Proceedings of the Thirtieth International Florida Artificial Intelligence Research Society Conference

Enforcing Relational Matching Dependencies with Datalog for Entity Resolution

Zeinab Bahmani, Leopoldo Bertossi[*]
Carleton University, Ottawa, Canada.
zbahmani@connect.carleton.ca, bertossi@scs.carleton.ca

Abstract

Entity resolution (ER) is about identifying and merging records in a database that represent the same real-world entity. Matching dependencies (MDs) have been introduced and investigated as declarative rules that specify ER policies. An ER process induced by MDs over a dirty instance leads to multiple clean instances, in general. General *answer sets programs* have been proposed to specify the MD-based cleaning task and its results. In this work, we extend MDs to *relational MDs*, which capture more application semantics, and identify classes of relational MDs for which the general ASP can be automatically rewritten into a stratified Datalog program, with the single clean instance as its standard model.

1 Introduction

The presence in a database of duplicate, but non-identical representations of the same external entity leads to uncertainty. Applications running on top of the database or a query answering process may not be able to tell them apart, and the results may lead to ambiguity, semantic problems, such as unintended inconsistencies, and erroneous results. In this situation, the database has to be cleaned. The whole area of *entity resolution* (ER) deals with identifying and merging database records in a database that refer to the same real-world entity (Bleiholder and Naumann 2008; Elmagarmid, Ipeirotis and Verykios 2007). In so doing, duplicates are eliminated from the database, while at the same time new tuples are created through the merging process. ER is one of the most common and difficult problems in data cleaning.

In the last few years there has been strong and increasing interest in providing declarative and generic solutions to data cleaning problems (Bertossi and Bravo 2013), in particular, in logical specifications of the ER process. In this direction, *matching dependencies* (MDs) have been proposed (Fan 2008; Fan et al. 2009). They are declarative rules that assert that certain attribute values in relational tuples have to be merged, i.e. made identical, when certain similarity conditions hold between possibly other attribute values in those tuples.

Example 1. Consider the relational predicate $R(A, B)$, with attributes A and B. The symbolic rule $R[A] \approx R[A] \rightarrow R[B] \doteq R[B]$ is an MD specifying that, if for any two database tuples $R(a_1, b_1), R(a_2, b_2)$ in an instance D, when A-values are similar, i.e. $a_1 \approx a_2$, then their B-values have to be made equal (merged), i.e. b_1 or b_2 (or both) have to be changed to a value in common.

Let us assume that $\approx$ is reflexive and symmetric, and that $a_2 \approx a_3$, but $a_2 \not\approx a_1 \not\approx a_3$. The table on the left-hand side (LHS) below provides the extension for predicate R in D. In it some duplicates are not "resolved", e.g. the tuples (with tuple identifiers) t_1 and t_2 have similar – actually equal – A-values, but their B-values are different.

$R(D)$	A	B
t_1	a_1	b_1
t_2	a_1	b_2
t_3	a_2	b_3
t_4	a_3	b_4

$R(D')$	A	B
t_1	a_1	b_1
t_2	a_1	b_1
t_3	a_2	b_5
t_4	a_3	b_5

D does not satisfy the MD, and is a *dirty* instance. After applying the MD, we could get the instance D' on the right-hand side (RHS), where values for B have been identified. D' is *stable* in the sense that the MD holds in the traditional sense of an implication and "$=$" on D', which we call a *clean instance*. In general, for a dirty instance and a set of MDs, multiple clean instances may exist. Notice that if we add the MD $R[B] \approx R[B] \rightarrow R[A] \doteq R[A]$, creating a set of *interacting* MDs, a merging with one MD may create new similarities that enable the other MD. ■

A *dynamic semantics* for MDs was introduced in (Fan et al. 2009), that requires pairs of instances: a first one where the similarities hold, and a second where the mergings are enforced, e.g. D and D' in Example 1. MDs, as introduced in (Fan et al. 2009), do not specify what values to use when merging two attribute values.

The semantics was refined and extended in (Bertossi, Kolahi and Lakshmanan 2012) by means of *matching functions* (MFs) providing values for equality enforcements. An MF induces a lattice-theoretic structure on an attribute's domain. Actually, a *chase-based* semantics for MD enforcement was proposed. On this basis, given an instance D and a set Σ of MDs, wrt. which D may contain duplicates, the *chase procedure* may lead to several different *clean and stable solutions* D'. Each of them can be obtained by means of a provably terminating, but non-deterministic, iterative procedure that

[*]Contact author. Research funded by NSERC Discovery.
Copyright © 2016, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

enforces the MDs through application of MFs. The set of all such clean instances is denoted by $\mathcal{C}(D, \Sigma)$. Each clean instance can be seen as the result of an *uncertainty reduction* process. If at the end there are several possible clean instances, uncertainty is still present, and expressed through this class of *possible worlds*. Identifying cases for which a single clean instance exists is particularly relevant: for them uncertainty can be eliminated.

In (Bahmani et al. 2012), a declarative specification of this procedural data cleaning semantics was proposed. More precisely, a general methodology was developed to produce, from D, Σ and the MFs, an *answer set program* (ASP) (Gelfond and Lifschitz 1991; Brewka, Eiter and Truszczynski 2011) whose models are exactly the clean instances in the class $\mathcal{C}(D, \Sigma)$. The ASP enables reasoning in the presence of uncertainty due to multiple clean instances. Computational implementations of ASP can be then used for reasoning, for computing clean instances, and for computing *certain query answers* (aka. *clean answers*), i.e. those that hold in all the clean instances (Bahmani et al. 2012). Disjunctive ASPs, aka. *disjunctive Datalog programs with stable model semantics* (Eiter, Gottlob and Mannila 1997), are used (and provably required) for this task.

For some classes of MDs, for any given initial instance D, the class $\mathcal{C}(D, \Sigma)$ contains a *single clean instance* that can be computed in polynomial time in the size of D. Some sufficient syntactic and MF-dependent conditions were identified in (Bertossi, Kolahi and Lakshmanan 2012). In this work we identify a new important "semantic" class of MDs, where the initial instance is also considered. This is the *similarity-free attribute intersection class* (the *SFAI* class) of *combinations of MDs and initial instances*. Members of this class also have *(polynomial-time computable) single clean instances*. For all these classes, we show that the general ASP mentioned above can be automatically and syntactically transformed into an equivalent *stratified Datalog* program with the single clean instance as its *standard model*, which can be computed bottom-up from D in polynomial time in the size of D (Abiteboul, Hull, and Vianu 1995; Ceri, Gottlob and Tanca 1989).

Relational ER has been approached by the machine learning community (Bhattacharya and Getoor 2007). The idea is to learn from examples a classifier that can be used to determine if an arbitrary pair of records (or tuples), r_1, r_2, are duplicates (or each other) or not. In order to speed up the process of learning and applying the classifier, usually *blocking* techniques are applied (Whang et al. 2009). They are used to group records in clusters (blocks), for further comparison of pairs within clusters, but never of two records in different clusters. Interestingly, as reported in (Bahmani, Bertossi, and Vasiloglou 2015), MDs can be used in the blocking phase. As expected, MDs were also used during the final merging phase, after the calls to the classifier. However, the use at the earlier stage is rather surprising. The kind of MDs in this case turn out to belong, together with the initial instance, to the SFAI class. Actually, this allowed implementation of MD-based blocking by means of Datalog.

The reason for using MDs at the blocking stage is that they may convey semantic relationships between records for different entities, and can then be used to *collectively* block records for different entities (Bhattacharya and Getoor 2007): blocking together two records for an entity, say of books, may depend on having blocked together related records for a different entity, say of authors. For these kinds of applications, to capture semantic relationships, MDs were extended with relational atoms (conditions) in the antecedents, leading to the class of *relational MDs*.

In this work we also introduce and investigate the class of relational MDs, we extend the single-clean instance classes mentioned above to the relational MD case, and we obtain in a uniform manner Datalog programs for the enforcement of MDs in these classes. For lack of space, our presentation is based mainly on representative examples.

2 Background

We consider relational schemas $\mathcal{R}$ with a possibly infinite data domain U, a finite set of database predicates, e.g. R, and a set of built-in predicates, e.g. $=, \neq$. Each $R \in \mathcal{R}$ has attributes, say $A_1, \ldots, A_n$, each of them with a domain $Dom_{A_i} \subseteq U$. We may assume that the A_is are different, and different predicates do not share attributes. However, different attributes may share the same domain.

An instance D for $\mathcal{R}$ is a finite set of ground atoms (or tuples) of the form $R(c_1, \ldots, c_n)$, with $R \in \mathcal{R}$, $c_i \in Dom_{A_i}$. We will assume that tuples have identifiers, as in Example 1. They allow us to compare extensions of the same predicate in different instances, and trace changes of attribute values. Tuple identifiers can be accommodated by adding to each predicate $R \in \mathcal{R}$ an extra attribute, T, that acts as a key. Then, tuples take the form $R(t, c_1, \ldots, c_n)$, with t a value for T. Most of the time we leave the tuple identifier implicit, or we use it to denote the whole tuple. More precisely, if t is a tuple identifier in an instance D, then t^D denotes the entire atom, $R(\bar{c})$, identified by t. Similarly, if $\mathcal{A}$ is a list of attributes of predicate R, then $t^D[\mathcal{A}]$ denotes the tuple identified by t, but restricted to the attributes in $\mathcal{A}$. We assume that tuple identifiers are unique across the entire instance.

For a schema $\mathcal{R}$ with predicates $R_1[\bar{L}_1]$, $R_2[\bar{L}_2]$, with lists of attributes $\bar{L}_1, \bar{L}_2$, resp., a *matching dependency* (MD) (Fan et al. 2009) is an expression of the form:

$$\varphi\colon\ R_1[\bar{X}_1] \approx R_2[\bar{X}_2] \ \longrightarrow\ R_1[\bar{Y}_1] \doteq R_2[\bar{Y}_2]. \quad (1)$$

Here, $\bar{X}_1, \bar{Y}_1$ are sublists of $\bar{L}_1$, and $\bar{X}_2, \bar{Y}_2$ sublists of $\bar{L}_2$. The lists $\bar{X}_1, \bar{X}_2$ (also $\bar{Y}_1, \bar{Y}_2$) are *comparable*, i.e. the attributes in them, say X_1^j, X_2^j, are *pairwise comparable* in the sense that they share the same data domain Dom_j on which a binary similarity (i.e. reflexive and symmetric) relation $\approx_j$ is defined.

The MD (1) intuitively states that if, for an R_1-tuple t_1 and an R_2-tuple t_2 in an instance D the attribute values in $t_1^D[\bar{X}_1]$ are similar to attribute values in $t_2^D[\bar{X}_2]$, then the values $t_1^D[\bar{Y}_1]$ and $t_2^D[\bar{Y}_2]$ have to be made identical. This update results in another instance D', where $t_1^{D'}[\bar{Y}_1] = t_2^{D'}[\bar{Y}_2]$ holds. W.l.o.g., we may assume that the list of attributes on the RHS of MDs contain only one conjunct (attribute).

For a set Σ of MDs, a pair of instances (D, D') satisfies Σ if whenever D satisfies the antecedents of the MDs, then D'

satisfies the consequents (taken as equalities). If $(D, D) \not\models \Sigma$, we say that D is "dirty" (wrt. Σ). On the other hand, an instance D is *stable* if $(D, D) \models \Sigma$ (Fan et al. 2009).

We now review some elements in (Bertossi, Kolahi and Lakshmanan 2012). In order to *enforce* an MD on two tuples, making values of attributes identical, we assume that for each comparable pair of attributes A_1, A_2 with domain (in common) Dom_A, there is a binary *matching function* (MF) $m_A : Dom_A \times Dom_A \rightarrow Dom_A$, such that $m_A(a, a')$ is used to replace two values $a, a' \in Dom_A$ whenever necessary. MFs are idempotent, commutative, and associative. Similarity relations and MFs are treated as built-in relations.

A chase-based semantics for entity resolution with MDs is as follows: starting from an instance D_0, we identify pairs of tuples t_1, t_2 that satisfy the similarity conditions on the left-hand side of an MD φ, i.e. $t_1^{D_0}[\bar{X}_1] \approx t_2^{D_0}[\bar{X}_2]$ (but not the identity in its RHS), and apply an MF on the values for the right-hand side attribute, $t_1^{D_0}[A_1], t_2^{D_0}[A_2]$, to make them both equal to $m_A(t_1^{D_0}[A_1], t_2^{D_0}[A_2])$. We keep doing this on the resulting instance, in a *chase-like* procedure (Abiteboul, Hull, and Vianu 1995), until a stable instance is reached (cf. (Bertossi, Kolahi and Lakshmanan 2012) for details), i.e. a *clean instance*. An instance D_0 may have several (D_0, Σ)-clean instances. $\mathcal{C}(D_0, \Sigma)$ denotes the set of clean instances for D_0 wrt. Σ.

For given D and Σ, the class of clean instances can be specified as the stable models of a logic program $\Pi(D_0, \Sigma)$ in $Datalog^{\vee, not}$, i.e. a disjunctive Datalog program with weak negation and stable model semantics (Gelfond and Lifschitz 1991; Eiter, Gottlob and Mannila 1997), with rules of the form: $A_1 \vee \ldots \vee A_n \leftarrow P_1, \ldots, P_m,\ not\ N_1, \ldots,\ not\ N_k$. Here, $0 \leq n, m, k$, and A_i, P_j, N_s are (positive) atoms. Rules with $n = 0$ are called *program constraints* and have the effect of eliminating the stable models of the program (without them) that make their bodies (RHS of the arrow) true. When $n = 1$ and $k = 0$, we have (plain) *Datalog* programs. When $n \geq 1$ and *not* is stratified, we have *disjunctive, stratified Datalog* programs, denoted $Datalog^{\vee, not, s}$. The subclass with $n = 1$ is *stratified Datalog*, denoted $Datalog^{not, s}$.

We now introduce general cleaning programs by means of a representative example (for full generality and details, see (Bahmani et al. 2012)). Let D_0 be a given, possibly dirty initial instance wrt. a set Σ of MDs. The *cleaning program*, $\Pi(D_0, \Sigma)$, that we will introduce here, contains an $(n+1)$-ary predicate R_i', for each n-ary database predicate R_i. It will be used in the form $R_i'(T, \bar{Z})$, where T is a variable for the tuple identifier attribute, and $\bar{Z}$ is a list of variables standing for the (ordinary) attribute values of R_i.

For every attribute A in the schema, with domain Dom_A, the built-in ternary predicate M_A represents the MF m_A, i.e. $M_A(a, a', a'')$ means $m_A(a, a') = a''$. $X \preceq_A Y$ is used as an abbreviation for $M_A(X, Y, Y)$. For attributes A without a matching function, $\preceq_A$ becomes the equality, $=_A$. For lists of variables $\bar{Z}_1 = \langle Z_1^1, \ldots Z_1^n \rangle$ and $\bar{Z}_2 = \langle Z_2^1, \ldots Z_2^n \rangle$, $\bar{Z}_1 \preceq \bar{Z}_2$ denotes the conjunction $Z_1^1 \preceq_{A_1} Z_2^1 \wedge \ldots \wedge Z_1^n \preceq_{A_n} Z_2^n$. Moreover, for each attribute A, there is a built-in binary predicate $\approx_A$. For two lists of variables $\bar{X}_1 = \langle X_1^1, \ldots X_1^l \rangle$ and $\bar{X}_2 = \langle X_2^1, \ldots X_2^l \rangle$ representing comparable attribute values, $\bar{X}_1 \approx \bar{X}_2$ denotes the conjunction $X_1^1 \approx_1 X_2^1 \wedge \ldots \wedge X_1^l \approx_l X_2^l$.

In intuitive terms, program $\Pi(D_0, \Sigma)$ has rules to *implicitly simulate* a chase sequence, i.e. rules that enforce MDs on pairs of tuples that satisfy certain similarities, create newer versions of those tuples by applying matching functions, and make the older versions of the tuples unavailable for other rules. The main idea is making stable models of the program correspond to valid chase sequences leading to clean instances.

When the conditions for applying an MD hold, we have the choice between matching or not.[1] If we do, the tuples are updated to new versions. Old versions are collected in a predicate, and tuples that have not participated in a matching that was possible never become old versions (see the last denial constraint under 2. in Example 2, saying that the RHS of the arrow cannot be made true).

The program eliminates, using *program constraints*, instances (models of the program) that are the result of an *illegal* set of applications of MDs, i.e. they cannot put them in a linear (chronological) order representing chase steps. This occurs when matchings use old versions of tuples that have been replaced by new versions. To ensure that the matchings are enforced according to an order that correctly represents a chase, pairs of matchings are stored in an auxiliary relation, *Prec*. The last two program constraints under 6. in the example make *Prec* a linear order. In particular, matchings performed using old versions of tuples are disallowed.

Example 2. Consider relation $R(A, B)$ with extension in D_0 as below; and assume that exactly the following similarities hold: $a_1 \approx a_2$, $b_2 \approx b_3$; and the MFs are as follows:

$M_B(b_1, b_2, b_{12})$,
$M_B(b_2, b_3, b_{23})$,
$M_B(b_1, b_{23}, b_{123})$,
$M_B(b_3, b_4, b_{34})$.

$R(D_0)$	A	B
t_1	a_1	b_1
t_2	a_2	b_2
t_3	a_3	b_3

Σ contains the MDs:

$$\varphi_1 : R[A] \approx R[A] \rightarrow R[B] \doteq R[B],$$
$$\varphi_2 : R[B] \approx R[B] \rightarrow R[B] \doteq R[B],$$

which are *interacting* in that the set of attributes in the RHS of φ_1, namely $\{R[B]\}$, and the set of attributes in the LHS of φ_2, namely $\{R[B]\}$, have non-empty intersection. For the same reason, φ_2 also interacts with itself. Enforcing Σ on D_0 results in two alternative chase sequences, each enforcing the MDs in a different order, and two final stable clean instances D_1 and D_2'.

D_0	A	B
t_1	a_1	b_1
t_2	a_2	b_2
t_3	a_3	b_3

$\Rightarrow_{\varphi_1}$

D_1	A	B
t_1	a_1	b_{12}
t_2	a_2	b_{12}
t_3	a_3	b_3

D_0	A	B
t_1	a_1	b_1
t_2	a_2	b_2
t_3	a_3	b_3

$\Rightarrow_{\varphi_2}$

D_1'	A	B
t_1	a_1	b_1
t_2	a_2	b_{23}
t_3	a_3	b_{23}

$\Rightarrow_{\varphi_1}$

D_2'	A	B
t_1	a_1	b_{123}
t_2	a_2	b_{123}
t_3	a_3	b_{23}

The cleaning program $\Pi(D_0, \Sigma)$ is as follows:

[1]Matching is merging, or making identical, two attribute values on the basis of the MDs.

1. $R'(t_1,a_1,b_1)$. $R'(t_2,a_2,b_2)$. $R'(t_3,a_3,b_3)$. (plus M_B facts)

2. $$\begin{aligned}&\mathit{Match}_{\varphi_1}(T_1,X_1,Y_1,T_2,X_2,Y_2)\ \vee\\ &\quad \mathit{NotMatch}_{\varphi_1}(T_1,X_1,Y_1,T_2,X_2,Y_2)\ \leftarrow\\ &\qquad R'(T_1,X_1,Y_1),\ R'(T_2,X_2,Y_2),\ X_1\approx X_2,\ Y_1\neq Y_2.\\ &\mathit{Match}_{\varphi_2}(T_1,X_1,Y_1,T_2,X_2,Y_2)\ \vee\\ &\quad \mathit{NotMatch}_{\varphi_2}(T_1,X_1,Y_1,T_2,X_2,Y_2)\ \leftarrow\\ &\qquad R'(T_1,X_1,Y_1),\ R'(T_2,X_2,Y_2),\ Y_1\approx Y_2,\ Y_1\neq Y_2.\end{aligned}$$

$$\begin{aligned}&\mathit{Match}_{\varphi_i}(T_1,X_1,Y_1,T_2,X_2,Y_2)\ \leftarrow\\ &\qquad \mathit{Match}_{\varphi_i}(T_2,X_2,Y_2,T_1,X_1,Y_1).\quad (i\in\{1,2\})\end{aligned}$$

$$\begin{aligned}\mathit{OldVersion}_R(T_1,\bar{Z}_1)\ \leftarrow\ & R'(T_1,\bar{Z}_1),\ R'(T_1,\bar{Z}_1'),\\ & \bar{Z}_1\preceq\bar{Z}_1',\ \bar{Z}_1\neq\bar{Z}_1'.\end{aligned}$$

$$\begin{aligned}\leftarrow\ & \mathit{NotMatch}_{\varphi_i}(T_1,X_1,Y_1,T_2,X_2,Y_2),\\ &\quad not\ \mathit{OldVersion}_R(T_1,X_1,Y_1),\\ & not\ \mathit{OldVersion}_R(T_2,X_2,Y_2).\qquad (i\in\{1,2\})\end{aligned}$$

3. $$\begin{aligned}R'(T_1,X_1,Y_3)\ \leftarrow\ & \mathit{Match}_{\varphi_1}(T_1,X_1,Y_1,T_2,X_2,Y_2),\\ & M_B(Y_1,Y_2,Y_3).\\ R'(T_1,X_1,Y_3)\ \leftarrow\ & \mathit{Match}_{\varphi_2}(T_1,X_1,Y_1,T_2,X_2,Y_2),\\ & M_B(Y_1,Y_2,Y_3).\end{aligned}$$

4. $$\begin{aligned}&\mathit{Prec}(T_1,X_1,Y_1,T_2,X_2,Y_2,T_1,X_1,Y_1',T_3,X_3,Y_3)\ \leftarrow\\ &\quad \mathit{Match}_{\varphi_i}(T_1,X_1,Y_1,T_2,X_2,Y_2),\\ &\quad \mathit{Match}_{\varphi_j}(T_1,X_1,Y_1',T_3,X_3,Y_3),\\ &\quad Y_1\preceq Y_1',\ Y_1\neq Y_1'.\qquad (i,j\in\{1,2\})\end{aligned}$$

5. $$\begin{aligned}&\mathit{Prec}(T_1,X_1,Y_1,T_2,X_2,Y_2,T_1,X_1,Y_1,T_3,X_3,Y_3)\ \leftarrow\\ &\quad \mathit{Match}_{\varphi_i}(T_1,X_1,Y_1,T_2,X_2,Y_2),\\ &\quad \mathit{Match}_{\varphi_j}(T_1,X_1,Y_1,T_3,X_3,Y_3),\ M_B(Y_1,Y_3,Y_4),\\ &\quad Y_1\neq Y_4.\qquad (i,j\in\{1,2\})\end{aligned}$$

6. $$\begin{aligned}&\mathit{Prec}(T_1,\bar{Z}_1,T_2,\bar{Z}_2,T_1,\bar{Z}_1,T_2,\bar{Z}_2)\ \leftarrow\\ &\quad \mathit{Match}_{\varphi_i}(T_1,\bar{Z}_1,T_2,\bar{Z}_2).\qquad (i\in\{1,2\})\end{aligned}$$

$$\begin{aligned}\leftarrow\ & \mathit{Prec}(T_1,\bar{Z}_1,T_2,\bar{Z}_2,T_1,\bar{Z}_1',T_3,\bar{Z}_3),\\ & \mathit{Prec}(T_1,\bar{Z}_1',T_3,\bar{Z}_3,T_1,\bar{Z}_1,T_2,\bar{Z}_2),\\ & (T_1,\bar{Z}_1,T_2,\bar{Z}_2)\neq(T_1,\bar{Z}_1',T_3,\bar{Z}_3).\end{aligned}$$

$$\begin{aligned}\leftarrow\ & \mathit{Prec}(T_1,\bar{Z}_1,T_2,\bar{Z}_2,T_1,\bar{Z}_1',T_3,\bar{Z}_3),\\ & \mathit{Prec}(T_1,\bar{Z}_1',T_3,\bar{Z}_3,T_1,\bar{Z}_1'',T_4,\bar{Z}_4),\\ & not\ \mathit{Prec}(T_1,\bar{Z}_1,T_2,\bar{Z}_2,T_1,\bar{Z}_1'',T_4,\bar{Z}_4).\end{aligned}$$

7. $$\begin{aligned}R^c(T_1,X_1,Y_1)\ \leftarrow\ & R'(T_1,X_1,Y_1),\\ & not\ \mathit{OldVersion}_R(T_1,X_1,Y_1).\end{aligned}$$

The program constraint under 2. (last in the list) ensures that all new, applicable matchings have to be eventually carried out. The last set of rules (one for each database predicate) collect the final, clean extensions of them.

Program $\Pi(D_0,\Sigma)$ has two stable models, whose R^c-atoms are shown below:

$M_1=\{...,R^c(t_1,a_1,b_{12}),R^c(t_2,a_2,b_{12}),R^c(t_3,a_3,b_3)\}$,

$M_2=\{...,R^c(t_1,a_1,b_{123}),R^c(t_2,a_2,b_{123}),R^c(t_3,a_3,b_{23})\}$.

From them we can read off the two clean instances D_1, D_2' for D_0 that were obtained from the chase. ■

The cleaning program $\Pi(D_0,\Sigma)$ allows us to reason in the presence of uncertainty as represented by the possibly multiple clean instances. Actually, it holds that there is a one-to-one correspondence between $\mathcal{C}(D_0,\Sigma)$ and the set $\mathit{SM}(\Pi(D_0,\Sigma))$ of stable models of $\Pi(D_0,\Sigma)$. Furthermore, the program $\Pi(D_0,\Sigma)$ without its program constraints belongs to the class $\mathit{Datalog}^{\vee,not,s}$, the subclass of programs in $\mathit{Datalog}^{\vee,not}$ that have *stratified negation* (Eiter and Gottlob 1995). As a consequence, its stable models can be computed bottom-up by propagating data upwards from the underlying extensional database (that corresponds to the set of *facts* of the program), and making sure to minimize the selection of true atoms from the disjunctive heads. Since the latter introduces a form of non-determinism, a program may have several stable models. If the program is non-disjunctive, i.e. belongs to the $\mathit{Datalog}^{not,s}$, it has a single stable model that can be computed in polynomial time in the size of the extensional database D. The program constraints in $\Pi(D_0,\Sigma)$ make it unstratified (Gelfond and Kahl 2014). However, this is not a crucial problem because they act as a filter, eliminating the models that make them true from the class of models computed with the bottom-up approach.

3 Relational MDs

We now introduce a class of MDs that have found useful applications in blocking for learning a classifier for ER (Bahmani, Bertossi, and Vasiloglou 2015). They allow bringing additional relational knowledge into the conditions of the MDs. Before doing so, notice that an explicit formulation of the MD in (1) in classical predicate logic is:[2]

$$\varphi:\ \forall t_1t_2\,\forall\bar{x}_1\bar{x}_2\ (\,R_1(t_1,\bar{x}_1)\wedge R_2(t_2,\bar{x}_2)\ \wedge \bigwedge x_1^j\approx_j x_2^j\ \longrightarrow\ y_1\doteq y_2),\qquad (2)$$

with $x_1^j,y_1\in\bar{x}_1,\ x_2^j,y_2\in\bar{x}_2$. The t_i are variables for tuple IDs. $\mathit{LHS}(\varphi)$ and $\mathit{RHS}(\varphi)$ denote the sets of atoms on the LHS and RHS of φ, respectively. Atoms $R_1(t_1,\bar{x}_1)$ and $R_2(t_2,\bar{x}_2)$ contain all the variables in the MD; and similarity and identity atoms involve one variable from each of R_1,R_2.

Now, *relational MDs* may have in their LHSs, in addition to the two *leading atoms*, as R_1,R_2 in (2), additional database atoms, from more than one relation, that are used to give context to similarity atoms in the MD, and capture additional relational knowledge via additional conditions. Relational MDs extend "classical" MDs.

Example 3. With predicates $\mathit{Author}(\mathit{AID},\mathit{Name},\mathit{PTitle},\mathit{ABlock})$, $\mathit{Paper}(\mathit{PID},\mathit{PTitle},\mathit{Venue},\mathit{PBlock})$ (with ID and block attributes), this MD, φ, is relational:

$$\begin{aligned}&\underline{\mathit{Author}(t_1,x_1,y_1,bl_1)}\ \wedge \mathit{Paper}(t_3,y_1',z_1,bl_4)\ \wedge\ y_1\approx y_1'\ \wedge\\ &\underline{\mathit{Author}(t_2,x_2,y_2,bl_2)}\ \wedge\ \mathit{Paper}(t_4,y_2',z_2,bl_4)\ \wedge\ y_2\approx y_2'\ \wedge\\ &\qquad x_1\approx x_2\ \wedge\ y_1\approx y_2\longrightarrow\ bl_1\doteq bl_2,\end{aligned}$$

with implicit quantifiers, and underlined leading atoms (they contain the identified variables on the RHS). It contains similarity comparisons involving attribute values for both relations *Author* and *Paper*. It specifies that when the *Author*-tuple similarities on the LHS hold, and their papers are similar

[2]Similarity symbols can be treated as regular, built-in, binary predicates, but the identity symbol, $\doteq$, would be non-classical.

to those in corresponding *Paper*-tuples that are in the *same* block (an implicit similarity captured by the join variable bl_4), then blocks bl_1, bl_2 have to be made identical. This blocking policy uses relational knowledge (the relationships between *Author* and *Paper* tuples), plus the blocking decisions already made about *Paper* tuples. ■

4 Single-Clean-Instance Classes

First we introduce some notation. For an MD φ, $ALHS(\varphi)$ denotes the set of (non-tid) attributes (with predicates) appearing in similarities in the LHS of φ (including equalities, implicit or not). Similarly, $ARHS(\varphi)$ contains the attributes appearing *in identities* in the RHS. In Example 3: $ALHS(\varphi) = \{Author[Name],\ Author[PTitle],\ Paper[PTitle],\ Paper[PBlock]\}$, $ARHS(\varphi) = \{Author[ABlock]\}$.

As shown in (Bertossi, Kolahi and Lakshmanan 2012), for the classical case of *similarity-preserving* MDs (i.e. whose MFs satisfy $a \approx_A a'$ implies $a \approx m_A(a', a''))$, the chase-procedure computes a single clean instance in polynomial time in the size of the initial instance. The same holds for the classical case of *non-interacting* MDs. Now, a set Σ of possibly relational MDs is *non-interacting* if there are no $\varphi_1, \varphi_2 \in \Sigma$ (possibly the same), with $ARHS(\varphi_1) \cap ALHS(\varphi_2) \neq \emptyset$. Relational similarity-preserving MDs are trivially defined by using similarity preserving MFs. Through simple changes in the proofs given in (Bertossi, Kolahi and Lakshmanan 2012) for classical similarity-preserving and non-interacting MDs, it is possible to prove that, for both classes, for a given initial instance D, there is a single resolved instance that can be computed in polynomial time in the size of D. We say that these classes of MDs have the *single-clean instance* property, in short, *they are SCI*.

There is another class of *combinations of relational MDs* Σ *and initial instances* D that lead to a single *clean* instance:[3] That of *similarity-free attribute intersection* (SFAI) combinations (Σ, D).

Definition 1. Let Σ be a set of relational MDs and D an instance. The combination (Σ, D) has the SFAI property (or is SFAI) if, for every $\varphi_1, \varphi_2 \in \Sigma$ (which could be the same) and attribute $R[A] \in ARHS(\varphi_1) \cap ALHS(\varphi_2)$, it holds: If $S_1, S_2 \subseteq D$ with $R(\bar{c}) \in S_1 \cap S_2$, then $LHS(\varphi_1)$ is false in S_1 or $LHS(\varphi_2)$ is false in S_2.[4] ■

Non-interacting sets of MDs are trivially SFAI for every initial instance D. In general, different orders of MD enforcements may result in different clean instances, because tuple similarities may be broken during the chase with interacting MDs and non-similarity-preserving MFs, without reappearing again (Bertossi, Kolahi and Lakshmanan 2012). With SFAI combinations, two similar tuples, i.e. with similar attribute values, in the original instance D -or becoming similar along a chase sequence- may have the similarities broken in a chase sequence, but they will reappear later on in the same and the other chase sequences. Thus, different orders of MD enforcements cannot lead in the end to different clean instances.

Contrary to the *syntactic* class of non-interacting (relational) MDs and the MF-dependant class of similarity-preserving MDs, SFAI is a *semantic* class that depends on the initial instance (but not on subsequent instances obtained through the chase). Checking the SFAI property for (Σ, D) can be done by posing Boolean conjunctive queries (with similarity built-ins) to D; actually for each pair φ_1, φ_2 in Σ, a query, $\mathcal{Q}^A_{\varphi_1,\varphi_2}$, if $A \in ARHS(\varphi_1) \cap ALHS(\varphi_2)$, and a query, $\mathcal{Q}^B_{\varphi_2,\varphi_1}$, if $B \in ARHS(\varphi_2) \cap ALHS(\varphi_1)$.[5]

Example 4. (ex. 2 cont.) Consider the same classical MDs and MFs, but now with $a_1 \approx a_2$, $b_3 \approx b_4$, and new instance:

$R(D)$	A	B
t_1	a_1	b_1
t_2	a_2	b_2
t_3	a_3	b_3
t_4	a_4	b_4

The MDs are interacting, and both *applicable* on D, i.e. their LHSs are true. We can check the SFAI property for the combination (Σ, D) posing the following, implicitly existentially quantified, Boolean conjunctive queries to D:[6]

$$\mathcal{Q}^{R[B]}_{\varphi_1,\varphi_2}: \quad R(t_1,x_1,y_1) \wedge R(t_2,x_2,y_2) \wedge x_1 \approx x_2 \wedge R(t_3,x_3,y_3) \wedge y_2 \approx y_3,$$
$$\mathcal{Q}^{R[B]}_{\varphi_2,\varphi_2}: \quad R(t_1,x_1,y_1) \wedge R(t_2,x_2,y_2) \wedge y_1 \approx y_2 \wedge R(t_3,x_3,y_3) \wedge y_2 \approx y_3.$$

which take the value *false* in D. Then, (Σ, D) is SFAI. This is consistent with the easily verifiable observation that, no matter how the MDs are applied, a single clean instance, D' above, is always achieved. ■

$R(D')$	A	B
t_1	a_1	b_{12}
t_2	a_2	b_{12}
t_3	a_3	b_{34}
t_4	a_4	b_{34}

The example shows that it is possible to decide in polynomial time in the size of D if a combination (Σ, D) is SFAI: The number of queries does not depend on D, and they can be answered in polynomial time in data. Furthermore, it is possible to prove from the definition and the chase that SFAI (sets of) MDs are also SCI. However, in Section 5 we will indirectly show that this holds, by presenting stratified Datalog programs that implicitly represent the chase procedure based on them. The SCI property follows also from this.

5 Datalog Programs for SRI Classes

The general ASPs for classical MDs can be easily changed to deal with relational MDs, by including in the rule bodies the new relational atoms as extra conditions.

It is possible to take a set of MDs of the three kinds introduced in Section 4, generate an ASP for them of the general form of Section 2, and next, appealing to a general semantic property in common for those three classes, automatically rewrite the program into a stratified Datalog program.

[3] More precisely, it is duplicate-free wrt. the MDs, i.e. no additional enforcements thereof are possible

[4] We informally say that φ_1 is not applicable in S_1, etc.

[5] E.g. $R[B] \approx R[B] \rightarrow R[A] \doteq R[A]$, $R[A] \approx R[A] \rightarrow R[B] \doteq R[B]$ give rise to two SFAI tests (two queries).

[6] For each of the intersections: $ARHS(\varphi_1) \cap ALHS(\varphi_2) = \{R[B]\}$, and $ARHS(\varphi_2) \cap ALHS(\varphi_2) = \{R[B]\}$.

The rewriting is based on the facts that: (a) We do not need rules or constraints for the *Prec* predicate, because imposing a linear order of matchings is not needed; basically all MDs can be applied in parallel. (b) For the same reason, we do not need disjunctive heads, as each applicable MD can be applied without affecting the results obtained by the applications of the others. That is, we do not have to withhold any matchings (via the *NotMatch* predicates). (c) Old versions of tuples can be used in future MDs enforcements without any undesirable impact on the result.

In essence, the semantic property of the three classes, which can be expressed and used as a systematic rewriting mechanism of the general cleaning ASP, is that: *When confronted with match or not match, we can safely match; and the matchings do not need to be linearly ordered. Also, old versions of tuples can be still used for matchings*. The general transformation is illustrated by means of an example.

Example 5. (ex. 4 cont.) The general cleaning program for Σ in Example 2 depends on the initial instance only through the program facts. Then, the same program can be used in Example 4, but with the facts corresponding to $R(D_0)$ replaced by those corresponding to $R(D)$. Since (Σ, D) is SFAI, the cleaning program can be automatically rewritten into the following residual program (with enumeration as Example 2):

1. $R(t_1, a_1, b_1).\ R(t_2, a_2, b_2).\ R(t_3, a_3, b_3).\ R(t_4, a_4, b_4).$

2. $\mathit{Match}_{\varphi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2) \ \leftarrow R(T_1, X_1, Y_1),$
$R(T_2, X_2, Y_2),\ X_1 \approx X_2,\ Y_1 \neq Y_2.$
$\mathit{Match}_{\varphi_2}(T_1, X_1, Y_1, T_2, X_2, Y_2) \ \leftarrow R(T_1, X_1, Y_1),$
$R(T_2, X_2, Y_2),\ Y_1 \approx Y_2,\ Y_1 \neq Y_2.$
$\mathit{OldVersion}(T_1, X_1, Y_1) \ \leftarrow \ R(T_1, X_1, Y_1),$
$R(T_1, X_1, Y_1'),\ Y_1 \preceq Y_1',\ Y_1 \neq Y_1'.$

3. $R(T_1, X_1, Y_3) \ \leftarrow \ \mathit{Match}_{\varphi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2),$
$M_B(Y_1, Y_2, Y_3).$
$R(T_1, X_1, Y_3) \ \leftarrow \ \mathit{Match}_{\varphi_2}(T_1, X_1, Y_1, T_2, X_2, Y_2),$
$M_B(Y_1, Y_2, Y_3).$

7. $R^c(T_1, X_1, Y_1) \ \leftarrow R(T_1, X_1, Y_1),$
$\mathit{not}\ \mathit{OldVersion}(T_1, X_1, Y_1).$

This program does not have disjunctive heads or program constraints. We still need the *OldVersion* predicate to collect (the final versions of) the tuples in a clean instance. ■

The general ASP programs of Section 2 can be run on ASP solvers, such as *DLV* (Leone et al. 2006; Bahmani et al. 2012). However, the specialized stratified Datalog programs of this section can be run with implementations of Datalog. Actually, for their use in classification-based ER reported in (Bahmani, Bertossi, and Vasiloglou 2015), the programs were specified using *LogicQL* and run on top of the Datalog-supporting *LogicBlox* platform (Aref et al. 2015).

6 Conclusions

Matching dependencies (MDs) are an important addition to the declarative approaches to data cleaning, in particular, to the common and difficult problem of entity resolution (ER). We have shown that MDs can be extended to capture additional semantic knowledge, which is important in applications, in particular, to machine learning.

Computing with MDs has a relatively high data complexity (Bertossi, Kolahi and Lakshmanan 2012), but some classes of MDs (possibly in combination with an instance) can be identified for which ER can be done in polynomial time in data. Even more, it is possible to automatically produce Datalog programs that can be used to do ER with them.

References

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

Aref, M., ten Cate, B., Green, T.J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T., and Washburn, G. Design and Implementation of the LogicBlox System. Proc. SIGMOD 2015, pp. 1371-1382.

Bahmani, Z., Bertossi, L., Kolahi, S. and Lakshmanan, L. Declarative Entity Resolution via Matching Dependencies and Answer Set Programs. Proc. KR'12, AAAI Press, 2012, pp. 380-390.

Bahmani, Z., Bertossi, L. and Vasiloglou, N. ERBlox: Combining Matching Dependencies with Machine Learning for Entity Resolution. *Int. J. of Approximate Reasoning*, 2017, 83:118–141.

Bertossi, L. and Bravo, L. Generic and Declarative Approaches to Data Quality Management. In S. Sadiq (ed.), *Handbook of Data Quality - Research and Practice*, Springer, 2013, pp. 181-212.

Bertossi, L., Kolahi, S. and Lakshmanan, L. Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. *Theory of Computing Systems*, 2013, 52(3):441-482.

Bhattacharya, I. and Getoor, L. Collective Entity Resolution in Relational Data. *TKDD*, 2007, 1(1).

Bleiholder, J. and Naumann, F. Data Fusion. *ACM Computing Surveys*, 2008, 41(1).

Brewka,G., Eiter, T. and Truszczynski, M. Answer Set Programming at a Glance. *Comm. of the ACM*, 2011, 54(12), pp. 93-103.

Ceri, S., Gottlob, G. and Tanca, L. *Logic Programming and Databases*. Springer, 1989.

Eiter, T., Gottlob, G. and Mannila, H. Disjunctive Datalog. *ACM Trans. Database Syst.*, 1997, 22(3):364-418.

Eiter, T. and Gottlob, G. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Math. and Artif. Intell.*, 1995, 15(3-4):289-323.

Elmagarmid, A., Ipeirotis, P. and Verykios, V. Duplicate Record Detection: A Survey. *IEEE Transactions in Knowledge and Data Engineering*, 2007, 19(1):1-16.

Fan, W. Dependencies Revisited for Improving Data Quality. Proc. PODS 2008, pp. 159-170.

Fan, W., Jia, X., Li, J. and Ma, S. Reasoning about Record Matching Rules. *PVLDB*, 2009, 2(1):407-418.

Gelfond, M. and Lifschitz, V. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 1991, 9(3/4):365-386.

Gelfond, M. and Kahl, J. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents*. Cambridge U. Press, 2014.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Log.*, 2006, 7(3):499-562.

Whang, S.E., Menestrina, D., Koutrika, G., Theobald, M. and Garcia-Molina, H. Entity Resolution with Iterative Blocking. Proc. Sigmod, 2009, pp. 219-232.