Query Rewriting using Datalog for Duplicate Resolution*

Jaffer Gardezi
University of Ottawa, SITE.
Ottawa, Canada
jgard082@uottawa.ca

Leopoldo Bertossi
Carleton University, SCS
Ottawa, Canada
bertossi@scs.carleton.ca

Abstract. Matching Dependencies (MDs) are a recent proposal for declarative entity resolution. They are rules that specify, given the similarities satisfied by values in a database, what values should be considered duplicates, and have to be matched. On the basis of a chase-like procedure for MD enforcement, we can obtain clean (duplicate-free) instances; actually possibly several of them. The clean answers to queries (which we call the resolved answers) are invariant under the resulting class of instances. In this paper, we investigate a query rewriting approach to obtaining the resolved answers (for certain classes of queries and MDs). The rewritten queries are specified in stratified Datalog^{not,s} with aggregation. In addition to the rewriting algorithm, we discuss the semantics of the rewritten queries, and how they could be implemented by means of a DBMS.

1 Introduction

For various reasons, databases may contain different coexisting representations of the same external, real world entity. This can occur, for example, because of errors or because the data comes from different sources using different formats. Those "duplicates" can be entire tuples or values within them. To obtain accurate information, in particular, query answers from the data, those tuples or values should be merged into a single representation.

Identifying and merging duplicates is a process called *entity resolution* (ER) [17, 23]. Matching dependencies (MDs) are a recent proposal for declarative duplicate resolution [24, 25]. An MD expresses, in the form of a rule, that if the values of certain attributes in a pair of tuples are similar, then the values of other attributes in those tuples should be matched (or merged) into a common value.

For example, the MD $R_1[X_1] \approx R_2[X_2] \rightarrow R_1[Y_1] \doteq R_2[Y_2]$ is a symbolic expression saying that, if an R_1-tuple and R_2-tuple have similar values for their attributes X_1, X_2, then their values for attributes Y_1, Y_2 should be made equal. This is a *dynamic* dependency, in the sense that its satisfaction is checked against a pair of instances: the first one where the antecedent holds, and the second one where the identification of values takes place. This semantics of MDs was sketched in [25].

In this paper we use a refinement of that original semantics that was put forth in [30] (cf. also [31]). It improves wrt the latter in that it disallows changes that are irrelevant to

* Research supported by the NSERC Strategic Network on Business Intelligence (BIN ADC05) and NSERC/IBM CRDPJ/371084-2008.

the duplicate resolution process. Actually, [30] goes on to define the clean versions of the original database instance D_0 that contains duplicates. They are called the *resolved instances* (RIs) of D_0 wrt the given set M of matching dependencies. A resolved instance is obtained as the fixed point of a chase-like procedure that starts from D_0 and iteratively applies or enforces the MDs from M. Each step of this chase generates a new instance by making equal values that are identified as duplicates by the MDs.

In [30] it was shown that resolved instances always exist, and that they have certain desirable properties. For example, the set of allowed changes is just restrictive enough to prevent irrelevant changes, while still guaranteeing existence of resolved instances. The resolved instances that minimize the *overall number* of attribute value changes (associated to a same tuple identifier) wrt the original instance are called *minimally resolved instances* (MRIs). On this basis, given a query $\mathcal{Q}$ posed to a database instance D_0 that may contain duplicates, we define the *resolved answers* wrt Σ as the query answers that are true of all the minimally resolved instances [30].

The concept of resolved query answer has similarities to that of *consistent query answer* (CQA) in a database that fails to satisfy a set of integrity constraints [4, 10, 12]. The consistent answers are invariant under the *repairs* of the original instance. However, data cleaning and CQA are different problems. For the former, we want to compute a clean instance, determined by MDs; for the latter, the goal is obtaining semantically correct query answers. MDs are not (static) ICs. In principle, we could see clean instances as repairs, treating MDs similarly to static FDs. However, the existing repair semantics do not capture the matchings as dictated by MDs (cf. [30, 31] for a more detailed discussion).

In this paper, we investigate the *problem of computing the resolved answers*, simply called *resolved answer problem* (RAP). The motivation for addressing this problem is that even in a database instance containing duplicates, much or most of the data may be duplicate-free. One can therefore obtain useful information from the instance without having to perform data cleaning on the instance. This would be convenient if the user does not want, or cannot afford, to go through a data cleaning process. In other situations the user may not have write access to the data being queried, or any access to the data sources, as in virtual data integration systems [35, 13].

In [29] we identified classes of MDs and conjunctive queries for which RAP can be solved in polynomial time in data complexity. Furthermore, a recursively-defined predicate was introduced for identifying the sets of duplicate values within a database instance. This predicate can be combined with a query, opening the ground for a query rewriting approach to RAP.

In this paper we present a *query rewriting methodology for the RAP problem* (for the identified classes of MDs and queries). It can be used to rewrite the original query $\mathcal{Q}$ into a new query $\mathcal{Q}'$, in such a way that the latter, posed as usual to the original instance D_0, returns the resolved answers to the original query. More precisely, we make the following contributions:

1. We show that queries $\mathcal{Q}$ (in a restricted but broad class of conjunctive queries) that expect to obtain resolved answers from a given dirty database D can be rewritten into a (non-disjunctive) recursive Datalog^{not,s} query $\mathcal{Q}'$ with stratified negation and aggregation. $\mathcal{Q}'$ posed to D returns the answers to $\mathcal{Q}$.

As expected, such a query can be computed in polynomial time in the size of the initial database. The recursion arises from the fact that identifying duplicate values requires computing the transitive closure of binary similarity operators. Transitivity is not assumed for similarity operators, and in fact, common similarity relations used in practice, such as those based on the edit distance and related string similarity metrics, are not transitive. Aggregation is needed to enforce the minimality constraint, since this involves finding the frequency of occurrence of values within a set of duplicates.

2. We analyze the queries resulting from the rewriting mechanism in terms of the combination of aggregation and recursion. We discuss semantic issues of these queries, their runtime, and their implementability on top of a DBMS.

To the best of our knowledge, this is the first result on query rewriting in the context of MD-based entity resolution. Furthermore, our rewritings into Datalog are non-trivial, in the sense that they are not the result of translations into Datalog of first-order rewritings. Our rewriting uses in an essential manner the elements of the resulting Datalog queries, namely recursion and aggregation. It is worth mentioning that the polynomial-time rewritings for conjunctive queries proposed for consistent query answering have been all been first-order (FO) [4, 20, 28, 37].

On the other hand, the general answer set programs that have been proposed as *repair programs* [5, 7, 33, 22, 19], that specify database repairs and can be used for highly expressive query rewritings, have a higher expressive power and complexity than Datalog programs with stratified negation and aggregation.[1] The attempts in [11] to obtain lower complexity programs for CQA from repair programs for some tractable classes of queries and constraints led back to FO rewritings. Thus classical, i.e. non-disjunctive and stratified, Datalog was missed as an "intermediate" language for CQA.

This paper is organized as follows. In Section 2 we introduce basic concepts and notation on MDs. In Section 3, we define the important concepts used in this paper, in particular, (minimally) resolved instances and resolved answers to queries. Section 4 contains the main results of this paper, that includes a query rewriting algorithm for a special case of the resolved query answer problem. Section 5 concludes the paper and discusses related and future work. In the Appendix, we consider possible implementations of the queries rewritten in Datalog. Proofs of results can be found in [29].

2 Preliminaries

We consider a relational schema $\mathcal{S}$ that includes an enumerable, possibly infinite domain U, and a finite set $\mathcal{R}$ of database predicates. Elements of U are represented by lower case letters near the beginning of the alphabet. $\mathcal{S}$ determines a first-order (FO) language $L(\mathcal{S})$. An instance D for $\mathcal{S}$ is a finite set of ground atoms of the form $R(\bar{a})$, with $R \in \mathcal{R}$, say of arity n, and $\bar{a} \in U^n$. $R(D)$ denotes the extension of R in D. Every predicate $R \in \mathcal{S}$ has a set of attribute, denoted $attr(R)$. As usual, we sometimes refer to attribute A of R by $R[A]$. We assume that all the attributes of a predicate are different, and that we can identify attributes with *positions* in predicates, e.g. $R[i]$, with

[1] Under the common assumption that the polynomial hierarchy does not collapse.

$1 \leq i \leq n$. If the ith attribute of predicate R is A, for a tuple $t = (c_1, \ldots, c_n) \in R(D)$, $t_R^D[A]$ (usually, simply $t_R[A]$ or $t[A]$ if the instance is understood) denotes the value c_i. For a sequence $\bar{A}$ of attributes in $attr(R)$, $t[\bar{A}]$ denotes the tuple whose entries are the values of the attributes in $\bar{A}$. Attributes have and may share subdomains of U.

In the rest of this section, we summarize some of the assumptions, definitions, notation, and results from two previous papers, [30] and [29], that we will need.

We will assume that every relation in an instance has an auxiliary attribute, a surrogate key, holding values that act as *tuple identifiers*. Tuple identifiers are never created, destroyed or changed during the duplicate resolution process. They do not appear in MDs, and are used to identify different versions of the same original tuple that result from the matching process. We usually leave them implicit; and "tuple identifier attributes" are commonly left out when specifying a database schema. However, when explicitly represented, they will be the "first" attribute of the relation. For example, if $R \in \mathcal{R}$ is n-ary, $R(t, c_1, \ldots, c_n)$ is a tuple with id t, and is usually written as $R(t, \bar{c})$. We usually use the same symbol for a tuple's identifier as for the tuple itself. Tuple identifiers are unique over the entire instance.

Two instances over the same schema that share the same tuple identifiers are said to be *correlated*. In this case it is possible to unambiguously compare their tuples, and as a result, also the instances.

As expected, some of the attribute domains, say A, have a built-in binary similarity relation $\approx_A$. That is, $\approx_A \subseteq Dom(A) \times Dom(A)$. It is assumed to be reflexive and symmetric. Such a relation can be extended to finite lists of attributes (or domains therefor), componentwise. For single attributes or lists of them, the similarity relation is is generically denoted with $\approx$.

A *matching dependency* (MD) [24], involving predicates R, S, is an expression (or rule), m, of the form

$$m: \; R[\bar{A}] \approx S[\bar{B}] \; \rightarrow \; R[\bar{C}] \doteq S[\bar{E}], \tag{1}$$

with $\bar{A} = (A_1, ..., A_k)$, $\bar{C} = (C_1, ..., C_{k'})$ lists of (different) attributes from $attr(R)$; and $\bar{B} = (B_1, ..., B_k)$, $\bar{E} = (E_1, ..., E_{k'})$ lists of attributes from $attr(S)$.[2]

The set of attributes on the left-hand-side (LHS) of the arrow in m is denoted with $LHS(m)$. Similarly for the right-hand-side (RHS).

In relation to (1), the attributes in a *corresponding pair* (A_i, B_i) or (C_i, E_i) are assumed to share a common domain; and in particular, a *similarity relation* $\approx_i$. In consequence, the condition on the LHS of (1) means that, for a pair of tuples t_1 in R and t_2 in S, $t_1[A_i] \approx_i t_2[B_i]$, $1 \leq i \leq k$. Similarly, the expression on the RHS means $t_1[A_i] \doteq t_2[B_i]$, $1 \leq i \leq k'$. Here, $\doteq$ means that the values should be updated to the same value.

Accordingly, the intended semantics of the MD in (1) is that, for an instance D, if any pair of tuples, $t_1 \in R(D)$ and $t_2 \in S(D)$, satisfy the similarity conditions on the LHS, then for the same tuples (or tuple ids), the attributes on the RHS have to take the same values [25], possibly through updates that may lead to a new version of D.

We assume that all sets M of MDs are in *standard form*, i.e. for no two different MDs $m_1, m_2 \in M$, $LHS(m_1) = LHS(m_2)$. All sets of MDs can be put in this form.

[2] We assume that the MDs are defined in terms of the same schema $\mathcal{S}$.

MDs in a set M can interact in the sense that a matching enforced by one of them may create new similarities that lead to the enforcement of another MD in M. This intuition is captured through the *MD-graph*.

Definition 1. [31] Let M be a set of MDs in standard form. The *MD-graph* of M, denoted $MDG(M)$, is a directed graph with a vertex m for each $m \in M$, and an edge from m to m' iff $RHS(m) \cap LHS(m') \neq \emptyset$.[3] If $MDG(M)$ contains edges, M is called *interacting*. Otherwise, it is called *non-interacting* (NI). □

3 Matching Dependencies and Resolved Answers

Updates as prescribed by an MD m are not arbitrary. The updates based on m have to be justified by m, as captured through the notion of *modifiable value* in an instance.

Definition 2. Let D be an instance, M a set of MDs, and $\mathcal{P}$ be a set of pairs (t, G), where t is a tuple of D and G is an attribute of t. (a) For a tuple $t_R \in R(D)$ and C an attribute of R, the value $t_R^D[C]$ is *modifiable wrt* $\mathcal{P}$ if there exist $S \in \mathcal{R}$, $t_S \in S(D)$, an $m \in M$ of the form $R[\bar{A}] \approx S[\bar{B}] \rightarrow R[\bar{C}] \doteq S[\bar{E}]$, and a corresponding pair (C, E) of $(\bar{C}, \bar{E})$ in m, such that $(t_S, E) \in \mathcal{P}$ and one of the following holds:

1. $t_R[\bar{A}] \approx t_S[\bar{B}]$, but $t_R[C] \neq t_S[E]$.
2. $t_R[\bar{A}] \approx t_S[\bar{B}]$ and $t_S[E]$ is modifiable wrt $\mathcal{P} \setminus \{(t_S, E)\}$.

(b) The value $t_R^D[C]$ is *modifiable* if it is modifiable wrt $\mathcal{V} \setminus \{(t_R, C)\}$, where $\mathcal{V}$ is the set of all pairs (t, G) with t a tuple of D and G an attribute of t. □

Definition 2 is recursive. The base case occurs when either case 1 applies (with any $\mathcal{P}$) or when there is no tuple/attribute pair in $\mathcal{P}$ that can satisfy part (a). Notice that recursion must terminate eventually, since the latter condition must be satisfied when $\mathcal{P}$ is empty, and each recursive call reduces the size of $\mathcal{P}$.

Example 1. Consider $m\colon R[A] \approx R[A] \rightarrow R[B] \doteq R[B]$ on schema $R[A, B]$, and the following instance. Assume that the only non-trivial similarities are $a_1 \approx a_2 \approx a_3$ and $b_1 \approx b_2$. Since $a_2 \approx a_3$ and $c_1 \neq c_3$, $t_2[B]$ and $t_3[B]$ are modifiable (base case). With case 2 of Definition 2, since $a_1 \approx a_2$, and $t_2[B]$ is modifiable, we obtain that $t_1[B]$ is modifiable.

$R(D)$	A	B
t_1	a_1	c_1
t_2	a_2	c_1
t_3	a_3	c_3
t_4	b_1	c_3
t_5	b_2	c_3

For $t_5[B]$ to be modifiable, it must be modifiable wrt $\{(t_i, B) \mid 1 \leq i \leq 4\}$, and via t_4. According to case 2 of Definition 2, this requires $t_4[B]$ to be modifiable wrt $\{(t_i, B) \mid 1 \leq i \leq 3\}$. However, this is not the case since there is no t_i, $1 \leq i \leq 3$, such that $t_4[A] \approx t_i[A]$. Therefore $t_5[B]$ is not modifiable. A symmetric argument shows that $t_4[B]$ is not modifiable. □

[3] That is, they share at least one corresponding pair of attributes.

Definition 3. [30] Let D, D' be correlated instances, and M a set of MDs. (D, D') *satisfies* M, denoted $(D, D') \models M$, iff: 1. For any pair of tuples $t_R \in R(D)$, $t_S \in S(D)$, if there exists an $m \in M$ of the form $R[\bar{A}] \approx S[\bar{B}] \rightarrow R[\bar{C}] \doteq S[\bar{E}]$ and $t_R[\bar{A}] \approx t_S[\bar{B}]$, then for the corresponding tuples (i.e. with same ids) $t'_R \in R(D')$ and $t'_S \in S(D')$, it holds $t'_R[\bar{C}] = t'_S[\bar{E}]$.
2. For any tuple $t_R \in R(D)$ and any attribute G of R, if $t_R[G]$ is *non-modifiable*, then $t'_R[G] = t_R[G]$. □

Intuitively, D' in Definition 3 is a new version of D that is produced after a single update. Since the update involves matching values (i.e. making them equal), it may produce "duplicate" tuples, i.e. that only differ in their tuple ids. They would possibly be merged into a single tuple in the a data cleaning process. However, we keep the two versions. In particular, D and D' have the same number of tuples. Keeping or eliminating duplicates will not make any important difference in the sense that, given that tuple ids are never updated, two duplicates will evolve in exactly the same way as subsequent updates are performed. Duplicate tuples will never be subsequently "unmerged".

This definition of MD satisfaction departs from [25], which requires that updates preserve similarities. Similarity preservation may force undesirable changes [30]. The existence of the updated instance D' for D is guaranteed [30]. Furthermore, wrt [25], our definition does not allow unnecessary changes from D to D'. Definitions 2 and 3 imply that only values of attributes that appear to the right of the arrow in some MD are subject to updates. Hence, they are called *changeable attributes*.[4]

Definition 3 allows us to define a *clean instance* wrt M as the result of a chase-like procedure, each step being satisfaction preserving.

Definition 4. [30] (a) A *resolved instance* (RI) for D wrt M is an instance D', such that there are instances $D_1, D_2, ...D_n$ with: $(D, D_1) \models M$, $(D_1, D_2) \models M$,..., $(D_{n-1}, D_n) \models M$, $(D_n, D') \models M$, and $(D', D') \models M$. We say D' is *stable*. (b) D' is a *minimally resolved instance* (MRI) for D wrt M if it is a resolved instance and it minimizes the overall number of attribute value changes wrt D and in relation with the same tuple ids. (c) $MRI(D, M)$ denotes the class of MRIs of D wrt M. □

Example 2. Consider the MD $R[A] \approx R[A] \rightarrow R[B] \doteq R[B]$ on predicate R, and the instance D. It has several resolved instances, among them, four that

$R(D)$	A	B
t_1	a_1	c_1
t_2	a_1	c_2
t_3	b_1	c_3
t_4	b_1	c_4

$R(D_1)$	A	B
t_1	a_1	c_1
t_2	a_1	c_1
t_3	b_1	c_3
t_4	b_1	c_3

$R(D_2)$	A	B
t_1	a_1	c_1
t_2	a_1	c_1
t_3	b_1	c_1
t_4	b_1	c_1

minimize the number of changes. One of them is D_1. A resolved instance that is not minimal in this sense is D_2. □

It can be shown that an RI exists for any instance [30]. It follows immediately that every instance has an MRI.

[4] Not to be confused with "modifiability", that applies to tuples.

In this work, as in [30, 31], we are investigating what we could call "the pure case" of MD-based entity resolution. It adheres to the original semantics outlined in [25], which does not specify how the matchings are to be done, but only which values must be made equal. That is, the MDs have implicit existential quantifiers (for the values in common). The semantics we just introduced formally captures this pure case. We find situations like this in other areas of data management, e.g. with referential integrity constraints, tuple-generating dependencies in general [1], schema mappings in data exchange [8], etc. A non-pure case that uses matching functions to realize the matchings as prescribed by MDs is investigated in [15, 16, 6].

The *resolved* answers to a query are *certain* for the class of MRIs for D wrt M.

Definition 5. [30] Let $\mathcal{Q}(\bar{x})$ be a query expressed in the first-order language $L(\mathcal{S})$ associated to schema $\mathcal{S}$ of an instance D. A tuple of constants $\bar{a}$ from U is a *resolved answer* to $\mathcal{Q}(\bar{x})$ wrt the set M of MDs, denoted $D \models_M \mathcal{Q}[\bar{a}]$, iff $D' \models \mathcal{Q}[\bar{a}]$, for every $D' \in MRI(D, M)$. We denote with $ResAn(D, \mathcal{Q}, M)$ the set of resolved answers to $\mathcal{Q}$ from D wrt M. □

Example 3. (example 1 continued) Since the only MRI for the original instance D is $R(D') = \{\langle t_1, a_1, c_1\rangle, \langle t_2, a_2, c_1\rangle, \langle t_3, a_3, c_1\rangle, \langle t_4, b_1, c_3\rangle, \langle t_5, b_2, c_3\rangle\}$, the resolved answers to the query $\mathcal{Q}(x, y)$: $R(x, y)$ are $\{\langle a_1, c_1\rangle, \langle a_2, c_1\rangle, \langle a_3, c_1\rangle, \langle b_1, c_3\rangle, \langle b_2, c_3\rangle\}$. □

The problem of deciding whether or not $\bar{a} \in ResAn(D, \mathcal{Q}, M)$ for a given tuple $\bar{a}$ is called the *resolved answer* decision problem.

4 Query Rewriting for Resolved Answers

In this section, we present a query rewriting method for retrieving the resolved answers for certain classes of queries and sets of MDs. We provide an intuitive and informal presentation of the rewritten queries. For precise details and a proof of correctness, see [29].

It has been shown in previous work that the resolved answer decision problem is generally intractable [30, 31, 29]. However, there are tractable cases of the problem that are practically relevant [29]. Two of those cases are considered here: that of *non-interacting* (NI) sets of MDs (cf. Definition 1), and that of *hit-set-cyclic* (HSC) sets of MDs, that we now define.

Definition 6. A set M of MDs is *hit-simple-cyclic* iff the following hold: (a) In all MDs in M and in all their corresponding pairs, the two attributes (and predicates) are the same. (b) In all MDs $m \in M$, at most one attribute in $LHS(m)$ is changeable. (c) Each vertex v_1 in $MDG(M)$ is on at least one cycle, or there is a vertex v_2 on a cycle with at least two vertices such that there is an edge from v_1 to v_2. □

Example 4. For schema $R[A, C, F, G]$, consider the following set M of MDs:

$$m_1\colon\ R[A] \approx R[A] \ \rightarrow\ R[C, F, G] \doteq R[C, F, G],$$
$$m_2\colon\ R[C] \approx R[C] \ \rightarrow\ R[A, F, G] \doteq R[A, F, G].$$

Set M obviously satisfies (a) and (b) of Definition 6. Also, $MDG(M)$ consists of a single cycle through the two vertices, so M satisfies (c). M is then HSC.

Predicate R subject to the given M has two "keys", $R[A]$ and $R[C]$. Such relations are common in practice. For example, R may be used in a database about people: $R[A]$ could be used for the person's name, $R[C]$ the address, and $R[F]$ and $R[G]$ for non-distinguishing information, e.g. gender and age. □

HSC sets have properties similar to those of NI sets wrt the resolved answer problem [29]. For both classes, the value positions identified as duplicates are the same for all MRIs, and they are characterized through the equivalence classes of the *tuple-attribute closure*, which we now define.

Definition 7. [29] Let $M = \{m_i \mid i = 1, \ldots, n\}$ be a set of MDs, with $m_i\colon\ R_i[\bar{A}_i] \approx_i S_i[\bar{B}_i] \ \rightarrow\ R_i[\bar{C}_i] \doteq S_i[\bar{E}_i]$. (a) The *previous set* of m_i, denoted $PS(m_i)$, is the set of all MDs $m_j \in M$ with a path in $MDG(M)$ from m_j to m_i. (b) For an instance D, and tuples ids t_1, t_2 for R, S, resp. (i.e. ids for tuples $t_1 \in R(D), t_2 \in S(D)$): $(t_1, C_i) \approx' (t_2, E_i) \ :\Longleftrightarrow\ t_1[\bar{A}_j] \approx_j t_2[\bar{B}_j]$, where (C_i, E_i) is a corresponding pair of $(\bar{C}_i, \bar{E}_i)$ in m_i and $m_j \in PS(m_i)$. (c) The *tuple-attribute closure* (TA closure) of M wrt D, denoted $TA^{M,D}$, is the reflexive, symmetric, and transitive closure of $\approx'$. □

Example 5. (example 4 continued) In this case, $PS(m_1) = PS(m_2) = \{m_1, m_2\}$. Consider the instance D, where the only similarities are: $a_i \approx a_j,\ b_i \approx b_j, d_i \approx d_j,$

$R(D)$	A	B
t_1	a_1	d_1
t_2	a_2	e_2
t_3	b_1	e_1
t_4	b_2	d_2

$e_i \approx e_j$, with $i, j \in \{1, 2\}$. The relations $(t_{i mod 4+1}, A) \approx' (t_{(i+1) mod 4+1}, A)$ and $(t_{i mod 4+1}, B) \approx' (t_{(i+1) mod 4+1}, B)$, $0 \leq i \leq 3$, hold. The TA closure is given by $\{TA(t_i, x, t_j, x) \mid 1 \leq i, j \leq 4,\ x \in \{A, B\}\}$. Notice that this relation involves just tuple ids and attributes. However, it depends on D through the similarity conditions in (b) in the definition. □

For the set of MDs as in Definition 7, the TA closure can be specified by Datalog rules. The database predicates in them have a first argument (attribute) to explicitly represent the tuple id. More precisely, for $1 \leq i \leq n$, for each corresponding pair (C, E) of $(\bar{C}_i, \bar{E}_i)$, and for each $m_j \in PS(m_i)$, we have the rule:[5]

$$(t_1, R_i[C]) \approx' (t_2, S_i[E]) \ \leftarrow\ R_i(t_1, \bar{x}), S_i(t_2, \bar{y}), t_1[\bar{A}_j] \approx_j t_2[\bar{B}_j].$$

Additionally, for all attributes A of R_i and ids t of tuples in R_i, we have

$$TA(t, A, t, A) \leftarrow R_i(t, \bar{x}); \tag{2}$$

similarly for S_i. For arbitrary tuple ids t_1, t_2, and t_3, and attributes A, B, and C,

$$TA(t_1, A, t_2, B) \ \leftarrow\ TA(t_2, B, t_1, A), \tag{3}$$

$$TA(t_1, A, t_2, B) \ \leftarrow\ (t_1, A) \approx' (t_2, B), \tag{4}$$

$$TA(t_1, A, t_3, C) \ \leftarrow\ TA(t_1, A, t_2, B), (t_2, B) \approx' (t_3, C). \tag{5}$$

[5] Remember that the first argument in R_i, S_i stands for the tuple id.

Rules (4) and (5) express that TA is the transitive closure of relation $\approx'$. Rules (2) and (3), that TA is reflexive and symmetric. A related concept is the one of *attribute closure*.

Definition 8. [31] Let M be a set of MDs on schema $\mathcal{S}$. (a) The symmetric binary relation $\doteq_r$ relates attributes $R[A]$, $S[B]$ of $\mathcal{S}$ whenever there is an MD m in M where $R[A] \doteq S[B]$ appears in $RHS(m)$. (b) The *attribute closure* of M is the reflexive, symmetric, transitive closure of $\doteq_r$. (c) $E_{R[A]}$ denotes the equivalence class of attribute $R[A]$ in the attribute closure of M. □

Example 6. Let M be the set of MDs

$$R[A] \approx_1 S[B] \rightarrow R[C] \doteq S[D],$$
$$S[E] \approx_2 T[F] \wedge S[G] \approx T[H] \rightarrow S[D, K] \doteq T[J, L],$$
$$T[F] \approx_3 T[H] \rightarrow T[L, N] \doteq T[M, P].$$

The equivalence classes of T_{at} are $E_{R[C]} = \{R[C], S[D], T[J]\}$, $E_{S[K]} = \{S[K],\ T[L], T[M]\}$, and $E_{T[N]} = \{T[N], T[P]\}$. □

It is easy to show that if a pair (u_1, A), (u_2, B) is in the same equivalence class of tuple-attribute closure, then A and B must be in the same equivalence class of attribute closure.

Definition 9. Let M be a set of MDs and D and instance and a a data value. For an equivalence class E of $TA^{M,D}$, the *frequency of a in E* is the quantity $freq^D(a, E) := |\{(t, A) \mid (t, A) \in E,\ t[A] = a \text{ in } D\}|$. □

Proposition 1. [29] For M an NI or HSC set of MDs, and D an instance, each MRI for D wrt M is obtained by setting, for each equivalence class E of $TA^{M,D}$, all the values for $t[A]$, with $(t, A) \in E$, to a value a that maximizes $freq^D(a, E)$. □

Example 7. (example 5 continued) The two equivalence classes of TA closure are $E_1 = \{(t_i, A) \mid 1 \leq i \leq 4\}$ and $E_2 = \{(t_i, B) \mid 1 \leq i \leq 4\}$. All values in the A (B) column of the table have frequency 1 in E_1 (E_2). Thus, there are 16 MRIs, obtained by setting all values in each column to a common value chosen from those in the column. □

Now we turn to resolved query answering. Proposition 1 tells us that the minimally resolved instances for an instance D can be obtained by identifying most frequently occurring values. Thus, resolved query answers from D can be computed by imposing this requirement on the original query. As a consequence, the rewritten queries will become aggregate queries.

In Datalog notation, aggregate queries take the form $P(\bar{a}, \bar{x}, Agg(\bar{u})) \leftarrow B(\bar{y})$, where P is answer collecting predicate, the body $B(\bar{y})$ represents a conjunction of literals all of whose variables are among those in $\bar{y}$, $\bar{a}$ is a list of constants, $\bar{x} \cup \bar{u} \subseteq \bar{y}$, and Agg is an aggregate operator such as $Count$ or Sum. The variables $\bar{x}$ are the "group-by" variables. That is, for each fixed value $\bar{b}$ for $\bar{x}$, aggregation is performed over all tuples that make $B\frac{\bar{x}}{\bar{b}}$, the instantiation of B on $\bar{b}$ for $\bar{x}$, true. $Count(\bar{u})$ counts the number of distinct values of $\bar{u}$, while $Sum(\bar{u})$ sums over all $\bar{u}$, whether distinct or not.

Our query rewriting methodology for computing resolved answers will be applicable to a certain class of conjunctive queries, the called UJCQ queries defined below. In [29] a counterexample for the general applicability to all conjunctive queries is given.

Definition 10. [29] For a set M of MDs, a conjunctive query $\mathcal{Q}$ without built-ins is an *unchangeable join conjunctive query* (UJCQ query) if there are no existentially quantified variables in a join in $\mathcal{Q}$ in the position of a changeable attribute. For fixed, M, $UJCQ$ denotes this class of queries. □

In the rest of this paper we assume that the we have a fixed and finite set M of MDs that satisfies the hypothesis of Proposition 1. The queries posed to the initial, possibly non-resolved instance belong to $UJCQ$.

The rewritten queries will be in Datalog^{not,s} [1], i.e. Datalog queries with stratified negation and aggregation, and the built-ins $\neq$ and $\leq$. For simplicity, the rewriting makes use of tuple identifiers only. In the absence of such a surrogate key, whole tuples could be used instead of identifiers.

Given a $UJCQ$ query $\mathcal{Q}$, with answer predicate Q,

$$Q(\bar{x}) \ \leftarrow \ R_1(\bar{v}_1), R_2(\bar{v}_2), ..., R_n(\bar{v}_n), \tag{6}$$

the rewritten query $\mathcal{Q}'$ is the conjunction of the rewritings Q_i of each of the R_i, to be given in (8) below, i.e.

$$Q'(\bar{x}) \ \leftarrow \ Q_1(\bar{v}_1), Q_2(\bar{v}_2), ..., Q_n(\bar{v}_n). \tag{7}$$

Now, for a fixed atom $R_i(\bar{v}_i)$ in (6), let C be the set of changeable attributes corresponding to a free variable in $\bar{v}_i$, i.e. also appearing in $Q(\bar{x})$. We denote the list of such variables by $\bar{v}_C$.

If C is empty, then its rewriting becomes $Q_i(\bar{v}_i) \leftarrow R_i(\bar{v}_i)$. Intuitively, this is because, by Definition 10, only attributes corresponding to free variables can participate in joins, so changes to values of attributes corresponding to bound variables cannot affect satisfaction of the body in (6).

Suppose C is non-empty, and consider $R_i[A] \in C$. From Proposition 1, deciding whether or not all MRIs have the same value v for $R_i[A]$ for a given tuple id t will involve finding the frequency of v in E for the equivalence class E of the TA closure to which $(t, R_i[A])$ belongs. We introduce aggregation operators to express this count. The values to be counted are values for attributes in $E_{R_i[A]}$ (cf. the remark following Example 6).

We introduce a predicate $C^{R_i[A]}$, with an attribute at the start of its attribute list whose value is the attribute in $E_{R_i[A]}$ over whose values aggregation is performed. For an attribute A and list of variables $\bar{v}$, we denote with v_A the variable holding the value for A. For each $S[B] \in E_{R_i[A]}$, we have the rule

$$C^{R_i[A]}(S[B], t_1, v_{S[B]}, Count(t_2)) \ \leftarrow \ TA(t_1, R_i[A], t_2, S[B]), R_i(t_1, \bar{u}), S(t_2, \bar{v}),$$

in which all predicate arguments are variables except for the attribute labels $S[B]$ and $R_i[A]$, that are constants.

In each tuple in the head predicate of the above expression, the value of the $Count$ expression is $|\{t \mid (t, S[B]) \in E,\ t[S[B]] = v_{S[B]}\}|$, where E is the equivalence class of the TA closure to which $(t_1, R_i[A])$ belongs.

To find the frequency of the value of $v_{S[B]}$ in E, this count must be extended to all attributes in $E_{R_i[A]}$. We introduce the predicate $Total^{R_i[A]}$ for this purpose:

$$Total^{R[A_i]}(t, v, Sum(z)) \ \leftarrow \ C^{R_i[A]}(x, t, v, z).$$

Tuples in $Total^{R_i[A]}$ specify in their last argument the frequency of v in the equivalence class of the TA-closure to which $(u, R[A_i])$ belongs.

To compare these aggregate quantities for different values of v, we use the $Compare$ predicate:

$$Compare^{R[A_i]}(t, v) \ \leftarrow \ Total^{R_i[A]}(t, v, z_1),\ Total^{R_i[A]}(t, v', z_2), z_1 \leq z_2, v' \neq v.$$

Tuples in $Compare^{R_i[A]}$ consist of a tuple identifier t in R_i and a value v for attribute $R_i[A]$. For such a pair (t, v) there is another value v' whose frequency in the equivalence class of the TA closure to which $(t, R_i[A])$ belongs is at least as large as that of v.

In order for a value to be a "certain" for a given attribute $R_i[A]$ in a given tuple, the tuple and value must not occur as a tuple in $Compare^{R_i[A]}$. Let $\bar{v}_i'$ be $\bar{v}_i$ with all variables in $\bar{v}_C$ replaced with new variables. Then,

$$Q_i(\bar{v}_i) \ \leftarrow \ R_i(t, \bar{v}_i'), \bigwedge_{R_i[A] \in C} not\ Compare^{R_i[A]}(t, v_{R_i[A]}), \\ Total^{R_i[A]}(t, v_{R_i[A]}, z). \tag{8}$$

Example 8. Consider the schema $R[ABC], S[EFG], U[HI]$ with non-interacting MDs:

$$R[A] \approx S[E] \rightarrow R[B] \doteq S[F], \\ S[E] \approx U[H] \rightarrow S[F] \doteq U[I];$$

and the $UJCQ$ query:
$Q(x, y, z) \leftarrow R(x, y, z), S(u, v, z), U(p, q).$

Since the S and U atoms have no free variables holding the values of changeable attributes, they remain unchanged. Therefore, the rewritten query $\mathcal{Q}'$ has the form

$$Q' \ \leftarrow \ R'(x, y, z), S(u, v, z), U(p, q), \tag{9}$$

where R' is the rewritten form of R. The only free variable holding the value of a changeable attribute is y. This variable corresponds to attribute $R[B]$, which belongs to the equivalence class $E_{R[B]} = \{R[B], S[F], U[I]\}$. Therefore, we have the rules:

$$C^{R[B]}(R[B], t_1, y, Count(t_2)) \leftarrow TA(t_1, R[B], t_2, R[B]), R(t_1, x', y', z'), R(t_2, x, y, z).$$
$$C^{R[B]}(S[F], t_1, y, Count(t_2)) \leftarrow TA(t_1, R[B], t_2, S[F]), R(t_1, x', y', z'), S(t_2, x, y, z).$$
$$C^{R[B]}(U[I], t_1, y, Count(t_2)) \leftarrow TA(t_1, R[B], t_2, U[I]), R(t_1, x', y', z'), U(t_2, x, y).$$

$$Total^{R[B]}(t, y, Sum(u)) \leftarrow C^{R[B]}(x, t, y, u).$$
$$Compare^{R[B]}(t, y) \leftarrow Total^{R[B]}(t, y, z_1),\ Total^{R[B]}(t, y'', z_2),\ z_1 \leq z_2,\ y'' \neq y.$$

The rewriting of R becomes

$$R'(x, y, z) \leftarrow R(t, x, y', z), \mathit{not}\ \mathit{Compare}^{R[B]}(t, y), \mathit{Total}^{R[B]}(t, y, w). \qquad (10)$$

Thus, the rewriting of the original query is the stratified Datalog program [1] with aggregation consisting of rules (9), (10), plus the five rules preceding (10). □

In order to obtain the resolved answers to a query on a possibly non-resolved instance D, the resulting Datalog program can be run on D in polynomial time in the size of D. Remarks on implementation and an example are included in the appendix.

5 Conclusions

This paper considered a novel approach based on query rewriting to the duplicate resolution problem within the framework of matching dependencies. The transformed queries return the resolved answers to the original query, which are the answers that are true in all minimally resolved instances.

We used minimal resolved instances (MRIs) as our model of a clean database. Another possibility is to use arbitrary, not necessarily minimal, resolved instances (RIs). While MRIs have the advantage of being "closer" to the original instance than RIs, they have the downside of being overly restrictive.

In practice, update values are typically chosen by applying a merging function to the sets of duplicates [9, 15, 16], rather than by imposing a minimal change constraint. RIs are more flexible in that they take into account all ways of choosing the update values that lead to a clean database. We are currently investigating query answering over RIs, and have identified several tractable cases of the problem that are not tractable in the case of MRIs.

Matching dependencies first appeared in [24], and their semantics is given in [25]. The original semantics was refined in [15, 16], including the use of *matching functions* (MFs) for matching two attribute values. The approach in [15, 16] uses a chase to define clean instances. The MDs are applied one at a time to pairs of tuples, rather than all at once to all tuples as in the present paper. Another important difference is that here we do not use MFs to do a mathcing, but implicit existential quantifiers for the values in common. When the update values are determined by the matching functions there is no uncertainty arising from different possible choices for update values. Rather, the different clean instances are produced by applying the MDs in different orders. Clean answers are obtained by taking a glb (or lub) over the clean instances wrt a partial ordering that is based on semantic domination of one value by another.

The alternative refinement of the semantics used in this paper was first introduced in [30, 31]. A thorough complexity analysis, as well as the derivation of a query rewriting algorithm for the resolved answer problem was done in [29].

Our work in some ways resembles work on query answering over ontologies [32, 18]. As in our duplicate resolution setting, a chase is applied repeatedly to an initial instance, terminating in a "repaired" instance which is a fixed point of the chase rules. The set of chase rules can include tuple generating dependencies (TGDs) and equality generating dependencies (EGDs). Despite these similarities, our chase differs from those based on EGDs and TGDs in that it does not generate new tuples, but modifies values in existing tuples. Also, despite the fact that MDs are similar to EGDs, issues

arise as a result of the non-transitivity of similarity operators that do not occur in the case of EGDs.

In [3], Datalog is used for identifying groups of tuples that could be merged. However, they do not do the merging (a main contribution in our approach) or base their approach on MDs. Actually, that work could be considered as complimentary to ours, in the sense that, in essence, the authors address the problem of identifying similarities. This is the starting point for the actual matchings that we address in this paper.

References

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

[2] F. Afrati and P. Kolaitis. Repair checking in inconsistent databases: Algorithms and complexity. *Proc. ICDT*, 2009, pp. 31-41.

[3] A. Arasu, Ch. Ré and D. Suciu. Large-scale deduplication with constraints using dedupalog. Proc. ICDE 2009, pp. 952-963.

[4] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. *Proc. PODS*, 1999, pp. 68-79.

[5] M. Arenas, L. Bertossi, and J. Chomicki. Answer sets for consistent query answering in inconsistent databases. *Theory and Practice of Logic Programming*, 2003, 3(4-5):393-424.

[6] Z. Bahmani, L. Bertossi, S. Kolahi and L. Lakshmanan. Declarative Entity Resolution via Matching Dependencies and Answer Set Programs. Proc. KR 2012, pp. 380-390.

[7] P. Barceló, L. Bertossi, and L. Bravo. Characterizing and computing semantically correct answers from databases with annotated logic and answer sets. In *Semantics in Databases*, Springer LNCS 2582, 2003, pp. 1-27.

[8] P. Barcelo. Logical foundations of relational data exchange. *SIGMOD Record*, 2009, 38(1):49-58.

[9] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. Euijong Whang, and J. Widom. Swoosh: A generic approach to entity resolution. *VLDB Journal*, 2009, 18(1):255-276.

[10] L. Bertossi. Consistent query answering in databases. *ACM Sigmod Record*, 2006, 35(2):68-76.

[11] L. Bertossi. From database repair programs to consistent query answering in classical logic. Proc. AMW, 2009, CEUR-WS, Vol-450.

[12] L. Bertossi. *Database Repairing and Consistent Query Answering*, Morgan & Claypool, Synthesis Lectures on Data Management, 2011.

[13] L. Bertossi and L. Bravo. Consistent query answers in virtual data integration systems. In *Inconsistency Tolerance*, Springer LNCS 3300, 2004, pp. 42-83.

[14] L. Bertossi, L. Bravo, E. Franconi, and A. Lopatenko. The complexity and approximation of fixing numerical attributes in databases under integrity constraints. *Information Systems*, 2008, 33(4):407-434.

[15] L. Bertossi, S. Kolahi, and L. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. *Proc. ICDT*, 2011.

[16] L. Bertossi, S. Kolahi and L. Lakshmanan. Data cleaning and query answering with matching dependencies and matching functions. *Theory of Computing Systems*, DOI: 10.1007/s00224-012-9402-7, 2012.

[17] J. Bleiholder and F. Naumann. Data fusion. *ACM Computing Surveys*, 2008, 41(1):1-41.

[18] A. Cali, G. Gottlob, T. Lukasiewicz and A. Pieris. A logical toolbox for ontological reasoning. *ACM Sigmod Record*, 2011, 40(3):5-14.

[19] M. Caniupan and L. Bertossi. The consistency extractor system: answer set programs for consistent query answering in databases. *Data & Knowledge Engineering*, 2010, 69(6), pp. 545-572.

[20] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 2005, 197(1/2):90-121.
[21] M. Consens and A. Mendelzon. Low complexity aggregation in graphlog and datalog. *Theoretical Computer Science*, 1993, 116(1/2):95-116.
[22] T .Eiter, M., Fink, G., Greco, G. and D. Lembo. Repair localization for query answering from inconsistent databases. *ACM Trans. Database Syst.*, 2008, 33(2).
[23] A. Elmagarmid, P. Ipeirotis, and V. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowledge and Data Eng.*, 2007, 19(1):1-16.
[24] W. Fan. Dependencies revisited for improving data quality. *Proc. PODS*, 2008, pp. 159-170.
[25] W. Fan, X. Jia, J. Li, and S. Ma. Reasoning about record matching rules. *Proc. VLDB*, 2009, pp. 407-418.
[26] S. Flesca, F. Furfaro, and F. Parisi. Querying and repairing inconsistent numerical databases. *ACM Trans. Database Syst.*, 2010, 35(2).
[27] E. Franconi, A. Laureti Palma, N. Leone, S. Perri, and F. Scarcello. Census data repair: A challenging application of disjunctive logic programming. *Proc. LPAR*, 2001, pp. 561-578.
[28] A. Fuxman and R. Miller. First-order query rewriting for inconsistent databases. *J. Computer and System Sciences*, 2007, 73(4):610-635.
[29] J. Gardezi, L. Bertossi. Query answering under matching dependencies for data cleaning: Complexity and algorithms. arXiv:1112.5908v1.
[30] J. Gardezi, L. Bertossi, and I. Kiringa. Matching dependencies with arbitrary attribute values: semantics, query answering and integrity constraints. *Proc. LID*, 2011, pp. 23-30.
[31] J. Gardezi, L. Bertossi, and I. Kiringa. Matching dependencies: semantics, query answering and integrity constraints. *Frontiers of Computer Science*, Springer, 2012, 6(3):278-292.
[32] G. Gottlob, G. Orsi, and A. Pieris. Ontological queries: rewriting and optimization. *Proc. ICDE*, 2011, pp. 2-13.
[33] G. Greco, S. Greco, and E. Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Trans. Knowledge and Data Eng.*, 2003, 15(6):1389-1408.
[34] K. Kline, D. Kline, and B. Hunt. *SQL in a Nutshell*. O'Reilly, 2004.
[35] M. Lenzerini. Data integration: a theoretical perspective. Proc. PODS 2002, pp. 233-246.
[36] J. Wijsen. Database repairing using updates. *ACM Trans. Database Systems*, 2005, 30(3):722-768.
[37] J. Wijsen. On the first-order expressibility of computing certain answers to conjunctive queries over uncertain databases. *Proc. PODS*, 2010, pp. 179-190.

Appendix: Implementation Issues

The output of the query rewriting algorithm that we presented is a recursive Datalog program that involves both negation and aggregation. It is easy to see that it is stratified with respect to both negation and aggregation. That is, each predicate can be assigned a number (its stratum), such that, for each rule, the strata of body predicates defined with aggregation operators and also of predicates in negative literals are lower than that of the head. The semantics of such a program is the same as that of stratified Datalog without aggregation, i.e. Datalog^{not,s} [1]: Proceeding from the lowest to highest stratum, the rules defining the predicates in a given stratum are evaluated until no further changes to these predicates are produced [21].

Such queries are supported by the SQL3 standard. Also, the similarity operators used in the queries could be implemented as user-defined functions, which are supported by most database vendors [34].

Example 9. (example 1 continued) We show how to implement our query rewriting technique in SQL, to obtain the resolved answers to the query $\mathcal{Q}(x, y) : R(x, y)$. The TA closure can be expressed in SQL as a recursive view, as follows (since there is only one changeable attribute, we include only tuple identifiers in the view, not the attribute):

```
WITH TA(tid, tid') AS(
SELECT r1.tid, r2.tid
FROM R r1, R r2
WHERE r1.A ≈ r2.A
UNION ALL
SELECT ta1.tid', r'.tid
FROM TA ta1, R r, R r'
WHERE r.A ≈ r'.A AND ta1.tid' = r.tid)
```

We need a definition for predicate $C^{R[B]}$. In this case, we have only one rule, since there is only one changeable attribute. It is also defined as a view (again, excluding the attribute-valued argument):

```
WITH C(X1, X2, X3) AS
SELECT ta.tid AS X1, r2.B AS X2, Count(r2.tid) AS X3
FROM TA ta, R r1, R r2
WHERE ta.tid = r1.tid AND ta.tid' = r2.tid
GROUP BY ta.tid, r.B
```

Finally, the following query retrieves the resolved answers:

```
SELECT r.A AS A, c.X2 AS B
FROM R r, C c
WHERE r.tid = c.X1, c.X3 > ALL(SELECT c'.X3
FROM C c'
WHERE c'.X1 = r.tid)
```

We give below the extensions of the intermediate views and the query answer (the tuple id of t_i is i):

TA	tid	tid'
	1	2
	1	3
	2	3
	4	5
	2	1
	3	1
	3	2
	5	4

C	X_1	X_2	X_3
	1	c_1	2
	1	c_3	1
	2	c_1	2
	2	c_3	1
	3	c_1	2
	3	c_3	1
	4	c_3	2

$\mathcal{Q}$	A	B
	a_1	c_1
	a_2	c_1
	a_3	c_1
	b_1	c_3
	b_2	c_3

□

While it is possible to execute the rewritten queries using only the interface provided by a typical RDBMS, the runtime may be unacceptable if this approach is used. The calculation of the TA closure predicate generally requires $O(n^2)$ applications of a (possibly expensive) comparison operator, with n the size of the database, in order to identify all pairs of similar values.

In duplicate resolution, the runtime cost of this step is usually reduced by applying some approximation technique, such as blocking or sliding window approaches [23]. While it may be possible to implement some of these techniques in SQL, it may be better to subject the data to an initial preprocessing step if the user has sufficient access privileges to the data. This step would generate a new table relating pairs of similar tuples. While the transitive closure computation could also be included in this step, this may not be necessary depending on the data. Computing the transitive closure of a binary relation is an $O(n + m)$ operation, where n is the number of elements and m is the number of pairs of related elements. If the number of pairs of duplicate values is $O(n)$, then transitive closure can be computed in $O(n)$ time.