Cleaning Data with Constraints and Experts (Technical Report)

Ahmad Assadi
Tel Aviv University
Tel Aviv, Israel
ahmadassadi@mail.tau.ac.il

Tova Milo
Tel Aviv University
Tel Aviv, Israel
milo@post.tau.ac.il

Slava Novgorodov
Tel Aviv University
Tel Aviv, Israel
slavanov@post.tau.ac.il

ABSTRACT

Popular techniques for data cleaning use integrity constraints to identify errors in the data and to automatically resolve them, e.g. by using predefined priorities among possible updates and finding a minimal repair that will resolve violations. Such automatic solutions however cannot ensure precision of the repairs since they do not have enough evidence about the actual errors and may in fact lead to wrong results with respect to the ground truth. It has thus been suggested to use *domain experts* to examine the potential updates and choose which should be applied to the database.

However, the sheer volume of the databases and the large number of possible updates that may resolve a given constraint violation, may make such a manual examination prohibitory expensive. The goal of the DANCE system presented here is to help to optimize the experts work and reduce as much as possible the number of questions (updates verification) they need to address. Given a constraint violation, our algorithm identifies the *suspicious tuples* whose update may contribute (directly or indirectly) to the constraint resolution, as well as the possible dependencies among them. Using this information it builds a graph whose nodes are the suspicious tuples and whose weighted edges capture the likelihood of an error in one tuple to occur and affect the other. Page-rank style algorithm then allows us to identify the most beneficial tuples to ask about first. Incremental graph maintenance is used to assure interactive response time. We implemented our solution in the DANCE system and show its effectiveness and efficiency through a comprehensive suite of experiments.

1 INTRODUCTION

Data cleaning is a long-standing problem that has attracted much research interest in the past years in the databases community. Many key business decisions are made based on underlying databases. Yet, real-life databases sometimes contain incomplete, wrong or inconsistent data, that may lead to incorrect output and bad decision making. Consequently, much effort has been targeted to the development of techniques to clean the underlying data.

Popular techniques for data cleaning use data-integrity and consistency rules to identify errors in the data and to automatically resolve them, e.g. by finding a *minimal repair* that will resolve the constraints violation [35], or by using predefined *priorities* among possible resolutions [21]. Such automatic solutions, however, cannot ensure the precision of the repairs since they do not have enough evidence about the actual errors and thus may, in fact, lead to wrong results with respect to the ground truth. In order to overcome the limitations of such automatic techniques it has been suggested to use *domain experts* that have extensive knowledge about the ground truth, to examine the potential updates and choose which should be applied to the database [9, 21, 32, 36].

Conference'17, July 2017, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

However, the sheer volume of the databases and the large number of possible updates that may resolve a given constraint violation, may make such a manual examination prohibitory expensive. The goal of the DANCE system presented here is to help to optimize the experts' work and reduce as much as possible the number of questions (updates verification) they need to address. As we will describe, our algorithms effectively prune the search space to minimize the amount of interaction with the experts while, at the same time, tries to maximize the potential "cleaning benefit" derived from the experts' answers. DANCE can be used to optimize the initial cleaning of a database as well as to assist in its ongoing maintenance - whenever a constraint violation is reported, DANCE can take over to efficiently clean the underlying database by interacting with the experts.

Given a constraint violation, our algorithm first identifies the tuples in the database whose update may contribute (directly or indirectly) to the constraint resolution. We call those *suspicious tuples*. Database constraints may be inter-related and thus when analyzing a constraint violation these relationships must be taken into consideration. To determine which tuples should be considered first, we examine for each suspicious tuple t (1) the potential effects of updates to t, namely what tuples may potentially become unsuspicious if t is found to be incorrect and correspondingly updated/removed, (2) the number of potential updates (attribute errors) to t that may lead to such an effect, and (3) the uncertainty, if known, for the values in the database relation to which t belongs. Using this information we build a graph whose nodes are the suspicious tuples and whose weighted edges capture the likelihood of an error in one tuple to occur and affect the other. Page-rank style algorithm is then used to identify the most beneficial tuples to ask about first.

Example 1.1. To illustrate let us consider the following simple example. The database in Figure 1 shows a portion of UEFA Champions League 2016/17 statistics database. The dark gray rows represent wrong tuples and lightgray rows represent missing tuples. The *Games* relation describes the results of a match between two teams, it stores the team's name, goals score and the stage. The *Teams* relation describes a football team, it stores the team name and country. The *Countries* relation describes the name of the country and number of teams that advanced to the group stage. We consider in our work integrity constraints described by standard tuple-generating and condition-generating dependencies [18]. The following two integrity constraints are relevant to this database: (i) two teams from the same country cannot play against each other on a group stage, and (ii) if a country has at least one representative, its team must appear in the teams table. These are captured by the following constraints.

- $Games(x_1, x_2, x_3, x_4, x_5) \wedge x_5 = "GroupStage" \wedge Teams(x_1, y_1) \wedge Teams(x_2, y_2) \rightarrow y_1 \neq y_2$
- $Countries(x_1, x_2) \wedge x_2 > 0 \rightarrow Teams(y_1, x_1)$

We assume that all the given constraints are correct and reflect the ground truth. In our running example, the constraints are derived

Games

team1	team2	goals1	goals2	stage
Celtic	Manchester City	3	3	Group Stage
Celtic	Barcelona	0	3	Group Stage
Celtic	Hapoel Beer Sheva	5	2	Qualification
Celtic	Astana	2	1	Qualification
Celtic	Lincoln Red Imps	3	0	Qualification

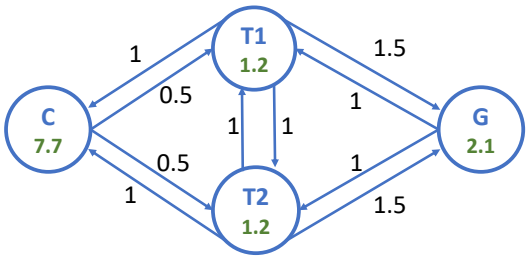
Teams

name	country
Celtic	UK
Manchester City	UK
Hapoel Beer Sheva	Israel
CSKA Moscow	Russia
Astana	Kazakhstan

Countries

name	num_teams
Israel	1
Russia	2
UK	5
England	4
Scotland	1

Figure 1: Sample of UEFA Champions League DB



β_{Teams}= 0.5, β_{Games}= 0.9, $\beta_{Countries}$= 0.5

T1 = Teams(Celtic, UK)
T2 = Teams(Manchester City, UK)
G = Games(Celtic, M. City, 3, 3, Gr. St.)
C = Countries(UK, 5)

Number of attributes for fix:
Rule #1: T1=2(all), T2=2(all),
G=3 (Celtic, M.City, Gr. St.)
Rule #2: T1=1(UK), T2=1(UK), C=2(all)

Figure 2: Suspicious tuples graph

from UEFA official regulation. Since the database is aggregated from multiple sources it contains mistakes and violates some of the constraints. One can notice for instance that the database mistakenly associates both the Celtic and the Manchester City football clubs to the United Kingdom. However, despite the fact that Celtic and Manchester City are actually located in the United Kingdom they belong to distinct federations (that represent Scotland and England separately), hence can play against each other.

When applying the integrity constraints to the database, we discover several inconsistencies. Each such inconsistency involves several tuples that when assigned together to the atoms in the body of the constraint yielded a constraint violation. For example, a violation of the first constraint involves a set of three tuples: *Games(Celtic, Manchester City, 3, 3, Group Stage)*, *Teams(Manchester City, UK)*, *Teams(Celtic, UK)*, whose existence in the database lead to the violation. Intuitively, each of the tuples is *suspicious* and at least one is wrong and needs to be updated/deleted (otherwise the constraint is incorrect which we assume is not the case). Also note that since the two constraints are inter-related, when a given tuple is suspicious other tuples become suspicious as well. Consider for example the second constraint, that requires that for each country in the *Countries* relation with a positive number of teams, there must be at least one team in *Teams* relation from this country. Relation *Countries* contains the tuple *(UK, 5)*, which enforces the existence of teams from United Kingdom. Since the *Teams(Celtic, UK)* and *Teams(Manchester City, UK)* tuples are *suspicious* (and may generally both be wrong), we may suspect also the tuple *Countries(UK, 5)*.

Which of these four suspicious tuples is more beneficial to consult about first with the expert? To determine this we build a directed graph whose nodes are the suspicious tuples and whose (weighted) edges capture the dependency between the suspicious tuples. Let β be the uncertainty of the values in the relation R to which a tuple t belongs to where β is between 0 (all the values are valid) and 1 (all the values are wrong). Intuitively, there is an edge from tuple s to t with a weight $n \times \beta$ if there are n attributes in t that one can change in order to eliminate at least one violating assignment that involves s. For example, data from official UEFA website will get a β close to 0 while user-generated content in the other relations should get much more We use 0.5 as a default value. The graph for the four tuples that we obtain is depicted in Figure 2 (ignore for now the number labels on the nodes).

To decide which tuple to verify first, we process the graph using a PageRank-style [11] algorithm, to rank the nodes, and ask the experts about the nodes with the highest rank. Intuitively, to minimize the number of questions, we would like to catch early errors whose correction may have the largest effect. When answers are gathered, the database is updated accordingly, and incremental computation is applied to update the graph and identify the next candidates. The resulting ranks for our running examples are depicted on the nodes, and so we will ask about C (which is indeed incorrect and will be removed, instead *(England, 4)* and *(Scotland, 1)* will be inserted by the expert), $T1$ (incorrect, updated to *(Celtic, Scotland)*) and $T2$ (incorrect, updated to *(Manchester City, England)*). G is then no longer suspicious and no constraint is violated.

Data cleaning with the help of experts has been previously considered in [9] where the goal was to update the database for eliminating incorrect query answers. The problem studies there is simpler because it does not consider transitive dependencies as the ones entailed by constraints. While [9] notes that further optimization could have been achieved by using the available database constants, this path is not followed there. Interestingly, our experiments show that, even in the absence of constraints, when applied to the same problem the algorithm presented here achieves superior performance compared to [9], since it better factors the dependencies between suspicious tuples.

A first prototype of DANCE was demonstrated in [8]. The short paper accompanying the demonstration gave only a high level overview of the system's capabilities and user interface whereas the present paper provides a detailed description of the model and the algorithms underlying the solution.

Our contributions can be summarized as follows.

(1) We formulate and present a constraints based framework for data cleaning with experts. Under this framework, the database is updated by (minimally) interacting with domain experts in order to fix the violations of the constraints.

(2) To address this problem we focus on a *suspicious tuples* group of tuples, that are the potential cause of the constraints violation, which we infer by analyzing the available data and constraints.
(3) We present an effective algorithm that, using the suspicious tuples and the inferred dependencies among them, builds a weighted graph that captures the potential "cleaning benefit" that a correction/verification to one suspicious tuple may yield to its neighbors. Page-rank style ranking, applied to the graph, determines the order of questions issued to the experts. Incremental computation is applied to maintain the graph as experts answers are gathered and further knowledge about the ground truth accumulates.
(4) We further show that our solution can be applied to repair database errors signaled through the identification of wrong query answers. This allows to compare DANCE to previous work in this context such as [9].
(5) We have implemented our solution in the DANCE prototype system and applied it to real use cases, demonstrating the efficiency of our constraints-based approach using domain experts. We performed experimental evaluations on real datasets showing how our algorithms consistently outperform alternative baseline algorithms, and effectively clean the data while asking fewer questions.

Outline of paper. Section 2 provides the basic definition and formalisms. The graph construction is explained in Section 3 and its incremental maintenance is described in Sections 4. Section 6 explains how the same principles may be used not only to correct constraints violation but also to handle errors identified in query answers. The implementation of DANCE prototype system, as well as experimental results, are described in Section 7. Related work is in Section 8, and we conclude in Section 9.

2 PRELIMINARIES

We start formalizing the database, the types of questions that may be posed to experts in order to clean it, and the supported integrity constraints. We will then define the set of suspicious tuples over which the cleaning algorithm is applied.

2.1 Basic definitions

Database We assume a relational schema S to be a finite set $\{R_1, ..., R_m\}$ of relational symbols, each with a fixed arity. A database instance D of S is a set $\{R_1^D, ..., R_m^D\}$, where R_i^D is a finite relation of the same arity as R_i. We use R_i to denote both the relation symbol and the relation R_i^D that interprets it. We refer to a tuple t of a relation R or fact $R(t)$ interchangeably.
Let $\mathcal{V}$ be a fixed set of variables and C be a fixed set of constants called the underlying vocabulary. Under the open world assumption, a fact that is in D is true and a fact that is not in D can be true or false. To model real-world data, we adopt the truly open world assumption where a fact that is in D can also be true or false, in addition to the assumption that a fact that is not in D can be true or false. In other words, we assume that a given database can contain mistakes, in addition to being incomplete. The truth of a tuple is given by the ground truth database D_G that contains all true tuples and only them. Hence, a database D is dirty w.r.t. D_G if $D \neq D_G$.

Questions to the Expert We assume, for simplicity of presentation, that there is a domain expert that has an extensive knowledge about the ground truth database D_G. (Otherwise, multiple experts may be consulted and standard techniques [29] may be applied to aggregate multiple answers to the same question.) In each question to the expert, the system can ask two types of questions.

- The first type is an *update question*, where the expert is asked to examine and possibly update a database tuple t, denoted by $q = Update(t)$. The answer of an update question can be one of the following options:
  – t is correct as is, denoted by $Answer(q) = true$, which means that $t \in D_G$.
  – t is wrong and should be deleted, denoted by $Answer(q) = false$, which means that $t \notin D_G$.
  – Update the tuple t with another tuple t', denoted by $Answer(q) = t'$, which means that $t \notin D_G$ but $t' \in D_G$.
- The second type is a *fill question*. During the operation of DANCE, the system can decide (e.g. based on a constraint) that a new tuple should be added to the database. While adding it, DANCE may fill some of the fields automatically and ask the expert to complete the rest. The missing values may be required to satisfy a given constraint, for example, $q = Fill(R(val_1, val_2, missing))$ *where* $missing > 20$.

Constraints The integrity constraints in DANCE are database assertions that are given as a part of the schema or added to the system at run time. We consider in our work constraints similar to the standard tuple-generating and equality-generating dependencies [6].

Specifically, the first type of constraints that we consider here are *tuple-generating constraints (tgcs)* in the spirit of the *tuple-generating dependencies with arithmetic comparisons* of [6]. These are a first order logic formulas of the form:

$$\forall x_1, ..., x_n \varphi(x_1, ..., x_n) \rightarrow \exists z_1, ..., z_m R(x_1, ..., x_n, z_1, ..., z_m)$$

where the left hand side (LHS) of the implication, φ, is a conjunction of relational atoms over the quantified variables and conditions. A condition is a boolean expression of the form v *op* w where v, w are quantified variables or constants and *op* is a boolean operation defined on the variables' domain. For example, if v and w value are numbers then $op \in \{=, \neq, \leq, \geq, >, <\}$. The right hand side (RHS) contains only one relational atom. Intuitively, given a particular combination of tuples satisfying the constraint of the LHS, tgcs expresses an assertion about the existence of a tuple in the instance on the RHS.
The second type of constraints is *condition-generating constraints (cgcs)*. They have the same form as tgcs but the RHS is a conjunction of conditions, and are defined as the *arithmetic-comparison-generating dependencies* of [6]. Both tgcs and cgcs have a "safety" restriction i.e. all LHS variables should appear in relational atom.

Example 2.1. Consider the database in Figure 1 and the constraints from Example 1.1.

- $Games(x_1, x_2, x_3, x_4, x_5) \wedge x_5 = "GroupStage" \wedge Teams(x_1, y_1) \wedge Teams(x_2, y_2) \rightarrow y_1 \neq y_2$
- $Countries(x_1, x_2) \wedge x_2 > 0 \rightarrow Teams(y_1, x_1)$

The first constraint is a cgc and the second is a tgc.

Assignments and constraints satisfaction Let $Var(\varphi)$ be the variables that appear in the constraint φ. An assignment $v: Var(\varphi) \rightarrow C$ for a constraint φ is a mapping from the constraint variables to constants. We denote by $Assign(\varphi)$ all the assignments for the constraint φ. An assignment v satisfies a relational atom $R(\bar{x})$ if and only if $R(v(\bar{x})) \in D$, denoted by $v \models_D R(\bar{x})$. In a similar way an assignment v satisfies a constraint $\varphi \rightarrow \psi$ if and only if $v \not\models_D \varphi$ or $v \models_D \varphi$ and $v \models_D \psi$, denoted by $v \models_D \varphi \rightarrow \psi$.
By the definition, $v \not\models_D \varphi \rightarrow \psi$ if and only if $v \models_D \varphi$ and $v \not\models_D \psi$.

We say that a database D satisfies a constraint φ if and only if, for all assignment $v \in Assign(\varphi)$ it holds that $v \vDash_D \varphi$.

Remark. Let φ be a constraint and $A_1, \ldots, A_n$ be the relational atoms in φ. By the constraints definition, each variable x in φ can be assigned to any attribute in any relational atom, for example, in the second constraint from Example 2.1, the variable x_1 is assigned to the attribute *name* in the relational atom $Countries(x_1, x_2)$ and to the attribute *country* in the relational atom $Teams(y_1, x_1)$. To simplify the exposition, in the rest of this paper we assume that, in each constraint, it holds that each variable could be assigned to at most one relational atom attribute. For example, the first constraint in Example 2.1 satisfies our assumption. This assumption is not restrictive because each constraint can be easily converted to a logically equivalent constraint that satisfies the requirement by duplicating each variable that is mapped to more than one attribute and adding suitable equality conditions of the variable duplicates. For example, the second constraint from Example 2.1 can be converted to the constraint:

$$Countries(x_1, x_2) \wedge x_2 > 0 \wedge x_1 = x_{1'} \rightarrow Teams(y_1, x_{1'})$$

2.2 Suspicious tuples

In our context, the cleaning process is triggered when the given set of constraints is not satisfied. Note that the constraints are satisfied by the ground truth database D_G but violated by the dirty database D. To clean D, we identify the *suspicious tuples* that may be (directly or transitively) the cause of the problem. Some auxiliary definitions are useful here. We first define the notion of *violation set* - a set of tuples that is a direct cause of a constraint violation. Namely, when assigned together to atoms of the constraint they cause its violation. Next, we define the notion of a *Proof tuples* - the tuples that (through the same or other constraints) assert the existence of some violating sets members. The suspicious tuples are then the union of the violation and proof tuples.

Violation sets Intuitively, we define a *violation set* of a constraint φ in a database D as a minimal set of tuples in D that implies $\exists v$ s.t. $v \nvDash_D \varphi$, denoted by $Vio(\varphi, D)$.
The formal definition depends on the constraint type (cgc or tgc):

- **Cgc:** Let $\varphi : \forall \bar{x} A_1 \wedge \ldots \wedge A_k \wedge C_1 \wedge \ldots \wedge C_j \rightarrow \exists \bar{z} C_{j+1} \wedge \ldots \wedge C_t$ be a cgc where A_i is a relational atom and C_i is a constraint. Then $\{t_1, ..., t_k\} \in Vio(\varphi, D)$ if and only if exists an assignment v s.t. $v \nvDash_D \varphi$ and $\forall 1 \leq i \leq k : t_i = A_i(v(\bar{x}))$.
  In other words, $Vio(\varphi, D)$ contains all sets of tuples $\{t_1, ..., t_k\}$ that caused the database D to violate the constraint φ.
- **Tgc:** Let $\varphi : \forall \bar{x} A_1 \wedge \ldots \wedge A_k \wedge C_1 \wedge \ldots \wedge C_j \rightarrow \exists \bar{z} R(\bar{x}, \bar{z})$ be a tgc where A_i is a relational atom and C_i is a constraint. Then $\{t_1, ..., t_k, \tilde{t}\} \in Vio(\varphi, D)$ if and only if exists an assignment v s.t. $v \nvDash_D \varphi$ and $\forall 1 \leq i \leq k : t_i = A_i(v(\bar{x}))$ and $\tilde{t}$ is the RHS that is not satisfied. In $\tilde{t}$ the bounded variables of the RHS are replaced with a wildcard "*" which represents an unknown value, as the value is not defined by the assignment $v(\bar{x})$.

In other words, each tuples set $T \in Vio(\varphi, D)$ is a set of tuples that caused the database D to violate the constraint φ. In case that φ is a tgc, T may also contain a tuple template that may be missing from D.
Therefore, each $T \in Vio(\varphi, D)$ must contain at least one tuple that is wrong or its last tuple may be missing from the database, otherwise, the constraint is invalid and this contradicts our assumption about the constraints correctness.

For a set of constraints, the violation sets are the union of the violation sets of the individual constraints.

$$Vio(f, D) = \bigcup_{\varphi \in f} Vio(\varphi, D)$$

Example 2.2. Consider the database and the constraints from Example 1.1. The first constraint represents the constraint: two teams from the same country cannot play against each other on a group stage. Therefore, according to the *Games* and *Teams* table, the set of tuples *{ Games(Celtic, M. City, 3, 3, Group Stage), Teams(Celtic, UK), Teams(M. City, UK)}* is a violation. The second constraint represents the constraint: if a country has at least one representative, its team must appear in the teams' table. Therefore, the set of tuples *{ Countries(Israel, 1), Teams(*, Israel)}* is also a violation, because *Israel* has no representatives in the *Teams* table.

Proof tuples To determine which tuples (transitively) assert the existence of tuples in the violation sets, we examine each individual attribute and identify the tuples that assert its value. We call these set of tuples tuple-value proof.
Tuple Values Proof Let $t = (v_1, ..., v_n)$ be a tuple of the relation R in D with arity n, let $V = (v_{i_1}, ..., v_{i_m})$ be a sub-tuple of the tuple t and let $\varphi = \forall \bar{x} A_1 \wedge \ldots \wedge A_k \wedge C_1 \wedge \ldots \wedge C_j \rightarrow \exists \bar{z} R$ be a tuple generating constraint, where A_i and R are relational atoms and C_i is a condition. Intuitively, a set of tuples $\{t_1, ..., t_k\}$ "proves" the validity of the values of V in t if by using only φ and the assumption that the tuples $\{t_1, ..., t_k\}$ are valid we can conclude that the values of V in t are also valid. Formally, we say that $\{t_1, ..., t_k\}$ proves the values of V in t via φ if and only if exists an assignment $v(\bar{x}, \bar{z})$ s.t. :

(1) $v \vDash_D \varphi$
(2) $\forall 1 \leq i \leq k \; t_i = A_i(v(\bar{x}))$
(3) $t = R(v(\bar{x}, \bar{z}))$
(4) All the values in V are mapped by the assignment v to a variable quantified within the head of the constraint φ.

We denote the proofs of the values of V in t via φ by the set $ValProof(\varphi, t, V)$.

Example 2.3. Consider the database in Figure 1 and let φ_2 be the second constraint from Example 2.1. Let $V = (UK)$ and t= Teams(Celtic, UK). It holds that the set of tuples {Countries(UK, 5)} $\in ValProof(\varphi_2, t, V)$. Because the tuple Countries(UK, 5) implies the existence of the value UK in t. Observe, if $V = (Celtic)$ then ValProof(φ_2, t, V)=$\emptyset$ because the value *Celtic* in t is mapped to the variable y_1 which is not quantified within the head of φ_2.

Relevant proofs Note however that not all attribute values (and their proofs) are suspicious.

First, as we interact with the expert, some attribute values may be verified to be correct. This happens when the expert asserts, as an answer to an update question, that the full tuple is correct, or when she fills in herself the values as an answer to a fill question. For a tuple t in D, we denote by $VVal(D, t)$ its validated attribute values. We will not include them (and their proofs) in the suspicious set.

Second, not all the attributes of the violation tuples contribute to the violation. For a constraint φ, we call the variables that appear in a conditional atom *constraint conditioned variables* and denote the set of such variables $CVars(\varphi)$. We call the values assigned to the conditional variables the *conditional values*. One may notice that, if there is a violation $T \in Vio(\varphi, D)$ then the assignment v that caused the violation satisfies the body of φ

and dissatisfy the head and changing the assignment of the non-conditioned variables will not change the dissatisfaction of the head, because, in case φ is a cgc, the head of φ contains only conditional atoms and they contain only conditional variables and in case φ is a tgc, all the variables in the relational atom of the head are conditional variables that defined in the body[1]. As a result, in order to resolve the violation T, the expert must perform one of the three following actions: (i) remove at least one of the tuples in T that is responsible for the satisfaction of the constraint's body, (ii) insert a tuple which completes the incomplete tuple of T (in case φ is a tgc) or (iii) update the conditional values of T. Consequently we are interested only in proofs that assert these *conditional values* (values that assigned to conditional variables).

Example 2.4. Let φ_1 be the first constraint from Example 2.1. Consider the assignment $v(x_1, x_2, x_3, x_4, x_5, y_1, y_2)$= (Celtic, M. City, 3, 3, Group Stage, UK, UK), it holds that $x_3, x_4 \notin CVars(\varphi_1)$. Hence changing the value of x_3 or x_4 will not resolve the violation.

In summary, for a tuple t and a constraint φ,we are interested only in tuple value proofs of its conditional attributes whose value has not been verified yet. We denote these by $Proof(t, \varphi)$ and define them formally as follows:

$$Proof(t, \varphi) = \{P \in ValProof(\varphi, t, V) \mid V \subset t[CVars(\varphi)] \setminus VVal(D, t)\}$$

where $t[CVars(\varphi)]$ are the values of the conditional attributes in t.

Note that when the all the conditional values of a tuple t are validated, it holds that, t may no longer be responsible for the dissatisfaction of φ in the database D, in which case $t[CVars(\varphi)] \setminus VVal(D, t) = \emptyset$ and $Proof(t, \varphi) = \emptyset$.

Suspicious tuples We are now ready to define the set of suspicious tuples. First we close transitively the set of tuple value proofs, considering the proofs of previously identified tuples. For that we define the proofs of a database D via a set of constraints f as follows:

- $Proofs^0(f, D) = \bigcup_{\varphi \in f} Vio(\varphi, D)$, the proofs calculation starts from the violations
- Let $S^{i-1} = \bigcup Proofs^{i-1}(f, D)$ then,

  $$Proofs^i(f, D) = \bigcup_{\varphi \in f} [\bigcup_{t \in S^{i-1}} Proof(t, \varphi)]$$

  Note that by $Proof(t, \varphi)$ definition, we use $ValProof$ to calculate it. Then we use it to execute a step of $Proofs$ recursion.
- $Proofs(f, D) = \bigcup_{i \in \mathbb{N}} Proofs^i(f, D)$

Observe that the proofs of a database D can be computed in a polynomial time using a fixpoint-based recursive algorithm.

For a database D and a set f of constraints, the set of suspicious tuples includes all the tuples in the proofs, excluding those who has already been verified by the expert (through update questions). We denote the set validated tuples by $Validated(f, D)$. The suspicious tuple are then

$$Susp(f, D) = \bigcup_{P \in Proofs(f, D)} P \setminus Validated(f, D)$$

Note that the *violation sets* are included in the *suspicious tuples* because they are included in the formal definition of *proofs*.

[1] Note that rules such as, e.g., $R(a, b) \rightarrow R(b, a)$ are not valid in our setting, since it is assumed (see Section 2.1) that rules are written s.t. in each constraint, every variable is assigned to at most one relational atom attribute.

Also by the tgcs violation definition, in addition to complete tuples $Susp(f, D)$ may also contain tuples templates (i.e. tuples that contain unknown missing value represented by the wildcard "*".)

3 BUILDING THE TUPLES GRAPH

As we mentioned in the Introduction, to determine which tuples should be considered first (the next question that will be posed to the expert), DANCE builds a graph whose nodes are the suspicious tuples and whose weighted edges capture the likelihood of an error in one tuple to occur and affect the other. We call this graph the *tuples graph*. Page-rank style algorithm is then applied to the graph to identify the most beneficial tuples to ask about first. For regular tuples we will ask an update questions $q = Update(t)$, whereas for tuple templates we will ask a fill question $q = Fill(t)$.

The tuples graph is an edge-weighted directed graph $G(V, E, W)$ where V is the set of nodes, E the set of edges and $W : E \rightarrow \mathbb{R}$ is the weight function. We formally define them below.

3.1 Vertices and edges

The graph vertices $V = \{v_t | t \in Susp(f, D)\}$ are the set of suspicious tuples. The graph edges capture the potential effect of updates to t, namely what tuples may potentially become unsuspicious if t is found to be incorrect and correspondingly updated/removed. To define this formally we use the following auxiliary definition.

More formally, recall that the expert's answer to the question $q = Update(t)$ is interpreted as a database edit e_q where e_q could be (i) do nothing, in case that $Answer(q) = true$, (ii) delete t, in case that $Answer(q) = false$, or (iii) replace t by t', in case $Answer(q) = t'$. We say that *a tuple t could cancel a violation/proof set T* if and only if, the answer to the question $Update(t)$ could potentially generate an edit e_q such that, the tuples set $T' = T \oplus e_q$ will not be a violation/proof set.

To illustrate, consider the database and the constraints from Example 1.1 and the violations from Example 2.2 (which defined the set of tuples *{Games(Celtic, M. City, 3, 3, Group Stage), Teams(Celtic, UK), Teams(M. City, UK)}* as a violation set). Notice that the tuple *t = Teams(Celtic, UK)* could cancel the violation because, if the system asks the expert to update this tuple one possible (and in fact correct) answer may be *t'=Teams(Celtic, Scotland)*. If the system replaces the tuple t by t' in D, the set *{ Games(Celtic, M. City, 3, 3, Group Stage), Teams(Celtic, Scotland), Teams(M. City, UK)}* will no longer be a violation set.

Consider two suspicious tuples t_{src}, t_{dst} and their corresponding vertices $v_{t_{src}}, v_{t_{dst}}$ respectively. We include in E an edge $e = (v_{t_{src}}, v_{t_{dst}})$ if and only if the tuple t_{dst} could cancel at least one proof/violation set T that contains t_{src}. Note that from the definition of proof/violation sets, this happens if and only if t_{src} and t_{dst} both participate in T. Consequently,

$$e = (v_{t_{src}}, v_{t_{dst}}) \in E \iff \exists T \in Proofs(f, D) : v_{t_{src}}, v_{t_{dst}} \in T$$

Note that this in particular means that all edges are bidirectional, and the tuples graph is a union of a collection of cliques where each clique is defined by some violation/proof set.

Example 3.1. Consider the database from Example 1.1 and let φ_1, φ_2 be the first and the second constraints from Example 2.1 respectively. As we mentioned in Example 2.2, the set of tuples *{Games(Celtic, M. City, 3, 3, Group Stage), Teams(Celtic, UK), Teams(M. City, UK)}* is a violation and by the *Proofs* definition,

it holds that, the tuple *Countries(UK, 5)* is a proof of the tuples *Teams(Celtic, UK)* and *Teams(M. City, UK)* by the constraint φ_2. Therefore, the four tuples *Teams(M. City, UK)*, *Teams(Celtic, UK)*, *Countries(UK, 5)* and *Games(Celtic, M. City, 3, 3, Group Stage)* are suspicious.

Figure 2 depicts the Tuples Graph of these four suspicious tuples. The tuples *T1*, *T2* and *G* are connected to each other because of the violation {*G, T1, T2*}. The existence of the proofs {C,T1} and {C,T2} implies the connection between T1, T2 and C.

The weights on the graph edges will be explained in details in the next section.

3.2 Edge weights

The edge weights capture the likelihood of an error in one tuple to occur and to affect the other. We assume that for each relation R we are given a parameter $0 \leq \beta \leq 1$ that describes the uncertainty of the values in R. (If no such information is available a default value of 0.5 is used). Intuitively, the weight w_e assigned to an edge $e = (v_{t_{src}}, v_{t_{dst}})$ is $n \times \beta$ where β is the uncertainty of the values in the relation to which t_{dst} belongs, and n is overall number of attributes in t_{dst} that one can update in order to eliminate a violation/proof set that involves t_{src} and t_{dst}.

Algorithm 1 Calculating the weight of the edge e

```
1:  procedure CalculateEdgeWeight(v_{t_src}, v_{t_dst}, β)
2:      V ← {T | T ∈ Proofs(f, D) ∧ t_src, t_dst ∈ T}
3:      w_e ← 0
4:      Attrs ← getConditionalAttributes(t_dst)
5:      foreach T ∈ V do
6:          Δ ← 0
7:          counter ← 0
8:          foreach Attr_i ∈ Attrs do
9:              b ← hasCancellationUpdate(Attr_i, T, t_dst)
10:             if b == true then
11:                 counter++
12:             end if
13:         end for
14:         Δ ← counter * β
15:         w_e ← w_e + Δ
16:     end for
17:     return w_e
18: end procedure
```

This is formalized in Algorithm 1 for calculating w_e. The input of the algorithm includes the endpoints of the edge $v_{t_{src}}$ and $v_{t_{dst}}$ and the uncertainty β of the values in the relation to which the t_{dst} belongs. The variable V is assigned the proofs/violations sets the include t_{src} and t_{dst}. In lines 3 and 4 the weight w_e is initialized to 0 and $Attrs$ is assigned to the set of all conditional attributes of t_{dst}. Then, in line 5, the algorithm iterates over all the relevant proofs In each such iteration it counts the number of conditional values in t_{dst} that may be updated in order to cancel T, multiply this number by the uncertainty measure and adds the result to the overall weight. To check whether an updated to conditional value at the attribute $Attr_i$ may cancel T (line 9), we use the function *hasCancellationUpdate* whose implementation is be explained below.

hasCancellationUpdate The function *hasCancellationUpdate* gets as input (i) an attribute $Attr$, (ii) a violation/proof set T and (iii) a tuple t. It then checks whether there is an update to the tuple t, yielding a tuple t' that differs from t at the attribute $Attr$, s.t. $T \setminus \{t\} \cup \{t'\}$ is no longer a violation/proof set. For that it considers the constraint φ that is responsible for T being a violation/proof set. It generates a new constraint φ' that captures the desired updates, and tests whether φ' is satisfiable. We next explain how φ' is defined, then prove that it is satisfiable if and only if a desired update exists.

We have that $T \in Vio(\varphi, D)$ or $T \in Proofs(\varphi, D) \setminus Vio(\varphi, D)$. Let us first consider the first case, where $T \in Vio(\varphi, D)$. In this case, there must be an assignment v, such that $v \not\models_D \varphi$ and exist a relational atom $R_i(x_1^t, ..., x_k^t)$ in φ where $R_i(v(x_1^t), ..., v(x_k^t)) = t$. Let $x' \in \{x_1^t, ..., x_k^t\}$ be the variable at the attribute $Attr$ in the relational atom $R_i(x_1^t, ..., x_k^t)$. Observe that, by updating t by t' a new assignment v' is generated from v, where $\forall x \in Var(\varphi)$, $v(x) \neq v'(x)$ if and only if $x = x'$. Also if $v' \models_D \varphi$ then updating t by t' cancels T. This is enforced by the constraint φ' as follows. φ' contains the conditional atoms of φ with an addition of a conditional atom for each variable x that appears in φ:

- If $x = x'$ then the condition is $x \neq v(x)$ because we want an assignment v' that differs from v at the variable x'.
- If $x \neq x'$ then the condition is $x = v(x)$.

We can show the following:

CLAIM 3.2. *φ' is satisfiable iff there is an update to the tuple t, yielding a tuple t' that differs from t at the attribute Attr, s.t. $T \setminus \{t\} \cup \{t'\}$ is no longer a violation/proof set.*

PROOF. If φ' is satisfiable then there is a satisfying assignment v' for the constraint φ', then by φ' definition, v' satisfies all the conditional atoms in φ. Also, because v' differs from v only at the variable x', v' satisfies all the relational atoms in φ. However, v' does not satisfy the atom $R_i(x_1^t, ..., x_k^t)$, because it is not ensured that the tuple $R_i(v'(x_1^t), ..., v'(x_k^t))$ exist in the database. But if we say that $R_i(v'(x_1^t), ..., v'(x_k^t)) = t'$ then v' will satisfy φ if we update t by t'. The fact that $v' \models_D \varphi$ implies that updating t by t' will cause the cancellation of T. The other direction of the claim also holds because if φ' is not satisfiable, then each assignment v' doesn't satisfy at least one of the conditional atoms of φ or it doesn't yield a tuple t' that differs from t at the attribute $Attr$. □

The constraint φ' is of the form $C_1 \wedge ... \wedge C_j \rightarrow C_{j+1} \wedge ... \wedge C_n$ where each C_i is a condition of the form $v\ op\ w$, v and w are constants or variables and op is an operation on the domain of v and w. In our implementation we use databases that contain two data types: reals and strings, and constraints with predicates $\{=, \neq, \leq, \geq, >, <\}$ for reals and $\{=, \neq\}$ for strings. To check the satisfiability of constraints over integers we use linear programming whereas for constraint over strings we have implemented a simple algorithm that checks whether the given set of string (in)equalities contain a contradiction. In the general case, this is a *constraint satisfaction problem (CSP)* [25] whose complexity depends on the values domain and the allowed constraints language.

Example 3.3. To illustrate, let φ_1 be the first constraint from Example 2.1 and $T = \{Games(Celtic, M.City, 3, 3, GroupStage), Teams(Celtic, UK), Teams(M.City, UK)\}$ the violation from Example 2.2. In the call hasCancellationUpdate("country", T, Teams(Celtic, UK)) the generated constraint φ'' is:

$$\varphi'' : x_1 = Celtic \wedge x_2 = M.City \wedge x_3 = 3 \wedge x_4 = 3 \wedge x_5 = GroupStage \wedge y_1 \neq UK \wedge y_2 = UK \rightarrow y_1 \neq y_2$$

The call will return true, because the constraint is satisfiable by assigning *Scotland* to y_1 and keeping the other values as is.

The case where $T \in Proofs(\varphi, D) \setminus Vio(\varphi, D)$ works similarly, with the difference that now there must be an assignment v, such

that $v \models_D \varphi$ and we are looking for an assignment v' that is generated from v as above but where $v' \not\models_D \varphi$. It can be solved by negating the constraint φ'' and looking for a satisfying assignment. This is done by using the algorithms from the first case (when $T \in Vio(\varphi, D)$), to check the satisfiability of the negation of φ''.

3.3 Node Weights

Finally, to decide which tuple to verify first, we process the graph using a PageRank-style algorithm [11], to rank the nodes, and ask the experts about the nodes with the highest rank. Intuitively, the higher rank for a tuple captures the potential for higher influence in terms errors (violations) elimination.

To complete our running example, the graph in Figure 2 is the tuples graph for the suspicious tuple from Example 3.1 . The edge weights are calculated with $\beta = 0.5$ for all relations. For instance, the weight of the edge from G to $T1$ is 1.5 because there are 3 values in G that can be updated in order to cancel the violation (the values are "ManCity", "Celtic", "Group Stage") and $\beta = 0.5$. The numbers on the graph nodes are their ranks after running our PageRank algorithm on the graph. The node C has the highest rank (7.7). Therefore, DANCE will ask the experts to check and update the tuple C.

4 INCREMENTAL GRAPH MAINTENANCE

As the computation advances and we get more answers from the users, the graph is updated to reflect the current database state and the (remaining) constraints violations. To explain how this is done, let us first examine what validated values can be inferred from the users answers, and how those affects the graph's shape.

Inferring validated values. As explained in Section 2, the *validated values* $VVal(D)$ are the set of tuples attributes that have been validated by the experts. As the computation advances and we obtain more answers from the users, the set grows. In particular, when DANCE poses a $q = Update(t)$ question to the expert, if the answer is $true$ then we know that t is correct and all its attributes are added to $VVal(D)$, and if the answer is a new tuple t' then we know that t' is correct and its attributes are added.

A simple observation is that more validated values can be inferred by following the constraints. Recall from Section 2, that a tgc expresses an assertion about the existence of a tuple assignment to the RHS, given a tuples assignment satisfying the constraint of the LHS. Therefore, we can use the tgcs to infer new valid values from the current valid values set. Let $\varphi : \forall \bar{x} A_1 \wedge ... \wedge A_k \wedge C_1 \wedge ... \wedge C_j \rightarrow \exists \bar{z} R(\bar{x}, \bar{z})$ be a tgc where A_i is a relational atom and C_i is a conditional atom. Let $\{t_1, ..., t_k, t\}$ a set of tuples where exists an assignment v such that $v \models_D \varphi$ and $\forall 1 \leq i \leq k : t_i = A_i(v(\bar{x}))$ and $t = R(v(\bar{x}), v(\bar{z}))$. We can infer new valid values based on the following claim:

CLAIM 4.1. *Given an assignment as described above, if the values assigned to the conditional variables of the LHS are valid then so are the values assigned to the conditional variables of the RHS.*

The proof is immediate from the definition of conditional variables and tgd semantics.

Example 4.2. To illustrate, consider the database from Example 1.1 and let φ_2 be the second constraint from Example 2.1. Let $v(x_1, x_2, y_1)$ ={UK, 5, Celtic} a satisfying assignment for φ_2. Let *{Countries(UK, 5), Teams(Celtic, UK)}* be the set that is defined by v. Also assume that the expert validated the value UK in the tuple *Countries(UK, 5)*, this means that there exists a country named UK. Therefore by φ_2 there must be at least one team from the country *UK*. Hence the value *UK* in the tuple *Teams(Celtic, UK)* is valid, nevertheless the whole tuple may still be incorrect, i.e. "Celtic".

The inference performed in Claim 4.1 can be repeated recursively to add further validated values, until a fix point is achieved. In our implementation (described in the following section) we will refer to this as the $databaseUpdate$ function.

Graph Update. We can now use the updated set of validated values to update the graph, following the definitions is Section 2. For that we maintain provenance of how $Proof(t, \varphi)$ and $Proofs(t, \varphi)$ and $Susp(f, D)$ were computed in the previous iteration. Then we update them as follows: We remove from each $Vio(\varphi, D)$ and $Proof(t, \varphi)$ all the tuples that their conditional values are validated. We then recursively update $Proofs^i(f, D)$ by removing the tuples derived from these removed proofs, yielding an updated value for $Proofs(f, D)$. By definition, the updated set $Susp(f, D)$ of suspicious tuples now includes only the remaining tuples, and all other nodes can be deleted from the graph. Similarly, edges that no longer belong to $Proofs(f, D)$ can be omitted.

In an analogous manner, note that answers to fill questions $q = Fill(t)$ insert new values to the database which may lead to new constraint violations. The Proofs for these new violations can be computed following the iterative definition from Section 2, and the new suspicious tuples (corresponding edges) are added to the graph.

To conclude we note that to further speed up computation time, an incremental page rank computation [17] can be applied to update the node ranks. As the graphs in our experiments turn out to be of moderate size, we did not use this here.

5 PUTTING EVERYTHING TOGETHER

To complete the picture, we explain how the different functionalities of our system are combined together.

The *System Manager* module is the main component of DANCE that interact with the databases and the experts. Given a set of constraints f (tgcs and cgcs) it runs Algorithm 2 in order to edit the database D, to obtain D' that satisfies f. The algorithms puts together the components described in the previous sections. In line 2 it calculates the violations of f in D. In line 3 it calculates the proofs based on the violations V.

After calculating the proofs and the violations, Algorithm 2 sets $Susp$ to be the set of the suspicious tuples (line 4), then it runs on iterations until all constraints are satisfied (which means that $Susp = \emptyset$). In each iteration it executes the commands in lines 6-13. In line 6 it calculates the tuples graph including the edges weights. In line 7 it executes PageRank on the tuples graph and identifies the tuple t with the highest rank, which the expert will be asked about. In line 8 the expert is asked to update the tuple t, the result is stored in t'. Note that if $t' = t$ then t is valid and if $t' = NULL$ then t must be deleted from D, else t must be replaced with t'. The corresponding database update is performed by function $databaseUpdate$ (line 9) which also infers the consequent set of validated values. After updating the validated values, the algorithm executes the function $tryToFillIncompleteTuples()$ (line 10) that iterates over all incomplete tuples (tuples that contain wildcards "*"). For each one of these values (denoted by t) it checks if the known values of t are validated, and if so asks the expert to also fill the unknown values of t, which is done by posing the question $Fill(t', true)$ to the experts. In lines 11- 13 the algorithm

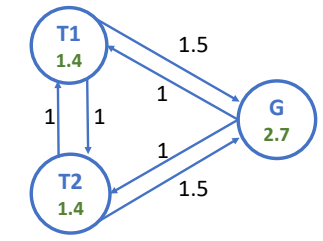


Figure 3: Suspicious tuples graph (second iteration)

recalculates the violations, proofs and the suspicious tuples. The graph is updated accordingly in the following interaction.

Example 5.1. Consider the database and the constraints from Example 1.1. Figure 2 is the tuples graph of the first iteration of Algorithm 2 where the algorithm executed on the database from Example 1.1 with injection of the two constraints from Example 2.1. The numbers on an edge is its weight and on a vertex is the vertex rank (PageRank result). As it appears in the graph, the tuple *Countries(UK,5)* has the highest rank, therefore the expert will be asked to update the tuple which will cause its deletion. After deleting the tuple *Countries(UK,5)*, the algorithm will move to the next iteration. Figure 3 is tuples graph of the second iteration. As is appears in the graph, the tuple *Countries(UK,5)* was deleted and the tuples *Teams(Celtic, UK)* and *Teams(M. City, UK)* have the highest ranks. The algorithm will ask the expert to update one of them randomly, because their ranks are equal. In the third iteration the algorithm will ask the expert to update the other team tuple, that will imply the satisfaction of the constraints by the database.

Since the ground truth database is finite, and the experts are assumed to provide correct answers, each iteration brings us closer to the ground truth database, and thus the algorithm is guaranteed to converge to a database D' where all the constraints f are satisfied (where $Susp(f, D) = \emptyset$).

Algorithm 2 The Manager Main Algorithm

```
1:  procedure managerMain(D,f)
2:      V ← calculateViolations(D, f)
3:      P ← calculateProofs(D, f, V)
4:      Susp ← calculateSuspiciousTuples(V, P)
5:      while Susp ≠ ∅ do
6:          G ← buildTuplesGraph(D, f, Susp)
7:          t ← getMaxRankedTuple(G)
8:          t' ← expertsUpdate(t)
9:          databaseUpdate(t, t')
10:         tryToFillIncompleteTuples()
11:         V ← calculateViolations(D, f)
12:         P ← calculateProofs(D, f, V)
13:         Susp ← calculateSuspiciousTuples(V, P)
14:     end while
15:     return
16: end procedure
```

6 QUERY ORIENTED DATA CLEANING

To conclude the presentation of DANCE we show that our solution can also be applied to repair database errors signaled through the identification of wrong query answers. This will alow us to experimentally compare DANCE to previous work in this context such as [9].

The scenario considered is the following. After executing a query Q on the database D (where Q is a conjunctive Datalog query without negations on relational atoms), a user may examine the query result and identify a tuple $r \in Q(D)$ that is a wrong answer and should not be part of the result. This should trigger a data cleaning process whose goal is to identify (and correct) the errors in the database that led to the wrong answer.

We show below that an assertion that a given tuple should not be included in the query result can be expressed as a simple constraint violation in our formalism, and thus our algorithms can be triggered to resolve the violation. Specifically, given an assertion $r \notin Q(D)$, DANCE translates the removed answer r to a cgc φ_r such that for each database D, if the database satisfies the constraint φ_r then $r \notin Q(D)$. Therefore, adding the constraint φ_r to the system's set of constraints f will guarantee the answer deletion. We next describe the construction of φ_r from the wrong answer $r \in Q(D)$.

Let Q be a conjunctive Datalog query without negations on relational atoms of the form:

$$Q(x_1, ..., x_n) : -A_1 \wedge ... \wedge A_k \wedge C_1 \wedge ... \wedge C_m$$

Where each A_i is a relational atom and each C_i is a conditional atom. Let $r = (\alpha_1, ..., \alpha_n) \in Q(D)$ be the wrong answer. Then we define φ_r as:

$$\varphi_r = C_{x_1} \wedge ... \wedge C_{x_n} \wedge A_1 \wedge ... \wedge A_k \wedge C_1 \wedge ... \wedge C_m \rightarrow false$$

Where each C_{x_i} is a conditional atom that demands that x_i is equal to α_i. Formally, $\forall 1 \leq i \leq n$, C_{x_i} is the conditional atom $(x_i = \alpha_i)$.

THEOREM 6.1. *Given a database D, a query Q, an answer $r = (\alpha_1, ..., \alpha_n)$ and its corresponding constraint φ_r, the following condition holds: if D satisfies the constraint φ_r, then $r \notin Q(D)$.*

Proof. If D satisfies φ_r then each assignment v also satisfies φ_r, therefore, v necessarily does not satisfy the LHS because the RHS is always false. It means that $v \not\models_D (C_{x_1} \wedge ... \wedge C_{x_n} \wedge A_1 \wedge ... \wedge A_k \wedge C_1 \wedge ... \wedge C_m)$. It follows that there is only two distinct options:

(1) $v \not\models_D A_1 \wedge ... \wedge A_k \wedge C_1 \wedge ... \wedge C_m$:
In this case, by the query definition, it holds that $(v(x_1), ..., v(x_n)) \notin Q(D)$. especially $r \notin Q(D)$

(2) $v \not\models_D C_{x_1} \wedge ... \wedge C_{x_n}$:
In this case, $(v(x_1), ..., v(x_n)) \neq r$ therefor $r \notin Q(D)$

From theorem 6.1 we can conclude that, if the constraint φ_r is injected to the database D then it holds that $r \notin Q(D')$ where D' is the database after editing D and D' satisfies φ_r.

Example 6.2. Consider the database from Example 2.2 and the query:

$$Q(x_1) : -Countries(x_1, x_2) \wedge x_2 > 2$$

It returns all the countries that have more than 2 representative teams. Running the query on the database will return the values $\{UK\}$ therefore the expert may specify that she wants to remove the answer UK. DANCE will build the constraint:

$$\varphi_{(UK)} : \mathbf{x_1} = \mathbf{UK} \wedge Countries(x_1, x_2) \wedge x_2 > 2 \rightarrow false$$

Injecting the constraint $\varphi_{(UK)}$ to the DANCE will cause the deletion of the tuple {Countries(UK, 5)} that will remove the answer (UK) from $Q(D)$.

7 EXPERIMENTS

In this section we first describe the system architecture of DANCE, and then present the experimental results on several datasets competing with other algorithms using different setups.

7.1 System Architecture

The architecture of DANCE is depicted in Figure 4. The experts interact with the system through the *User Interface*. They can provide integrity constraints that are added to the *constraints DB*. The *Violation Detector* continuously validates the *DB* against the constraints. When violations are detected, they are passed to the *Graph Builder* along with their corresponding proofs (see Section 2 for technical details). The *Graph Builder* is responsible for generating and maintaining the graph (as detailed in Section 3). Once the graph is generated, the modules runs the PageRank style algorithm to determine the most promising question to the experts (tuple for verification/update). The module works in an incremental manner, hence once the graph is generated it is maintained and updated incrementally (see Section 4 for details). Based on the experts answer (update/verification) the *DB* is updated accordingly. The process repeats until there are no more violations. As mentioned in Section 6, DANCE can also support a query-oriented cleaning mode. This flow is depicted in the system architecture figure with dashed lines. The query and the tuples to be removed are passed to the *Query-based Constraints Generator*, which generates the relevant constraint and adds it to *constraints DB*. From there things run as described above.

7.2 Experiments

We have implemented the DANCE prototype system, using Java and SQLite as the DBMS. All experiments have been executed on an Intel i7 2.4Ghz processor and 16GB RAM. We run experiments over real-life data sets and examined the system performance both in terms of the number of questions posed to the experts and the running time. We examined the full system, and contribution of separate components of our solution. We start by describing the considered algorithms, then the datasets and finally describe our experiments.

Algorithms. The main algorithm in DANCE builds the tuples graphs and ranks the nodes so that their rank reflects their potential importance for the database cleaning. This is achieved by assigning to the edges weights that not only reflect the potential influence tuple updates but also the uncertainty of the values in the corresponding relations. To assess the importance of this ranking, we compare DANCE to three alternatives alternative algorithms.

- Random: a naïve algorithm that randomly picks tuples in the graph to ask about
- DANCE v1: a simplified version of DANCE where all edges are assigned an equal weight (equals to 1)
- DANCE v2: a simplified variant of DANCE where the uncertainty of the values of all the relations are the same (by setting $\beta = 0.5$ for all relation).

We will see that the full fledged algorithm yields fewer questions than its restricted variants.

We have also compared DANCE to a related previous work, described next. As mentioned in the Introduction, data cleaning with the help of experts has been previously considered in the QOCO system [9] where the goal was to update the database for eliminating incorrect query answers. It is easy to show that an assertion that a given tuple should not be included in the query

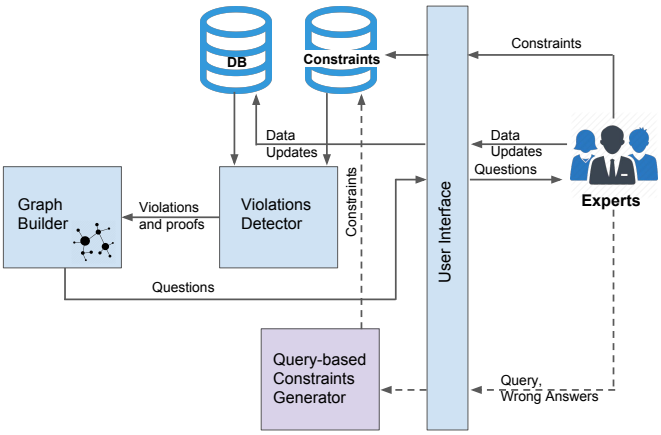


Figure 4: DANCE framework architecture.

result can be expressed as simple constraint violation in our formalism (we omit the translation algorithm for space constraints). We are thus able to compare the performance of DANCE to that of QOCO for solving the same problem. Since QOCO allows users only to add and delete tuples, whereas DANCE allows also tuple update, we also examine here a restricted variant of DANCE :

- DANCE v3: a simplified version of DANCE that does not include tuple updates

Interestingly, our experiments show that, even in the absence of constraints (and even without tuple updates), when applied to the same problem our algorithm requires fewer question compared to QOCO, as it better factors the dependencies between suspicious tuples.

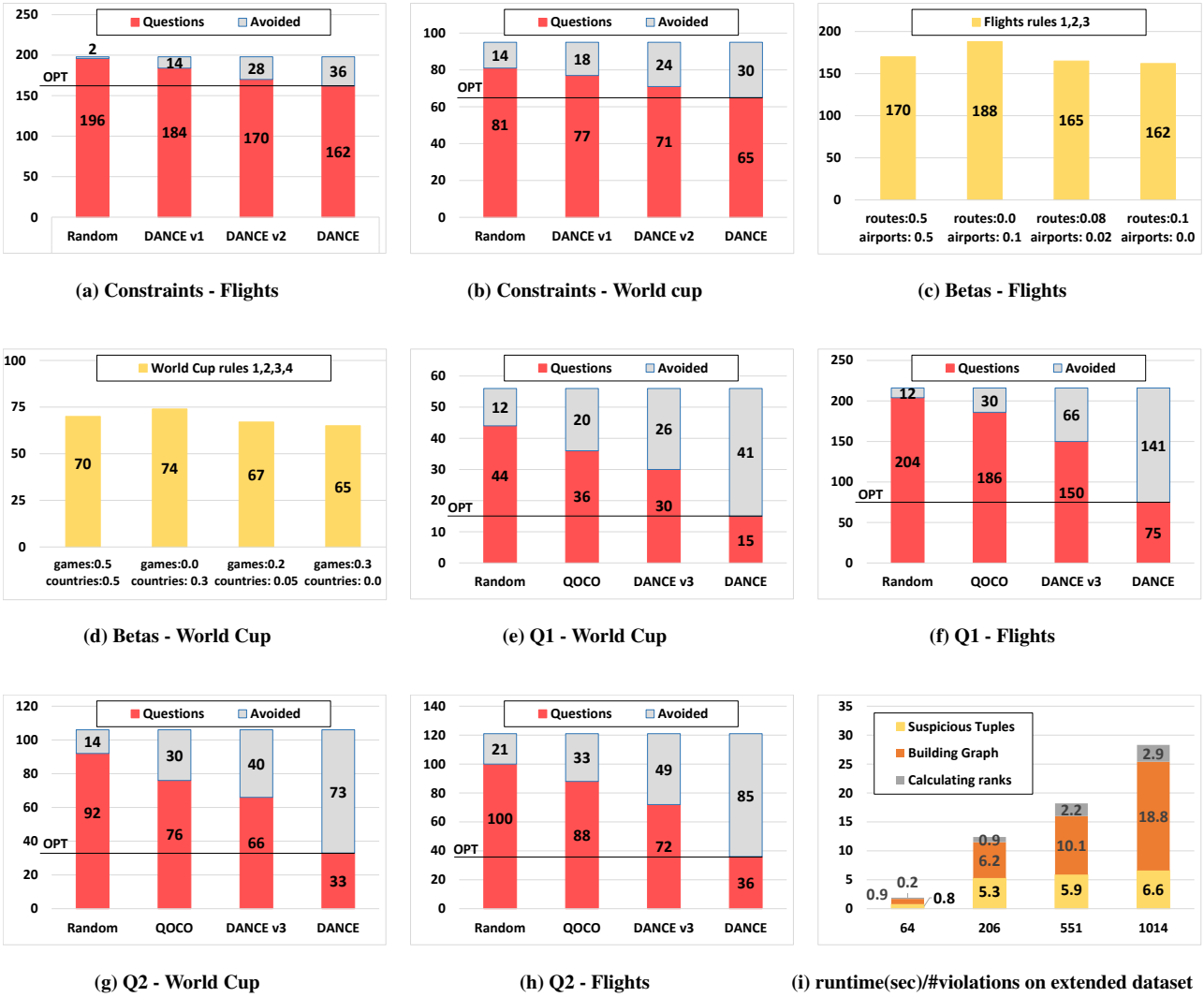
Datasets, Constraints and Queries. We consider three datasets. The first dataset is a soccer-related. It contains information about World Cup games, goals, players, teams, etc. and consists of around 5000 tuples. The teams and players relations are derived from the FIFA official data [1] and are thus assigned $\beta = 0$. The games relation is derived using automatic website scraping tools from sites such as [4, 5]. We first cleaned the database by comparing the games data with reference data from FIFA official data and used the cleaned database as our ground truth database, with the expert answers following this ground truth. Sampling the games data and comparing to the ground truth we derived an uncertainty measure β of 30%. We have experimented with various integrity constraints based on FIFA competition rules and show here the results for the following representative constraints, informally described below.

- φ_1^{WC}: If the winning and losing teams scores are not equal, then their penalties are equal to 0.
- φ_2^{WC}: If the winning and losing teams penalties are not equal, then their scores are equal.
- φ_3^{WC}: If the winning and losing teams penalties are equal, then the winning team score is bigger than the losing team.

For the comparison with QOCO we examine the following two queries and what it takes to remove wrong tuples from the result.

- Q_1^{WC}: All games between a country from Asian and any other country.
- Q_2^{WC}: All games of 'Round of 16' that ended without penalties.

Since QOCO does not exploit constraints, to make the comparison as "fair" as possible, we assume in this experiment no constraints (other than the assertions on the erroneous query answers).



(a) Constraints - Flights

(b) Constraints - World cup

(c) Betas - Flights

(d) Betas - World Cup

(e) Q1 - World Cup

(f) Q1 - Flights

(g) Q2 - World Cup

(h) Q2 - Flights

(i) runtime(sec)/#violations on extended dataset

Figure 5: Experimantal results

Our second dataset is a flights database from [3]. This database records information about flights all around the world, and was last updated on 2012. It contains data about flights (68K tuples is a routes relation), airports (8.2K tuples in an airports relation) and airlines (6.1K tuples in an airlines relation). We first cleaned the flights database by comparing the data with current reference data from Google flights website [2] and used it as our ground truth database, again, with the expert answers corresponding to this ground truth. By sampling the data and comparing to the ground truth we derived the uncertainty measures β for routes, airlines and airports to be 10%, 5% and 0% respectively. For our experiments we used the following three real-life constraints that follow from the fact that after 2012 there was a political conflict between Russia and each of Ukraine, Egypt and Turkey that caused the cancellation of the direct flights between that countries.

- φ_1^{Fl}: There are no direct flights between Russia and Ukraine.
- φ_2^{Fl}: There are no direct flights between Russia and Egypt.
- φ_3^{Fl}: There are no direct flights between Russia and Turkey.

For the comparison with QOCO we examine here the following two representative queries (again, assuming for fairness that no constraints are available for DANCE as well).

- Q_1^{Fl}: All direct flights from any country to China or to Greenland.
- Q_2^{Fl}: All direct flights from Russia or from USA to any country.

To test scalability of performance, we used a third dataset which was synthetically constructed by taking the flights databse mentioned above and replicating data (with variations) to achieve a 400K tuples dataset. Each tuple was replicated between 3 to 6 times, while generating new unique primary keys (airport ID for airports, pair of source and destination airport IDs for the flight, etc), by padding a number between 1-6 to the original key. The constraints used for the third dataset experiments are similar to the second dataset, with the addition of an extra constraint:

- φ_4^{Fl}: There are no 2 different airlines with same code.

. This constraint is added especially to stress-test the system, since it has many violations in the dataset.

Experiments. Our first experiment compares the performance of DANCE, in terms on the number of questions posed to the experts, to that of Random and the restricted variants DANCE v1 and v2. The results for the two datasets are depicted in Figures 5a and 5b. In both figures, each vertical bar corresponds to one of the algorithms. The height of each bar shows the maximal number of possible questions (the number of suspicious tuples at the beginning of the experiment). The lower part of the bar (in red) denotes the number of questions asked by the algorithm to fix all

violations. The remaining part represents the number of avoided questions (compared to the maximum). The horizontal (black) line indicates the number of questions the would have been asked by an optimal algorithm that knows the underlying ground truth and asks only about the actual erroneous tuples. In both cases Random shows the worst performance, then come DANCE v1, v2 and finally DANCE . This demonstrates the importance of the ingredients in our solution.

To better understand the results we examined the effect of choosing appropriate β values on the performance of our algorithms. The results of varying β values are depicted in Figures 5c and 5d. We can see that while the use of β values that reflect the uncertainty measure is useful, rough estimation suffices for obtaining good results.

Our second experiment compares the performance of DANCE, with and without tuples updates, to that of QOCO, for the queries listed above. To get a clearer perspective on the performance of the algorithm, we also add the results of Random for the same problems. The results are depicted in Figures 5e-5h. The bars for each of the algorithm have the same structure as in the first experiment and the horizontal black line indicates again the optimum. We can see that both variants of DANCE perform better than QOCO. This is interesting since the improved performance is achieved even without this use of additional constraints. (In the presence of constraints the gap grows. We omit the results or space constraints). We can also see that allowing users to update tuples, rather than only add and delete, results in fewer needed updates.

To conclude this section we examine running time of DANCE as a function of the number of suspicious tuples. We consider here the extended flights database which contains around 400K tuples. The results are shown in Figure 5i. To vary the number of suspicious tuple have run four experiments each with a different set of constraints (φ_1^{Fl} in the 1st bar of figure, φ_1^{Fl} and φ_2^{Fl} in the 2nd bar, $\varphi_1^{Fl} - \varphi_3^{Fl}$ in the 3rd, and all the four constraints in the last bar). For each experiment we have measured the run time of the main algorithm (Algorithm 2) in seconds from its start until finding the first question that will be posed to the expert. The number of suspicious tuples in each experiment (column) are depicted on the X axis. In each case we also detail the time spent on each part of the algorithm. As expected the time grows with the number of suspicious tuples, but in all our experiments was below 30 seconds. The iterations took just 1 to 4 seconds in all cases, due to our incremental graph maintenance, thus sufficiently fast to maintain an interactive user experience and work as a real time data cleaning system.

8 RELATED WORK

Data cleaning has attracted much attention in recent years. A large set of work focuses on fully-automated cleaning, using dedicated object similarity measures, probabilistic and statistical methods, and machine learning techniques [12, 24, 31, 35]. As mentioned in the Introduction a problem with automatic solutions is that they cannot ensure precision of the repairs since they do not have enough evidence about the ground truth and may in fact lead to wrong results [9]. It has thus been suggested to use domain experts to examine the data and choose which updates should be applied.

Multiple data cleaning tools leverage the crowd to assist in data cleaning (e.g. [9, 14, 16, 21, 30, 32, 36]), typically using the crowd to identify problematic spots in the data, e.g. by running queries and validating the results or by iteratively generating cleaning task for the crowd. [32] introduces the idea of cleaning only a sample of data to obtain unbiased query results with confidence intervals. [9] uses experts to identify errors in query answers and attempts to minimize the number of posed questions. However, as mentioned in Section 7, ignores the databases constraints and its performance is inferior than ours even in the absence of constraints. Our work complements these previous efforts by using the set of integrity constraints to identify data errors and to effectively use the crowd (domain experts in our case) by identifying the potentially most influential errors.

Several data cleaning tools employ integrity constraints in the cleaning process (e.g. [13, 15, 21, 23, 30, 31, 36]). Some of the papers (e.g. [19, 33]) rely on high quality reference data. Others are fully automatics (hence suffer from the problems mentioned above) and use predefined preferences among updates and/or minimal-repair strategy. When no unique update may be inferred from the available preferences, systems like [21] and [36] turn to experts to assist in the constraint resolution. But they not optimize the experts exploration of the possible updates space. Our work may be integrated into such systems to optimize the experts work in such scenarios. The authors of recent related research [30] propose a framework for detecting functional dependencies (FDs) violations. Their main focus is finding the (subset of) FDs that can detect the errors and studying different types of questions that can be asked from the experts under a limited budget (e.g. verifying if the proposed FD is correct) in order to detect the errors in the data. Our efforts are complimentary, since we are not focusing on identifying FDs and only detecting the data errors, but given a set of FDs we are trying to find and also fix underlying violations.

A complementary line of work employs data mining tools for outlier detection [20] to complete missing values, correct illegal values and identify duplicate records. Incorporating these tools in our framework to provide experts with autocorrelation-style suggestions for data correction is an interesting future research direction. Another data cleaning research [28] that deals with inconsistencies in databases introduces mining of forbidden itemsets, i.e. invalid value combinations in dirty data. This work is also complementary to ours as it tries to avoid forbidden itemsets, while in our settings we are trying to clean the data under a set of given integrity constrains.

The very recent [7] is the most related to our work. They study user-guided cleaning of Knowledge Bases w.r.t violations of tgds and a subset of denial constraints, called contradiction detecting dependencies. While they too handle tgds these include only equalities and they do not support cgds. This, together with the different data model, make the works incomparable but complementary.

Crowdsourcing, using ordinary users and domain experts, has been an active field of research in recent years. Crowdsourcing is employed for a a variety of cleaning-related tasks such as entity resolution [34], duplicate detection [10], schema matching [37], and filling up missing data [27]. Our system may be employed to resolve constraints violation in the database generated by these methods.

An related line of work deals with the development of algorithms to ensure answers quality, e.g. by aggregating multiple answers [29], or evaluating workers quality [22]. As mentioned, to simplify our presentation we assumed that there is a domain expert that has an extensive knowledge about the ground truth database. The above techniques can be used to aggregate multiple users answers.

9 CONCLUSIONS

We presented DANCE, a system that assists in the efficient resolution of integrity constraints violation. DANCE identifies the suspicious tuples whose update may contribute to the violation resolution, and builds a graph that captures the likelihood of an error in one tuple to occur and affect the other. PageRank style algorithm identifies the most beneficial tuples to ask about first. Incremental graph maintenance is used to assure interactive response time. Our experimental results on several different real-world datasets demonstrate the promise that DANCE is an effective and efficient tool for data cleaning.

There are several directions for future research. Supporting a richer constraint language, and in particular constraints on aggregations (i.e. a national team cannot have more then 23 active players in the World Cup) is challenging. Violations of such constraints may be corrected in multiple ways, hence it is interesting to investigate how to choose among them in the most efficient way. Also, in the current architecture, the constraints are assumed to be given as input. We plan to integrate our approach with mechanisms that infer additional constraints, or corrections to existing constraints, possibly also with the help of the experts, in order to improve the cleaning process. Parallel processing for speeding up the computation is another intriguing future direction.

REFERENCES

[1] Fifa official site. http://www.fifa.com/.

[2] Google flights. https://www.google.com/flights/.

[3] Open flights. http://openflights.org/data.html.

[4] Open football. https://github.com/openfootball/.

[5] World cup history. http://www.worldcup-history.com/.

[6] F. N. Afrati, C. Li, and V. Pavlaki. Data exchange in the presence of arithmetic comparisons. In *EDBT*, pages 487–498, 2008.

[7] A. Arioua and A. Bonifati. User-guided repairing of inconsistent knowledge bases. In *EDBT*, 2018.

[8] A. Assadi, T. Milo, and S. Novgorodov. DANCE: Data Cleaning with Constraints and Experts. In *ICDE*, pages 1409–1410, 2017.

[9] M. Bergman, T. Milo, S. Novgorodov, and W. C. Tan. Query-oriented data cleaning with oracles. In *SIGMOD*, pages 1199–1214, 2015.

[10] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *SIGKDD*, pages 39–48, 2003.

[11] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[12] F. Chiang and R. J. Miller. A unified model for data and constraint repair. In *ICDE*, pages 446–457, 2011.

[13] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *ICDE*, pages 458–469, 2013.

[14] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye. KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. In *SIGMOD*, pages 1247–1261, 2015.

[15] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, pages 315–326, 2007.

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

[17] P. K. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Incremental page rank computation on evolving graphs. In *WWW*, pages 1094–1095, 2005.

[18] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

[19] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *Proceedings of the VLDB Endowment*, 3(1-2):173–184, 2010.

[20] U. M. Fayyad. Mining databases: Towards algorithms for knowledge discovery. *IEEE Data Eng. Bull.*, 21(1):39–48, 1998.

[21] F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC data-cleaning framework. *PVLDB*, 6(9):625–636, 2013.

[22] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. HCOMP '10, pages 64–67, 2010.

[23] S. Kolahi and L. V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *ICDT*, pages 53–62, 2009.

[24] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg. Activeclean: interactive data cleaning for statistical modeling. *PVLDB*, 9(12):948–959, 2016.

[25] V. Kumar. Algorithms for constraint satisfaction problems: A survey. 1992.

[26] H. Park and J. Widom. Crowdfill: collecting structured data from the crowd. In *SIGMOD*, pages 577–588, 2014.

[27] J. Rammelaere, F. Geerts, and B. Goethals. Cleaning data with forbidden itemsets. In *ICDE*, pages 897–908, 2017.

[28] V. C. Raykar, S. Yu, L. H. Zhao, A. K. Jerebko, C. Florin, G. H. Valadez, L. Bogoni, and L. Moy. Supervised learning from multiple experts: whom to trust when everyone lies a bit. In *ICML*, pages 889–896, 2009.

[29] S. Thirumuruganathan, L. Berti-Equille, M. Ouzzani, J.-A. Quiane-Ruiz, and N. Tang. Uguide: User-guided discovery of fd-detectable errors. In *SIGMOD*, pages 1385–1397, 2017.

[30] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous data cleaning. In *ICDE*, pages 244–255, 2014.

[31] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. In *SIGMOD*, pages 469–480, 2014.

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

[33] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. *PVLDB*, 6(6):349–360, 2013.

[34] M. Yakout, L. Berti-Equille, and A. K. Elmagarmid. Don't be scared: use scalable automatic repairing with maximal likelihood and bounded changes. In *SIGMOD*, pages 553–564, 2013.

[35] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 4(5):279–289, 2011.

[36] C. J. Zhang, Z. Zhao, L. Chen, H. V. Jagadish, and C. C. Cao. Crowdmatcher: crowd-assisted schema matching. In *SIGMOD*, pages 721–724, 2014.