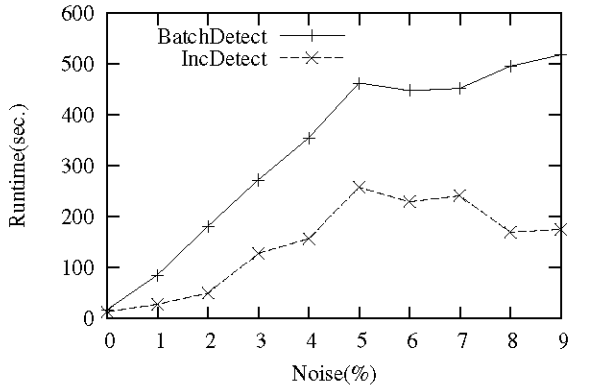


(c) Scalability in number of constraints

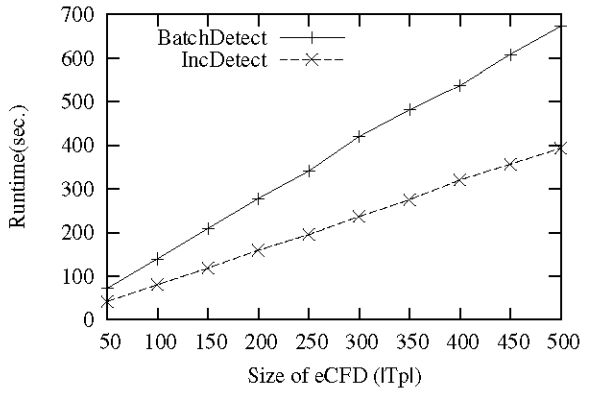
Fig. 5. BATCHDETECT



(a) Scalability in |D|



(b) Scalability in noise



(c) Scalability in number of constraints

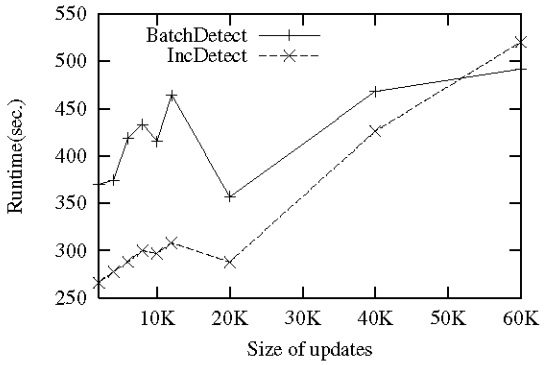
Fig. 6. BATCHDETECT vs INCDETECT

noise% = 5%, we varied |D| from 10k to 100k in 10k increments. Moreover, fixing |D| = 100k, we varied noise% from 0% to 9% in 1% increments. The results are presented in Figs. 5(a) and 5(b). As expected, BATCHDETECT scales well *w.r.t.* the size of the datasets and the error rate.

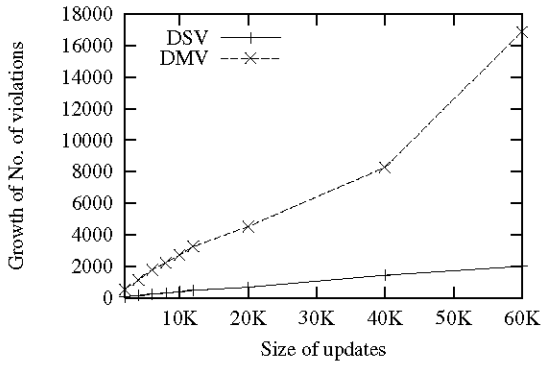
We then set |D| = 100k and noise% = 5%, and studied the impact of varying the complexity of eCFDs of Σ on the cost of BATCHDETECT. We selected an eCFD from Σ and varied its |Tp| from 50 to 500 in 50 increments. As shown in Fig. 5(c) BATCHDETECT scales linearly in |Tp|.

Experiment 2: Incremental vs. Batch. In the second set of experiments we compared the cost of incremental detection vs. its batch counterpart in response to database updates. We use $|\Delta D^+|$ and $|\Delta D^-|$ to indicate the number of tuples to be inserted into and deleted from D, respectively. We always ensure that ΔD^+ and ΔD^- do not overlap. As opposed to the first set of experiments, here BATCHDETECT was applied to the data after database updates are executed.

First, we fixed $|\Delta D^+|$ = 10k and $|\Delta D^-|$ = 10k and repeated the same set of experiment sets as above, *i.e.,* we investigated



(a) Scalability in size of updates



(b) Effect on number of violation changes.

Fig. 7. Effect of updates

the scalability of INCDETECT in response to the size of datasets, the error rate and the complexity of eCFDs. The results are shown in Figs. 6(a), 6(b), and 6(c). In each figure we show the running time of INCDETECT and BATCHDETECT, in response to both tuple insertions and deletions. The results tell us that INCDETECT scales well *w.r.t.* $|D|$, noise% and $|T_p|$ and more importantly, performs better than BATCHDETECT. We note that the running time, reported in Figs. 6(a) and 6(b), not only depend on the size of the updates but also on which tuples are part of the update. This explains why the curves show some irregular behavior, although we averaged over different updates.

Second, we fixed $|D| = 100K$, noise% = 5%, $|T_p| = 10$ and varied $|\Delta D^+|$ and $|\Delta D^-|$ from 2k to 12k in 2k increments and from 20k to 60k in 20k increments. Note that $|D|$ is indeed fixed, since we delete and insert the same number of tuples. We compared the cost of INCDETECT vs. the cost of BATCHDETECT in response to the number of updates. As shown in Fig. 7(a), INCDETECT outperforms BATCHDETECT when the size of the updates is relatively small, and is slightly better for larger ones. However, as expected, BATCHDETECT outperforms INCDETECT for very large updates. Indeed, in our experiments this happens when around 50% of the data is updated. Overall, we may conclude that INCDETECT works extremely well and scales up in a similar way as BATCHDETECT.

Finally, in Fig. 7(b), we report the growth of the number of single (resp. multiple) tuple violations, denoted by DSV (resp. DMV), in the database before and after updates, for an increasing number of updates. On our datasets, we observed that the number of single-tuple violations grows linearly in the number of updates. However, the number of multiple-tuple violations increases dramatically for large updates. This also explains why BATCHDETECT performs better for large updates (see Fig. 7(a)). Indeed, maintaining the auxiliary information by INCDETECT incurs a large overhead when the number of violations changes too much.

Summary. We may conclude the following from our experimental evaluation. (1) BATCHDETECT and INCDETECT scale well *w.r.t.* when the dataset size, the error rate and the complexity of eCFDs increase. (2) INCDETECT significantly outperforms BATCHDETECT in response to both tuple insertions and deletions, for reasonably-sized updates. (3) BATCHDETECT performs better than INCDETECT when more than 50% of the data is updated.

VII. RELATED WORK

Conditional functional dependencies (CFDs) were proposed in [1] for data cleaning. In [1], the intractability of the satisfiability and implication problems for CFDs is proved, and a batch algorithm for detecting inconsistencies based on CFDs and SQL is developed. In [8], approximation of the satisfiability analysis and incremental inconsistency detection are addressed for CFDs. This work differs from [1], [8] in that we study eCFDs, a class of dependencies more expressive than CFDs, and show that the complexity bounds for CFDs carry over to these dependencies. In addition, we develop effective techniques to tackle detection issues for eCFDs; in particular, incremental detection methods were not considered in [1], [8] for multiple CFDs. Database repairing techniques were developed in [9] based on CFDs, which we do not consider in this paper.

As observed in [1], previous work on constraint-based data cleaning has mostly focused on two topics, both introduced in [2]: *repairing* is to find another database that is consistent and minimally differs from the original database (*e.g.,* [3], [4], [5]); and *consistent query answering* is to find an answer to a given query in every repair of the original database (*e.g.,* [2], [6]). A variety of constraint formalisms have been used in data cleaning, ranging from standard FDs and inclusion dependencies [2], [3], [4], denial constraints [10] to logic programs (see [11] for a survey). To our knowledge, except [1], [9] no prior work on data cleaning has studied pattern tableaux as those embedded in eCFDs.

Closer to eCFDs are dependencies of [12], [13], [14], [15] developed for constraint databases. Constraints of [13], also referred to as 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 [14]

extend [13] by allowing $\xi \rightarrow (Z \rightarrow W)$, where ξ is an arbitrary constraint that is not necessarily an FD. In a nutshell, these dependencies are to apply FD $Z \rightarrow W$ only to the subset of a relation that satisfies $X \rightarrow Y$ or ξ. They cannot express even CFDs since $Z \rightarrow W$ does not allow patterns with constants as found in CFDs and eCFDs. More expressive are constraint-generating dependencies (CGDs) of [12] and constrained tuple-generating dependencies (CTGDs) of [15], of the form $\forall \bar{x}(R_1(\bar{x}) \wedge \ldots \wedge R_k(\bar{x}) \wedge \xi(\bar{x}) \rightarrow \xi'(\bar{x}))$ and $\forall \bar{x}(R_1(\bar{x}) \wedge \ldots R_k(\bar{x}) \wedge \xi \rightarrow \exists \bar{y}(R'_1(\bar{x}, \bar{y}) \wedge \ldots \wedge R'_s(\bar{x}, \bar{y}) \wedge \xi'(\bar{x}, \bar{y}))$, respectively, where R_i, R'_j are relation symbols, and ξ, ξ' are arbitrary constraints. While both CGDs and CTGDs can express CFDs and eCFDs, little is known about the complexity of their satisfiability and implication analyses, or effective algorithms for checking these dependencies. Indeed, for CGDs, the complexity of these analyses is an open issue in the presence of constants *or* finite-domain attributes, even when ξ and ξ' are $(=, \neq)$ constraints; for CTGDs the satisfiability and implication problems are already undecidable even in the absence of ξ, ξ' and constants. That is, the expressive power of these dependencies comes with the price of high complexity. Furthermore, none of the prior results applies to CFDs or eCFDs. We are not aware of any applications of these constraints in data cleaning.

Codd tables, variable tables and conditional tables have been studied for incomplete information [16], [17], which also allow both variables and constants in the specifications. As clarified in [1], these formalisms differ from eCFDs and CFDs in that each of these tables is used as a representation of possibly infinitely many relation instances, one instance for each instantiation of variables in the table. No instance represented by these table formalisms can include two tuples that result from different instantiations of a table tuple. In contrast, all pattern tuples in a pattern tableau of an eCFD or CFD constrain a *single* relation instance, which can contain any number of tuples that are all instantiations of unnamed variables in the same pattern tuple.

The satisfiability problem is not an issue for standard FDs: one can specify arbitrary FDs without worrying about their satisfiability. There has been work on heuristic algorithms for the satisfiability analysis of first-order logic constraints (see, *e.g.,* [18], [19]), but attributes with finite domains were not considered there, and those algorithms do not yield an effective method for eCFD satisfiability checking.

Incremental methods have been studied for *checking* constraints (see, *e.g.,* [20] for a survey). However, we are not aware of any previous work on incrementally checking multiple constraints via a fixed number of SQL queries.

VIII. Conclusions

We have proposed eCFDs, an extension of CFDs that can capture more errors in real-life data than CFDs. We have shown that the satisfiability and implication analyses of eCFDs have the same complexity bounds as their CFD counterparts. We have also revised the detection technique for CFD violations [1] such that eCFD violations can also be handled efficiently using SQL. Thus, *despite the increased expressive power*, eCFDs incur *no extra complexity* in static analyses or inconsistency detection. Moreover, we have developed an incremental SQL-based technique for eCFD violation detection, as well as an approximation algorithm for the satisfiability analysis of eCFDs, the most important decision problem for eCFDs. Our experimental results show that our batch and incremental detection methods are effective.

One topic for future work is to develop algorithms for eliminating eCFD violations and *repairing* data. Another practical topic is to find effective methods for automatically discovering eCFDs from data samples; we defer the full treatment of eCFD discovery to another publication.

Acknowledgment

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

References

[1] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis, "Conditional functional dependencies for data cleaning," in *ICDE*, 2007.
[2] M. Arenas, L. E. Bertossi, and J. Chomicki, "Consistent query answers in inconsistent databases," in *PODS*, 1999.
[3] 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.
[4] J. Chomicki and J. Marcinkowski, ""Minimal-Change Integrity Maintenance Using Tuple Deletions"," *Information and Computation*, vol. 197, no. 1-2, pp. 90–121, 2004.
[5] E. Franconi, A. L. Palma, N. Leone, S. Perri, and F. Scarcello, "Census data repair: a challenging application of disjunctive logic programming," in *LPAR*, 2001.
[6] J. Wijsen, "Condensed representation of database repairs for consistent query answering," in *ICDT*, 2003.
[7] C. H. Papadimitriou, *Computational Complexity*. Addison Wesley, 1994.
[8] Anonymous, "Conditional functional dependencies for capturing data inconsistencies," *Submitted for journal publication*, 2007.
[9] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma, "Improving data quality: Consistency and accuracy," in *VLDB*, 2007.
[10] A. Lopatenko and L. Bertossi, "Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics," in *ICDT*, 2007.
[11] L. Bertossi, "Consistent query answering in databases," *SIGMOD Rec.*, vol. 35, no. 2, pp. 68–76, 2006.
[12] M. Baudinet, J. Chomicki, and P. Wolper, "Constraint-Generating Dependencies," *JCSS*, vol. 59, no. 1, pp. 94–115, 1999.
[13] P. D. Bra and J. Paredaens, "Conditional dependencies for horizontal decompositions," in *Colloquium on Automata, Languages and Programming*, 1983.
[14] M. J. Maher, "Constrained dependencies," *Theoretical Computer Science*, vol. 173, no. 1, pp. 113–149, 1997.
[15] M. J. Maher and D. Srivastava, "Chasing Constrained Tuple-Generating Dependencies," in *PODS*, 1996.
[16] T. Imieliński and W. L. Jr, "Incomplete information in relational databases," *JACM*, vol. 31, no. 4, pp. 761–791, 1984.
[17] G. Grahne, *The Problem of Incomplete Information in Relational Databases*. Springer, 1991.
[18] F. Bry, N. Eisinger, H. Schütz, and S. Torge, "SIC: Satisfiability checking for integrity constraints," in *DDLP*, 1998.
[19] R. Manthey, "Satisfiability of integrity constraints: Reflections on a neglected problem," in *FMLDO*, 1990.
[20] G. Ramalingam and T. W. Reps, "A categorized bibliography on incremental computation," in *POPL*, 1993.