DECLARATIVE ENTITY RESOLUTION VIA MATCHING DEPENDENCIES AND COMBINING MATCHING DEPENDENCIES WITH MACHINE LEARNING FOR ENTITY RESOLUTION

by
Zeinab Bahmani

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

School of Computer Science

at

CARLETON UNIVERSITY

Ottawa, Ontario
September, 2017

Table of Contents

List of Tables

List of Figures

Abstract

Entity resolution (ER) is an important problem in data cleaning. It is about identifying and merging records in a database that represent the same external entity. Relatively recently, declarative rules called matching dependencies (MDs) have been proposed for specifying similarity conditions under which attribute values in database records are merged. An ER process supported by MDs over a dirty instance may lead to multiple clean instances.

In this thesis, we first present disjunctive answer set programs that capture through their models the class of alternative clean instances obtained after an ER process based on MDs. With these programs, we can obtain clean answers to queries by skeptically reasoning from the program. As an important practical case of ER, we provide a declarative reconstruction of the so-called *union-case ER* methodology, as presented through a generic approach to ER, the so-called Swoosh approach. We extend our ASP-based account of the union-case of Swoosh with negative rules.

In this work, we extend MDs to *relational* MDs, which capture more application semantics, and identify classes of relational MDs for which the proposed declarative specifications for ER via MDs can be automatically rewritten into stratified Datalog programs.

We also show the process and the benefits of integrating four components of ER: (a) Building a classifier for duplicate/non-duplicate record pairs using machine learning (ML) techniques; (b) Use of relational MDs for supporting the blocking phase of ML; (c) Record merging on the basis of the classifier results; and (d) The use of the declarative language *LogiQL* -an extended form of Datalog supported by the *LogicBlox* platform- for all activities related to data processing, and the specification and enforcement of MDs.

To my love, Mohammad, and
my beloved parents

Acknowledgements

First and foremost, I offer my sincerest gratitude to my supervisor, Dr Leopoldo Bertossi. Working under supervision of Professor Bertossi was an invaluable experience, that allowed me to learn many aspects of science and life. He has supported me throughout my thesis, with his patience and knowledge. He has a been a true professional. The work presented in this thesis would not have been possible without his advice and support.

This dissertation would not have been possible without the support of my parents, Naser and Fatemeh, who have been there for me, supporting and motivating me all the time. They have given everything without expecting anything in return. I extend my appreciation to my brothers, Hossein and Mehdi, who were there whenever help was needed.

Many of my friends helped beyond reasonable expectation. My heartfelt appreciation goes to all them, in particular to: Elnaz, Najmeh, and Amine.

Above all, I thank God for all his blessings and mercies. I pray to God to guide me to the right path in this life, and make me a useful member of the community.

Chapter 1

Introduction

Entity resolution (ER) is a common and difficult problem in data cleaning that has to do with handling unintended multiple representations in a database of the same external objects. This problem is also known as deduplication, reference reconciliation, merge-purge, etc.. Multiple representations lead to uncertainty in data and the problem of managing it. Cleaning the database reduces uncertainty. In more precise terms, ER is about the identification and fusion of database records (think of rows or tuples in relational tables) that represent the same real-world entity. For some surveys see [Bleiholder and Naumann, 2008,Elmagarmid et al., 2007,Koudas et al., 2006,Winkler, 1999]. As a consequence, ER usually goes through two main consecutive phases: (a) detecting duplicates, and (b) merging them into single representations.

Efficient methods of data cleaning are essential to modern database applications. Data warehouses can contain large amounts of data, which, because they come from different sources, tend to suffer from high degrees of dirtiness as a result of different keys, formats, etc. To clean such data manually would be very time consuming and error-prone. It is important, therefore, to find ways of automating the process of duplicate resolution, and of applying generic methods that can be adapted to different situations.

As in most of the research and literature on data cleaning, the ER problem is confronted with data that reside in a single repository, e.g. a relational database. Also much in common with other data cleaning problems, ER is attacked on the basis of mechanisms and algorithms that lack a declarative specification or a clear semantics. However, quality requirements and mechanisms for enforcing them should be expressed in declarative terms, e.g. by means of classic integrity constraints, logical quality constraints or declarative cleaning rules.

A prominent example of declarative cleaning rules is provided by the proposal

of *matching dependencies* (MDs) [Fan, 2008, Fan et al., 2009]. MDs are specifically for ER. In intuitive terms, they express that under certain similarity conditions on attribute values for two tuples, some other attribute values in those tuples should be identified or matched, i.e. made equal. As we will see in this thesis in more general terms, MDs help identify duplicate data and enforce their merging by exploiting semantic or domain knowledge.

Example 1.0.1 Consider the relational schema $\mathcal{R} = \{R(A, B)\}$, with a predicate R with attributes A and B. The symbolic expression in (1.1) is an MD requiring that,

$$R[A] \approx R[A] \longrightarrow \quad R[B] \doteq R[B], \tag{1.1}$$

if for any two database tuples $R(a_1, b_1), R(a_2, b_2)$ in an instance D for the schema, when the values for attributes A are similar, i.e. $a_1 \approx a_2$, then their values for attribute B have to be made equal (matched), i.e. b_1 or b_2 (or both) have to be changed to a value in common.

Let us assume that $a_1 \approx a_2$, and $b_2 \approx b_3$, and $\approx$ is reflexive and symmetric. The table on the left-hand side (LHS) below is the extension for predicate R in D. In it, a duplicate is not resolved, since the tuples (with tuple identifiers) t_1 and t_2 have similar values for attribute A, but the values for B are different.

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

$R(D')$	A	B
t_1	a_1	b_5
t_2	a_2	b_5
t_3	a_3	b_3

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. In principle, nothing prevents us from choosing a new value b_5 from the data domain to do the matching. The MD holds in the traditional sense of an implication on D'. We call D' a *clean* instance. □

The introduction of MDs and their declarative formulation have become important additions to data cleaning research. A *dynamic semantics* for MDs was introduced

in [Fan et al., 2009], that requires a pair of instances: a first one where the similarities hold and a second one where the matchings are enforced, like D and D' in Example 1.0.1.

The MDs, as introduced in [Fan et al., 2009], do not specify how to match values. As we did in the example, we could even pick up a new value, e.g. b_5 above, for the value in common. This semantics was refined and extended in [Bertossi et al., 2013], using *matching functions* (MFs) to guide the matchings, one for each of the participating attribute domains. The MFs induce a lattice-theoretic structure on the latter [Bertossi et al., 2013]. An alternative dynamic semantics was introduced in [Gardezi et al., 2012]. It is not using MFs, but matchings have to be *justified* (so as in [Bertossi et al., 2013]) and *minimal*, i.e. a minimum number of changes to attribute values is applied to satisfy the MDs.

In this work we revisit the approach to ER via MDs and MFs introduced in [Bertossi et al., 2013]. In that scenario, a "dirty" instance D w.r.t. a set Σ of MDs may lead to several different *clean and stable solutions* D', each of which can be obtained by means of a provably terminating, but non-deterministic, chase-like procedure [Bertossi et al., 2013]. The latter involves enforcing MDs iteratively by means of applying MFs. The set of all such clean instances is denoted by $\mathcal{C}(D, \Sigma)$.

Example 1.0.2 (ex. 1.0.1 cont.) Assume that we add the MD $\varphi_2 : R[B] \approx R[B] \rightarrow R[B] \doteq R[B]$, creating a set of MDs $\Sigma = \{\varphi_1, \varphi_2\}$. Moreover, suppose that matching function m_B is as follows: $m_B(b_1, b_2) = b_{12}$, $m_B(b_2, b_3) = b_{23}$, $m_B(b_1, b_{23}) = b_{123}$. Enforcing Σ on D results in two chase sequences, and two final stable clean instances D_1 and D_2''.

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

$\Rightarrow_{\varphi_1}$

$R(D_1)$	A	B
t_1	a_1	b_{12}
t_2	a_2	b_{12}
t_3	a_3	b_3

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

$\Rightarrow_{\varphi_2}$

$R(D_1'')$	A	B
t_1	a_1	b_1
t_2	a_2	b_{23}
t_3	a_3	b_{23}

$\Rightarrow_{\varphi_1}$

$R(D_2'')$	A	B
t_1	a_1	b_{123}
t_2	a_2	b_{123}
t_3	a_3	b_{23}

Observe that, for instance D and the set of MDs Σ, two clean instances exist, namely D_1, D_2'', where the former results from the application of φ_1 on the pair of violating tuples t_1^D, t_2^D, and the latter results from first application of φ_2 on the pair of tuples t_2^D, t_3^D, and then φ_1 on the pair of tuples $t_1^{D_1''}, t_2^{D_1''}$. □

In [Bertossi et al., 2013], the clean answers to a query were introduced as those that are *certain*, i.e. true of all the clean instances (cf. Section 2 for details). They are invariant across the class $\mathcal{C}(D, \Sigma)$, and then are intrinsically "clean" answers. The problem of deciding, computing and approximating clean answers was also investigated. Clearly, computing clean answers via an explicit and materialized computation of all clean instances is prohibitively expensive and should be avoided whenever possible. Indeed, for a given initial instance D, we could have exponentially many clean instances (in the size of D) [Bertossi et al., 2013].

1.1 Declarative MD-based ER

ER becomes particularly crucial in data integration [Motro and Anokhin, 2006], and even more difficult in virtual data integration systems (VDIS). Logic-based specifications of the intended solutions of a generic VDIS have been proposed, used and investigated [Bertossi and Bravo, 2004]. As a consequence, logic-based specifications of ER or generic approaches to ER, that could be combined with the specifications of the integration process, become particularly relevant. Declarative, logic-based approaches to ER are particularly appropriate for their amalgamation with queries and query answering processes via some sort of query rewriting.

Answer set programming is a relatively new declarative programming paradigm [Brewka et al., 2011]. It has been successfully used to implicitly specify in general logical terms all the solutions of general combinatorial problems. In this thesis, we

introduce answer set programs (ASPs), in the form of *disjunctive Datalog* with stable model semantics [Gelfond and Lifschitz, 1991, Eiter et al., 1997], to specify the class $\mathcal{C}(D, \Sigma)$ of clean instances for D w.r.t. Σ. For each instance D and set Σ of MDs, we show how to build an answer set program $\Pi(D, \Sigma)$ whose stable models are in one-to-one correspondence with the instances in $\mathcal{C}(D, \Sigma)$.

The *cleaning program* $\Pi(D, \Sigma)$ axiomatizes the class $\mathcal{C}(D, \Sigma)$. Hence reasoning from/with the program amounts to reasoning with the full class $\mathcal{C}(D, \Sigma)$. In particular, clean query answers can be obtained from the original instance D by skeptical (aka. cautious) reasoning from the program.

Answer set programs have been used before in *consistent query answering* (CQA) [Arenas et al., 1999 , Bertossi, 2006, Chomicki, 2007, Bertossi, 2011], in the form of *repair programs*, that specify the *repairs* of a database instance w.r.t. a set of integrity constraints (ICs) [Arenas et al., 2003, Greco et al., 2003, Barcelo et al., 2003, Eiter et al., 2008, Caniupan and Bertossi, 2010, Franconi et al. 2001]. However, *MDs cannot be treated as classical ICs.* In particular, the matching functions and the lattice-theoretic structure of the domains, with the induced domination order, create a substantially different scenario, where new challenges arise. Furthermore, the semantics of MDs is quite different from that of classical ICs, and repair techniques for CQA cannot be straightforwardly used for ER via MDs or for clean query answering (cf. [Gardezi et al., 2012]).

We statically analyze the cleaning programs, in terms of syntactic structure and complexity, and establish that they belong to the class $Datalog^{\vee, not, s}$, the subclass of programs in $Datalog^{\vee, not}$ that have stratified negation [Eiter and Gottlob, 1995]. In particular, we show that their expressive power is appropriate for our application in ER, and is in line with the computational complexity of computing clean instances and clean query answers as established in [Bertossi et al., 2013]. We also show how to use cleaning programs with the skeptical semantics for the computation of clean answers from the original database, with a data complexity that matches the intrinsic data complexity of clean query answering.

1.2 Declarative Swoosh ER: The Union Case

The Swoosh approach to ER was proposed in [Benjelloun et al., 2009], as a generic and procedural specification of ER mechanisms. Special attention receives the common "union-case" of ER, that treats individual records as sets of triples of the form $(id, attr, value)$, i.e. as *objects*. An ER step basically matches values by producing their union; and the resulting value dominates the original values w.r.t. information contents.

The Swoosh's ER methodology is generic, but not declarative, in the sense that the semantics of the system is not captured in terms of a logical specification of the instances resulting from the cleaning process.

In this work we provide a *declarative version* of the union-case of Swoosh. It uses some extensions with sets and functional terms of the logic programming paradigm. We experiment with this approach using the *DLV-Complex* system [Calimeri et al., 2009] that supports such extensions.

Swoosh has been extended with negative rules [Whang et al., 2009a]. In this thesis, we extend our ASP-based account of the union-case of Swoosh by considering negative rules. This sometimes requires calls/access to external experts. In our approach they will be simulated as a separate program or as calls to external predicates [Eiter et al., 2005].

1.3 Relational MDs

In MDs, tuples for different relations may be related via attributes in common. The way attribute values in tuples in certain relations are merged, as a result of enforcing an MD, may influence the way attribute values for tuples in other relations are merged. Furthermore, in an extended form of MDs we could consider additional relational atoms in the LHS, as conditions for merging. For capturing all this, in this thesis we extend the class of matching dependencies introduced in Section 1 to the larger class of *relational MDs*, where semantic information is used to express relationships between different relations and their corresponding similarity conditions.

The chase-based semantics developed in [Bertossi et al., 2013] for MDs can be

applied to relational MDs without any relevant change: the new relational conditions in the LHSs of them have to be made true to enforce the MDs.

Example 1.3.1 With predicates $Author(AID, Name, BTitle, ABlock)$, $Book(BID, BTitle, Editor, BBlock)$ (with ID and block attributes), this MD, φ, is relational:

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

It contains similarity comparisons involving attribute values for both relations *Author* and *Book*. It specifies that when the *Author*-tuple similarities on the LHS hold, and their books are similar to those in corresponding *Book*-tuples that are in the *same* block (an implicit similarity, actually equality, captured by the join variable bl_4), then blocks bl_1, bl_2 have to be made identical. □

The introduction of Relational MDs is motivated by the application of MDs in *ERBlox* (cf. Chapter 7), but applications can be easily foreseen in other areas where declarative relational knowledge may be useful in combination with matching and merging.

1.4 A New Class of Well-behaved Relational MDs

Under the semantics of MDs introduced in [Bertossi et al., 2013], it is possible that, for a given initial instance D and set Σ of MDs, multiple clean instances exist, as shown in Example 1.0.2. This makes it interesting to identify relevant sets of MDs for which a single clean instance can be obtained starting from the initial one. It has been established that this is the case for sets of MDs with *similarity preserving* MFs, i.e., for every a, a', a'': $a \approx a'$ implies $a \approx MF(a', a'')$. The same happens with sets of *interaction-free* MDs, i.e. no attribute appears both in a RHS and LHS of MDs in the set [Bertossi et al., 2013]. In the both cases, the unique clean instance can be obtained in polynomial time in the size of the initial instance [Bertossi et al., 2013].

Example 1.4.1 (ex. 1.0.2 cont.) Σ is a set of interacting MDs since attribute B appears both in the RHS of φ_1 and LHS of φ_2.

Furthermore, suppose that matching function m_B is similarity preserving. Then, enforcing Σ on D results in a single clean instance D'_3, even if we apply the MDs on D in different orders:

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

$\Rightarrow_{\varphi_1}$

$R(D'_1)$	A	B
t_1	a_1	b_{12}
t_2	a_2	b_{12}
t_3	a_3	b_3

$\Rightarrow_{\varphi_2}$

$R(D'_2)$	A	B
t_1	a_1	b_{12}
t_2	a_2	b_{123}
t_3	a_3	b_{123}

$\Rightarrow_{\varphi_2}$

$R(D'_3)$	A	B
t_1	a_1	b_{123}
t_2	a_2	b_{123}
t_3	a_3	b_{123}

Here, $t_2[B] \approx t_3[B]$ holds in the initial instance D. After enforcing φ_1 on D, $t_2[B] \approx t_3[B]$ still holds in D'_1. This is because of similarity preserving matching function m_B. Due to $t_2^{D'_1}[B] \approx t_3^{D'_1}[B]$, we can apply φ_2 on D'_1. Similarly, $t_1[B] \approx t_2[B]$ keeps holding in D'_1, D'_2, D'_3. □

Enforcing Relational MDs that are similarity-preserving (i.e. that use similarity-preserving matching functions) leads to single clean instances, because only new additional conditions have to be verified before enforcing the MDs. In this work, we generalize the interaction-free class to the relational case.

In addition to the above well-behaved classes of relational MDs, in this thesis we identify a new class of combinations of relational MDs and initial instances, called the *blocking* class, for which single clean instances can be obtained starting from the initial instances.

Let D be a given, possibly dirty initial instance w.r.t. a set Σ of relational MDs where one of the three well-behaved classes of MDs with similarity preserving MFs, interaction-free MDs and blocking combination of the initial instance and the set of MDs holds for D, Σ. Then, we say that (D, Σ) have the *unique clean instance property* (UCI property). Make notice that the first two cases do not depend on D.

1.5 Specialized Cleaning Programs for UCI Class

In Section 1.1, general sets of MDs can be represented by means of disjunctive answer set programs, with the possibly multiple clean instances corresponding to the stable models of the program. The programs belong to the class $Datalog^{\vee, not, s}$, disjunctive Datalog programs with stratified negation.

The fact that the enforcement of Σ on D leads to a single clean instance, when (D, Σ) have the unique clean instance property, gives us the hope that we can use a computationally well-behaved extension of plain Datalog to enforce these MDs on D.

In this work we present a uniform methodology to specialize the general cleaning programs to produce programs for cases with the unique clean instance property. In other words, the general programs $\Pi(D, \Sigma)$ are specialized to the classes with the unique clean instance property, obtaining residual programs $\Pi^U(D, \Sigma)$ in non-disjunctive Datalog with stratified negation, $Datalog^{not, s}$.

1.6 The *ERBlox* Approach to ER

In this work we describe our *ERBlox* system which enables/supports ML-techniques for ER. Indeed, different ML techniques can be used for the classification model. To be more precise, *ERBlox* implements ER applying to ML techniques, and the specification and enforcement of relational MDs.

For duplicate detection, one of the main ER phases, one must first analyze multiple pairs of records, comparing the two records in them, and discriminating between: *pairs of duplicate records* and *pairs of non-duplicate records*. This classification problem is approached with machine learning (ML) methods, to learn from previously known or already made classifications (a training set for supervised learning), building a *classification model* (a classifier) for deciding about other record pairs [Christen and Goiser, 2010, Elmagarmid et al., 2007].

In principle, every two records (forming a pair) have to be compared, and then classified. Most of the work on applying ML to ER work at the record level [Rastogi et al., 2011, Christen and Goiser, 2010, Christen, 2008], and only with some of the attributes, or their features, i.e. numerical values associated to them, may be

involved in duplicate detection. The choice of relevant sets of attributes and features is application dependent.

In our *ERBlox* system MDs have to be specified in a declarative manner, and at some point enforced, by producing changes on the data. For this purpose, we use the *LogicBlox* platform, a data management system developed by the LogicBlox[1] company, that is centered around its declarative language, *LogiQL*. *LogiQL* supports relational data management and, among several other features [Aref et al., 2015], an extended form of Datalog with stratified negation [Ceri et al., 1989]. *LogicBlox* is expressive enough for the kind of MDs considered in *ERBlox* system (more details follow below).

ERBlox has four main components or modules, that are executed in this order:

(a) MD-based collective blocking,

(b) ML-based classification model construction,

(c) Duplicate detection, and

(d) MD-based merging (of duplicates).

In the following, we briefly explain each component.

1.6.1 MD-based collective blocking

ER may be a task of quadratic complexity since it requires comparing every two records. To reduce the large number of two-record comparisons, *blocking techniques* are used [Steorts et al., 2014, Baxter et al., 2003, Herzog, 2007, Whang et al., 2009a]. Commonly, a single record attribute, or a combination of attributes, the so-called *blocking key*, is used to split the database records into blocks. Next, under the assumption that any two records in different blocks are unlikely to be duplicates of each other, only every two records in a same block are compared for duplicate detection.

Although blocking will discard many record pairs that are obvious non-duplicates, some true duplicate pairs might be missed (by putting them in different blocks), due to errors or typographical variations in attribute values. More interestingly, similarity

[1] www.logicblox.com

between blocking keys alone may fail to capture the relationships that naturally hold in the data and could be used for blocking. Thus, entity blocking based only on blocking key similarities may cause low recall[2]. This is a major drawback of traditional blocking techniques.

In our *ERBlox* system we consider different and coexisting entities. For each of them, there is a collection of records. Records for different entities may be related via attributes in common or referential constraints. However, we do not assume that integrity constraints (ICs) always exist. Blocking can be performed on each of the participating entities, and the way records for an entity are placed in blocks may influence the way the records for another entity are assigned to blocks. This is called "collective blocking". Semantic information, in addition to that provided by blocking keys for single entities, can be used to state relationships between different entities and their corresponding similarity criteria. So, blocking becomes a collective and intertwined process involving several entities. In the end, the records for each individual entity will be placed in blocks associated to that entity.

Example 1.6.1 Consider two entities, Author and Paper. For each of them, there is a set of records (for all practical purposes, think of database tuples in a single table for each entity). For Author we have records with $\mathbf{a} = \langle name, \ldots, affiliation, \ldots, paper\ title, \ldots \rangle$, with $\{name, affiliation\}$ the blocking key formed by two attributes; and for Paper, records of the form $\mathbf{p} = \langle title, \ldots, author\ name, \ldots \rangle$, with attribute *title* the blocking key. We want to group Author and Paper records at the same time, in an entwined process.

We block together two Author entities on the basis of the similarities of authors' names and affiliations. Assume that Author entities $\mathbf{a}_1, \mathbf{a}_2$ have similar names, but their affiliations are not. So, the two records would not be put in the same block. However, $\mathbf{a}_1, \mathbf{a}_2$ are authors of papers (in Paper records) $\mathbf{p_1}, \mathbf{p_2}$, resp., which have been put in the same block (of papers) on the basis of similarities of paper titles. In this case, additional *semantic knowledge* might specify that if two papers are in the same block, then corresponding Author records that have similar author names should be

[2]Recall is the ratio of the number of duplicate record-pairs found to the total number of duplicate record-pairs in the database.

put in the same block too. Then, $\mathbf{a}_1$ and $\mathbf{a}_2$ would end up in the same block.

In this example, we are blocking Author and Paper entities, separately, but collectively and in interaction. □

Collective blocking is based on blocking keys and *the enforcement* of semantic information about the *relational closeness* of entities Author and Paper, which is captured by a set of relational MDs. So, we propose "MD-based collective blocking".

MD-based collective blocking uses relational MDs to specify the blocking strategy. Relational MDs express conditions in terms of blocking key similarities and also relational closeness -the semantic knowledge- to assign two records to a same block, by making the block identifiers identical. Then, under MD-based collective blocking different records of possibly several related entities are simultaneously assigned to blocks through the enforcement of MDs (cf. Section 7.3 for details).

Example 1.6.2 (ex. 1.6.1 cont.) We could use the following relational MD for blocking Author records. In it there are similarity comparisons involving attributes for both entities Author and Paper:

$$Author(x_1, y_1, bl_1) \quad \wedge \quad Paper(y_1, z_1, bl_4) \wedge Author(x_2, y_2, bl_2) \wedge \\ Paper(y_2, z_2, bl_4) \wedge x_1 \approx_1 x_2 \wedge z_1 \approx_2 z_2 \longrightarrow bl_1 \doteq bl_2. \tag{1.2}$$

It specifies that when the Author record similarities on the LHS hold, and corresponding papers are in the same block, then blocks bl_1, bl_2 have to be made identical. □

Although in general a set of relational MDs may lead to alternative final instances through its enforcement [Bertossi et al., 2013], in the case of MD-based collective blocking, for each entity, a unique set of blocks is generated. The reason is that the combination of the set of relational MDs and the initial database instance falls into a newly identified, well-behaved blocking class, which we introduced in Section 1.4 (and Section 6.1.1).

1.6.2 ML-based classification model construction and duplicate detection

On the machine learning side (item (b) above), the problem is about building and implementing a model for the detection of pairs of duplicate records. The classification model is trained using record-pairs known to be duplicates or non-duplicates. We used the established classification algorithm *support vector machines* (SVM) [Vapnik, 2009]. We used the *Ismion*[3] implementation of it due to the in-house expertise at LogicBlox.

After records are divided in blocks, the proper duplicate detection process starts (item (c) above). It is carried out by comparing every two records in a block, and classifying the pair as "duplicates" or "non-duplicates" using the trained ML model at hand.

1.6.3 MD-based merging

In the end, records in duplicate pairs are considered to represent the same external entity, and have to be *merged* into a single representation, i.e. into a single record. This second phase is also application dependent. MDs were originally proposed to support this task.

The sets of MDs used in (a) and (d) are different, and play different roles. The blocking phase, (a) above, is a non-traditional, novel use of MDs, whereas the application of MDs for the proper merging, (d) above, corresponds to the intended use of MDs [Fan, 2008]. In both cases, they are application-dependent, but have a canonical representation in the system, as Datalog rules. The MDs are then enforced by applying (running) those rules. In general a set of MDs may lead to multiple final instances through its enforcement [Bertossi et al., 2013]. But, in the case of (d), the set of MDs leads to a single, duplicate-free instance for each entity. This is because the MDs in the set turn out to be interaction-free (as introduced in [Bertossi et al., 2013]).

As presented in Section 1.1, general sets of MDs are expressed by means of cleaning ASPs. However, both classes of MDs used by *ERBlox*, have the unique clean instance property. They are expressed by Datalog with stratified negation, as developed in

[3]http://www.ismion.com

Section 1.5, which is supported by *LogicQL*.

For experimentation with the *ERBlox* system, we used as dataset a snapshot of Microsoft Academic Search (MAS)[4] that includes 250K authors, 2.5M papers, and a training set. We also used, independently, datasets from DBLP and Cora Citation. The experimental results show that our system improves ER recall and precision over traditional, *standard* blocking techniques [Jaro, 1989], where just blocking-key similarities are used. Actually, MD-based collective blocking leads to higher precision and recall on the given datasets.

Our work also shows the integration under a single system of different forms of data retrieval, storage and transformation, on one side, and machine learning techniques, on the other. All this is enabled by the use of optimized Datalog-rules declaration and execution as supported by the *LogicBLox* platform.

1.6.4 Thesis Structure

This thesis is structured as follows. In Chapter 2 we recall some definitions and concepts. Chapter 3 presents the state of the art in ER, MDs, and blocking techniques. In Chapter 4 we describe the objectives of the thesis. Chapter 5 contains ASPs for ER process based on MDs and a declarative reconstruction of union-case of Swoosh. Chapter 6 introduces relational MDs and blocking combinations of MDs and initial instances. The general cleaning ASPs are specialized for the cases with the unique clean instance property in Chapter 6. Chapter 7 proposes MD-based collective blocking, and describes the *ERBlox* system. Experimental results are shown in Chapter 7. Chapters 8 and 9 provide related work, some final conclusions and future work.

Some of the results of this thesis have been published in [Bahmani et al., 2012, Bahmani et al., 2016, Bahmani et al., 2017].

[4] http://academic.research.microsoft.com. As of January 2013.

Chapter 2

Background

2.1 Relational Databases

We consider relational schemas $\mathcal{R}$ with a possibly infinite data domain U, whose elements are called *constants*, 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 a fixed set of 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 $R \in \mathcal{R}$, $c_i \in Dom_{A_i}$. The *active domain* of an instance D, denoted $Adom(D)$, is the finite set of all constants from U that appear in D. We will assume that tuples have unique, global identifiers, as in Example 1.0.1. They allow us to compare extensions of the same predicate in different instances, and trace changes of attribute values in tuples. 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.

Schema $\mathcal{R}$ determines a language $L(\mathcal{R})$ of first-order (FO) predicate logic. A conjunctive query is a formula of $L(\mathcal{R})$ of the form $\mathcal{Q}(\bar{x})\colon\ \exists \bar{y}(P_1(\bar{x}_1) \wedge \cdots \wedge P_m(\bar{x}_m))$, where $P_i \in \mathcal{R}$, $\bar{y}, \bar{x}_1, \cdots, \bar{x}_m$ are lists of variables, $\bar{x} = (\cup_i \bar{x}_i) \smallsetminus \bar{y}$ is the list free variables of the query, say $\bar{x} = x_1 \cdots x_k$. An answer to $\mathcal{Q}$ in instance D is a sequence $\langle a_1, \ldots, a_k \rangle \in U^k$ that makes $\mathcal{Q}$ true in D, denoted $D \models \mathcal{Q}[a_1, \ldots, a_k]$. $\mathcal{Q}(D)$ denotes

the set of answers to $\mathcal{Q}$ in D.

2.2 Matching Dependencies

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

$$\varphi\colon \quad R_1[\bar{X}_1] \approx R_2[\bar{X}_2] \;\longrightarrow\; R_1[\bar{Y}_1] \doteq R_2[\bar{Y}_2]. \tag{2.1}$$

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. $LHS(\varphi)$ and $RHS(\varphi)$ denote the sets of atoms on the LHS and RHS of φ, respectively. Actually, (2.1) can be seen as an abbreviation for

$$\varphi\colon \quad \bigwedge_j R_1[X_1^j] \approx_j R_2[X_2^j] \;\longrightarrow\; \bigwedge_k R_1[Y_1^k] \doteq R_2[Y_2^k].$$

This MD 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.

Let D, D' be "corrected" instances, i.e. they have same tuple ids. 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" (w.r.t. Σ). On the other side, an instance D is *stable* if $(D, D) \models \Sigma$ [Fan et al., 2009].

In order to *enforce* MDs on pairs of tuples, making values of attributes identical, we assume, according to [Bertossi et al., 2013], that for each comparable pair of attributes A_1, A_2 with domain (in common) Dom_A, there is a binary *matching function* (MF) $m_A : Dom_A \times Dom_A \to Dom_A$, such that $m_A(a, a')$ is used to replace two values $a, a' \in Dom_A$ whenever necessary. We expect MFs to be idempotent, commutative, and associative [Bertossi et al., 2013, Benjelloun et al., 2009].

The structure (Dom_A, m_A) forms a *join semilattice*, that is, a partial order with

a least upper bound (*lub*) for every pair of elements. The induced partial order $\preceq_A$ on Dom_A is defined by: $a \preceq_A a'$ whenever $m_A(a, a') = a'$. The *lub* coincides with m_A: $lub_{\preceq_A}\{a, a'\} = m_A(a, a')$ [Bertossi et al., 2013]. We also assume the existence of the greatest lower bounds, $glb(a, a')$. In this thesis, we will assume by default that similarity relations and MFs are built-in relations.

A chase-based semantics for entity resolution with MDs is given in [Bertossi et al., 2013]: 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 a matching dependency φ, i.e., $t_1^{D_0}[\bar{X}_1] \approx t_2^{D_0}[\bar{X}_2]$ (but not 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, until a stable instance is reached. Below we make all this precise.

Definition 2.2.1 Let D, D' be database instances with the same set of tuple identifiers, Σ be a set of MDs, and $\varphi : R_1[\bar{X}_1] \approx R_2[\bar{X}_2] \rightarrow R_1[\bar{Y}_1] \doteq R_2[\bar{Y}_2]$ be an MD in Σ. Let t_1, t_2 be an R_1-tuple and an R_2-tuple identifiers, respectively, in both D and D'. Instance D' is the *immediate result of enforcing* φ on t_1, t_2 in instance D, denoted $(D, D')_{[t_1,t_2]} \models \varphi$, if

(a) $t_1^D[\bar{X}_1] \approx t_2^D[\bar{X}_2]$, but $t_1^D[\bar{Y}_1] \neq t_2^D[\bar{Y}_2]$;

(b) $t_1^{D'}[\bar{Y}_1] = t_2^{D'}[\bar{Y}_2] = m_A(t_1^D[\bar{Y}_1], t_2^D[\bar{Y}_2])$; and

(c) D, D' agree on every other tuple and attribute value. □

Notice that there are no unnecessary changes.

Definition 2.2.2 For an instance D_0 and a set of MDs Σ, an instance D_k is (D_0, Σ)-*clean* if D_k is stable, and there exists a finite sequence of instances $D_1, \ldots, D_{k-1}$ such that, for every $i \in [1, k]$, $(D_{i-1}, D_i)_{[t_1^i, t_2^i]} \models \varphi$, for some $\varphi \in \Sigma$ and tuple identifiers t_1^i, t_2^i. □

As shown in Example 1.0.2, an instance D_0 may have several (D_0, Σ)-clean instances. $\mathcal{C}(D_0, \Sigma)$ denotes the set of clean instances for instance D_0 w.r.t. Σ. However,

there is a unique clean instance that is computable in polynomial time in $|D_0|$ if one of the following holds [Bertossi et al., 2013]:

(a) MFs used by Σ are *similarity-preserving*, i.e., for every a, a', a'': $a \approx a'$ implies $a \approx m_A(a', a'')$.

(b) Σ is *interaction-free* (IF), i.e. no attribute appears both in a RHS and LHS of MDs in Σ.

For example, the set $\Sigma_1 = \{R[A] \approx T[B] \rightarrow R[C] \doteq T[D], \quad T[D] \approx S[A] \rightarrow T[A] \doteq S[B]\}$ is interacting due to the presence of attribute $T[D]$. $\Sigma_2 = \{R[A] \approx T[B] \rightarrow R[C] \doteq T[D], \quad T[A] \approx S[A] \rightarrow T[C] \doteq S[C]\}$ is non-interacting.

Let D be a given, possibly dirty initial instance w.r.t. a set Σ of MDs where one of the two well-behaved classes of similarity preserving MFs and IF MDs holds for D, Σ. Then, we say that (D, Σ) have *unique clean instance property* (UCI property).

The domain-level relations $a \preceq_A a'$ can be thought of in terms of relative information contents [Bertossi et al., 2013]. They can be lifted to a *tuple-level partial order*, defined by: $t_1 \preceq t_2$ iff $t_1[A] \preceq_A t_2[A]$, for each attribute A. This in turn can be lifted to a *relation-level partial order*: $D_1 \sqsubseteq D_2$ iff $\forall t_1 \in D_1 \; \exists t_2 \in D_2 \; t_1 \preceq t_2$.

When a tuple t^D in instance D is updated to $t^{D'}$ in instance D' by enforcing an MD and applying an MF, it holds $t^D \preceq t^{D'}$; and the instances D and D' satisfy: $D \sqsubseteq D'$. If instances are all "reduced", *Red*, by eliminating tuples that are dominated by others, the set of reduced instances with $\sqsubseteq$ forms a lattice. That is, we can compute the *glb* and the *lub* of every finite set of instances w.r.t. $\sqsubseteq$. This is a useful result that allows us to compare sets of query answers w.r.t. $\sqsubseteq$. The *glb* of two instances D_1, D_2 of schema $\mathcal{R}$ w.r.t. $\sqsubseteq$ is defined as

$$glb_{\sqsubseteq}\{D_1, D_2\} = Red(\{t_1 \curlywedge t_2 \mid t_1 \in D_1, t_2 \in D_2, \text{and } t_1, t_2 \text{ are } R\text{–tuples}\}), \qquad (2.2)$$

where $R \in \mathcal{R}$, and $t_1 \curlywedge t_2$ is the tuple t, such that $t[A] = glb_{\preceq_A}\{t_1^{D_1}[A], t_2^{D_2}[A]\}$ for every attribute A in R.

Example 2.2.1 Consider the following instances

$R(D_1)$	*name*	*addr*
	John Doe	25 Main st., Ottawa
	J. Doe	25 Main st., Ottawa

$R(D_2)$	*name*	*addr*
	John Doe	Main st., Ottawa
	J. Doe	25 Main st., Vancouver

The instance $\{t' \curlywedge t'' \mid t' \in D_1, t'' \in D_2\}$ is:

$R(D_1)$	*name*	*addr*
	John Doe	Main st., Ottawa
	J. Doe	25 Main st.
	J. Doe	Main st., Ottawa

We have "Main st." $\preceq$ "Main st., Ottawa" $\preceq$ "25 Main st., Ottawa", "J. Doe" $\preceq$ "John Doe", "Main st." $\preceq$ "25 Main st." $\preceq$ "25 Main st., Ottawa". After reduction, we obtain

$R(D_1)$	*name*	*addr*
	John Doe	Main st., Ottawa
	J. Doe	25 Main st.

which is the $glb_{\sqsubseteq}\{D_1, D_2\}$. □

The set of *clean answers* to a query $\mathcal{Q}$ from instance D w.r.t. Σ is formally defined by $Clean^{D}_{\Sigma}(\mathcal{Q}) := glb_{\sqsubseteq}\{\mathcal{Q}(D') \mid D' \in \mathcal{C}(D, \Sigma)\}$ where each $\mathcal{Q}(D')$ is reduced. [Bertossi et al., 2013].

If the query is boolean, i.e. a sentence, then, for an instance D, $\mathcal{Q}(D) := \{yes\}$ when $\mathcal{Q}$ is true in D, and $\{no\}$, otherwise. It is also assumed that $\{no\} \preceq \{yes\}$, but $\{\text{yes}\} \not\preceq \{\text{no}\}$, creating a two-valued lattice. Accordingly, $\{no\} \sqsubseteq \{yes\}$ is defined.

Example 2.2.2 Consider the MD $\varphi : R[name] \approx R[name] \longrightarrow R[addr] \doteq R[addr]$ applied to the instance D_0 below. Assume that John Doe $\approx$ J. Doe, Jane Doe $\approx$ J. Doe. Moreover, suppose that matching function m_B is as follows: m_{addr}(Main st., Ottawa, 25 Main st., Canada) = 25 Main st., Ottawa Canada, m_{addr}(25 Main, st. Vancouver , 25 Main st., Canada) = 25 Main st., Vancouver Canada.

$R(D_0)$	*name*	*addr*
	John Doe	Main st., Ottawa
	J. Doe	25 Main st., Canada
	Jane Doe	25 Main st., Vancouver

Enforcing φ on D_0 results in two clean instances D' and D'':

$R(D')$	*name*	*addr*
	John Doe	25 Main st., Ottawa Canada
	J. Doe	25 Main st., Ottawa Canada
	Jane Doe	25 Main st., Vancouver

$R(D'')$	*name*	*addr*
	John Doe	Main st., Ottawa
	J. Doe	25 Main st., Vancouver Canada
	Jane Doe	25 Main st., Vancouver Canada

For the query $\mathcal{Q}(x)$: $R(\text{“J. Doe”}, x)$, it holds: $\mathit{Clean}_{\Sigma}^{D_0}(\mathcal{Q}) = \{25 \text{ Main st., Canada}\}$. This is because $\mathcal{Q}(D') = \{25 \text{ Main st., Ottawa Canada}\}$, $\mathcal{Q}(D'') = \{25 \text{ Main st., Vancouver Canada}\}$, and $\mathit{glb}_{\sqsubseteq}\{\{25 \text{ Main st., Ottawa Canada}\}, \{25 \text{ Main st., Vancouver Canada}\}\} = \{25 \text{ Main st., Canada}\}$.

For the Boolean query $\mathcal{Q}$: $R(\text{J. Doe}, 25 \text{ Main st. Vancouver Canada})$, we have $\mathcal{Q}(D') = \{no\}$, $\mathcal{Q}(D'') = \{yes\}$. Therefore, $\mathit{Clean}_{\Sigma}^{D}(\mathcal{Q}) = \{no\}$. □

2.3 Datalog

Datalog is a logic programming language for deductive databases, first formalized by [Ullman, 1989]. A comprehensive review of Datalog was published by [Ceri et al., 1989].

A literal, $p(t_1, ..., t_k)$ is a predicate applied to its arguments, each of which is either a constant or a variable. A variable is an identifier that starts with an upper-case letter. A constant is one that starts with a lower-case letter. Let $P_0, P_1, \ldots, P_n$ be literals, a Horn clause in Datalog has the form

$$P_0 \leftarrow P_1, \ldots, P_m.$$

Semantically, it means if $P_1, \ldots, P_n$ are true then P_0 is also true. The left-hand side is called the *head* and the right-hand side is called the *body*. A clause with an empty body is called a *fact*. A clause with a nonempty body is called a *rule*.

Datalog is often used in *deductive databases*. In such settings, data records in the database are represented as Datalog facts, and the deductive engine is implemented as a Datalog program that runs on the inputs from the database. The Datalog facts representing the original database are called the *extensional database* (EDB), and the Datalog facts computed by the deductive engine are called the *intensional database* (IDB).

Datalog is more powerful than SQL, which is based on relational calculus, because Datalog predicates can be recursively defined [Ullman, 1989] [1]. Other query languages based on first-order logic are also incapable of expressing many simple recursive queries like finding the transitive closure of an input graph [Benedikt et al., 1998].

Enhanced definitions of Datalog including negation and their effects have been extensively studied in literature (see, e.g., [Ullman, 1989]). Negations can occur in the rule bodies of Datalog programs in different places: In front of EDB relations and in front of IDB relations.

There are several proposals on defining the "right" meaning to rules with negated predicates. The straightforward case involves Datalog programs with *stratified negation*, $Datalog^{not,s}$, [Chandra and Harel, 1985]. Stratified negation restricts the use of negation to non-recursive rules.

2.4 Disjunctive Datalog with Stable Models Semantics

We will use logic programs Π in $Datalog^{\vee,not}$, i.e. disjunctive Datalog programs with weak negation [Gelfond and Lifschitz, 1991, Eiter et al., 1997]. They are formed by a finite number of rules of the form

$$A_1 \vee \ldots \vee A_n \leftarrow P_1, \ldots, P_m,\ not\ N_1, \ldots,\ not\ N_k,$$

where $0 \leq n, m, k$, and A_i, P_j, N_s are (positive) atoms. All the variables in the A_i, N_s appear among those in the P_j. The constants in program Π form the (finite) Herbrand universe H of the program. The ground version of program Π, $gr(\Pi)$, is obtained by instantiating the variables in Π in all possible ways using values from H. The

[1] Some ideas from Datalog are incorporated to SQL:1999 in order to support recursive queries [Horwitz and Teitelbaum, 1986].

Herbrand base HB of Π consists of all the possible atoms obtained by instantiating the predicates in Π with constants in H.

A subset M of HB is a model of Π if it satisfies $gr(\Pi)$, i.e.: For every ground rule $A_1 \vee \ldots \vee A_n \leftarrow P_1, \ldots, P_m,\ not\ N_1, \ldots,\ not\ N_k$ of $gr(\Pi)$, if $\{P_1, \ldots, P_m\} \subseteq M$ and $\{N_1, \ldots, N_k\} \cap M = \emptyset$, then $\{A_1, \ldots, A_n\} \cap M \neq \emptyset$. M is a minimal model of Π if it is a model of Π, and Π has no model that is properly contained in M. $MM(\Pi)$ denotes the class of minimal models of Π.

Now, for $S \subseteq HB(\Pi)$, transform $gr(\Pi)$ into a new, positive program $gr(\Pi)^S$ (i.e. without *not*), as follows: Delete every rule $A_1 \vee \ldots \vee A_n \leftarrow P_1, \ldots, P_m,\ not\ N_1,$ $\ldots,\ not\ N_k$ for which $\{N_1, \ldots, N_k\} \cap S \neq \emptyset$. Next, transform each remaining rule $A_1 \vee \ldots \vee A_n \leftarrow P_1, \ldots, P_m,\ not\ N_1, \ldots,\ not\ N_k$ into $A_1 \vee \ldots \vee A_n \leftarrow P_1, \ldots, P_m$. Now, S is a *stable model* of Π if $S \in MM(gr(\Pi)^S)$. Every stable model of Π is also a minimal model of Π.

With $Datalog^{\vee,not,s}$, we denote the subclass of programs in $Datalog^{\vee,not}$ that have *stratified negation* [Eiter and Gottlob, 1995]. For these programs, the set of predicates $\mathcal{P}$ can be partitioned into a sequence $\mathcal{P}_1, \ldots, \mathcal{P}_k$ in such a way that, for every $P \in \mathcal{P}$:

1. If $P \in \mathcal{P}_i$ and predicate Q appears in a head of a rule with P, then $Q \in \mathcal{P}_i$.

2. If $P \in \mathcal{P}_i$ and Q appears positively in the body of a rule that has P in the head, then $Q \in \mathcal{P}_j$, with $j \leq i$.

3. If $P \in \mathcal{P}_i$ and Q appears negatively in the body of a rule that has P in the head, then $Q \in \mathcal{P}_j$, with $j < i$.

If a program is stratified, then its stable models can be computed bottom-up by propagating data upwards from the underlying extensional database, 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.

Programs in *Datalog* and $Datalog^{not,s}$ have a single stable model that can be computed in polynomial time in the size of the extensional database. This single stable model can be computed in a bottom-up manner starting from the extensional database. In general, disjunctive Datalog programs and those in $Datalog^{not}$ (without stratified negation) may have multiple stable models.

The (likely) higher expressive power of $Datalog^{\vee,not}$ w.r.t. *Datalog* and $Datalog^{not,s}$ is reflected in, or caused by, the (probable) difference in computational complexity. The problem of deciding if a ground atom A is entailed by a program $\Pi \in Datalog^{\vee,not}$, i.e. if A is true in all the stable models of Π, is Π_2^P-complete in the size of the EDB. This decision problem is also referred to as skeptical (cautious) query answering. The same problem can be solved in polynomial time for programs in *Datalog* and $Datalog^{not,s}$ (cf. [Dantsin et al., 2001] for more details).

The expressive power of $Datalog^{\vee,not}$ has been useful and necessary for applications to database repairs and consistent query answering (CQA) [Caniupan and Bertossi, 2010] due to the inherently rather high complexity of CQA [Bertossi, 2011].

2.5 Support Vector Machines

The *support-vector machines* technique (SVM) [Vapnik, 2009] is a form of kernel-based learning. SVM can be used for classifying vectors in an inner-product vector space $\mathcal{V}$ over $\mathbb{R}$. Vectors are classified in two classes, say with labels 0 or 1. The classification model is a hyper-plane in $\mathcal{V}$: vectors are classified depending on the side of the hyperplane they fall.

The hyper-plane has to be learned through an algorithm applied to a training set of examples, say $E = \{(\mathbf{e}_1, f(\mathbf{e}_1)), (\mathbf{e}_2,\ f(\mathbf{e}_2)), (\mathbf{e}_3, f(\mathbf{e}_3)),\ \ldots, (\mathbf{e}_n,\ f(\mathbf{e}_n))\}$. Here, $\mathbf{e}_i \in \mathcal{V}$, and for the real-valued *feature* (function) f: $f(\mathbf{e}_i) \in \{0, 1\}$.

The SVM algorithm finds an optimal hyperplane, $\mathcal{H}$, in $\mathcal{V}$ that separates the two classes in which the training vectors are classified. Hyperplane $\mathcal{H}$ has an equation of the form $\mathbf{w} \bullet \mathbf{x} + b$, where $\bullet$ denotes the inner product, $\mathbf{x}$ is a vector variable, $\mathbf{w}$ is a weight-vector of real values, and b is a real number. Now, a new vector $\mathbf{e}$ in $\mathcal{V}$ can be classified as positive or negative depending on the side of $\mathcal{H}$ it lies. This is determined by computing $h(\mathbf{e}) := sign(\mathbf{w} \bullet \mathbf{e} + b)$. If $h(\mathbf{e}) > 0$, $\mathbf{e}$ belongs to class 1; otherwise, to class 0.

It is possible to compute real numbers $\alpha_1, \ldots, \alpha_n$, the coefficients of the "support vectors", such that the classifier h can be computed through: $h(\mathbf{e}) = sign(\sum_i \alpha_i \cdot f(\mathbf{e}_i) \cdot \mathbf{e}_i \bullet \mathbf{e} + b)$ [Flach, 2014].

Chapter 3

State of the Art

3.1 A Generic Approach to ER: Swoosh

Swoosh, a generic approach to entity resolution [Benjelloun et al., 2009], considers a general *match function*, $Match(\cdot, \cdot)$, taking values *true* or *false*; and a general *merge function*, μ. The match function *Match* and the merge function μ are defined at the record (or tuple) level.

More precisely, we consider a finite set *Rec* of tuples, i.e. ground atoms of the form $R(a_1, \ldots, a_n)$, for a relational predicate $R(A_1, \ldots, A_n)$, where the A_i are attributes, with domains Dom_{A_i}, and $a_i \in Dom_{A_i}$. For $r_1, r_2 \in Rec$, $Match(r_1, r_2)$ takes the value *true* if r_1, r_2 match; otherwise, *false*. In the former case, the actual matching is the tuple $\mu(r_1, r_2) \in Rec$.

When *Match* and μ have the *ICAR* properties (idempotency, commutativity, associativity and representativity), there is a natural domination partial order on *Rec*, the *merge domination*: $r_1 \leq_s r_2$ iff $Match(r_1, r_2) = true$ and $\mu(r_1, r_2) = r_2$ [Benjelloun et al., 2009]. Domination can be extended to a partial order $\preceq_S$ on database instances.

Given an instance D, the *merge closure* of D is defined as the smallest set of records $\bar{D}$, such that includes D, and for every $r_1, r_2 \in \bar{D}$, when $M(r_1, r_2) = true$, also $\mu(r_1, r_2) \in \bar{D}$. For an instance D, its *Swoosh entity resolution* is defined as the (unique) instance $ER^S(D)$ that satisfies the conditions: (a) $ER^S(D) \subseteq \bar{D}$. (b) $\bar{D} \preceq_s ER^S(D)$. (c) No strict subset of $ER^S(D)$ satisfies the first two conditions [Benjelloun et al., 2009].

There is a particular, but still common and broad, class of match and merge functions that is based on *union of values*. This is the *union-case* for Swoosh (UC Swoosh), on which we concentrate in Section 5.4. In it, attribute values are represented as sets of finer granularity values, like objects, i.e. sets of attribute/value pairs. A common way of merging records is via their union, as objects.

Example 3.1.1 Consider the instance D_0 below.

$R(D)$	$Name$	$Street$	$City$
r_1	*J. Doe*	55	*Toronto*
r_2	*J. Doe*	*Grenadier*	*Vancouver*

The records are represented as objects as follows:

$$\begin{aligned} r_1 &= \{\underline{\langle \text{Name}, \{\text{J. Doe}\}\rangle}, \langle \text{Street}, \{55\}\rangle, \langle \text{City}, \{\text{Toronto}\}\rangle\} \\ r_2 &= \{\underline{\langle \text{Name}, \{\text{J. Doe}\}\rangle}, \langle \text{Street}, \{\text{Grenadier}\}\rangle, \langle \text{City}, \{\text{Vancouver}\}\rangle\} \end{aligned}$$

If they are considered to be similar on the basis of the underlined values, they can be merged into:

$$\begin{aligned} \mu(r_1, r_2) = \{&\underline{\langle \text{Name}, \{\text{J. Doe}\}\rangle}, \langle \text{Street}, \{55, \text{Grenadier}\}\rangle, \\ &\langle \text{City}, \{\text{Toronto}, \text{Vancouver}\}\rangle\}, \end{aligned}$$

leading to a unique clean instance. □

More generally, if S_1, S_2 are (sets of) values for attribute A, they are merged via a local merge function μ_A defined by $\mu_A(S_1, S_2) := S_1 \cup S_2$, e.g. $\mu_{City}(\{\text{Toronto}\}, \{\text{Vancou ver}\}) := \{\text{Toronto}, \text{Vancouver}\}$. The "global" merge function μ can be defined in terms of the local merge functions μ_A. The match function can also be defined in terms of local, component-based match functions. The resulting merge and match functions satisfy the ICAR properties [Benjelloun et al., 2009, Bertossi et al., 2013].

Example 3.1.2 Consider the instance D below. Attribute A takes as values finite sets of elements from the domain of an underlying, lower-level attribute $\underline{A}$. E.g. $a_1, a_2 \in Dom_{\underline{A}}, \{a_1, a_2\} \in Dom_A$. (Similarly for attribute B.) Two tuples match when the values for attribute A match, which happens when there is a pair of values in the A-sets that match: For values S_1, S_2 for A, $Match_A(S_1, S_2)$ holds when there are $v_1 \in S_1, v_2 \in S_2$ with $Match_{\underline{A}}(v_1, v_2) = true$, where $Match_{\underline{A}}$ is a lower-level match function.

Assume $Match_{\underline{A}}(a_1, a_2)$, $Match_{\underline{A}}(a_2, a_3)$ are true. The following is an ER process starting from D:

$R(D)$	A	B
	$\{a_1\}$	$\{b_1\}$
	$\{a_2\}$	$\{b_2\}$
	$\{a_3\}$	$\{b_3\}$

$\Rightarrow$

$R(D')$	A	B
	$\{a_1, a_2\}$	$\{b_1, b_2\}$
	$\{a_2, a_3\}$	$\{b_2, b_3\}$

$\Rightarrow$

$ER^S(D)$	A	B
	$\{a_1, a_2, a_3\}$	$\{b_1, b_2, b_3\}$

Here, we are not using tuple identifiers and we are also getting rid of dominated tuples, as Swoosh does. However, if we had tuple identifiers, keeping them along the ER process, the final instance above would have had three identical tuples, modulo the tuple id. □

If the merge and match functions satisfy the ICAR properties, for any instance D_0, the Swoosh entity resolution $ER^S(D_0)$ exists and is unique [Benjelloun et al., 2009, Proposition 2.1]. Although ER is well defined, it may be infinite. Even when it is finite, its computation may be very expensive. However, if the ICAR properties are satisfied by the match and merge functions for any instance D_0, like merge and match functions in UC Swoosh, then it ensures that the ER computation is tractable [Benjelloun et al., 2009].

In [Whang et al., 2009a], the original Swoosh approach to ER is extended with *negative rules*, that impose constraints on the merge results. More specifically, negative rules are used to avoid inconsistencies (e.g. with respect to semantic constraints) that could be introduced by indiscriminate matching.

Example 3.1.3 Consider the database instance D, shown blow, which is to be resolved (to represent tuples better, we are using r_i with $1 \leq i \leq 3$).

$R(D)$	*name*	*phone*	*gender*
r_1	$\{Mishael\}$	$\{7654321\}$	$\{\}$
r_2	$\{Michael\}$	$\{\}$	$\{M\}$
r_3	$\{Mishale\}$	$\{7654322\}$	$\{F\}$

If two tuples r, r' have similar names, then $M(r, r') = \mathit{true}$. Assume $\mathit{Mishael} \approx \mathit{Michael}$, $\mathit{Mishael} \approx \mathit{Mishale}$. Suppose first r_1 and r_2 are merged into new tuple r_{12}.

$R(D_1)$	*name*	*phone*	*gender*
r_{12}	$\{\mathit{Mishael},\ \mathit{Michael}\}$	$\{7654321\}$	$\{M\}$
r_3	$\{\mathit{Mishale}\}$	$\{7654322\}$	$\{F\}$

Using the original Swoosh approach, we could obtain the following final result:

$R(D_2)$	*name*	*phone*	*gender*
r_{123}	$\{\mathit{Mishael},\ \mathit{Michael},\ \mathit{Mishale}\}$	$\{7654321, 7654322\}$	$\{F, M\}$

However, a *negative rule* prohibiting for a person to be both M and F, would avoid reaching this instance. This might require to appeal to an external expert, to make the right merge decisions.

Assume there is another negative rule stating an inconsistency in an ER instance if there exist two persons with an identical phone number. Here, $R(D_2)$ is consistent w.r.t this negative rule. □

In [Whang et al., 2009a], it is formally defined what the *valid* ER instance is in the presence of negative rules. Given an initial instance D and the merge closure $\bar{D}$, *a valid* entity resolution instance of D is a consistent set of tuples D' that satisfies the following conditions: (a) $D' \subseteq \bar{D}$, (b) $\forall r_1 \in \bar{D} - D'$: $\exists r_2 \in D'$ such that r_2 dominates r_1 or $D' \cup \{r_1\}$ is inconsistent, (c) no strict subset of D' satisfies the first two conditions, (d) no other instance satisfying the first three conditions dominate D'.

In general, there can be more than one valid ER instance. In [Whang et al., 2009a], it is discussed how a domain expert can *guide* the ER process to capture a desirable and valid set of tuples in the ER instance. In other words, the expert looks at the tuples, and selects one that is consistent, non dominated and more desirable to have in the final instance in order to prevent inconsistencies.

3.2 The Union Case for Swoosh via MDs

As Section 3.1, we assume that records correspond to ground tuples of a single relational predicate, say R, and *Rec* denotes the set of records.

A direct MD-based reconstruction of union case of Swoosh has been proposed in [Bertossi et al., 2013]. More precisely, a possibly denumerable domain D_{A_i} is considered for each attribute A_i of R, $1 \leq i \leq n$, and a reflexive and symmetric similarity relation $\approx_{A_i}$ is defined for each domain. Now, the domain for each A_i of R becomes Dom_{A_i} which consists of all subsets of D_{A_i}, i.e. Dom_{A_i} contains sets. Thus, the elements of *Rec* are of the form $R(s_1, ..., s_n)$, with each s_i being a set that belongs to Dom_{A_i}.

The similarity relation $\approx_{A_i}$ on D_{A_i} induces a similarity relation $\approx_{\{A_i\}}$ on Dom_{A_i}, as follows: $s_1 \approx_{\{A_i\}} s_2$ holds iff there exists $a_1 \in s_1$ and $a_2 \in s_2$ such that $a_1 \approx_{A_i} a_2$.

Matching functions $m_{\{A_i\}}$ on $Dom_{A_i} \times Dom_{A_i}$ are considered such that $m_{\{A_i\}}(s_1, s_2) = s_1 \cup s_2$. Matching functions $m_{\{A_i\}}$ are similarity preserving w.r.t. $\approx_{\{A_i\}}$. In fact, if $s_1 \approx_{\{A_i\}} s_2$, then there are $a_1 \in s_1$, $a_2 \in s_2$ with $a_1 \approx_{A_i} a_2$. Since a_2 also belongs to $s_2 \cup s_3$, for every $s_3 \in Dom_{A_i}$, it holds $s_2 \cup s_3 = m_{\{A_i\}}(s_2, s_3) \approx_{\{A_i\}} s_1$. Based on these definitions, given $r_1 = R(\bar{s}^1)$, $r_2 = R(\bar{s}^2)$: (a) $Match(r_1, r_2)$ holds iff for some i, $s_i^1 \approx_{\{A_i\}} s_i^2$. (b) When $Match(r_1, r_2) = true$, $\mu(r_1, r_2) = R(m_{\{A_1\}}(s_1^1, s_1^2), ..., m_{\{A_n\}}(s_n^1, s_n^2))$.

Example 3.2.1 (ex. 3.1.3 cont.) Consider domain D_{name} for attribute *name*, and similarity relation $\approx_{name}$ for this domain. Then, Dom_{name} consists of all subsets of D_{name}. Therefore, we have $Dom_{name} = \{\{Mishael,\ Michael,\ Mishale\}, \{Mishael,\ Michael\}\ , \{Mishale\}, \ldots\}$. $\{Mishael,\ Michael\} \approx_{\{name\}} \{Mishale\}$ holds since $Mishael \approx_{name} Mishale$, and $m_{\{name\}}(\{Mishael,\ Michael\}, \{Mishale\}) = \{Mishael,\ Michael,\ Mishale\}$. □

Now, UC Swoosh framework with *Match* and μ on Dom_A are reconstructed by means of the set Σ^S of MDs $R[A_i] \approx_{\{A_i\}} R[A_i] \rightarrow R[A_j] \doteq R[A_j]$ for $1 \leq i, j \leq n$ where the RHSs have to be applied by matching functions $m_{\{A_j\}}$. Consistently with the MD framework, it is also assumed that tuples of *Rec* have tuple identifiers which are the first and extra attribute of relation R. In consequence, the elements of D and

(D,Σ^S)-clean instance D^m under the MD framework are of the form $R(t,\bar{s})$, and those in D and the entity resolution instance $ER^S(D)$ obtained directly via UC Swoosh are the records r of the form $R(\bar{s})$. There is a single (D,Σ^S)-clean instance D^m since matching function $m_{\{A_i\}}$ is similarity preserving.

The following example explicitly shows how Σ^S can be defined for reconstructing union case of Swoosh by MDs.

Example 3.2.2 (ex. 3.1.2 cont.) Assume that instance $R(D)$ has tuple identifiers t_i with $1 \leq i \leq 3$). For reconstructing UC Swoosh framework by MDs, we consider Σ^S consisting of the following MDs. Observe that two tuples match when the values for attribute A match. Then, they should be merged.

$$\varphi_1 : R[A] \approx_{\{A\}} R[A] \longrightarrow \quad R[A] \doteq R[A]$$
$$\varphi_2 : R[A] \approx_{\{A\}} R[A] \longrightarrow \quad R[B] \doteq R[B]$$

$R(D^m)$	A	B
t_1	$\{a_1, a_2, a_3\}$	$\{b_1, b_2, b_3\}$
t_2	$\{a_1, a_2, a_3\}$	$\{b_1, b_2, b_3\}$
t_3	$\{a_1, a_2, a_3\}$	$\{b_1, b_2, b_3\}$

More precisely, the MDs state that if the values for attribute A are similar in two tuples based on $\approx_{\{A\}}$, then they should have identical values for attributes A and B meaning that the two tuples should be merged. □

In [Bertossi et al., 2013], it is proved that in UC Swoosh the clean instance of D resulting from the chase procedure (cf. Definition 2.2.2) with MDs Σ^S, similarity relations $\approx_{\{A_i\}}$ and matching functions $m_{\{A_i\}}$ is equivalent to the Swoosh entity resolution $ER^S(D)$ (more precisely, they are equivalent if we look at the reduced version of the clean instance).

3.3 Rules and Ontologies for Duplicate Detection and Merging

In the previous sections we have mostly concentrated on the merge part of entity resolution. However, identifying similarities and duplicates is also an important and

common problem [Bleiholder and Naumann, 2008]. There are some declarative approaches to duplicate detection. They could be naturally combined with declarative approaches to merging.

A declarative framework for collective entity deduplication of large data sets using domain-specific soft and hard constraints is proposed in [Arasu et al., 2009]. The constraints specify the deduplication. They use a novel Datalog-style language, *Dedupalog*, to write the constraints as rules. The deduplication process tries to satisfy all the hard constraints, but minimizing the number of violations to the soft constraints. Dedupalog is used for identifying groups of tuples that are candidates for merging. They do not do the merging or base their work on MDs.

Another declarative approach to ER is presented in [Sais et al., 2007]. The emphasis is placed mainly on the detection of duplicates rather than on the actual merging. An ontology expressed in a logical language based on RDF-S and OWL-DL is used for this task. Reconciliation rules are captured by SWRL, a rule language for the semantic web. Also negative rules that prevent reconciliation of certain values can be expressed, much in the spirit of Swoosh with negative rules [Whang et al., 2009a].

Recently, a declarative framework for entity linking is developed in [Burdick et al., 2015]. The framework is based on the use of constraints. The authors consider ER as a problem of defining links between entities. The salient feature of this framework is linking entities that are not necessarily of the same type, for example, linking a manager with her company. The adopted constraints enable the declarative listing of all the reasons as to why two entities should be linked.

A clustering-based approach to collective deduplication is proposed in [Bhattacharya and Getoor, 2007]. While traditional deduplication techniques assume that only similarities between attribute values are available, in relational data the entities are assumed to have additional relational information that can be used to improve the deduplication process. For example, when resolving author names, looking at co-authors can be helpful as an author may regularly work together with the same co-authors, which would then help in identifying both as the same individuals. This process is referred as *collective deduplication*.

More precisely, in [Bhattacharya and Getoor, 2007], a relationship graph is built

whose nodes are the entities (records), and edges connect nodes if there is a logico-relational relationship between them as captured by logical constraints. The graph supports the propagation of similarity information to related entities. In particular, the similarity between two nodes is calculated as the weighted sum of the attribute-value similarity and their relational similarity (as captured through the graph). Experimental results in [Bhattacharya and Getoor, 2007] show that this form of *collective deduplication* outperforms traditional deduplication.

3.4 On-the-fly ER in Data Integration

The problem of ER has a prominent role in the data integration literature. Actually, a virtual data integration system (VDIS) that is fed by several independent sources cannot be cleaned as a stand alone, single database. The cleaning has to be done on-the-fly, at query answering time. However, most existing ER techniques are aimed at the offline processing of static databases [Bertossi et al., 2013, Gardezi et al., 2012, Dong et al., 2012, Arasu et al., 2009], and limited work exists on the applicability of query rewriting methodologies for computing clean answers under ER.

Several works have considered on-the-fly ER techniques [Bhattacharya and Getoor, 2007, Ioannou et al., 2001, Sismanis et al., 2009]. The technique in [Bhattacharya and Getoor, 2007] answers queries collectively using a two-step "expand and resolve" algorithm. It retrieves the related records for a query using two expansion operators, and then answers the query by only considering the extracted records. However, this approach just considers selection queries where the type of the condition attribute is a string, e.g. a query to to retrieve all books written by author "J. Doe". It does not take into account other kinds of selection queries, such as range queries. This approach is computationally very expensive. Hence, it is not scalable to large databases.

The approach in [Ioannou et al., 2001] is also on-the-fly ER, but it solves a different problem, that of queries under data uncertainty by connecting ideas of ER and probabilistic databases. The term query refers to a combination of (attribute-name/value) pairs and each record retrieved as an answer is accompanied by a probability that this record will be selected amongst all possible worlds. More specifically, possible

duplicate records (which have a probability attached to them) are kept in a database, and the attributes values of all records have their attached probabilities. These probabilities show the confidence one has in the correctness of the attribute values in a record. When a query record is given to such a probabilistic database, a matching and merging step of entity records is guided based on the probabilities, and all merged records that fulfil the query terms are being returned to the user.

In [Sismanis et al., 2009], the authors handle entity uncertainty at query time for OLAP applications. This approach assumes the existence of a record-to-cluster mapping table and its goal is to answer group-by OLAP queries by retrieving results in the form of strict ranges.

More recently, a query rewriting-based methodology for polynomial-time clean query answering under MDs is presented in [Gardezi and Bertossi, 2012]. The rewriting uses recursive Datalog with stratified negation and aggregation, but applies to a restricted class of conjunctive queries and MDs.

3.5 Unifying Data Repairing and Deduplication

Integrity constraints (ICs) capture the semantics of data and are expected to be satisfied by a database in order to keep its correspondence with the outside reality it is modeling. For several reasons, databases may become inconsistent with respect to a given set of ICs. A database instance D, that is expected to satisfy certain integrity constraints may fail to do so. In this case, a *repair* of D is a database D' that does satisfy the integrity constraints and *minimally departs* from D. Different forms of minimality can be applied and investigated [Arenas et al., 1999 , Greco et al., 2003, Bertossi, 2011]. A number of studies have been proposed to interleave data repairing and deduplication in one single framework [Fan et al., 2011, Geerts et al., 2013, Geerts et al., 2014].

In [Fan et al., 2011], the interaction of MDs and ICs on single databases has been partially investigated. In that case, the ICs are conditional functional dependencies (CFDs). In particular, this work on the interaction between MDs and data repairs combines record matching and data repairing for better data quality. MDs are used for ER, while CFDs are used to specify certain equalities of values within a given relation.

In order to obtain a clean instance, certain pre-defined strategies are followed (e.g., using master data), to actually force the correction of attribute values. However, the work does not have a unified formal semantics, with a precise definition of a clean instance.

Furthermore, [Geerts et al., 2013] develops a uniform framework to solve repair problems in a single database instance that involves different kinds of constraints, including equality-generating dependencies (egds), MDs, CFDs. The core contribution of the paper consists in the definition of a novel semantics for the data cleaning problem. More specifically, the authors reformulate repairing as finding intended instances using a chase procedure. The chase uses the notion of the partial order to clean a dirty instance as the process of upgrading its quality, similar to [Bertossi et al., 2013].

Following this approach, [Geerts et al., 2014] builds on [Geerts et al., 2013], and making some contributions to data transformation and data cleaning. In particular, they propose a general framework for schema mapping and data cleaning that can be used to generate solutions to data transformation scenarios, and to repair conflicts and inconsistencies with respect to a class of constraints. More specifically, they propose a new semantics representing a conservative extension of previous semantics for schema mappings and data repairing. Based on the semantics, a chase is introduced to compute the intended instances. Similar to [Geerts et al., 2013], the notion of partial order is incorporated to the chase.

3.6 Support Vector Machines Techniques for ER

ER can be handled as a supervised learning problem, if training data is present. In this direction, classification techniques has been applied, in particular SVMs [Bilenko and Mooney, 2003a, Bilenko and Mooney, 2003b, Christen and Goiser, 2010].

For supervised classification techniques, selection of training examples may be manual, semi-automatic or automatic. With manual selection entities have to be chosen and labeled by a user. Semi-automatic selection still requires a human for labeling, but entity pairs are automatically chosen for labeling. Automatic selection provides and labels training examples automatically without any inspection by a

user [Kopcke and Rahm, 2010].

In [Bilenko and Mooney, 2003a, Bilenko and Mooney, 2003b], the MARLIN (Multiply Adaptive Record Linkage with Induction) framework a SVMs classifier has been developed to learn the costs for edit operations (such as character inserts, deletes or substitutions). The goal is to have a better separation of the string pairs associated to duplicate pairs from those associated to non-duplicate pairs, by learning the mentioned costs. The training data required for this approach consists of pairs of strings and their duplicate and non-duplicate status.

In [Christen, 2008], an automatic classification approach is proposed for duplicate detection based on a SVMs technique. The approach consists of two steps. Firstly, training examples are automatically selected from the set of all weight vectors W. Two strategies for automatic selection of training examples are supported: threshold- and nearest-based. For specifying duplicate and non-duplicate training vectors, the threshold method selects entity vectors whose all similarity values are within a certain distance to the exact similarity value 1 or to total dissimilarity 0. The nearest method sorts the similarity vectors of the entity vectors according to their distances from the vectors containing only exact similarities and only total dissimilarities, resp, and then selects the nearest entity vectors for training. In this way, the duplicate training examples set W_D, and the non-duplicate training examples set W_N are created.

Based on the initial training set $W_T = W_D \cup W_N$, a first SVM classifier is trained. Then, the set of comparison vectors W_U that is not selected for training data, i.e. $W_U = W \setminus W_T$, is classified using first trained SVM classifier. In the second step, the comparison vectors from W_U that were classified to be furthest away from the SVM decision boundary are removed from W_U and added into the training set W_T. In this way, a new training set W'_T is made, and a second SVM classifier is trained on this enlarged training set W'_T. This process of adding more comparison vectors into the training set followed by training a new SVM classifier is repeated until there is no unclassified weight vector in W_U.

3.7 Blocking Techniques for ER

Several blocking techniques have been proposed to significantly improve the efficiency of ER by avoiding a possibly expensive similarity computation for all record pairs and computing the similarity for only blocks of record pairs [Christen, 2011]. For a comprehensive survey on blocking see [Baxter et al., 2003, Draisbach and Naumann, 2009, Christen, 2011, Christen, 2012]. However, most prior approaches to blocking are inflexible for at least one of two reasons: (1) They allow blocking of only a single entity type in isolation, (2) They ignore valuable domain or semantic knowledge that can be used for blocking. Possible exceptions are [Rastogi et al., 2011, Nin et al., 2007].

In [Rastogi et al., 2011], similarity of blocking keys and relational closeness are considered for entity deduplication (not the merging). However, the semantics of relational closeness between blocking keys and entities is not thoroughly developed.

Semantic blocking in [Nin et al., 2007] completely disregards blocking keys and creates blocks by considering exclusively the relationships between entities. At its core lies a collaborative graph, where every node corresponds to an entity and every edge connects two associated entities. For instance, the collaborative graph for a bibliographic data collection can be formed by mapping every author to a node and adding edges between co-authors. In this context, blocks are created in the following way: for each node n, a new block is formed, containing all nodes connected with n through a path, whose length does not exceed a predefined limit.

From another point of view, blocking techniques are generally distinguished in two categories: those that produce disjoint blocks, such as standard blocking [Fellegi and Sunter, 1969], and those techniques that yield overlapping blocks with redundant comparisons, such as meta-blocking [Papadakis et al., 2014, Papadakis et al., 2016a], in an effort to achieve high recall in the context of highly heterogeneous data, such as the Web of Data [Bizer et al., 2009]. Redundancy comes at the cost of lower efficiency since it may increase the number of required pair-wise comparisons.

Meta-blocking [Papadakis et al., 2014, Papadakis et al., 2016a] has been introduced as a generic procedure that intervenes between the creation and the processing of blocks, transforming an initial set of blocks into a new one with substantially fewer

comparisons and equally high effectiveness. Basically, a meta-blocking technique receives as input an existing block collection, and transforms it to a new block collection that contains fewer unnecessary comparisons. To this end, it first builds an abstract graph representation of the original set of blocks, with the nodes corresponding to entities and the edges connecting the co-occurring ones. During the creation of this structure all redundant comparisons are discarded, while the superfluous ones can be removed by pruning the edges with the lowest weight.

Despite the significant enhancements in efficiency, meta-blocking techniques suffer from a crucial drawback: processing of voluminous datasets involves a significant overhead. The corresponding blocking graphs comprise millions of nodes that are strongly connected with billions of edges. Inevitably, the pruning of such graphs is very time-consuming [Papadakis et al., 2016a].

Iterative Blocking [Whang et al., 2009b] propagates all identified duplicates to the subsequently processed blocks so as to save repeated comparisons and to detect more duplicates. Hence, it improves both precision and recall. Similar to Meta-blocking, it targets redundant comparisons between duplicate entities.

Chapter 4

Thesis Contributions

This thesis makes the following specific contributions on the topic of MDs and entity resolution:

1. We present cleaning answer set programs (ASPs), in the form of *disjunctive Datalog* programs with stable model semantics. They capture through their models the class of alternative clean instances obtained after an ER process based on MDs.

2. Clean answers to a query posed to a "dirty" instance are not defined via a set-theoretic intersection (as usual in answer set programming), but via the lattice-theoretic *greatest lower bound* (glb) (cf. Section 2.2). We introduce some additional rules into a given query program that capture the *glb* in set-theoretic terms. In this way, the clean answers are obtained by set-theoretic cautious reasoning from a cleaning program.

3. As an important special and practical case of ER, we provide a declarative reconstruction of the so-called union-case ER methodology, as presented through a generic Swoosh approach to ER. Swoosh has been extended with negative rules. Accordingly, we extend our ASP-based account of the union-case of Swoosh by considering negative rules.

4. We introduce relational MDs, and identify a new class of combinations of relational MDs and initial instances that have good properties in terms of the number of clean instances: a single one, in our case, and computable in polynomial time.

5. We describe the *ERBlox* system in which relational MDs and machine learning methods, including preliminary tasks such as blocking, can all be integrated

into a single Datalog-driven platform. We built *ERBlox* on top of the *LogicBlox* platform. More specifically, MDs declaration and enforcement, data processing in general, and machine learning are all be integrated using the *LogiQL* language.

6. We propose MD-based collective blocking where relational MDs are used for blocking records before classification in the *ERBlox* system, which is a novel and not original intended use of MDs.

7. We show experiments with our *ERBlox* system using as dataset a snapshot of Microsoft Academic Search (MAS) (as of January 2013) that includes 250K authors, 2.5M papers, and a training set. We also use, independently, datasets from DBLP and Cora Citation.

8. We show that our system improves ER recall and precision over traditional, standard blocking techniques [Jaro, 1989], where just blocking-key similarities are used. Actually, MD-based collective blocking leads to higher precision and recall on the given datasets.

Chapter 5

Answer Set Programs for MD-based ER

A natural research goal is to come up with a general methodology to logically specify the result of an MD-based ER process. More precisely, the aim is to compactly and declaratively specify the class of clean instances for an instance D_0 subject to ER on the basis of a set Σ of MDs. In principle, a logic-based specification of that kind could be used to reason about/from the class of clean instances, in particular, enabling a process of clean query answering. In this chapter we present logic-based specifications such that the logical language of choice will be that of answer set programs (ASPs) [Brewka et al., 2011].

We start by showing that clean query answering is a non-monotonic process in the sense that the set of clean answers does not monotonically grow with the database instance.

Example 5.0.1 Consider the MD φ: $R\,[phone, addr] \approx R\,[phone, addr] \rightarrow R\,[addr] \doteq R\,[addr]$, and the following instance D:

$R(D)$	*name*	*phone*	*addr*
t_1	John Doe	(613)7654321	Bank St.
t_2	Alex Smith	(514)1234567	10 Oak St.

D is a stable, clean instance w.r.t. φ. Now consider the query asking for the address of John Doe: $\mathcal{Q}(z) : \exists y R(John\ Doe, y, z)$. In this case, $Clean^{D}_{\{\varphi\}}(\mathcal{Q}) = \mathcal{Q}(D) = \{\langle Bank\ St.\rangle\}$.

Now, suppose that D is updated into D':

$R(D')$	*name*	*phone*	*addr*
t_1	John Doe	(613)7654321	Bank St.
t_2	Alex Smith	(514)1234567	10 Oak St.
t_3	J.Doe	7654321	25 Bank St.

Assuming that "(613)7654321" $\approx$ "7654321", *Bank St.* $\approx$ *25 Bank St.*, and also $m_{addr}(Bank\ St., 25\ Bank\ St.) = 25\ Bank\ St.$, then D'' below is the only clean instance:

$R(D'')$	*name*	*phone*	*addr*
t_1	John Doe	(613)7654321	25 Bank St.
t_2	Alex Smith	6131234567	10 Oak St.
t_3	J.Doe	7654321	25 Bank St.

Now, $Clean^{D'}_{\{\varphi\}}(\mathcal{Q}) = \mathcal{Q}(D'') = \{\langle 25\ Bank\ St.\rangle\}$. Clearly, $\mathcal{Q}(D) \not\subseteq \mathcal{Q}(D')$, even though $D \subseteq D'$. The previous answer was lost after the update. □

This example shows the specification mentioned above must appeal to some sort of non-monotonic logical formalism. Intuitively, when an MD is enforced on two tuples of an instance in a single step of the chase procedure, the tuples are updated to newer versions. The older versions of the tuples are no longer available during the rest of the chase. The chase grows in terms of information contents.

In this chapter, we use *answer set programs* (ASPs) as the basic formalism to capture the result of this non-monotonic chase procedure. More precisely, given an instance D_0 and a set Σ of MDs, we propose logic programs $\Pi(D_0, \Sigma)$ with stable model semantics whose stable models correspond to the (D_0, Σ)-clean instances. On this basis, the clean answers to a query posed to D_0 can be obtained via *cautious reasoning* from the program.

The main idea is that program $\Pi(D_0, \Sigma)$ implicitly simulates the chase sequences, each one represented by a model of the program. For this, $\Pi(D_0, \Sigma)$ has rules to: (a) enforce MDs on pairs of tuples satisfying similarities conditions, (b) create newer versions of those tuples by applying MFs, (c) make the older versions of the tuples unavailable for further matchings, and (d) make each stable model correspond to *a valid chase sequence*, leading to a clean instance. The latter is the most intricate part.

The program $\Pi(D_0, \Sigma)$ explicitly eliminates, using *program constraints*, instances that are the result of *illegal* applications of MDs. A set of matching applications is illegal if we cannot put them in a chronological order to represent the steps of a chase. That is, there are some matchings that use old versions of tuples that have already been replaced by new versions.

To ensure that the matchings are enforced according to an order that correctly represents a chase, we will record pairs of matchings in an auxiliary relation, *Prec*, used by the cleaning program, and explicitly impose an order on *Prec* via program constraints.

5.1 Cleaning Programs for MDs

Let D_0 be a given, possibly dirty initial instance w.r.t. a set Σ of MDs. The *cleaning program* $\Pi(D_0, \Sigma)$ that we will introduce contains an $(n+1)$-ary program predicate R_i, for each n-ary database predicate R_i, plus other predicates to be introduced below. Predicate R_i will be used in the form $R_i(T, \bar{Z})$, where T is used for the tuple identifier database attribute, and $\bar{Z}$ is a list standing for the (ordinary) attributes of R_i. If you assume the database already has tuple identifiers (tid's), you do not need this distinction.

For every attribute A in the schema, with domain Dom_A, the 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 $M_A(X, Y, Y)$. When an attribute A (or its domain) does not have a matching function, because it is not affected by an MD, then $\preceq_A$ 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 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 componentwise comparable attribute values, $\bar{X}_1 \approx \bar{X}_2$ denotes the conjunction $X_1^1 \approx_1 X_2^1 \wedge \ldots \wedge X_1^l \approx_l X_2^l$.

For a given instance D_0 and set of MDs Σ, the program $\Pi(D_0, \Sigma)$ contains the rules in **1.-9.** below:

1. For every tuple (id) $t^{D_0} = R_j(\bar{a})$, the fact $R_j(t, \bar{a})$. We also need facts for the MFs as tables and similarity relations.

2. For each MD φ_j: $R_1[X_1] \approx R_2[X_2] \rightarrow R_1[A_1] \doteq R_2[A_2]$, the program rule:

$$Match_{\varphi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2) \vee NotMatch_{\varphi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2) \leftarrow$$
$$R_1(T_1, \bar{X}_1, Y_1),\ R_2(T_2, \bar{X}_2, Y_2), \bar{X}_1 \approx \bar{X}_2,\ Y_1 \neq Y_2.$$

Here, the $\bar{X}_i$s are lists of variables corresponding to lists of attributes on the LHS of the MD, whereas the Y_is are single variables corresponding to the attribute on the RHS of the MD. Also, the $\bar{Z}_i$s are lists of variables corresponding to all attributes in a tuple, and T_i is a variable standing for a tuple ID. We use this notation to make the association with attributes or lists thereof in MDs easier.

This rule is used to capture possible matchings when similarities hold for two tuples. Notice that predicate *Match* does not do the actual merging; and we need the freedom to match or not to match, to obtain different chase sequences (cf. below).

When the two predicates appearing in φ_j are the same, say R_1, the above rule becomes symmetric w.r.t. every two atoms $R_1(t_1, \bar{a}_1)$ and $R_1(t_2, \bar{a}_2)$ that satisfy the body of the rule. We need to make sure that if the matching takes place for these two tuples, then both $Match_{\varphi_j}(t_1, \bar{a}_1, t_2, \bar{a}_2)$ and $Match_{\varphi_j}(t_2, \bar{a}_2, t_1, \bar{a}_1)$ exist. Thus, for every such MD, we need a rule of the following form

$$Match_{\varphi_j}(T_2, \bar{X}_2, Y_2, T_1, \bar{X}_1, Y_1) \;\leftarrow\; Match_{\varphi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2).$$

3. *Match* does not take place if one of the involved tuples was used for another matching, and replaced by newer version. For each MD φ_j, we need:

$$\leftarrow NotMatch_{\varphi_j}(T_1, \bar{Z}_1, T_2, \bar{Z}_2), not\ OldVersion_1(T_1, \bar{Z}_1), not\ OldVersion_2(T_2, \bar{Z}_2).$$

This is a *program constraint* filtering out models that make the body true. More precisely, the program constraint states that if: (a) we have "live", never replaced versions of two tuples (ids) t_1 and t_2 from relations R_1 and R_2, respectively, (b) the similarity conditions holds for them according to an MD, and (c) both are not matched (together or with some other tuples), then the model should be rejected. That is, t_1 and t_2 have to be either matched together, or be replaced by newer versions (becoming unavailable). This constraint enforces at least one match for a tuple that satisfies some match condition.

4. Predicate $OldVersion_i$ contains different versions of every tuple (id) in relation

R_i which has been replaced by a newer version (during the ER process). This is captured by upward lattice navigation:

$$\mathit{OldVersion}_i(T_1, \bar{Z}_1) \ \leftarrow R_i(T_1, \bar{Z}_1),\ R_i(T_1, \bar{Z}_1'), \bar{Z}_1 \preceq \bar{Z}_1',\ \bar{Z}_1 \neq \bar{Z}_1'.$$

For each tuple identifier t there could be many atoms of the form $R_i(t, \bar{a})$ corresponding to different versions of the tuple associated with t that represent the evolution of the tuple during the enforcement of MDs.

For convenience, below we refer to the various atoms associated with a given tuple identifier t as versions of the tuple identifier t.

We haven't done the actual merging yet. This is done next.

5. Rules to insert new tuples into R_1, R_2, as a result of enforcing φ_j (M_j stands for the MF for the RHS of φ_j):

$$\begin{aligned} R_1(T_1, \bar{X}_1, Y_3) \ &\leftarrow\ \mathit{Match}_{\varphi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2),\ M_j(Y_1, Y_2, Y_3). \\ R_2(T_2, \bar{X}_2, Y_3) \ &\leftarrow\ \mathit{Match}_{\varphi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2),\ M_j(Y_1, Y_2, Y_3). \end{aligned}$$

All the previous rules tell us what can be done or not, but not exactly how to combine those possibilities. We need additional structure to create valid chase sequences, which are ordered sequences of instances within a partial order of instances. Those ordered sequences are captured by additional conditions (cf. below).

6. For every two matchings applicable to different versions of a tuple with a given identifier, we record in *Prec* the relative order of the matchings.

This predicate applies to *two pairs of tuples*, and to *two possible matchings of two tuples*. The matching applied to the smaller version of the tuple w.r.t. $\preceq$ has to precede the other.

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

A more clear reading of this predicate arguments could be as follows: $Prec(\langle T_1, \bar{Z}_1\rangle, \langle T_2, \bar{Z}_2\rangle | \langle T_1, \bar{Z}'_1\rangle, \langle T_3, \bar{Z}_3\rangle)$, emphasizing that two pairs are being related.

We need similar rules (four in total) for the cases where the common tuple identifier variable T_1 appears in different components of the two *Match* predicates (cf. rules **6.** in Example 5.1.1 below).

7. Each version of a tuple identifier can participate in more than one matching only if at most one of them changes the tuple. For every two matchings applicable to a single version of a tuple identifier, we record in *Prec* the relative order of the two matchings. The matching that produces a new version for the tuple has to come after the other matching. If both of the matchings do not produce a new version of the tuple, they can be applied in any order, making unnecessary to record their relative order in *Prec*.

$$Prec(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2, T_1, \bar{X}_1, Y_1, T_3, \bar{X}_3, Y_3) \leftarrow$$
$$Match_{\varphi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2), Match_{\varphi_k}(T_1, \bar{X}_1, Y_1, T_3, \bar{X}_3, Y_3), M_A(Y_1, Y_3, Y_4),$$
$$Y_1 \neq Y_4.$$

Here, M_A is the built-in matching function (relation) for attribute (generically denoted by) A to whose values variables Y_1, Y_2, Y_3 refer to. This rule says that, in case (a ground version of) a tuple $\langle T_1, \bar{X}_1, Y_1\rangle$ participates in two matching, via MDs φ_j and φ_k, and the tuple changes according to φ_k, as captured by the last two body atoms that use φ_k's matching function M_k, then the matching via φ_k must come after the matching via φ_j. By this same rule, the reverse *Prec*-order could also be true, but we will disallow having both by imposing conditions on *Prec*, making it a partial order (see below). By the rule(s) in 2. above, a stable model can always choose between doing a matching or not, and then choosing between one of the two possible *Prec*-orders. As in rules **6.** above, we need four rules of this form, for different possible appearances of the common variable T_1 (cf. rules **7.** in Example 5.1.1 below).

This rule disallows two matchings that produce incomparable versions of a tuple w.r.t. $\preceq$, because *Prec* is antisymmetric (due to rules **8.** below). As a consequence, every two matchings applicable to a given tuple identifier will fire one of the two rules

6. or **7.**, and they will have a relative order recorded in *Prec*, unless they both do not change the tuple.

8. With the definitions so far, *Prec* could still not be an order relation, exhibiting antisymmetry. Program constraints are used to make it an order, Consequently, rules for making *Prec* a reflexive, antisymmetric and transitive relation, respectively:

$$Prec(T_1, \bar{Z}_1, T_2, \bar{Z}_2, T_1, \bar{Z}_1, T_2, \bar{Z}_2) \ \leftarrow Match_{\varphi_j}(T_1, \bar{Z}_1, T_2, \bar{Z}_2).$$

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

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

Notice that we do not use *Prec* in body conditions. In consequence, the main rules around it are the last two program constraints. They are used to to eliminate instances (models) that result from illegal applications of MDs. As a consequence, each stable model will represent a particular version of that order; and different orders correspond to different models, and to different chase sequences.

9. Finally, rules to collect in R_i^c the latest version of each tuple for every predicate R_i; they are used to form the clean instances.

$$R_i^c(T_1, \bar{Z}_1) \ \leftarrow \ R_i(T_1, \bar{Z}_1),\ not\ OldVersion_i(T_1, \bar{Z}_1).$$

Notice that the rules in **2.** above are the only one that depend on an essential manner on the particular MDs at hand. Rules **1.** are just the facts that represent the initial, underlying database, the MFs as tables, and similarity relations. All the other rules are basically generic, and could be used by any cleaning program, as long as

there is a correspondence between the predicates $Match_\varphi$ with the MDs φ, for which the former have subindices for the latter.

Notice that, given a relational schema and a set of MDs on it, a program like the one above can be automatically created, and can be used for that schema and MDs. Only the facts of the program depend on the actual relational instance at had. An alternative to our approach would be to build a *single* program that can be used with any schema and finite set of MDs associated to it. Such a program is bound to be much more complex than those, specific but still generic, that we are proposing here.

Example 5.1.1 (ex. 1.0.2 cont.) The cleaning program $\Pi(D_0, \Sigma)$ has the following rules: (skipping rules **6.**)

1. $R(t_1, a_1, b_1)$. $R(t_2, a_2, b_2)$. $R(t_3, a_3, b_3)$., plus $M_B(b_1, b_2, b_{12})$. $M_B(b_2, b_3, b_{23})$. $M_B(b_1, b_{23}, b_{123})$. $a_1 \approx a_2$. $b_2 \approx b_3$.

From now on we will omit extensions for the MFs and similarity relations.

2.

$$Match_{\varphi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2) \vee NotMatch_{\varphi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2) \leftarrow R(T_1, X_1, Y_1),\ R(T_2, X_2, Y_2),\ X_1 \approx X_2,\ Y_1 \neq Y_2.$$

$$Match_{\varphi_2}(T_1, X_1, Y_1, T_2, X_2, Y_2) \vee NotMatch_{\varphi_2}(T_1, X_1, Y_1, T_2, X_2, Y_2) \leftarrow R(T_1, X_1, Y_1),\ R(T_2, X_2, Y_2),\ Y_1 \approx Y_2,\ Y_1 \neq Y_2.$$

$$Match_{\varphi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2) \leftarrow Match_{\varphi_1}(T_2, X_2, Y_2, T_1, X_1, Y_1).$$

(similarly for $Match_{\varphi_2}$)

3.

$$\leftarrow NotMatch_{\varphi_1}(T_1, X_1, Y_1, T_2, X_2, Y_2),\ not\ OldVersion_R(T_1, X_1, Y_1),\ not\ OldVersion(T_2, X_2, Y_2).$$

(similarly for $NotMatch_{\varphi_2}$)

4.

$$\mathit{OldVersion}(T_1, \bar{Z}_1) \ \leftarrow \ R(T_1, \bar{Z}_1),\ R(T_1, \bar{Z}'_1),\ \bar{Z}_1 \preceq \bar{Z}'_1,\ \bar{Z}_1 \neq \bar{Z}'_1.$$

5.

$$R(T_1, X_1, Y_3) \ \leftarrow \ \mathit{Match}_{\varphi 1}(T_1, X_1, Y_1, T_2, X_2, Y_2), M_B(Y_1, Y_2, Y_3).$$
$$R(T_1, X_1, Y_3) \ \leftarrow \ \mathit{Match}_{\varphi 2}(T_1, X_1, Y_1, T_2, X_2, Y_2), M_B(Y_1, Y_2, Y_3).$$

6.

$$\mathit{Prec}(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X_1, Y'_1, T_3, X_3, Y_3) \ \leftarrow$$
$$\mathit{Match}_{\varphi_j}(T_1, X_1, Y_1, T_2, X_2, Y_2), \mathit{Match}_{\varphi_k}(T_1, X_1, Y'_1, T_3, X_3, Y_3),$$
$$Y_1 \preceq Y'_1,\ Y_1 \neq Y'_1. \quad (\text{with } 1 \ \leq \ j,\ k \ \leq \ 2)$$

7.

$$\mathit{Prec}(T_1, X_1, Y_1, T_2, X_2, Y_2, T_1, X_1, Y_1, T_3, X_3, Y_3) \ \leftarrow$$
$$\mathit{Match}_{\varphi_j}(T_1, X_1, Y_1, T_2, X_2, Y_2), \mathit{Match}_{\varphi_k}(T_1, X_1, Y_1, T_3, X_3, Y_3),$$
$$M_B(Y_1, Y_3, Y_4), Y_1 \neq Y_4. \quad (\text{with } 1 \ \leq \ j,\ k \ \leq \ 2)$$

9.

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

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

$M_1 = \{..., R^c(t_1, a_1, b_{12}), R^c(t_2, a_2, b_{12}), R^c(t_3, a_3, b_3)\},$
$M_2 = \{..., R^c(t_1, a_1, b_{123}), R^c(t_2, a_2, b_{123}), R^c(t_3, a_3, b_{23})\}.$

We show some of other missing atoms in M_2 to describe how the second chase is captured by the program (for simplicity facts and atoms obtained by the symmetric rules are not shown):

$M_2 = \{Match_{\varphi_2}(t_2, a_2, b_2, t_3, a_3, b_3), R(t_2, a_2, b_{23}), R(t_3, a_3, b_{23}), OldVersion(t_2, a_2, b_2), OldVersion(t_3, a_3, b_3), NotMatch_{\varphi_1}(t_1, a_1, b_1, t_2, a_2, b_2), Match_{\varphi_1}(t_1, a_1, b_1, t_2\ , a_2, b_{23}), R(t_1, a_1, b_{123}), R(t_2, a_2, b_{123}), OldVersion(t_1, a_1, b_1), OldVersion(t_2, a_2, b_{23}), NotMatch_{\varphi_1}(t_1, a_1, b_{123}, t_2, a_2, b_{23}), R^c(t_1, a_1, b_{123}), R^c(t_2, a_2, b_{123}), R^c(t_3, a_3, b_{23}), Prec(\langle t_2, a_2, b_2\rangle, \langle t_3, a_3, b_3\rangle | \langle t_2, a_2, b_{23}\rangle, \langle t_1, a_1, b_1\rangle),\ NotMatch_{\varphi_1}(t_1, a_1, b_{123},\ t_2, a_2, b_2), NotMatch_{\varphi_2}(t_2, a_2, b_{123}, t_3, a_3, b_{23})\}$.

M_2 contains $Prec(\langle t_2, a_2, b_2\rangle, \langle t_3, a_3, b_3, \rangle | \langle t_2, a_2, b_{23}\rangle, \langle t_1, a_1, b_1\rangle)$, showing that two matchings are applicable to different versions of tuple t_2, and the matching applied to the smaller version of the tuple t_2 w.r.t. $\preceq$, i.e. $Match_{\varphi_2}(t_2, a_2, b_2, t_3, a_3,\ b_3)$, has to precede the other, i.e. $Match_{\varphi_1}(t_1, a_1, b_1,\ t_2, a_2, b_{23})$, (cf. rule **6.** above). By the first rule in **2.** above we have $NotMatch_{\varphi_1}(t_1, a_1, b_1, t_2, a_2, b_2)$ meaning that the matching $Match_{\varphi_1}(t_1, a_1, b_1, t_2, a_2, b_2)$ does not take place. Notice that the stable model M_2 does not make the body of the program constraint in 3. true since tuple $R(t_2, a_2, b_2)$ is used for another matching, and replaced by a newer version.

From the stable models M_1, M_2 we can read off the two clean instances D_1, D_2' for D_0 that were obtained from the chase. The stable models of the program can be computed using the *DLV* system [Leone et al., 2006]. The *DLV* code for this example can be found in Appendix A. □

Theorem 5.1.1 There is a one-to-one correspondence between $\mathcal{C}(D_0, \Sigma)$ and the set $SM(\Pi(D_0, \Sigma))$ of stable models of the cleaning program $\Pi(D_0, \Sigma)$. More precisely, the clean instances for D_0 w.r.t. Σ are exactly the restrictions of the elements of $SM(\Pi(D_0, \Sigma))$ to schema $\mathcal{R}^c$.

Proof: The proof of the theorem consists of two parts. For the first part, we need to show that for every (D_0, Σ)-clean instance D' (obtained through a chase as in Definition 2.2.2), we can construct a set of atoms $S^{D'}$ that is a stable model for the logic program $\Pi(D_0, \Sigma)$, and the atoms in $S^{D'}$ are obtained from the valid chase sequence starting from D_0 and leading to D_k, with $D_k = D'$, by enforcing MDs in Σ.

For D' a (D_0, Σ)-clean instance, there are instances $D_1, \ldots, D_{k-1}, D_k = D'$ such that, for every $j \in [1, k]$, $(D_{j-1}, D_j)_{[t_1^j, t_2^j]} \models \varphi$, for some $\varphi \in \Sigma$ and tuple identifiers

t_1^j, t_2^j. We construct $S^{D'}$, a set of atoms over relations of the logic program $\Pi(D_0, \Sigma)$, as follows.

- For every instance D_j, $j \in [0, k]$ and every tuple identifier t of a relation R of relational schema, S_{D_k} contains an atom $R(t, \bar{a})$, where $t^{D_j} = \bar{a}$. $S^{D'}$ also contains facts for the MFs as tables and similarity relations.

- For every tuple identifier t of relation R, $S^{D'}$ contains an atom $R^c(t, \bar{a})$, where $t^{D'} = \bar{a}$ for clean instance D'.

- For every instance D_j, $j \in [0, k-1]$ and every tuple identifier t of relation R such that $t^{D_j} \neq t^{D_k}$, $S^{D'}$ contains an atom $\mathit{OldVersion}(t, \bar{a})$, where $t^{D_j} = \bar{a}$.

- For every $j \in [1, k]$, identifiers t_1^j, t_2^j and MD φ, such that $(D_{j-1}, D_j)_{[t_1^j, t_2^j]} \models \varphi$, for some $\varphi \in \Sigma$, $S^{D'}$ contains an atom $\mathit{Match}_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$, where $t_1^j = \bar{a}_1$ and $t_2^j = \bar{a}_2$ in D_{j-1}. If the two relation names appearing in φ are the same, $S^{D'}$ also contains $\mathit{Match}_\varphi(t_2, \bar{a}_2, t_1, \bar{a}_1)$.

- For every $j, l \in [1, k]$, tuple identifiers t_1^j, t_2^l and MD φ, such that $t_1^j = \bar{a}_1$ in D_j, $t_2^l = \bar{a}_2$ in D_l, t_1^j, t_2^l satisfy the similarities in the left-hand side of φ, but not the equality in the right-hand side, S_{D_k} contains $\mathit{NotMatch}_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$.

- For every $j, l \in [1, k]$, tuple identifiers $t_1^j, t_1^l, t_2^j, t_3^l$ and MDs φ_1, φ_2, such that $(D_{j-1}, D_j)_{[t_1^j, t_2^j]} \models \varphi_1$, $(D_{l-1}, D_l)_{[t_1^l, t_3^l]} \models \varphi_2$, and $j \leq l$, $S^{D'}$ contains an atom $\mathit{prec}(t_1, \bar{a}_1, t_2, \bar{a}_2, t_1, \bar{a}'_1, t_3, \bar{a}_3)$, where $t_1^j = \bar{a}_1$, $t_2^j = \bar{a}_2$ in D_{j-1}, and $t_1^l = \bar{a}'_1$, $t_3^l = \bar{a}_3$ in D_{l-1}.

Next, we show that $S^{D'}$ is a stable model for the program $\Pi(D_0, \Sigma)$. For this, we need to show that $S^{D'}$ is a minimal model of $gr(\Pi(D_0, \Sigma))^{S^{D'}}$, the ground version of program $\Pi(D_0, \Sigma)$ that is obtained by transforming $gr(\Pi(D_0, \Sigma))$ into a positive program according to atoms in $S^{D'}$ (cf. Section 2.4). For obtaining $gr(\Pi(D_0, \Sigma))^{S^{D'}}$, from $gr(\Pi(D_0, \Sigma))$:

- Delete every rule **3.** that has a subgoal *not* $\mathit{OldVersion}(t_1, \bar{a}_1)$ in the body, with $\mathit{OldVersion}(t_1, \bar{a}_1) \in S^{D'}$,

- Delete every program constraint in **8.** that has a subgoal *not* $prec(t_1, \bar{a}_1, t_2, \bar{a}_2, t_1, \bar{a}'_1, t_3, \bar{a}_3)$ in the body, with $prec(t_1, \bar{a}_1, t_2, \bar{a}_2, t_1, \bar{a}'_1, t_3, \bar{a}_3) \in S^{D'}$, and
- Delete the negative subgoal *not* $OldVersion(t_1, \bar{a}_1)$ from the remaining rules.

We are left with a program $gr(\Pi(D_0, \Sigma))^{S^{D'}}$ with the following rules and constraints in their generic forms (all of them are ground):

1'. $R(t_1, \bar{c}_1).\ \ R(t_2, \bar{c}_2).\ \ \ldots$

2'.

$$Match_{\varphi_j}(t_1, \bar{a}_1, b_1, t_2, \bar{a}_2, b_2) \ \vee \ NotMatch_{\varphi_j}(t_1, \bar{a}_1, b_1, t_2, \bar{a}_2, b_2) \ \ \leftarrow$$
$$R(t_1, \bar{a}_1, b_1),\ R(t_2, \bar{a}_2, b_2), \bar{a}_1 \approx \bar{a}_2,\ b_1 \neq b_2.$$

3'. The following ground rule whenever $S^{D'}$ does not contain atoms $OldVersion(t_1, \bar{c}_1)$, $OldVersion(t_2, \bar{c}_2)$:

$$\leftarrow \quad NotMatch_{\varphi_j}(t_1, \bar{c}_1, T_2, \bar{c}_2).$$

Notice that if $S^{D'}$ contains $OldVersion(t_1, \bar{c}_1)$ or $OldVersion(t_2, \bar{c}_2)$, then $gr(\Pi(D_0, \Sigma))^{S^{D'}}$ does not have any ground version of rule **3.**.

4'.

$$OldVersion(t_1, \bar{c}_1) \ \ \leftarrow \ \ R(t_1, \bar{c}_1),\ R(t_1, \bar{c}'_1), \bar{c}_1 \preceq \bar{c}'_1,\ \bar{c}_1 \neq \bar{c}'_1.$$

5'.

$$R(t_1, \bar{a}_1, b_3) \ \ \leftarrow \ Match_{\varphi_j}(t_1, \bar{a}_1, b_1, t_2, \bar{a}_2, b_2),\ M_j(b_1, b_2, b_3).$$

6'.

$$Prec(t_1, \bar{c}_1, t_2, \bar{c}_2, t_1, \bar{c}'_1, t_3, \bar{c}_3) \ \ \leftarrow$$
$$Match_{\varphi_j}(t_1, \bar{c}_1, t_2, \bar{c}_2), Match_{\varphi_k}(t_1, \bar{c}'_1, t_3, \bar{c}_3), \bar{c}_1 \preceq \bar{c}'_1,\ \bar{c}_1 \neq \bar{c}'_1.$$

7'.

$$Prec(t_1, \bar{a}_1, b_1, t_2, \bar{a}_2, b_2, t_1, \bar{a}_1, b_1, t_3, \bar{a}_3, b_3) \leftarrow$$
$$Match_{\varphi_j}(t_1, \bar{a}_1, b_1, t_2, \bar{a}_2, b_2), Match_{\varphi_k}(t_1, \bar{a}_1, Y\beta_1, t_3, \bar{a}_3, b_3), M_A(b_1, b_3, b_4),$$
$$b_1 \neq b_4.$$

8'. The last program constraint in **8.** is the only one with a negative atom in the body, namely atom *not* $Prec(T_1, \bar{Z}_1, T_2, \bar{Z}_2, T_1, \bar{Z}_1'', T_4, \bar{Z}_4)$, so we obtain:

$$Prec(t_1, \bar{c}_1, t_2, \bar{c}_2, t_1, \bar{c}_1, t_2, \bar{c}_2) \leftarrow Match_{\varphi_j}(t_1, \bar{c}_1, t_2, \bar{c}_2).$$

$$\leftarrow Prec(t_1, \bar{c}_1, t_2, \bar{c}_2, t_1, \bar{c}_1', t_3, \bar{c}_3), Prec(t_1, \bar{c}_1', t_3, \bar{c}_3, t_1, \bar{c}_1, t_2, \bar{c}_2),$$
$$(t_1, \bar{c}_1, t_2, \bar{c}_2) \neq (t_1, \bar{c}_1', t_3, \bar{c}_3). \quad (5.1)$$

$$\leftarrow Prec(t_1, \bar{c}_1, t_2, \bar{c}_2, t_1, \bar{c}_1', t_3, \bar{c}_3), Prec(t_1, \bar{c}_1', t_3, \bar{c}_3, t_1, \bar{c}_1'', t_4, \bar{c}_4). \quad (5.2)$$

In (5.1), for interpreting the inequality of sequences, i.e., $(t_1, \bar{c}_1, t_2, \bar{c}_2) \neq (t_1, \bar{c}_1', t_3, \bar{c}_3)$, it unfolds into several constraints, each with one inequality between two values. Program $gr(\Pi(D_0, \Sigma))^{S^{D'}}$ has the constraint (5.2) whenever $S^{D'}$ does not contain the atom $Prec(t_1, \bar{c}_1, t_2, \bar{c}_2, t_1, \bar{c}_1'', t_4, \bar{c}_4)$.

The program constraint (5.2) has to do with making *Prec* a transitive relation, and is obtained when $S^{D'}$ does not contain $Prec(t_1, \bar{c}_1, t_2, \bar{c}_2, t_1, \bar{c}_1'', t_4, \bar{c}_4)$. If $S^{D'}$ contains this atom, the program constraint is discarded. Whereas, the program constraint (5.1) is related to making *Prec* antisymmetric, and it is always in $gr(\Pi(D_0, \Sigma))^{S^{D'}}$ because there is no negative atoms in the body. Therefore, it may happen that the program constraint (5.2) does not appear in $gr(\Pi(D_0, \Sigma))^{S^{D'}}$, but (5.1) does.

9'. The following ground rule whenever $S^{D'}$ does not contain $OldVersion(t_1, \bar{c}_1)$.

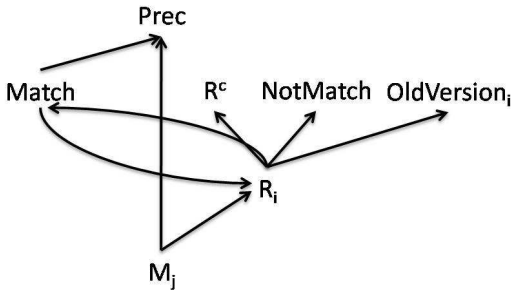
$$R^c(t_1, \bar{c}_1) \ \leftarrow \ R(t_1, \bar{c}_1).$$

As expected, the residual program $gr(\Pi(D_0, \Sigma))^{S^{D'}}$ depends on $S^{D'}$, the candidate to be a stable model for $\Pi(D_0, \Sigma)$. We used the atoms in $S^{D'}$ to simplify $gr(\Pi(D_0, \Sigma))$ by partially evaluating all rules with negative literals against $S^{D'}$. Atoms in $S^{D'}$ specify which ground rules of the forms **3.**, **8.**, **9.** should be in $gr(\Pi(D_0, \Sigma))^{S^{D'}}$ and with what ground atoms in the body. For example, if in a different candidate S', to be a stable model, there is atom $OldVersion(t_1, \bar{c}_1)$, then the ground rule $R^c(t_1, \bar{c}_1) \leftarrow \ R(t_1, \bar{c}_1), \ not \ OldVersion(t_1, \bar{c}_1)$ would not appear in the residual program $gr(\Pi(D_0, \Sigma))^{S'}$.

Program $gr(\Pi(D_0, \Sigma))^{S^{D'}}$ is positive disjunctive. Thus, it may have more than one minimal model [Eiter et al., 1997]. The minimal models of a positive disjunctive program $gr(\Pi(D_0, \Sigma))^{S^{D'}}$ can be obtained as the fix-points of a bottom-up evaluation of the program (cf. Section 2.4). For the bottom-up evaluation of $gr(\Pi(D_0, \Sigma))^{S^{D'}}$, we need to construct the predicate dependency graph for $gr(\Pi(D_0, \Sigma))^{S^{D'}}$.

The predicate dependency graph of a disjunctive logic program Π is the directed graph $DG(\Pi) = (V, E)$. The set V of nodes is the set of all predicates that occur in Π, and there is an edge from q to p, i.e. $(q, p) \in E \subseteq V \times V$, if and only if the predicate q occurs in the body of a rule with p as one of its head predicates [Ben-Eliyahu and Dechter, 1994].

We build the following dependency graph for $gr(\Pi(D_0, \Sigma))^{S^{D'}}$.



In this graph, due to rule **2'.**, there is an edge from predicate R to predicates *Match* and *NotMatch*. Notice that if the two relation names appearing in φ are the

same, then the dependency graph should have a self-loop on node *Match*.

In general, for fix-points evaluation of a positive disjunctive program Π, we first put all the extensional database (EDB) into an initial set M. Then, starting from the facts, we traverse the dependency graph upwards, propagating the facts through the rules, from right to left, iteratively. This procedure, starting from the initial set M starts producing and updating a set of sets MM due to the application of disjunctive rules that make true only one disjunct at a time. In other words, when we apply a disjunctive rule the number of sets satisfying the ground rules may start growing, in addition to the change of each single set in MM. The fix-points are reached when no new sets in MM are created and no new atoms are obtained in each single set in MM. At the end, we discard those sets in MM that violate a program constraint. Then, MM is the set of minimal models of Π.

Next, we show that we can reconstruct $S^{D'}$ as one of the sets in MM, say M', following the one path of the fix-point construction. We take advantage of our construction of $S^{D'}$ starting from the chase sequence $D_0, D_1, \ldots, D_k = D'$ with stable instance D'.

- First, we put all the atoms in **1'.** in a single set M.

- There are edges from predicates R, M_j to predicates $Match, NotMatch$ in the above dependency graph. Therefore, M should satisfy rules of the form **2'.**. For satisfying these rules, for every tuple identifiers t_1, t_2 and MD φ, such that $(D_0, D_1)_{[t_1,t_2]} \models \varphi$ with $t_1^{D_0} = \bar{a}_1$ and $t_2^{D_0} = \bar{a}_2$, we put $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$ in M. For every tuple identifiers t_3, t_4 and MD φ, such that $t_3^{D_0} = \bar{a}_3$, $t_4^{D_0} = \bar{a}_4$, $t_3^{D_0}, t_4^{D_0}$ satisfy the similarities in the left-hand side of φ, but not the equality in the right-hand side, M contains $NotMatch_\varphi(t_3, \bar{a}_3, t_4, \bar{a}_4)$.

- We traverse the dependency graph upwards. There is an edge from predicate *Match* to predicate R. Accordingly, M should satisfy rules of the form **5'.** in $gr(\Pi(D_0, \Sigma))^{S^{D'}}$. For satisfying these rules, M contains an atom $R(t, \bar{a})$, where $t^{D_1} = \bar{a}$.

- Next, we repeat steps 2. and 3., with D_0 and D_1 becoming D_i and D_{i+1}, respectively, with $(D_i, D_{i+1})_{[t_1^i, t_2^i]} \models \varphi$, for some $\varphi \in \Sigma$, and atoms are inserted into

$M' \in MM$, until no new atoms are obtained in M', the set we are constructing.

Actually, we are starting from M and creating an $M' \in MM$ that is being updated through an iterative process. We also put $NotMatch_{\varphi}(t_1, \bar{a}_1, t_2, \bar{a}_2)$ in M' for every $j, l \in [1, k]$, tuple identifiers t_1^j, t_2^l and MD φ, such that $t_1^j = \bar{a}_1$ in D_j, $t_2^l = \bar{a}_2$ in D^l, t_1^j, t_2^l satisfy the similarities in the left-hand side of φ, but not the equality in the right-hand side.

- Since there is an edge from predicate R to predicate *OldVersion*, M' should satisfy rules of the form **4'.**. Accordingly, for every instance D_j, $j \in [0, k-1]$ and every tuple identifier t of a relation R such that $t^{D_j} \neq t^{D_k}$, M' contains an atom $OldVersion(t, \bar{a})$, where $t^{D_j} = \bar{a}$.

- There are edges from predicates $Match, M_j$ to predicate *Prec*. Therefore, M' should satisfy rules of the forms **6'.**, **7'.**, and the first rule in **8'.** in $gr(\Pi(D_0, \Sigma))^{S^{D'}}$. Then, for every $j, l \in [1, k]$, tuple identifiers $t_1^j, t_1^l, t_2^j, t_3^l$ and MDs φ_1, φ_2, such that $(D_{j-1}, D_j)_{[t_1^j, t_2^j]} \models \varphi_1$, $(D_{l-1}, D_l)_{[t_1^l, t_3^l]} \models \varphi_2$, and $j \leq l$, M' contains an atom $prec(t_1, \bar{a}_1, t_2, \bar{a}_2, t_1, \bar{a}'_1, t_3, \bar{a}_3)$, where $t_1^j = \bar{a}_1$, $t_2^j = \bar{a}_2$ in D_{j-1}, and $t_1^l = \bar{a}'_1$, $t_3^l = \bar{a}_3$ in D_{l-1}.

- Finally, there is an edge from predicate R to predicate R^c. Accordingly, M' should satisfy rules of the form **9'.**. Then, for every tuple identifier t of relation R, M' contains an atom $R^c(t, \bar{a})$, where $t^{D_k} = \bar{a}$.

It is clear from this construction that $M' = S^{D'}$. Therefore, $S^{D'}$ is a minimal model for $gr(\Pi(D_0, \Sigma))^{S^{D'}}$, and then, a stable model for $\Pi(D_0, \Sigma)$.

For the second part of the theorem, we need to show that, for every stable model S of the program $\Pi(D_0, \Sigma)$, we can construct a (D_0, Σ)-clean instance D^S.

Let S be a stable model for the logic program $\Pi(D_0, \Sigma)$. For every relation R and every tuple identifier t of relation R such that $R^c(t, \bar{a}) \in S$, we let $t^{D^S} = \bar{a}$. To show that D^S is a (D_0, Σ)-clean instance, we need to construct instances $D_1, \ldots, D_k = D^S$, such that, for every $j \in [1, k]$, $(D_{j-1}, D_j)_{[t_1^j, t_2^j]} \models \varphi$, for some $\varphi \in \Sigma$ and tuple identifiers t_1^j, t_2^j. We use the following lemma for the proof.

Lemma 5.1.1 For every stable model S of the program $\Pi(D_0, \Sigma)$, the relation *prec* is a partial order on the set of sequences of constants, $\mathcal{M}^S = \{(t_1, \bar{a}_1, t_2, \bar{a}_2) \mid Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2) \in S, \text{ for some } \varphi \in \Sigma\}$.

Proof of Lemma 5.1.1: For the proof, we need to show that *Prec* is a reflexive, antisymmetric and transitive relation on the set of sequences of constants $\mathcal{M}^S$.

For proving the first property, we need to show that for all sequence of constants $(t_1, \bar{a}_1, t_2, \bar{a}_2) \in \mathcal{M}^S$, we have $Prec(\langle t_1, \bar{a}_1, t_2, \bar{a}_2\rangle|\langle t_1, \bar{a}_1, t_2, \bar{a}_2\rangle)$ in S. Let $(t_1, \bar{a}_1, t_2, \bar{a}_2) \in \mathcal{M}^S$. This means that we have $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2) \in S$. S is a stable model for $\Pi(D_0, \Sigma)$. This means that all the rules in $\Pi(D_0, \Sigma)$ should be satisfied by S. The body of the first rule in **8.** is satisfied by $Match_\varphi(t_1, \bar{a}, t_2, \bar{a}_2)$. Thus, for satisfying the head of that rule and, consequently, the whole rule, we should have $Prec(\langle t_1, \bar{a}_1, t_2, \bar{a}_2\rangle|\langle t_1, \bar{a}_1, t_2, \bar{a}_2\rangle)$ in S. Therefore, *Prec* is a reflexive relation on $\mathcal{M}^S$.

For showing the second property, let $Prec(\langle t_1, \bar{a}_1, t_2, \bar{a}_2\rangle|\langle t_1, \bar{a}'_1, t_3, \bar{a}_3\rangle)$ in S. Suppose *Prec* is symmetric on M^S. This means that we also have $Prec(\langle t_1, \bar{a}'_1, t_3, \bar{a}_3\rangle|\langle t_1, \bar{a}_1, t_2, \bar{a}_2\rangle)$ in S. This contradicts the fact that S is a stable model of $\Pi(D_0, \Sigma)$ because S would not satisfy the first program constraint in **8.** in $\Pi(D_0, \Sigma)$.

For proving that *Prec* is transitive on $\mathcal{M}^S$, we need to show that if $Prec(\langle t_1, \bar{a}_1, t_2, \bar{a}_2\rangle|\langle t_1, \bar{a}'_1, t_3, \bar{a}_3\rangle)$ and $Prec(\langle t_1, \bar{a}'_1, t_3, \bar{a}_3\rangle|\langle t_1, \bar{a}''_1, t_4, \bar{a}_4\rangle) \in S$, then we have $Prec(\langle t_1, \bar{a}_1, t_2, \bar{a}_2\rangle|\langle t_1, \bar{a}''_1, t_4, \bar{a}_4\rangle)$ in S. Let $Prec(\langle t_1, \bar{a}_1, t_2, \bar{a}_2\rangle|\langle t_1, \bar{a}'_1, t_3, \bar{a}_3\rangle)$ and $Prec(\langle t_1, \bar{a}'_1, t_3, \bar{a}_3\rangle|\langle t_1, \bar{a}''_1, t_4, \bar{a}_4\rangle)$ be in S. Atoms $Prec(\langle t_1, \bar{a}_1, t_2, \bar{a}_2\rangle|\langle t_1, \bar{a}'_1, t_3, \bar{a}_3\rangle)$ and $Prec(\langle t_1, \bar{a}'_1, t_3, \bar{a}_3\rangle|\langle t_1, \bar{a}''_1, t_4, \bar{a}_4\rangle)$ exist in S only if we have $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$, $Match_\varphi(t_1, \bar{a}'_1, t_3, \bar{a}_3)$, $Match_\varphi(t_1, \bar{a}''_1, t_4, \bar{a}_4)$ in S, and $\bar{a}_1 \preceq \bar{a}'_1$, $\bar{a}'_1 \preceq \bar{a}''_1$ (cf. rule **6.**).

By the associativity of matching functions (cf. Section 2.2), $\bar{a}_1 \preceq \bar{a}''_1$ holds. By applying rule **6.** with atoms $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$, $Match_\varphi(t_1, \bar{a}''_1, t_4, \bar{a}_4)$, and $\bar{a}_1 \preceq \bar{a}''_1$ as inputs for the body, and propagating to the head, we get $Prec(\langle t_1, \bar{a}_1, t_2, \bar{a}_2\rangle|\langle t_1, \bar{a}''_1, t_4, \bar{a}_4\rangle)$. Thus, *Prec* is transitive on $\mathcal{M}^S$.

Hence, the relation *Prec* is a partial order on the set of sequences of constants $\mathcal{M}^S$. □

The sequences of constants in $\mathcal{M}^S$ may have different length, e.g. $(t_1, a_1, b_1, t_2, a_3, b_3)$, $(t_5, a_1, c_1, d_1, t_6, a_2, c_2, d_2) \in \mathcal{M}^S$. This is because various relations of a relational

schema with different numbers of attributes may be involved in the MDs in Σ.

Every partial order can be extended to a linear order (that is compatible with the former). The reason why we need the linear order will be explained in below. So, in the following, $\leq$ is a fixed such linear order.

For every two match atoms $Match_{\varphi_j}(t_1, \bar{a}_1, t_2, \bar{a}_2)$ and $Match_{\varphi_l}(t_3, \bar{a}_3, t_4, \bar{a}_4)$ in S, we have $(t_1, \bar{a}_1, t_2, \bar{a}_2) \leq (t_3, \bar{a}_3, t_4, \bar{a}_4)$ when $prec(t_1, \bar{a}_1, t_2, \bar{a}_2, t_3, \bar{a}_3, t_4, \bar{a}_4)$ holds in S, and equality holds only when the two sequences of constants are identical.

We need to construct instances $D_1, \ldots, D_n = D^S$, for some $n \geq 0$, such that, for every $j \in [1, n]$, $(D_{j-1}, D_j)_{[t_1^j, t_2^j]} \models \varphi$, for some $\varphi \in \Sigma$ and tuple identifiers t_1^j, t_2^j. For this purpose, for every $i \in [1, k]$, $k = |\mathcal{M}^S|$, we construct instance D_i as follows. Let $(t_1, \bar{a}_1, t_2, \bar{a}_2)$ be the ith smallest sequence of constants in the linear order $\leq$. For every tuple identifier t, we let $t^{D_i} = t^{D_{i-1}}$ if $t \neq t_1^i, t_2^i$, and we let t_1^i, t_2^i in D_i be the result of enforcing φ, for some $\varphi \in \Sigma$, on $R(\bar{a}_1), R(\bar{a}_2)$. By the definition of a stable model, S is a minimal model for $\Pi(D_0, \Sigma)$. Due to this, $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$ exists in S only if the similarities in the left-hand side of the MD φ hold for $R(t_1, \bar{a}_1), R(t_2, \bar{a}_2)$, and the equality in the right-hand side does not hold. We thus have $(D_{i-1}, D_i)_{[t_1^i, t_2^i]} \models \varphi$. It remains to show that D_k is a stable instance, and it is actually equal to D^S. For proving them, we need the following lemma.

Lemma 5.1.2 Let R be a relation of a relational schema $\mathcal{R}$, and $k = |\mathcal{M}^S|$.

(a) For every $i \in [0, k]$, if $(D_i, D_{i+1})_{[t_1^i, t_2^i]} \models \varphi$, for some $\varphi \in \Sigma$, where t_1^i and t_2^i denote $R(\bar{a}_1)$ and $R(\bar{a}_2)$ in D_i, resp, and t_1^{i+1} and t_2^{i+1} denote $R(\bar{a}'_1)$ and $R(\bar{a}'_2)$ in D_{i+1}, resp, then $(t_1, \bar{a}_1, t_2, \bar{a}_2)$ is the $i+1$th smallest sequence of constants of $\mathcal{M}^S$ in the linear order $\leq$ and $R(t_1, \bar{a}'_1), R(t_2, \bar{a}'_2) \in S$.

(b) For every $i \in [1, k]$, if $(t_1, \bar{a}_1, t_2, \bar{a}_2)$ is the ith smallest sequence of constants of $\mathcal{M}^S$ in the linear order $\leq$, and $R(t_1, \bar{a}'_1), R(t_2, \bar{a}'_2)$ are the atoms obtained by applying rule **5.** with atom $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$, then $(D_{i-1}, D_i)_{[t_1^i, t_2^i]} \models \varphi$, for some $\varphi \in \Sigma$, where t_1^i and t_2^i denote $R(\bar{a}_1)$ and $R(\bar{a}_2)$ in D_{i-1}, resp, and t_1^{i+1} and t_2^{i+1} denote $R(\bar{a}'_1)$ and $R(\bar{a}'_2)$ in D_i, resp.

Proof of Lemma 5.1.2: The proof of this lemma is by an induction on i.

First, we prove (a). For $i = 0$, if $(D_0, D_1)_{[t_1, t_2]} \models \varphi$, then, for the MD $\varphi\colon R[A] \approx R[A] \rightarrow R[B] \doteq R[B]$, we have $t_1[A] \approx t_2[A]$, $t_1[B] \neq t_2[B]$ in D_0. Stable model S

contains all the facts of the initial instance D_0 (cf. rule **1.**). By rule **2.**, we get atom $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$ in S. This is because the necessary similarities in the RHS of rule **2.** hold between two atoms $R(t_1, \bar{a}_1)$, $R(t_2, \bar{a}_2)$. Thus, $(t_1, \bar{a}_1, t_2, \bar{a}_2)$ is the first smallest sequence of constants in the linear order $\leq$. Let $R(t_1, \bar{a}'_1)$, $R(t_2, \bar{a}'_2)$ be in D_1. Then, by the rule **5.**, $R(t_1, \bar{a}'_1), R(t_2, \bar{a}'_2) \in S$.

Suppose (a) holds for every $j < i$. Let $(D_{i-1}, D_i)_{[t_1^i, t_2^i]} \models \varphi$, where $\varphi : R[A] \approx R[A] \rightarrow R[B] \doteq R[B]$. We then have $t_1^i[A] \approx t_2^i[A]$ in D_{i-1} and $t_1^{i+1}[B] = t_2^{i+1}[B] = m_B(t_1^i[B], t_2^i[B])$ in D_i. Let t_1^i and t_2^i denote $R(\bar{a}_1)$ and $R(\bar{a}_2)$ in D_{i-1}, resp. Moreover, by the induction hypothesis, $R(t_1, \bar{a}_1), R(t_2, \bar{a}_2)$ are in S, and there is a sequence of constants in $\mathcal{M}^S$ that is the ith smallest sequence of constants in the linear order $\leq$. Since S is a stable model of $\Pi(D_0, \Sigma)$, and the necessary similarities in the RHS of rule **2.** hold between two atoms $R(t_1, \bar{a}_1)$, $R(t_2, \bar{a}_2)$, S contains $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$ (by rule **2.**). Therefore, $(t_1, \bar{a}_1, t_2, \bar{a}_2)$ is the $i+1$th smallest sequence of constants of $\mathcal{M}^S$ in the linear order $\leq$. Let t_1^{i+1} and t_2^{i+1} denote $R(\bar{a}'_1)$, $R(\bar{a}'_2)$ in D_i, resp. By the rule **5.**, $R(t_1, \bar{a}'_1), R(t_2, \bar{a}'_2) \in S$.

For proving the second part, for $i = 1$, let $(t_1, \bar{a}_1, t_2, \bar{a}_2)$ be the first smallest sequence of constants of $\mathcal{M}^S$ in the linear order $\leq$. This means that $R(t_1, \bar{a}_1)$, $R(t_2, \bar{a}_2)$ are facts in **1.**. Let $R(t_1, \bar{a}'_1), R(t_2, \bar{a}'_2)$ be the atoms obtained by applying rule **5.** with atom $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$ in the body, where $\varphi : R[A] \approx R[A] \rightarrow R[B] \doteq R[B]$. By the definition of a stable model, S is a minimal model for $\Pi(D_0, \Sigma)$. Due to this, $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$ exists in S only if the similarities in the left-hand side of the MD φ hold for $R(t_1, \bar{a}_1), R(t_2, \bar{a}_2)$, and the equality in the right-hand side does not hold. We thus have $R(t_1, \bar{a}_1)$, $R(t_2, \bar{a}_2)$ in D_0 and $t_1^{D_0} \neq t_1^{D_1}$, $t_2^{D_0} \neq t_2^{D_1}$, $t_1^{D_0}[A] \approx t_2^{D_0}[A]$ and $t_1^{D_1}[B] = t_2^{D_1}[B] = m_B(t_1^{D_0}[B], t_2^{D_0}[B])$. Therefore, we have $(D_0, D_1)_{[t_1, t_2]} \models \varphi$ with $R(t_1, \bar{a}'_1)$, $R(t_2, \bar{a}'_2)$ in D_1.

Assume that (b) holds for every $j < i$. Let $(t_1, \bar{a}_1,\ t_2, \bar{a}_2)$ be the ith smallest sequences of constants in the linear order $\leq$. Since S is a stable model for $\Pi(D_0, \Sigma)$, $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$ exists in S only if the similarities in the left-hand side of the MD φ hold for $R(t_1, \bar{a}_1), R(t_2, \bar{a}_2)$, and the equality in the right-hand side does not hold, with $\varphi : R[A] \approx R[A] \rightarrow R[B] \doteq R[B]$. Let $R(t_1, \bar{a}'_1), R(t_2, \bar{a}'_2)$ be the atoms obtained by applying rule **5.** with atom $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$ in the body. By the induction

hypothesis, we have $R(t_1^i, \bar{a}_1), R(t_2^i, \bar{a}_2)$ in D_{i-1}. Thus, $t_1^i[A] \approx t_2^i[A]$ in D_{i-1}. We then have $(D_{i-1}, D_i)_{[t_1^i, t_2^i]} \models \varphi$. Therefore, we have $t_1^{i+1}[B] = t_2^{i+1}[B] = m_B(t_1^i[B], t_2^i[B])$ in D_i. Thus, $R(t_1^{i+1}, \bar{a}_1')$, $R(t_2^{i+1}, \bar{a}_2')$ in D_i. □

We continue proving Theorem 5.1.1. Let S_R contain all the R atoms from the stable model S, i.e. $S_R = \{R(t, \bar{a}) \mid R(t, \bar{a}) \in S\}$. Let D_R be a set of R atoms constructed from instances $D_0, \ldots, D_k$ above, defined as $D_R = \{R(t, \bar{a}) \mid t$ is an R-tuple, and $t^{D_i} = \bar{a}$ for some $i \in [0, k]\}$. By Lemma 5.1.2, it holds $S_R = D_R$. Since D_k and D^S collect the largest version of each tuple identifier, w.r.t. $\preceq$, from the identical sets of atoms S_R and D_R, the two instances should be equal. Thus, there is a valid chase sequence $D_0, D_1, \ldots, D_k = D^S$. It remains to show that D_k is a stable instance, i.e. $(D_k, D_k) \models \Sigma$. Assume that D_k is not stable. This means that there is at least a pair of tuples t_1, t_2 in D_k such that they do not satisfy an MD $\varphi \in \Sigma$. Since $D^S = D_k$, we should have atom $Match_\varphi(t_1, \bar{a}_1, t_2, \bar{a}_2)$ in S. This contradicts the fact that S is a stable model of the program $\Pi(D_0, \Sigma)$. □

The restriction of the stable models to the relational schema $\mathcal{R}^c$ in Theorem 5.1.1 is due to the fact that they also have extensions for the auxiliary predicates used in the programs, as shown in Example 5.1.1.

5.2 Clean Query Answering

We can use the cleaning program $\Pi(D_0, \Sigma)$ to compute the clean answers to a query $\mathcal{Q}$ posed to D_0. In Section 2.2 the clean answers were defined by taking into account the underlying lattices, as the *glb* of all the sets of answers that can be obtained by separately evaluating the query on the clean instances. This is not the same as the usual certain (or skeptical) answers, i.e. the set-theoretic intersection of all the answers from every clean instance, and therefore it is not equivalent to classical skeptical query answering on the logic program. In this section we provide a mechanism for computing clean answers while still using skeptical query answering from the program.

Given an FO query $\mathcal{Q}(x_1, \ldots, x_n)$, with free variables standing for attributes

$A_1, \ldots, A_n$ of $\mathcal{R}$, and defined by a formula $\varphi(\bar{x})$, (with $\bar{x} = x_1, \ldots, x_n$), a non-disjunctive and stratified query program $\Pi(\mathcal{Q})$ can be obtained from φ, using a standard transformation [Lloyd, 1987]. It contains an answer predicate $Ans_{\mathcal{Q}}(\bar{x})$ to collect the answers to $\mathcal{Q}$, and rules defining it, of the form $Ans_{\mathcal{Q}}(\bar{x}) \leftarrow B(\bar{x}')$, where the Bs are conjunctions of literals (i.e. atoms or negations *not* A thereof). The R-atoms in $\mathcal{Q}$, with $R \in \mathcal{R}$, are replaced in $\Pi(\mathcal{Q})$ by R^c-atoms.

In [Bertossi et al., 2013], it is proved that the *glb* for every finite set of reduced instances exist. This means that for clean query answering we need reduced of answers sets (cf. Section 2.2). We can obtain reduced of answer sets by adding two new rules to $\Pi(\mathcal{Q})$:[1]

$$\begin{aligned} Ans^r_{\mathcal{Q}}(\bar{x}) &\leftarrow Ans_{\mathcal{Q}}(\bar{x}),\ not\ Dominated_{\mathcal{Q}}(\bar{x}). \\ Dominated_{\mathcal{Q}}(\bar{x}) &\leftarrow Ans_{\mathcal{Q}}(\bar{y}),\ \bar{x} \preceq \bar{y},\ \bar{x} \neq \bar{y}. \end{aligned}$$

The stable models S of $\Pi(D_0, \Sigma, \mathcal{Q}) := \Pi(D_0, \Sigma) \cup \Pi(\mathcal{Q})$ are the stable models of $\Pi(D_0, \Sigma)$ expanded with extensions $Ans^r_{\mathcal{Q}}(S)$ for predicate $Ans^r_{\mathcal{Q}}$. Those extensions, as database instances, are already reduced. Assume that $SM(\Pi(D_0, \Sigma, \mathcal{Q})) = \{S_1, \ldots, S_m\}$. By definition, we have $Clean^{D_0}_{\Sigma}(\mathcal{Q}) = glb_{\sqsubseteq}\{Ans^r_{\mathcal{Q}}(S_i) \mid i = 1, \ldots, m\}$. Moreover, from equation (2.2) which is defined in [Bertossi et al., 2013] for computing the *glb* of two instances, we obtain

$$glb_{\sqsubseteq}\{Ans^r_{\mathcal{Q}}(S_i) \mid i = 1, \ldots, m\} = Red_{\sqsubseteq}(\{glb_{\preceq}\{\bar{a}_1, \ldots, \bar{a}_m\} \mid \bar{a}_i \in Ans^r_{\mathcal{Q}}(S_i), i = 1, \ldots, m\}) \tag{5.3}$$

where $Red_{\sqsubseteq}$ produces the reduced version of a set under $\sqsubseteq$, i.e. the dominated elements are discarded from the set. We use (5.3) for computing the *glb* of reduced answer sets (cf. Proposition 5.2.1).

Now we show how the program $\Pi(D_0, \Sigma, \mathcal{Q})$ can be modified, so that the clean answers to query $\mathcal{Q}$ can be obtained by running the program under the skeptical semantics.

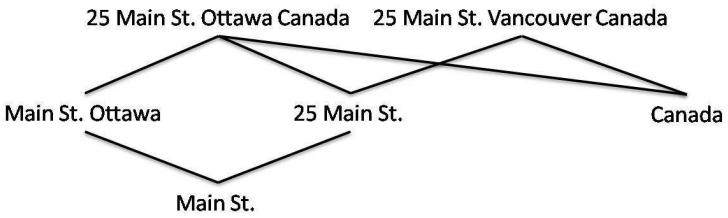
Given $Ans^r_{\mathcal{Q}}(S_i)$, i.e. the set of answers to $\mathcal{Q}$ from the clean instance corresponding to the stable model S_i, we define its *downward expansion* by:

[1]Notice that $\preceq$ in the second rule is defined in terms of the relations M_A.

$$Ans^{exp}_{\mathcal{Q}}(S_i) := \{\bar{b} \mid \bar{b} \preceq \bar{a}, \text{ for some } \bar{a} \in Ans^{r}_{\mathcal{Q}}(S_i)\}.$$

$Ans^{exp}_{\mathcal{Q}}(S_i)$ contains all the answers in $Ans^{r}_{\mathcal{Q}}(S_i)$ and everything below w.r.t. the $\preceq$ lattice. Since $Ans^{r}_{\mathcal{Q}}(S_i)$ is finite, $Ans^{exp}_{\mathcal{Q}}(S_i)$ is also finite, because we consider finite lattices.

Example 5.2.1 (ex. 3.1.2 cont.) Assume that S' and S'' are stables models of the program corresponding to D' and D''. For the query $\mathcal{Q}:\ \pi_{addr}(\sigma_{name=\text{"J. Doe"}}(R))$, it holds: $Ans^{r}_{\mathcal{Q}}(S') = \{25\ \text{ Main st. Ottawa Canada}\}$ and $Ans^{r}_{\mathcal{Q}}(S'') = \{25\ \text{ Main st. Vancouver Canada}\}$.



According to the above lattice, the downward expansions are as follows: $Ans^{exp}_{\mathcal{Q}}(S') = \{25\ \text{ Main st. Ottawa Canada}, \text{Main st. Ottawa}, 25\ \text{ Main st.}, \text{Main st.}, \text{Canada}\}$ and $Ans^{exp}_{\mathcal{Q}}(S'') = \{25\ \text{ Main st. Vancouver Canada}, 25\ \text{ Main st.}, \text{Main st.}, \text{Canada}\}$.

The set-theoretically intersection of extensions of predicates $Ans^{exp}_{\mathcal{Q}}(S')$, $Ans^{exp}_{\mathcal{Q}}(S'')$ is $Ans^{exp}_{\mathcal{Q}}(S') \cap Ans^{exp}_{\mathcal{Q}}(S'') = \{25\ \text{ Main st.}, \text{Main st.}, \text{Canada}\}$. As can be seen, the intersection has a dominated element, i.e. Main st. $\prec$ 25 Main st.. If the dominated element is deleted from $Ans^{exp}_{\mathcal{Q}}(S') \cap Ans^{exp}_{\mathcal{Q}}(S'')$, we get $Red_{\sqsubseteq}(\{Ans^{exp}_{\mathcal{Q}}(S') \cap Ans^{exp}_{\mathcal{Q}}(S')\}) = \{25\ \text{ Main st.}, \text{Canada}\}$. In this way, we reobtain $Clean^{D_0}_{\Sigma}(\mathcal{Q}) = \{25\ \text{ Main st.}, \text{Canada}\}$. □

Example 5.2.1 is a motivation for the following proposition.

Proposition 5.2.1 Let D_0 be an instance, Σ be a set of MDs, and $\mathcal{Q}$ be a query. Let $SM(\Pi(D_0, \Sigma, \mathcal{Q})) = \{S_1, \ldots, S_m\}$, then $Clean^{D_0}_{\Sigma}(\mathcal{Q}) = Red_{\sqsubseteq}(\bigcap \{Ans^{exp}_{\mathcal{Q}}(S_i) \mid i = 1, \ldots, m\})$.

Proof: For proving $Clean^{D_0}_{\Sigma}(\mathcal{Q}) = Red_{\sqsubseteq}(\bigcap \{Ans^{exp}_{\mathcal{Q}}(S_i) \mid i = 1, \ldots, m\})$, we need to

show that (a) $Clean_{\Sigma}^{D_0}(\mathcal{Q}) \subseteq Red_{\sqsubseteq}(\bigcap \{Ans_{\mathcal{Q}}^{exp}(S_i) \mid i = 1, \ldots, m\})$, and (b) $Red_{\sqsubseteq}(\bigcap \{Ans_{\mathcal{Q}}^{exp}(S_i) \mid i = 1, \ldots, m\}) \subseteq Clean_{\Sigma}^{D_0}(\mathcal{Q})$.

First, we prove (a). Let $\bar{a} \in Clean_{\Sigma}^{D_0}(\mathcal{Q})$. By (5.3), for all $\bar{a}' \in Ans_{\mathcal{Q}}^{r}(S_i)$, with $1 < i < m$, $\bar{a} \preceq \bar{a}'$ holds, and $\bar{a}$ is non-dominated. By definition of $Ans_{\mathcal{Q}}^{exp}(S_i)$, it thus follows that, for each S_i, $\bar{a} \in Ans_{\mathcal{Q}}^{exp}(S_i)$. Therefore, $\bar{a} \in \bigcap\{Ans_{\mathcal{Q}}^{exp}(S_i) \mid i = 1, \ldots, m\}$. Since $\bar{a}$ is non-dominated, $\bar{a} \in Red_{\sqsubseteq}(\bigcap \{Ans_{\mathcal{Q}}^{exp}(S_i) \mid i = 1, \ldots, m\})$. It thus follows that $Clean_{\Sigma}^{D_0}(\mathcal{Q}) \subseteq Red_{\sqsubseteq}(\bigcap \{Ans_{\mathcal{Q}}^{exp}(S_i) \mid i = 1, \ldots, m\})$.

For proving (b), let $\bar{a} \in Red_{\sqsubseteq}(\bigcap \{Ans_{\mathcal{Q}}^{exp}(S_i) \mid i = 1, \ldots, m\})$. It thus follows that $\bar{a}$ is non-dominated, and for each S_i, $\bar{a} \in Ans_{\mathcal{Q}}^{exp}(S_i)$. By definition of $Ans_{\mathcal{Q}}^{exp}(S_i)$, for each S_i, $\bar{a} \in Ans_{\mathcal{Q}}^{r}(S_i)$ holds. By (5.3), it thus follows that $\bar{a} \in Clean_{\Sigma}^{D_0}(\mathcal{Q})$. Hence, $Red_{\sqsubseteq}(\bigcap \{Ans_{\mathcal{Q}}^{exp}(S_i) \mid i = 1, \ldots, m\}) \subseteq Clean_{\Sigma}^{D_0}(\mathcal{Q})$.

We obtain $Clean_{\Sigma}^{D_0}(\mathcal{Q}) \subseteq Red_{\sqsubseteq}(\bigcap \{Ans_{\mathcal{Q}}^{exp}(S_i) \mid i = 1, \ldots, m\})$ and $Red_{\sqsubseteq}(\bigcap \{Ans_{\mathcal{Q}}^{exp}(S_i) \mid i = 1, \ldots, m\}) \subseteq Clean_{\Sigma}^{D_0}(\mathcal{Q})$. Thus, the two sets should be identical. □

As a consequence of this result, the clean answers can be obtained by taking the (set-theoretic) intersection of all sets $Ans_{\mathcal{Q}}^{exp}(S_i)$ (followed by a final reduction) instead of taking the *glb* over all sets $Ans_{\mathcal{Q}}^{r}(S_i)$. This can be achieved directly through $\Pi(D_0, \Sigma, \mathcal{Q})$ by adding to it the following rule:

$$Ans_{\mathcal{Q}}^{exp}(\bar{y}) \leftarrow Ans_{\mathcal{Q}}^{r}(\bar{x}),\ \bar{y} \preceq \bar{x},\ Dom_{\mathcal{L}}(\bar{y}). \tag{5.4}$$

Here, $Dom_{\mathcal{L}}(\cdot)$ is stands for the cartesian product of the finite domains Dom_A for the local lattices L_A.

The new rule will expand each stable model by adding finitely many $Ans_{\mathcal{Q}}^{exp}(\bar{b})$ atoms for every $Ans_{\mathcal{Q}}^{r}(\bar{a})$ atom, where $\bar{b} \preceq \bar{a}$. The values for $\bar{y}$ are taken from $Dom_{\mathcal{L}}$. Then each stable model will contain the atoms in the *glb* of all stable models, restricted to the $Ans_{\mathcal{Q}}^{exp}$ predicate, and therefore the intersection of all stable models followed by a final reduction will contain the *glb*. The final reduction is not captured by the program, and happens outside. This observation will help understand the next section. In this way we can obtain the clean answers to query $\mathcal{Q}$.

Example 5.2.2 (ex. 5.1.1 cont.) Consider the query $\mathcal{Q}:\ \pi_B(R)$. For the clean instances D_1, D_2' in Example 5.1.1 it holds: $\mathcal{Q}(D_1) = \{b_{12}, b_3\}$ and $\mathcal{Q}(D_2') = \{b_{123}, b_{23}\}$. We obtain $Clean_{\Sigma}^{D}(\mathcal{Q}) = glb_{\sqsubseteq}\{\mathcal{Q}(D_1), \mathcal{Q}(D_2')\} = glb_{\sqsubseteq}\{\{b_{12}, b_3\}, \{b_{123}, b_{23}\}\} = \{b_{12}, b_3\}$.

We will use the downward expansion approach. The rule defining $\mathcal{Q}$ is $Ans_{\mathcal{Q}}(y) \leftarrow R^c(x, y)$. From $SM(\Pi(D_0, \Sigma, \mathcal{Q})) = \{M_1', M_2'\}$, where stable models M_1' and M_2' contain atoms in M_1 and M_2, respectively, plus atoms involved in query answering, we have:

$Ans_{\mathcal{Q}}(M_1') = \{b_{12}, b_3\},\ \ Ans_{\mathcal{Q}}^{r}(M_1') = \{b_{12}, b_3\},\ \ Ans_{\mathcal{Q}}^{exp}(M_1') = \{b_1, b_2, b_3, b_{12}\},$

$Ans_{\mathcal{Q}}(M_2') = \{b_{123}, b_{23}\}\ \ Ans_{\mathcal{Q}}^{r}(M_2') = \{b_{123}\},\ \ Ans_{\mathcal{Q}}^{exp}(M_2') = \{b_1, b_2, b_3, b_{12}, b_{23}, b_{123}\}.$

Consequently, $Red_{\sqsubseteq}(Ans_{\mathcal{Q}}^{exp}(M_1') \bigcap Ans_{\mathcal{Q}}^{exp}(M_2')) = \{b_{12}, b_3\}$. □

5.2.1 Manifold programs and query answering

We have just described a way to compute, by means of the *downward expanded* programs, the clean answers to a query $\mathcal{Q}$. In this way we avoid a separate and off-line gathering of query answers from each of the stable models for later combination via the *glb*. The *manifold programs* (MF programs) [Faber and Woltran, 2011] offer another alternative for using a single ASP for the whole task. Here we will just sketch the way they can be used in this direction, actually in combination with an extension of ASP with sets and unions thereof [Calimeri et al., 2009]. More details on this extension will be given in Section 5.4.1.

Given a program Π, an MF program for Π, say $MF(\Pi)$, extends Π by collecting brave or skeptical atomic consequences from what would have been Π -now a part of $MF(\Pi)$- and using them for further processing by $MF(\Pi)$.

In our case, properly marked brave consequences from $\Pi(D_0, \Sigma, \mathcal{Q})$ of the form $Ans_{\mathcal{Q}}(\bar{a})^S$, with $S \in SM(\Pi(D_0, \Sigma, \mathcal{Q}))$, can be further used by $MF(\Pi(D_0, \Sigma, \mathcal{Q}))$ to compute the *glb*s. For this, $MF(\Pi(D_0, \Sigma, \mathcal{Q}))$ includes rules of the form (we give a high-level description of them):

$$
\begin{aligned}
glb_{\preceq}(\bar{x}, U) &\leftarrow U = \#Union(\{\bar{y}\}, U'), glb_{\preceq}(\bar{u}, U'),\ \bar{x} = glb^{t}_{\preceq}(\bar{u}, \bar{y}).\\
glb_{\preceq}(\bar{x}, \{\bar{x}\}) &\leftarrow Dom(\bar{x}).\\
PAns_{\mathcal{Q}}(\bar{x}) &\leftarrow glb_{\preceq}(\bar{x}, \{\bar{x}_1, \ldots, \bar{x}_m\}), Ans_{\mathcal{Q}}(\bar{x}_1)^{S_1}, \ldots, Ans_{\mathcal{Q}}(\bar{x}_m)^{S_m}.\\
Dominated^{p}_{\mathcal{Q}}(\bar{x}) &\leftarrow PAns_{\mathcal{Q}}(\bar{y}),\ \bar{x} \preceq \bar{y},\ \bar{x} \neq \bar{y}.\\
CAns(\bar{x}) &\leftarrow PAns_{\mathcal{Q}}(\bar{x}),\ not\ Dominated^{p}_{\mathcal{Q}}(\bar{x}).
\end{aligned}
$$

Here, $glb_{\preceq}(\bar{x}, U)$ is a binary predicate that says that tuple $\bar{x}$ is the $glb_{\preceq}$ of set U; and is defined by recursion and associativity: $glb_{\preceq}(\{\bar{y}\} \cup U') = glb^{t}_{\preceq}(\bar{y}, glb_{\preceq}(U'))$. $glb^{t}_{\preceq}(\bar{u}, \bar{y})$ is a function that produces the $glb_{\preceq}$ of two tuples. The first two rules use the extension of ASP with sets and operations with them (as in Section 5.4.1). They recursively compute the *glb* of a set. The domain predicate, *Dom*, is associated to the cartesian product of the finite attribute domains involved.

The third rule computes the *pre-answers* by combination into the *glb* $\bar{x}$ of brave answers obtained from the $Ans_{\mathcal{Q}}(\bar{x}_i)^{S_i}$. The next rule computes the dominated answers. The last one computes clean answers by discarding pre-answers that are dominated by other pre-answers. Notice that the "manifold part" of the program above is used to form a set of values for a higher-level aggregation. The sets and the aggregation do no appear in the properly manifold part.

5.3 Analysis of Cleaning Programs

In this section we investigate the properties of the cleaning programs in terms of their syntactic structure, and by doing so, shedding some light of their expressive power and computational complexity. At the same time, this analysis will provide upper-bounds for natural computational problems in relation to entity resolution via MDs.

Proposition 5.3.1 The cleaning programs $\Pi(D, \Sigma)$ belong to the class $Datalog^{\vee, not, s}$.

Proof: Since rules in **2.** of the cleaning programs $\Pi(D, \Sigma)$ contain disjunction and negation, the programs belong to the class $Datalog^{\vee, not}$ (cf. Section 2.3). It remains to show that the programs are stratified. The cleaning programs $\Pi(D, \Sigma)$

are stratified due to the following partition $\Pi(D, \Sigma) = P_1 \bigcup P_2 \bigcup P_3 \bigcup P_4$ of rules for which the conditions in [Eiter and Gottlob, 1995] hold.

- P_1 has facts in **1.**, rules in **2.**, **6.**, **7.**, and the first rule in **8.**,
- P_2 contains the rules in **5.**,
- P_3 has the rules in **4.**, and finally
- P_4 consists of the rules in **9.**, **3.**, and the remaining rules in **8.**. □

As a consequence of this result, the stable models of the programs introduced in Section 5.1 can be obtained by means of a bottom-up computation that defines the clean instances.

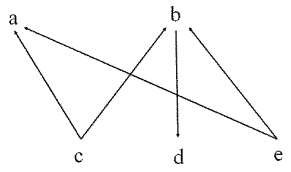
The data complexity of skeptical query evaluation for programs in $Datalog^{\vee, not, s}$ is the same as for programs with unstratified negation, i.e. for the class $Datalog^{\vee, not}$, i.e. Π_2^P-complete [Eiter and Gottlob, 1995, Dantsin et al., 2001, Gelfond and Leone, 2002].

Repair programs for CQA under ICs, also belong to the class $Datalog^{\vee, not, s}$ [Caniupan and Bertossi, 2010]; and their relatively high expressive power is really needed to specify database repairs, because the intrinsic data complexity of CQA is provably Π_2^P-complete (cf. [Bertossi, 2011] for a survey of complexity results in CQA). In the case of cleaning programs two natural questions arise. First, whether they provide an expressive power that exceeds the one needed for clean query answering. Secondly, whether we can obtain an informative upper bound on the complexity of clean query answering. Observe that we have— in principle so far— Π_2^P-complete program and a *co-NP*-complete problem [Bertossi et al., 2013], so the question is whether there is a gap.

These questions are closely related to the properties of the cleaning programs as determined by their syntactic structure. Actually, it turns out that their syntactic structure can be simplified. More precisely, a cleaning program can be transformed into one that that is non-disjunctive. To undertake this task, we need some terminology.

Let $\Pi \in Datalog^{\vee,not}$, and $gr(\Pi)$ be its ground version. The *dependency graph*, $DG(gr(\Pi))$, is a directed graph whose nodes are literals of $gr(\Pi)$. There is an arc from L_1 to L_2 iff there is a rule in $gr(\Pi)$ where L_1 appears positive in the body and L_2 appears in the head. Π is *head-cycle free* (HCF) iff $DG(gr(\Pi))$ has no cycle through two literals that belong to the head of a same rule [Ben-Eliyahu and Dechter, 1994].

Example 5.3.1 Consider the ground program $\Pi = \{a \vee b \leftarrow c, \quad d \leftarrow b, \quad a \vee b \leftarrow e, not\ f\}$.



Its dependency graph is shown in the figure besides. Π is HCF, because there is no cycle involving both a and b, the atoms that appear in the disjunctive head. □

HCF programs in $Datalog^{\vee,not}$ can be transformed into equivalent non-disjunctive programs, i.e. with the same stable models [Ben-Eliyahu and Dechter, 1994, Dantsin et al., 2001]. That is, they can be written as programs in $Datalog^{not}$. This actually holds for our cleaning programs.

Proposition 5.3.2 Every cleaning program $\Pi(D_0, \Sigma)$ is HCF, and hence can be transformed into an equivalent non-disjunctive program in $Datalog^{not}$.

Proof: Let us suppose that $\Pi(D, \Sigma)$ is not HCF. Then the program $\Pi(D, \Sigma)$ has a directed cycle in its dependency graph that goes through $Match_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2)$ and $NotMatch_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2)$ (the atoms that appear in the only disjunctive head), but $NotMatch_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2)$ cannot be involved in a directed cycle since there is no rule in program $\Pi(D, \Sigma)$ in which $NotMatch_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2)$ appears in the body of a rule having heads, a contradiction. □

The transformation is standard. Each disjunctive rule generates as many non-disjunctive rules as atoms in the head, by keeping one at a time in the head, and

moving the others in negated form to the body. In our case, the disjunctive rule

$$Match_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2) \ \vee \ NotMatch_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2) \ \leftarrow \\ R'_1(T_1, \bar{X}_1, Y_1),\ R'_2(T_2, \bar{X}_2, Y_2), \bar{X}_1 \approx \bar{X}_2,\ Y_1 \neq Y_2.$$

gives rise to two rules:

$$Match_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2) \ \leftarrow R'_1(T_1, \bar{X}_1, Y_1), R'_2(T_2, \bar{X}_2, Y_2), \\ not\ NotMatch_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2), \bar{X}_1 \approx \bar{X}_2,\ Y_1 \neq Y_2.$$

and

$$NotMatch_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2) \ \leftarrow R'_1(T_1, \bar{X}_1, Y_1), R'_2(T_2, \bar{X}_2, Y_2), \\ not\ Match_{\phi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2), \bar{X}_1 \approx \bar{X}_2,\ Y_1 \neq Y_2.$$

In general, for a HCF program, checking if a set of atoms is a stable model can be done in polynomial time [Gelfond and Leone, 2002]. However, checking if a set of atoms is contained in a stable model becomes an *NP*-complete problem [Ben-Eliyahu and Dechter, 1994]. In our case, by Theorem 5.1.1, checking if an instance D' is a clean instance (for D_0 and Σ), amounts to checking if D' is contained in stable model of $\Pi(D_0, \Sigma)$ (remember that the stable models of cleaning programs also contain atoms other than $\mathcal{R}$-atoms, e.g. those representing the "cleaning-history" (chase steps). That cleaning-history seems to be necessary to check if D' is a clean instance (just checking stability, i.e. if $(D', D') \models \Sigma$, is the easy part). In consequence, directly from Proposition 5.3.2 we can only obtain that checking if an instance is a clean instance belongs to *NP*.

The data complexity of skeptical query answering from programs in $Datalog^{not}$ is *co-NP*-complete [Dantsin et al., 2001]. In consequence, the decision problem of skeptical query answering from $\Pi(D_0, \Sigma)$ belongs to the class *co-NP*. From this result and Theorem 5.1.1, we obtain

Proposition 5.3.3 For a set Σ of MDs, and a FO query $\mathcal{Q}(\bar{x})$, deciding if a tuple $\bar{c}$ is a clean answer to $\mathcal{Q}$ from an instance D_0 belongs to the class *co-NP* (in the size of

D_0).[2] □

This result should be contrasted with the *co-NP*-complete data complexity of deciding clean query answers presented in [Bertossi et al., 2013, Theorem 3]. We have reobtained the membership of *co-NP* via cleaning programs, but, more importantly, we can conclude that our cleaning programs are not overkilling the problem of clean query answering, and that we need all the expressive power that they provide.

By the proof of *co-NP*-hardness for clean query answering in [Bertossi et al., 2013], we establish that *cautious query answering*, i.e. truth in all clean instances (as opposed to taking the *glb*), is also *co-NP*-hard. This result, combined with the reduction provided by Theorem 5.1.1, tells us that, among the HCF programs in $Datalog^{\vee,not}$, the cleaning programs are among the hardest.

Proposition 5.3.4 Skeptical query answering from cleaning programs is *co-NP*-complete. □

It is possible to obtain a non-disjunctive, stratified cleaning program when matching functions are similarity preserving or MDs are interaction-free. In these cases, the cleaning program has a single stable model, computable in polynomial time, which confirms via cleaning programs a similar result in [Bertossi et al., 2013] saying that there is a unique (D_0, Σ)-clean instance D_0 if matching functions are similarity preserving or MDs in Σ are interaction-free. This is done in Section 6.3 where the general programs $\Pi(D_0, \Sigma)$ are specialized to the well-behaved classes of similarity preserving MFs and interaction-free MDs, obtaining residual programs in $Datalog^{not,s}$.

5.4 Declarative Swoosh ER: The Union Case

5.4.1 Special cleaning programs for UC-Swoosh

In this section we use ASPs for the *declarative specification* of UC Swoosh (cf. Section 3.2). In this union case, an attribute, say A, can take as a value a whole, finite

[2] To be precise, we have to use program $\Pi(D_0, \Sigma, \mathcal{Q})$ expanded with rule (5.4), which actually adds to D_0 the extension of $Dom_{\mathcal{L}}$. However, the latter could be left as a fixed parameter.

set of values from an underlying, lower-level attribute, $\underline{A}$. For example, if $\underline{A} = \{a_1, a_2, a_3, \ldots, \}$, A can take as value $\{a_1, a_3\}$.

In this case the ASPs have to be able to represent sets and sets operations, such as set union. For this purpose we use an extension of disjunctive logic programs with stable model semantics that supports function terms and set terms, with built-in functions for their manipulation [Calimeri et al., 2008, Calimeri et al., 2009].

In this extension of ASP, basic terms are constants and variables, and complex terms, such as functional, list and set terms, are inductively defined: for terms $t_1, ..., t_n$: 1. A *functional term* is of the form $f(t_1, ..., t_n)$, where f is a function symbol. 2. A *list term* has any of the forms: (a) $[t_1, ..., t_n]$; (b) $[h|t]$, where h is a term, and t is a list term. 3. A *set term* is of the form $\{t_1, ..., t_n\}$, where the t_i are terms that do not contain any variables. Some functional terms, called *built-in functions*, have predefined meaning. They are prefixed with #, and we use them below, for sets, lists, membership, and union [Calimeri et al., 2009].

Given a database instance D of schema $R(A_i, \ldots, A_n)$, the *swoosh-program* $\Pi^{UCS}(D)$ that follows captures UC Swoosh. It contains the rules **1.-4.** below:

1. For every atom $R(\bar{s}) \in D$, $\Pi^{UCS}(D)$ contains a fact of the form $R(\bar{s})$. For every attribute A of R, that takes finite sets of values from an underlying domain $Dom_{\underline{A}}$, facts of the form $Match_{\underline{A}}(a_1, a_2)$, with $a_1, a_2 \in Dom_{\underline{A}}$ (cf. Example 3.1.2).

2. Two tuples in D match whenever for some attributes $A_i, A_j, A_k, \ldots$ of R, $1 \leq i, j, k \leq n$, there exists a pair of values, one in each of the set values for $A_i, A_j, A_k, \ldots$ that match. Hence, the rule:

$$R(\#Union(\bar{S}^1, \bar{S}^2)) \leftarrow R(\bar{S}^1),\ R(\bar{S}^2), \#Member(At_i, S_i^1), \#Member(At_j, S_j^1),$$
$$\#Member(At_k, S_k^1), \#Member(At'_i, S_i^2), \#Member(At'_j, S_j^2), \#Member(At'_k, S_k^2)$$
$$Match_{\underline{A_i}}(At_i, At'_i), Match_{\underline{A_j}}(At_j, At'_j), Match_{\underline{A_k}}(At_k, At'_k),\ \bar{S}^1 \neq \bar{S}^2.$$

With them we obtain the merge closure of the original instance, which is a set of records obtained by adding merges of matching records until a fixpoint is reached (cf. Section 3.1). Here, in records $R(\bar{S}^1), R(\bar{S}^2)$, the $\bar{S}^1$ and $\bar{S}^2$ are lists of variables corresponding to all attributes in a record. In other words, $\bar{S}^1 = \langle S_1^1, \ldots, S_n^1 \rangle$, a list

of variables, where each S_i^1, $1 \leq i \leq n$, is related to the attribute A_i; similarly for $\bar{S}^2$. $R(\#Union(\bar{S}^1,\ \bar{S}^2))$ is an abbreviation for the componentwise union of two records $R(\bar{S}^1), R(\bar{S}^2)$, namely: $R(\#Union(S_1^1, S_1^2),\ \ldots,\ \#Union(S_n^1, S_n^2))$. The $S_i^1,\ S_j^1,\ S_k^1,\ S_i^2,\ S_j^2,\ S_k^2,\ At_i,\ At_i',\ At_j,\ At_j',\ At_k,\ At_k'$ are variables, whereas in $\underline{A_i},\ \underline{A_j},\ \underline{A_k}$, the attributes are fixed. Notice that these rules both specify the match function based on the elements of the set values for attributes, and also the result of the merge.

Recall that the match function *Match* and the merge function μ in Swoosh are defined at the record level (cf. Section 3.1). This means that we apply union on each pair of attribute values of two records based on similarities of values of some attributes.

3. A rule defining tuple domination, basically via subset relation:

$$Dominated(\bar{S}^1) \ \leftarrow R(\bar{S}^1),\ R(\bar{S}^2),\ \#Union(\bar{S}^1, \bar{S}^2) = \bar{S}^2,\ \bar{S}^1 \neq \bar{S}^2.$$

By the ICAR properties of match and merge functions for the UC Swoosh, dominated tuples in the merge closure of D can be eliminated via merge domination, which is specified by the above rule.

4. A predicate that collects the result of the ER process:

$$Er(\bar{S}) \ \leftarrow \ R(\bar{S}),\ not\ Dominated(\bar{S}).$$

The facts in **1.** correspond to the elements of the initial instance, and the pairs of low-level attributes values that match. The merge closure of the instance is obtained with rules in **2.**. The natural domination partial order on the tuples in the merge closure of D is captured by rule **3.**. Rule **4.** collects those tuples of the merge closure $\bar{D}$ that are not dominated.

Example 5.4.1 (ex. 3.1.2 cont.) The specific rules are:

1.

$$R(\{a_1\},\{b_1\}).\ \ R(\{a_2\},\{b_2\}).\ \ R(\{a_3\},\{b_3\}).\ \ Match_{\underline{A}}(a_1,a_2).\ \ Match_{\underline{A}}(a_2,a_3).$$

2. As mentioned in Example 3.1.2, two tuples $R(S_1^1, S_2^1), R(S_1^2, S_2^2)$ in D match

when the values for attribute A match, which happens when there is a pair of values in the A-sets that match: For values S_1^1, S_1^2 for A, $Match_A(S_1^1, S_1^2)$ holds when there are $v_1 \in S_1^1, v_2 \in S_1^2$ with $Match_{\underline{A}}(v_1, v_2) = true$. Accordingly, we obtain the merge closure of instance D by the following rule:

$$\begin{aligned} R(\#Union(S_1^1, S_1^2), \#Union(S_2^1, S_2^2)) \quad &\leftarrow R(S_1^1, S_2^1),\ R(S_1^2, S_2^2), \\ &\#Member(At_1, S_1^1), \#Member(At_1', S_1^2), \\ &Match_{\underline{A}}(At_1, At_1'),\ (S_1^1, S_2^1) \neq (S_1^2, S_2^2). \end{aligned}$$

In this case we are matching via attribute A. Notice that we are still computing the union of values of attribute B to merge two tuples. This is because in Swoosh the match and the merge function are defined at the record level. If we also used B, we would have a similar, additional rule for it. Assume that two tuples match in D when the values for attribute A and B match. In this case, we need the following rule instead of the above rule:

$$\begin{aligned} R(\#Union(S_1^1, S_1^2), \#Union(S_2^1, S_2^2)) \quad &\leftarrow R(S_1^1, S_2^1), R(S_1^2, S_2^2), \\ &\#Member(At_1, S_1^1), \#Member(At_1', S_1^2), \\ &\#Member(At_2, S_2^1), \#Member(At_2', S_2^2), \\ &Match_{\underline{A}}(At_1,\ At_1'), Match_{\underline{B}}(At_2, At_2'),\ (S_1^1, S_2^1) \neq (S_1^2, S_2^2). \end{aligned}$$

3. For two non-identical tuples $R(S_1^1, S_2^1), R(S_1^2, S_2^2)$ in the merge closure $\bar{D}$, if the union of these tuples is $R(S_1^2, S_2^2)$, then $R(S_1^1, S_2^1)$ is dominated by $R(S_1^2, S_2^2)$. This is captured by the following rule:

$$\begin{aligned} Dominated(S_1^1, S_2^1) \quad \leftarrow\ & R(S_1^1, S_2^1),\ R(S_1^2, S_2^2), \\ & (\#Union(S_1^1, S_1^2), \#Union(S_2^1, S_2^2)) = (S_1^2, S_2^2), (S_1^1, S_2^1) \neq (S_1^2, S_2^2). \end{aligned}$$

4. Tuple $R(S_1^1, S_2^1)$ in the merge closure $\bar{D}$ that is not dominated by another tuple

in $\bar{D}$ is a tuple in Swoosh entity resolution instance $ER^S(D)$.

$$Er(S_1^1, S_2^1) \ \leftarrow R(S_1^1, S_2^1),\ \textit{not Dominated}(S_1^1, S_2^1).$$

This program, containing set terms and operations, can be run with *DLV-Complex* [Calimeri et al., 2009].[3] □

It is easy to verify that the program $\Pi^{UCS}(D)$ is stratified. Then, it has a single minimal model that can be computed bottom-up in polynomial time in the size of D. We establish in the following that this model, restricted to predicate Er, coincides with the ER instance procedurally computed in [Benjelloun et al., 2009], where it was shown that the ICAR properties make the ER computation tractable. In consequence, our declarative approach to UC Swoosh is in line with the results in [Benjelloun et al., 2009].

Theorem 5.4.1 The unique minimal model $M_{UCS}(D)$ of the program $\Pi^{UCS}(D)$ coincides with the unique Swoosh entity resolution instance $ER^S(D)$. More precisely, $\{Er(\bar{s}) \mid Er(\bar{s}) \in M_{UCS}(D)\} = ER^S(D)$. □

For the proof of Theorem 5.4.1, we need the following definition from [Benjelloun et al., 2009].

Definition 5.4.1 [Benjelloun et al., 2009] (a) Given an instance D, a derivation step $D \rightarrow D'$ is a transformation of instance D into instance D' obtained by applying one of the following two operations:

- Merge step: Given two tuples r_1 and r_2 of D s.t. $Match(r_1, r_2) = true$, and $\mu(r_1, r_2) = r_3 \notin D, D' = D \cup \{r_3\}$,
- Purge step: Given two tuples r_1 and r_2 of D such that r_2 dominates r_1, $D' = D - \{r_1\}$.

[3] http://www.mat.unical.it/dlv-complex. Cf. Appendix C for the DLV code.

(b) A derivation sequence $D \xrightarrow{\star} D_n$ is any non-empty sequence of derivation steps $D \rightarrow D_1 \rightarrow \ldots \rightarrow D_n$. A derivation sequence $D \xrightarrow{\star} D_n$ is maximal if there exists no instance D_{n+1} s.t. $D_n \rightarrow D_{n+1}$ is a valid derivation step. □

In [Benjelloun et al., 2009] it is also proved that, given match and merge functions that have the ICAR properties, for any instance D_0, $ER^S(D_0)$ is finite, and any maximal derivation sequence starting from D_0 computes $ER^S(D_0)$. This is used in the proof of Theorem 5.4.1.

Proof of Theorem 5.4.1: The proof of the theorem consists of two parts. In the first part, we show that for the unique $ER^S(D_0)$ instance D_k, we can construct a set of atoms S_{D_k} that is a minimal model for the logic program $\Pi^{UCS}(D_0)$.

Let D_k be an $ER^S(D_0)$ instance. That is, there is a maximal derivation sequence $D_1 \rightarrow D_2, \ldots, D_{k-2} \rightarrow D_{k-1}$ such that, for every $j \in [1, k]$, $D_{j-1} \rightarrow D_j$, is either a valid merge step or purge step. We construct S_{D_k}, a set of atoms over relations of the logic program $\Pi^{UCS}(D_0)$, as follows.

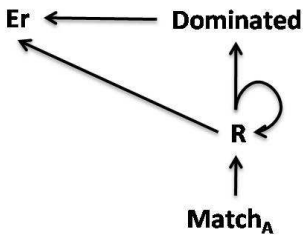
- For every instance D_j, $j \in [0, k]$ and every record $R(\bar{s}) \in D_j$, S_{D_k} contains an atom $R(\bar{s})$. Moreover, for every attribute A of R, that takes finite sets of values from an underlying domain $Dom_{\underline{A}}$, S_{D_k} contains facts of the form $Match_{\underline{A}}(a_1, a_2)$, with $a_1, a_2 \in Dom_{\underline{A}}$ if a_1, a_2 is a pair of values that match.

- For every record $R(\bar{s}) \in D_k$, S_{D_k} contains an atom $Er(\bar{s})$.

- For every instance D_j, $j \in [0, k-1]$ and every record $R(\bar{s}) \in D_j$ such that $R(\bar{s}) \notin D_k$, S_{D_k} contains an atom $Dominated(\bar{s})$.

Next, we show that S_{D_k} is a minimal model for the program $\Pi^{UCS}(D_0)$. The minimal model of a stratified datalog program can be obtained as the fix-point of the bottom-up evaluation of the program (cf. Section 2.4). For the bottom-up evaluation of $\Pi^{UCS}(D_0)$, we need to construct the predicate dependency graph for $\Pi^{UCS}(D_0)$. We

build the following dependency graph for $\Pi^{UCS}(D_0)$.



In general, for fix-points evaluation of a positive datalog program Π, we first put all the extensional database (EDB) into an initial set M. Then, starting from the facts, we traverse the dependency graph upwards, propagating the facts through the rules, from right to left, iteratively. The fix-point is reached when no new atoms are obtained in M. Then, M is a minimal model of Π.

Next, we show that we can reconstruct S_{D_k} as a minimal model M, following the fix-point construction. We take advantage of our construction of S_{D_k} starting from the maximal derivation sequence $D_0 \rightarrow D_1, \ldots, D_{k-1} \rightarrow D_k$, with entity resolution instance D_k.

- First we put all the atoms in **1**. in a single set M.

- There are edges from predicates $R, Match_A$ to predicate R in the above dependency graph. Therefore, M should satisfy rules of the form **2.**. For satisfying these rules, for every records $R(\bar{s}_1), R(\bar{s}_2)$, such that there is a derivation step $D \rightarrow D'$ with $Match(R(\bar{s}_1), R(\bar{s}_2)) = true$, and $\mu(R(\bar{s}_1), R(\bar{s}_2)) = R(\bar{s}_3) \notin D, D' = D \cup \{R(\bar{s}_3)\}$, we put $R(\bar{s}_3)$ in M.

- We traverse the dependency graph upwards. There is an edge from predicate R to predicate *Dominated* in the above graph. Accordingly, M should satisfy rules of the form **3.** in $\Pi^{UCS}(D_0)$. For satisfying these rules, for every records $R(\bar{s}_1), R(\bar{s}_2)$, such that there is a derivation step $D \rightarrow D'$ with $R(\bar{s}_2)$ dominates $R(\bar{s}_1)$, $D' = D - \{R(\bar{s}_1)\}$, M contains an atom $Dominated(\bar{s}_1)$.

- Finally, there are edges from predicates R, *Dominated* to predicate Er in the above graph. Accordingly, M should satisfy rules of the form **4.**. For satisfying

such rules, for every record $R(\bar{s}) \in D_k$, with entity resolution instance D_k, M contains an atom $Er(\bar{s})$.

It is clear from this construction that $M = S_{D_k}$. Therefore, S_{D_k} is a minimal model for $\Pi^{UCS}(D_0)$.

In the second part of the proof, we show that, for a unique minimal model S of the program $\Pi^{UCS}(D_0)$, we can reconstruct the $ER^S(D_0)$ instance D_S.

Let S be a minimal model for the logic program $\Pi^{UCS}(D_0)$. To show that D_S is $ER^S(D_0)$ instance, we need to construct a maximal derivation sequence $D_0 \xrightarrow{\star} D_S = ER^S(D_0)$. For this, we consider two sets $M = \{R(\bar{s}) \mid R(\bar{s}) \in S,\ R(\bar{s}) \notin D_0\}$, $P = \{Dominated(\bar{s}) \mid Dominated(\bar{s}) \in S\}$. We construct a derivation sequence using M, P as follows: starting from $R(\bar{s})$ such that $R(\bar{s}) \in D_0$, we do all necessary merge steps in any order using atoms from M to construct merge closure $\bar{D}_0$, thus we have the sequence $D_0 \rightarrow D_1, \ldots, D_{n-1} \rightarrow D_n$, where $n = |M|$. Next, we perform all purge steps in any order using atoms from P, we then have $\bar{D}_n \rightarrow D_{n+1}, \ldots, D_{m-1} \rightarrow D_{n+m} = D_S$, where $m = |P|$. We thus have the derivation sequence $D_0 \xrightarrow{\star} D_S$.

It remains to show that $D_0 \xrightarrow{\star} D_S$ is a maximal derivation sequence. S is a minimal model. Hence, $R(\bar{s})$ such that $R(\bar{s}) \notin D_0$ exists only if the similarity between two records holds, and the merge of two records does not exist. Similarly, $Dominated(\bar{s}^1)$ exists only if the union of two records $R(\bar{s}^1)$, $R(\bar{s}^2)$ is the record $R(\bar{s}^2)$. Thus, no additional purge steps are possible, and no additional merge steps are possible for extending the derivation sequence $D_0 \xrightarrow{\star} D_S$. Therefore, it is maximal. Hence, $D_S = ER^S(D_0)$ since every maximal derivation sequence starting from the initial instance D_0 results in a unique instance $ER^S(D_0)$. □

5.4.2 UC-Swoosh with negative rules

In [Whang et al., 2009a], the original Swoosh approach to ER is extended with *negative rules*, to impose constraints on the merge results; and interaction with an external expert.

In general, there can be more than one valid ER instance. In [Whang et al., 2009a], it is discussed how a domain expert can *guide* the ER process, to capture a desirable

and valid set of tuples in the ER instance. In other words, with the help of a domain expert, the ER process is started identifying tuples that wanted to be in ER. Actually, the expert looks at the tuples, and selects one that is consistent, not dominated, and more desirable to have in the final instance, e.g. to avoid inconsistencies.

It is possible to extend our ASP-based account of UC Swoosh by considering negative rules and the use of external experts. The latter is achieved via *HEX* programs that extend ASPs with calls to external sources [Eiter et al., 2005].

More specifically, HEX programs are non-monotonic logic programs admitting external atoms [Eiter et al., 2005]. By means of HEX programs, the new logic program for obtaining ER instance in the presence of negative rules delegates the task of identifying non-dominated and consistent tuples that is more desirable to be in the ER instance to an external computational source (e.g. an external deduction system).

The new logic program significantly differs from that for the union case of Swoosh without any negative rules. This is because an expert chooses non-dominated and consistent tuples from merge closure to be located in the ER instance as more desirable ones. Then those tuples from merge closure that are inconsistent w.r.t. tuples in ER would be unavailable to be chosen as tuples in ER. In contrast, in Swoosh without any negative rules, only non-dominated tuples are chosen from the merge closure to be in ER instance, i.e., without any needs to look at which tuples are already in the ER instance, and check consistency.

Example 5.4.2 (ex. 3.1.3 cont.) The following HEX program computes the ER instance in the presence of the given two negative rules: One prohibits for a person in the ER instance to be both M and F, the other one stats an inconsistency in an ER instance if there exist two persons with an identical phone number (functional dependency on the attribute *phone*). These negative rules are outside the program, (for simplicity facts are not shown in the following):

1. A rule to obtain the merge closure of D, which is a set of records obtained by

adding merges of matching records until a fixpoint is reached (cf. Section 3.1):

$$R'(\#Union(S_1^1, S_1^2), \#Union(S_2^1, S_2^2), \#Union(S_3^1, S_3^2)) \leftarrow\ R'(S_1^1, S_2^1, S_3^1),$$
$$R'(S_1^2, S_2^2, S_3^2), \#Member(A_1, S_1^1), \#Member(A_2, S_1^2),\ Match_{\underline{A}}(A_1, A_2),$$
$$(S_1^1, S_2^1, S_3^1) \neq (S_1^2, S_2^2, S_3^2).$$

2. This rule checks if a tuple is inconsistent w.r.t. the negative rule prohibiting for a person to be both M and F:

$$Inconsistent(\bar{S}) \ \leftarrow\ R'(\bar{S}), \#Member(A_1, S_3), \#Member(A_2, S_3)$$
$$A_1 \neq A_2.$$

We also have an inconsistency referring to the functional dependency on the attribute *phone* if there are two tuples with an identical phone number. This rule specifies those tuples that are inconsistent with tuples in ER and makes them unavailable to be chosen as tuples in ER.

$$Inconsistent(\bar{S}^2) \ \leftarrow\ Er(\bar{S}^1),\ R'(\bar{S}^2),\ S_2^1 = S_2^2,\ \bar{S}^1 \neq \bar{S}^2.$$

3. We need a rule to determine tuples of the merge closure that are not dominated, inconsistent and already in ER instance, in order to be selected by an expert to place them in ER instances, as more desirable ones:

$$Select(\bar{S}) \ \leftarrow\ R'(\bar{S}),\ not\ Inconsistent(\bar{S}), not\ Dominated(\bar{S}),$$
$$not\ Er(\bar{S}).$$

4. The following rule computes the predicate *Er* taking values from the predicate *#External*, which chooses via *#External[Select]* more desirable tuples from

tuples in *Select* to be located in *Er*, delegating this task to an external computational source that behaves as a domain expert.

$$Er(\bar{S}) \;\leftarrow\; \#External[Select](\bar{S}).$$

5. We need a rule to find dominated tuples.

$$Dominated(S_1^1, S_2^1, S_3^1) \;\leftarrow\; R'(S_1^1, S_2^1, S_3^1),\; R'(S_1^2, S_2^2, S_3^2),$$
$$(\#Union(S_1^1, S_1^2), \#Union(S_2^1, S_2^2), \#Union(S_3^1, S_3^2)) = (S_1^2, S_2^2, S_3^2),$$
$$(S_1^1, S_2^1, S_3^1) \neq (S_1^2, S_2^2, S_3^2).$$

□

The above logic program is non-disjunctive.

In general, the logic programs developed for UC-Swoosh with negative rules are non-disjunctive.

5.5 Conclusions

In this chapter, we have introduced and developed a declarative approach to ER. It is based on MDs, that can be used to specify details related to ER objectives, such as matchings of attribute values when other values are similar. Our work provides a declarative, model-theoretic specification of the process of *enforcement* of those MDs. The intended clean instances obtained from a dirty instance become the stable models of a specification given by a *cleaning* ASP.

We have provided a declarative specification for a usually procedural process. The prcoedure can be executed for ER and clean query answering using a standard ASP solver. Our focus has been on fundamental questions, e.g. required expressive power, complexity issues, and capturing of clean instances. We have made important steps towards those goals, and created the basis for further improvements, e.g. for clean query answering.

Our ASPs can be automatically generated from MDs, and run on ASP solvers. Indeed, we have used *DLV* and *DLV-Complex*. The *DLV* code for Example 5.1.1 and *DLV-Complex* code for Example 5.4.1 can be found in Appendix A. The programs run and terminate as expected.

We have also proposed a declarative specification of UC Swoosh, and UC Swoosh with negative rules, as important special and practical cases of ER.

As with repair programs for CQA, there is room for many optimizations [Caniupan and Bertossi, 2010, Eiter et al., 2008] (cf. Section 9.2). We will point to important extensions and research directions in Section 9.2, most importantly to applications in virtual data integration systems, where the system can be specified declaratively using ASPs. Indeed, explicitly computing clean instances is not practical, rather computing clean query answers on-the-fly from the ASP is the only realistic option.

Some of the results in this chapter have been published in [Bahmani et al., 2012].

Chapter 6

Relational MDs with Unique Clean Instance

In this chapter, we formally extend the class of matching dependencies introduced in Section 2.2, which we call *classical MDs* from now on, to the larger class of *relational MDs*. This extension is motivated by the application of MDs in *ERBlox* (cf. Chapter 7), but applications can be easily foreseen in other areas where declarative relational knowledge may be useful in combination with matching and merging.

We also identify classes of relational MDs for which a single clean instance exists, no matter how the MDs are enforced, that can be computed through the chase procedure in polynomial time in the size of the database. We say that the MDs (in some cases in combination with an initial instance) have the *unique clean instance property* (UCI property). In this direction, we introduce a class of combinations of relational MDs and initial instances, called *blocking* class. This class includes the sets of MDs used for the blocking component of *ERBlox* (cf. Chapter 7). There are two other classes, namely *MDs with similarity-preserving MFs* and *interaction-free MDs* (cf. Section 2.2), that have similar good properties [Bertossi et al., 2013]. However, these two classes do not depend on the initial data.

In this chapter, we also specialize the cleaning programs $\Pi(D, \Sigma)$ developed in Section 5.1, to obtain specific programs for enforcing sets of MDs in the three classes with the UCI property. We establish that the resulting residual programs belong to a computationally well-behaved extension of plain Datalog. In particular, the single clean instance can be computed with the program in polynomial time in the size of the initial instance.

6.1 Relational MDs

As defined in Section 2.2, *classical MDs* are formulas of the form:

$$\varphi\colon \quad \bigwedge_j R_1[X_1^j] \approx_j R_2[X_2^j] \;\longrightarrow\; \bigwedge_k R_1[Y_1^k] \doteq R_2[Y_2^k], \tag{6.1}$$

where attributes (treated as variables) X_1^j and X_2^j, and Y_1^k, Y_2^k, pairwise share the same data domain. We can consider only MDs with a single identity atom (with $\doteq$) in the RHSs. Accordingly, an explicit formulation of the MD in (6.1) is:

$$\varphi\colon \quad \forall t_1 t_2\, \forall \bar{x}_1 \bar{x}_2 (R_1(t_1, \bar{x}_1) \wedge R_2(t_2, \bar{x}_2) \wedge \bigwedge_j x_1^j \approx_j x_2^j \;\longrightarrow\; y_1 \doteq y_2), \tag{6.2}$$

with $x_1^j, y_1 \in \bar{x}_1,\ x_2^j, y_2 \in \bar{x}_2$. The t_i are used as variables for tuple IDs. As in previous chapters, we usually leave the universal quantifiers implicit. In (6.2), $\approx_j$ is a binary similarity relation on domain Dom_j.

In (6.2), $LHS(\varphi)$ contains, apart from similarity atoms, atoms $R_1(t_1, \bar{x}_1)$ and $R_2(t_2, \bar{x}_2)$, which contain all the variables in the MD, including those in the $RHS(\varphi)$. Identity atoms in the RHS of φ involve one variable from predicate R_1 and one from predicate R_2.

Example 6.1.1 For predicates $Emp(Tid, EID, Dpt, City)$, $Loc(Tid, Dpt, City)$, the following MD is classical:

$$Emp[Dpt] \approx Loc[Dpt] \;\longrightarrow\; Emp[City] \doteq Loc[City]. \tag{6.3}$$

The explicit formulation is:

$$Emp(t_1, eid, dpt_1, city_1) \wedge Loc(t_2, dpt_2, city_2) \wedge dpt_1 \approx dpt_2 \;\longrightarrow\; city_1 \doteq city_2, \tag{6.4}$$

which states that if for an *Emp*-tuple t_1 and an *Loc*-tuple t_2 in an instance D, the attribute value in $t_1^D[Dpt]$ is similar to attribute value in $t_2^D[Dpt]$, then the values $t_1^D[City]$ and $t_2^D[City]$ have to be made identical. The similarity and identity atoms in (6.4) involve one variable from predicate *Emp* and one from predicate *Loc*. □

In MDs, tuples for different relations may be related via attributes in common. The way attribute values in tuples in certain relations are merged, as a result of enforcing an MD, may influence the way attribute values for tuples in other relations are

merged. Furthermore, in an extended form of MDs we could consider additional relational atoms in the LHS, as conditions for merging. For capturing all this, we extend the class of classical MDs to the class of *relational MDs*, where semantic information is used to express relationships between different relations and their corresponding similarity conditions.

Definition 6.1.1 Given a relational schema $\mathcal{R}$, a *relational MD* is a sentence of the form:

$$\varphi\colon \quad \forall t_1 t_2 \bar{t}_3 \; \forall y_1 y_2 \bar{x} \; (R_1(t_1, y_1, \bar{x}_1) \; \wedge \; R_2(t_2, y_2, \bar{x}_2) \; \wedge \; \psi(\bar{t}_3, \bar{z}) \; \longrightarrow \; y_1 \doteq y_2), \quad (6.5)$$

where the following conditions hold:

- $R_1, R_2 \in \mathcal{R}$, the $\bar{x}_i$, etc. are lists of variables, and the y_i are single and distinct variables, the t_i are tid variables, and the $\bar{t}_i$ are lists of tid variables, all of them distinct.

- Variables y_1, y_2 appear each only in one of the different relational atoms (with predicates in $\mathcal{R}$) $R_1(t_1, y_1, \bar{x}_1)$, $R_2(t_2, y_2, \bar{x}_2)$ on the LHS, which are called the *leading atoms*. Variables y_1, y_2 (and its attributes) are called the *leading* variables (attributes).

- $\bar{x} = \bar{x}_1 \cup \bar{x}_2 \cup \bar{z}$, $(\{t_1, t_2\} \cup \bar{t}_3) \cap \bar{x} = \emptyset$ and $\{y_1, y_2\}, \bar{x}_1, \bar{x}_2, \bar{z}$ are not necessarily mutually disjoint.

- Formula $\psi(\bar{t}_3, \bar{z})$ is a conjunction of similarity atoms and relational atoms, where $\bar{z}$ contains y_1, or y_2 only if they appear in a similarity atom in $\psi(\bar{t}_3, \bar{z})$. $\square$

Notice that there are exactly two leading atoms in (6.5), because $\psi(\bar{t}_3, \bar{z})$ does not contain relational atoms with variables y_1 or y_2. The reason for having this restriction is that the relational MDs extend the classical MDs with additional relational atoms in the LHSs, as conditions for merging. In classical MDs, there are exactly two leading atoms. Moreover, if the relational MDs were allowed to have more than two leading atoms, then we would decide if enforcing the MDs would affect all the relations with a leading variable.

It is worth comparing classical MDs in (6.2) with this extended form, in which the arguments in the *relational part* of the MD, namely in $\psi(\bar{t}_3, \bar{z})$, may interact via variables in common (joins) with the arguments in the leading atoms.

Example 6.1.2 With predicates $Author(AID, Name, PTitle, ABlock)$, $Paper(PID, PTitle, Venue, PBlock)$ (with ID and block attributes), this MD, φ, is relational:

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

with 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 to those in corresponding *Paper*-tuples that are in the *same* block (an implicit similarity, actually equality, captured by the join variable bl_4), then blocks bl_1, bl_2 have to be made identical. □

Example 6.1.3 The following formulas

$$\begin{aligned} \varphi_1 \colon \quad & R_1(t_1, x_1, y_1) \ \wedge \ R_2(t_2, x_2, y_2) \ \wedge \ R_3(t_3, y_1, y_2) \ \wedge \ x_1 \approx x_2 \longrightarrow y_1 \doteq y_2, \\ \varphi_2 \colon \quad & R_1(t_1, x_1, y_1, x_2, y_2) \ \wedge \ x_1 \approx x_2 \longrightarrow y_1 \doteq y_2, \end{aligned}$$

are not relational MDs since the leading variables y_1, y_2 appear in more than two relational atoms in LHS of φ_1, and in a single relational atom in the LHS of φ_2. □

The LHS of a relational MD φ may contain more than two relational atoms, but only some variables from relational atoms appear in similarity atoms. We introduce $LSim(\varphi)$ to denote the set of such relational atoms.

Definition 6.1.2 For a relational MD φ, $LSim(\varphi)$ denotes the set of relational atoms appearing in $LHS(\varphi)$, where some of their variables (attributes) appear in similarity or implicit equality atoms in $LHS(\varphi)$. □

$LSim(\varphi)$ might not contain the leading atoms of φ. Only if the leading variables y_1 or y_2 appear in the similarity atoms in $LHS(\varphi)$, $LSim(\varphi)$ contains the leading atoms. $LSim(\varphi)$ might contain only one of the leading atoms.

Example 6.1.4 For schema $Author(Name, Aff, PapTitle, Bl\#)$, $Paper(PTitle, Kwd, Venue, Bl\#)$, the following is a properly relational MD:

$$\varphi\colon\ \underline{Author(t_1, x_1, y_1, p_1, bl_1)}\ \wedge\ \underline{Author(t_2, x_2, y_2, p_2, bl_2)}\ \wedge\ x_1 \approx x_2\ \wedge bl_1 \approx bl_2\ \wedge$$
$$Paper(t_3, p_1', z_1, w_1, bl_4)\ \wedge\ Paper(t_4, p_2', z_2, w_2, bl_4)\ \wedge\ p_1 \approx p_1'\ \wedge\ p_2 \approx p_2'$$
$$\longrightarrow\ bl_1 \doteq bl_2.$$

Here, the leading atoms are underlined. They contain the two variables bl_1, bl_2 that appear in the identity atom on the RHS. Notice that there is an implicit similarity atom (an equality) represented by the use of the shared (join) variable bl_4, and the two leading variables bl_1, bl_2 appear in a similarity atom in the $LHS(\varphi)$.

Due to occurrence of $x_1 \approx x_2$, $bl_1 \approx bl_2$, $p_1 \approx p_1'$, $p_2 \approx p_2'$, and the implicit similarity atom via variable bl_4, $LSim(\varphi) = \{Author(t_1, x_1, y_1, p_1, bl_1),\ Author(t_2, x_2, y_2, p_2, bl_2),\ Paper(t_3, p_1', z_1, w_1, bl_4),\ Paper(t_4,\ p_2', z_2, w_2, bl_4)\}$. In this case, $LSim(\varphi)$ contains the leading atoms, in addition to other relational atoms. □

The chase-based semantics developed for classical MDs can be applied to relational MDs without any relevant change: the new relational conditions in the LHSs of them have to be made true to enforce the MDs. Notice that a relational MD might be enforced on a set of tuples of an instance, as opposed to a classical MD which is only applied on two tuples. Thus, for a set of relational MDs Σ, (D_0, Σ)-clean instance is defined as follows:

Definition 6.1.3 For an instance D_0 and a set of relational MDs Σ, an instance D_k is (D_0, Σ)-*clean* if D_k is stable, and there exists a finite sequence of instances

$D_1, \ldots, D_{k-1}$ such that, for every $i \in [1, k]$, $(D_{i-1}, D_i)_{[t^i]} \models \varphi$, for some $\varphi \in \Sigma$ and a list of tuple identifiers t^i. □

For classical MDs (cf. Section 2.2), two special classes of MDs were identified: similarity-preserving MDs, and interaction-free (IF) MDs. They have the UCI property. Now, relational MDs that are similarity-preserving (i.e. that use similarity-preserving matching functions) are clearly UCI, because only new additional conditions have to be verified before enforcing the MDs.

We proceed now to generalize the interaction-free class to the relational case, and prepare the ground for introducing a new class of relational MDs, the blocking class.

Definition 6.1.4 (a) For a relational MD φ, $ALHS(\varphi)$ denotes the sets of attributes (with their predicates) appearing in similarity atoms in its LHS. [1] $ARHS(\varphi)$ denotes the set of attributes appearing in the identity atom (with $\doteq$) in its RHS.[2] Notice from (6.5) that variables y_1, y_2 in the RHS have implicit predicates, $R_1[Y_1]$, and $R_2[Y_2]$, resp.

(b) A set of relational MDs Σ is *interaction-free* (IF) if, for every $\varphi_1, \varphi_2 \in \Sigma$, $ARHS(\varphi_1) \cap ALHS(\varphi_2) = \emptyset$. Here, φ_1 and φ_2 can be the same. □

$ALHS(\varphi)$ contains the leading attributes only if the leading variables appear in similarity atoms in the $LHS(\varphi)$. However, $ARHS(\varphi)$ always contains the leading attributes. Notice that $ALHS(\varphi)$ is the set of attributes appearing in the atoms in $LSim(\varphi)$. In this way, an attribute in $ARHS(\varphi_1) \cap ALHS(\varphi_2)$ appears, as in 6.1.4(b), in a leading atom of φ_1 and also in an atom in $LSim(\varphi_2)$.

Example 6.1.5 Consider the initial instance D, and the set of relational MDs $\Sigma =$

[1] Attributes appearing in the implicit forms of similarity (equalities), represented by the use of the shared (join) variables, are also considered in $ALHS(\varphi)$.

[2] These are different from $LHS(\varphi)$ and $RHS(\varphi)$, used in Chapter 5 and in this chapter to denote the sets of *atoms* in the LHS and RHS of φ, respectively.

$\{\varphi_1, \varphi_2\}$ with:

$$\varphi_1: \quad Author(t_1, x_1, y_1, p_1, bl_1) \ \wedge \ Author(t_2, x_2, y_2, p_2, bl_2) \ \wedge \ x_1 \approx x_2 \ \wedge$$
$$Paper(t_3, p_1, z_1, w_1, bl_4) \ \wedge \ Paper(t_4, p_2, z_2, w_2, bl_4) \ \longrightarrow \ bl_1 \doteq bl_2.$$
$$\varphi_2: \quad Paper(t_1, p_1, z_1, w_1, bl_1) \ \wedge \ Paper(t_2, p_2, z_2, w_2, bl_2) \ \wedge \ z_1 \approx z_2 \ \wedge$$
$$Author(t_3, x_1, y_1, p_1, bl_3) \ \wedge \ Author(t_4, x_2, y_2, p_2, bl_3) \ \longrightarrow \ bl_1 \doteq bl_2.$$

Author(*D*)	*Name*	*Aff*	*PID*	*Bl#*
t_1	n_1	a_1	120	250
t_2	n_2	a_2	121	251
t_3	n_3	a_3	122	252

Paper(*D*)	*PID*	*Title*	*Key*	*Bl#*
t_4	120	$title_1$	k_1	302
t_5	122	$title_2$	k_2	300
t_6	121	$title_3$	k_3	300

Here, $ALHS(\varphi_2) = \{Paper[Title], \ Author[Bl\#]\}$ and $ARHS(\varphi_1) = \{Author[Bl\#]\}$. Then, $ARHS(\varphi_1) \cap ALHS(\varphi_2) = \{Author[Bl\#]\}$. So, $\{\varphi_1, \varphi_2\}$ is not IF. Here, enforcing φ_1 affects the *Bl#* values in *Author*-tuples, with predicate *Author* not appearing in the leading atoms of φ_2. □

In Example 6.1.4, $ALHS(\varphi) = \{Author[Name], Author[PTitle], Paper[PTitle], Paper[Bl\#]\}$ and $ARHS(\varphi) = \{Author[Bl\#]\}$. Since $ARHS(\varphi) \cap ALHS(\varphi) = \emptyset$, $\Sigma = \{\varphi\}$ is IF.

Notice that we could have more than one attribute in $ARHS(\varphi_1) \cap ALHS(\varphi_2)$, as illustrated in the following example.

Example 6.1.6 Consider the initial instance D_0, and the following set Σ of classical MDs:

$$\varphi_1: \quad R_1(t_1, x_1, y_1, z_1) \ \wedge R_2(t_2, x_2, y_2, z_2) \ \wedge x_1 \approx x_2 \rightarrow y_1 \doteq y_2,$$
$$\varphi_2: \quad R_1(t_1, x_1, y_1, z_1) \ \wedge R_2(t_2, x_2, y_1, z_2) \rightarrow z_1 \doteq z_2.$$

$R_1(D_0)$	A	B	C
t_1	a_1	b_1	c_1
t_2	a_2	b_2	c_2

$R_2(D_0)$	A	B	C
t_3	a_3	b_3	c_3
t_4	a_4	b_4	c_4

Here, $ARHS(\varphi_1) \cap ALHS(\varphi_2) = \{R_1[B], R_2[B]\}$. So, $\{\varphi_1, \varphi_2\}$ is not IF. $LSim(\varphi_2) = \{R_1(t_1, x_1, y_1, z_1),\ R_2(t_2, x_2, y_2, z_2)\}$, and atoms $R_1(t_1, x_1, y_1, z_1)$, $R_2(t_2, x_2, y_2, z_2)$ are leading atoms of φ_1. Then, enforcing φ_1 affects the A-values in R_1, R_2-tuples, with the R_1, R_2-atoms as leading atoms of φ_2. □

So as with similarity-preserving relational MDs, enforcing IF sets of relational MDs on an initial instance results in a single clean instance, which can be computed in polynomial time in the size of the initial instance. Accordingly, IF sets of relational MDs have the UCI property.

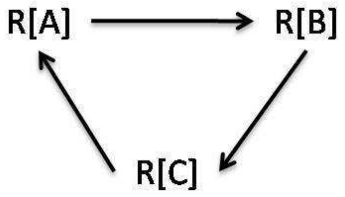
Next, we introduce a notion of *cyclic* (set of) relational MDs.

Definition 6.1.5 (a) For Σ a set of relational MDs, its directed *MD-graph*, $MDG(\Sigma)$, is defined as follows: (1) there is a node for every attribute $R[A]$ with $R[A] \in ALHS(\varphi)$ or $R[A] \in ARHS(\varphi)$, for some $\varphi \in \Sigma$; (2) for every $\varphi \in \Sigma$ and attribute $R[A] \in ALHS(\varphi)$ and attribute $R[B] \in ARHS(\varphi)$, there is an edge from $R[A]$ to $R[B]$.

(b) A set of MDs whose MD-graph contains a cycle is called *cyclic*. Otherwise, it is *acyclic*. □

Example 6.1.7 Consider the set Σ of MDs below. $MDG(\Sigma)$ shows that Σ is cyclic.

$\varphi_1 : R[A] \approx R[A] \rightarrow R[B] \doteq R[B]$,
$\varphi_2 : R[B] \approx R[B] \rightarrow R[C] \doteq R[C]$,
$\varphi_3 : R[C] \approx R[C] \rightarrow R[A] \doteq R[A]$.



□

Notice that we can recognize an interacting set of MDs from its MD-graph: Σ is interacting if there is a node in $MDG(\Sigma)$ with an in-coming and out-going edge. Every set of cyclic MDs is interacting, but not necessarily the other way around.

In next section, we will introduce a class of MDs that includes the sets of MDs used for the blocking component of *ERBlox* (cf. Chapter 7). Interestingly, this class has the UCI property.

6.1.1 Blocking Combinations of Relational MDs and Initial Instances

In this section we define the *blocking* class of sets of relational MDs, motivated by the application of MDs in *ERBlox* (cf. Chapter 7).

Definition 6.1.6 Let Σ be a set of relational MDs and D_0 an instance. (Σ, D_0) is a *blocking combination* (in short, is blocking) if the following conditions hold for every $\varphi_1, \varphi_2 \in \Sigma$ (which could be the same) and attribute $R[A] \in ARHS(\varphi_1) \cap ALHS(\varphi_2)$:

(a) Σ is a set of relational MDs of the form (6.5) and their formulas ψ in the LHSs are conjunctions of similarity atoms and relational atoms, but not $\approx$-similarities for $R[A]$. The implicit form of similarity (an equality) for variables associated to $R[A]$, represented by use of shared (join) variables, is not prohibited.

(b) If $R[A], R'[A']$ share the same domain, and there exists an equality involving variables associated to $R[A], R'[A']$ in $LHS(\varphi_2)$, there are no two tuples $t \in R(D_0), t' \in R'(D_0)$ with $t[A] = t'[A']$.

(c) The matching function MF for attribute $R[A]$ is such that for every pair of values a, a', $MF(a, a')$ takes one of the values a or a'. □

The blocking class requires that there are no two tuples $t \in R(D_0), t' \in R'(D_0)$ with $t[A] \approx t'[A']$ if there is a similarity atom with variables associated to $R[A], R'[A']$ in $LHS(\varphi_2)$. On the other hand, there is no MD in Σ with $\approx$-similarity for $R[A]$ in LHS. Therefore, we have condition (b) in Definition 6.1.6 stating that there are no two tuples in D_0 with $t[A] = t'[A']$. Clearly, if the condition (b) holds, then $R[A]$-attribute values are different from $R'[A']$-attribute values in D_0.

Notice that the condition (b) in Definition 6.1.6 is checked only against the initial instance, and not on later instances obtained along a chase sequence. Non-interacting sets of MDs are trivially blocking for every initial instance D_0.

Notice that a blocking combination (Σ, D_0) may be cyclic, as shown in the following example.

Example 6.1.8 Consider the initial instance D_0 and the set of relational MDs $\Sigma = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$ with:

$$\varphi_1: \quad Author(t_1, x_1, y_1, p_1, bl_1) \ \wedge \ Author(t_2, x_2, y_2, p_2, bl_2) \ \wedge \ x_1 \approx x_2 \ \wedge$$
$$Paper(t_3, p_1, z_1, w_1, bl_4) \ \wedge \ Paper(t_4, p_2, z_2, w_2, bl_4) \ \longrightarrow \ bl_1 \doteq bl_2.$$

$$\varphi_2: \quad Paper(t_1, p_1, z_1, w_1, bl_1) \ \wedge \ Paper(t_2, p_2, z_2, w_2, bl_2) \ \wedge \ z_1 \approx z_2 \ \wedge$$
$$Author(t_3, x_1, y_1, p_1, bl_3) \ \wedge \ Author(t_4, x_2, y_2, p_2, bl_3) \ \longrightarrow \ bl_1 \doteq bl_2.$$

$$\varphi_3: \quad Paper(t_1, p_1, z_1, w_1, bl_1) \wedge Paper(t_2, p_2, z_2, w_2, bl_2) \wedge$$
$$z_1 \approx z_2 \ \wedge \ w_1 \approx w_2 \rightarrow \ bl_1 \doteq bl_2.$$

$$\varphi_4: \quad Author(t_1, x_1, y_1, p_1, bl_1) \ \wedge \ Author(t_2, x_2, y_2, p_2, bl_2) \wedge$$
$$x_1 \approx x_2 \ \wedge \ y_1 \approx y_2 \rightarrow \ bl_1 \doteq bl_2.$$

Author	*Name*	*Aff*	*PID*	*Bl#*
t_1	n_1	a_1	120	250
t_2	n_2	a_2	121	251
t_3	n_3	a_3	122	252

Paper	*PID*	*Title*	*Key*	*Bl#*
t_4	120	$title_1$	k_1	300
t_5	122	$title_2$	k_2	303
t_6	121	$title_3$	k_3	304

Assume that for enforcing MDs on two tuples, the matching function m_{Bl} is used. It makes two block numbers identical: $m_{Bl}(i, j) := i$ if $j \leq i$.

For checking the conditions in Definition 6.1.6 for (Σ, D_0), we find four cases of interaction: (1) $ARHS(\varphi_1) \cap ALHS(\varphi_2) = \{Author[Bl\#]\}$, (2) $ARHS(\varphi_2) \cap ALHS(\varphi_1) = \{Paper[Bl\#]\}$, (3) $ARHS(\varphi_4) \cap ALHS(\varphi_2) = \{Author[Bl\#]\}$, (4) $ARHS(\varphi_3) \cap ALHS(\varphi_1) = \{Paper[Bl\#]\}$.

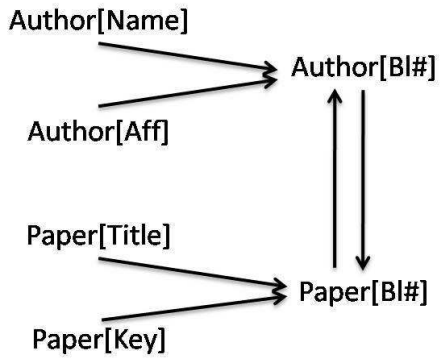
For the second case, $Paper(\bar{x})$ is a leading atom since $Paper[Bl\#]$ is the leading attribute of φ_2. $Paper(\bar{x})$ also belongs to $LSim(\varphi_1)$ because of the use of the join variable bl_4 in $LHS(\varphi_1)$.

(Σ, D_0) satisfies the condition (a) in Definition 6.1.6 since there is no $\approx$-similarity in the LHSs of the MDs in Σ for $Author[Bl\#]$ or $Paper[Bl\#]$. Notice that, in the case of φ_1, there is an implicit equality of blocks through the use of the same block variable in the two *Paper* atoms. For φ_2, there is an implicit similarity atom (an equality) represented by the use of the shared (join) variable bl_3.

There exists an equality involving attributes associated to $Author[Bl\#]$ in $LHS(\varphi_2)$. Then, for cases (1) and (3), we check if there are *Author*-tuples in D_0 with same blocking numbers. This does not hold in D_0. Thus, these two cases do not lead to a violation of the condition (b) in Definition 6.1.6.

There is an equality containing attributes associated to $Paper[Bl\#]$ in $LHS(\varphi_1)$. Therefore, for cases (2) and (4), (Σ, D_0) is not blocking if there are two *Paper*-tuples in D_0 with same blocking numbers. This does not hold in D_0. Then, the condition (b) in Definition 6.1.6 is satisfied by (Σ, D_0).

Moreover, $m_{Bl\#}(i, j) = j$ if $i < j$. Thus, (Σ, D_0) is a blocking combination. Notice that Σ is a cyclic set of MDs by the following $MDG(\Sigma)$.



For a negative example, consider a different initial instance D_1 with the same MDs:

$Author(D_1)$	$Name$	Aff	PID	$Bl\#$
t_1	n_1	a_1	122	250
t_2	n_2	a_2	121	251
t_3	n_3	a_3	122	252
t_4	n_4	a_4	123	253

$Paper(D_1)$	PID	$Title$	Key	$Bl\#$
t_5	120	$title_1$	k_1	302
t_6	122	$title_2$	k_2	300
t_7	121	$title_3$	k_3	300
t_8	123	$title_4$	k_4	303

For checking the condition (b) for cases (2) and (4) of interaction, two *Paper*-tuples t_6, t_7 in D_1 have same blocking numbers. Thus, (Σ, D_1) is not a blocking combination. □

As expected, the notion of blocking combinations can be applied to classical MDs.

Example 6.1.9 Consider predicate $R(A, B, C)$, the instance D_0, and the set of classical MDs Σ below:

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

$R(D_0)$	A	B	C
t_1	a_1	b_1	c_1
t_2	a_2	b_2	c_2
t_3	a_3	b_3	c_3
t_4	a_4	b_4	c_4

Assume that the matching function m_B acts as follows: $m_B(b_1, b_2) = b_2$, $m_B(b_1, b_3) = b_3$, $m_B(b_2, b_3) = b_3$, $m_B(b_3, b_4) = b_4$. Σ is interacting (i.e. not IF), because $ARHS(\varphi_1) \cap ALHS(\varphi_2) = \{R[B]\}$.

(Σ, D_0) satisfies condition (a) in Definition 6.1.6 because there is no $\approx$-similarity in the LHS of φ_1 or φ_2 for attribute $R[B]$.

Since there are no tuples t, t' in D_0 with $t[B] = t'[B]$, (Σ, D_0) satisfies condition (b) in Definition 6.1.6. Condition (c) in Definition 6.1.6 is also satisfied. Then, (Σ, D_0) is blocking. □

In the following example, condition (b) in Definition 6.1.6 is investigated more.

Example 6.1.10 (ex. 6.1.6 cont.) As mentioned in Example 6.1.6, $\{\varphi_1, \varphi_2\}$ is not IF since $ARHS(\varphi_1) \cap ALHS(\varphi_2) = \{R_1[B], R_2[B]\}$.

Suppose that the matching function m_B for identifying values of attributes $R_1[B]$, $R_2[B]$ acts as follows: $m_B(b_1, b_2) = b_2$, $m_B(b_1, b_3) = b_3$, $m_B(b_2, b_3) = b_3$.

(Σ, D_0) satisfies condition (a) in Definition 6.1.6 because there is no $\approx$-similarity in the LHSs of the MDs for $R_1[B]$ or $R_2[B]$.

For checking the condition (b) in Definition 6.1.6, $R_1[B], R_2[B]$ share the same domain, and there is only one equality including variables associated to $R_1[B], R_2[B]$ in $LHS(\varphi_2)$. There are no two tuples $t \in R_1(D_0), t' \in R_2(D_0)$ with $t[B] = t'[B]$. Then, the condition (b) in Definition 6.1.6 is satisfied by (Σ, D_0). Furthermore, the matching function $m_B(a, a')$ takes one of the values. Thus, (Σ, D_0) is a blocking combination.

For another example, consider $\Sigma_1 = \{\varphi_1, \varphi_3\}$ with same initial instance.

$$\varphi_3 : R_1(t_1, x_1, y_1, z_1) \ \wedge R_1(t_2, x_2, y_1, z_2) \rightarrow z_1 \doteq z_2.$$

In this case, $ARHS(\varphi_1) \cap ALHS(\varphi_3) = \{R_1\,[B]\}$. Conditions (a), (c) are checked in the same way as for (Σ, D_0), but not condition (b). In φ_3, there is not an equality for $R_1[B], R_2[B]$. Actually, there exists an equality for only $R_1[B]$ in $LHS(\varphi_3)$. In this case, we check if there are two tuples $t \in R_1(D_0), t' \in R_1(D_0)$ with $t[B] = t'[B]$. This does not hold in D_0. Then, the condition (b) in Definition 6.1.6 is satisfied by (Σ_1, D_0). □

The conditions in Definition 6.1.6 are strong for relational MDs. However, we require the blocking class to capture the sets of MDs used for the blocking component of *ERBlox* (cf. Chapter 7).

Blocking combinations lead to single clean instances. This also holds for the cyclic set of MDs, as illustrated in the following example.

Example 6.1.11 (ex. 6.1.8 cont.) Assume that the only similarities that hold are $a_1 \approx a_2$, $n_1 \approx n_2$, $n_1 \approx n_3$, $title_1 \approx title_3$, $title_1 \approx title_2$ and $k_1 \approx k_2$.

For the blocking combination of Σ and D_0, different orders of MD enforcements lead in the end to the same single clean instance. This behavior can be better appreciated below.

We will show that the enforcement of Σ on D_0 generates a unique clean instance, through different chase sequences. First, we show a possibly chase sequence $D_0, D_1, D_2, D_3, D_4, D_5, D_6$, with D_6 a stable instance.

As a result of enforcing φ_4 on D_0 first, the tuples t_1, t_2 get the identical values for $Author[Bl\#]$, as shown in the new instance D_1 (cf. Figure 6.1).

Author	*Name*	*Aff*	*PID*	*Bl#*
t_1	n_1	a_1	120	**251**
t_2	n_2	a_2	121	**251**
t_3	n_3	a_3	122	252

Paper	*PID*	*Title*	*Key*	*Bl#*
t_4	120	$title_1$	k_1	300
t_5	122	$title_2$	k_2	303
t_6	121	$title_3$	k_3	304

Figure 6.1: Instance D_1

Next, since t_4 and t_5 have similar values for $Paper[Title]$ and $Paper[Key]$, we

can enforce φ_3, leading to t_4, t_5 getting the same value for $Paper[Bl\#]$, as shown in instance D_2 (cf. Figure 6.2).

Author	*Name*	*Aff*	*PID*	*Bl#*
t_1	n_1	a_1	120	251
t_2	n_2	a_2	121	251
t_3	n_3	a_3	122	252

Paper	*PID*	*Title*	*Key*	*Bl#*
t_4	120	$title_1$	k_1	**303**
t_5	122	$title_2$	k_2	**303**
t_6	121	$title_3$	k_3	304

Figure 6.2: Instance D_2

As we can see, through MD enforcement new similarities may be created, in this case $t_1[Bl\#] = t_2[Bl\#]$ in D_1. Furthermore, the equality of values in t_1, t_2 for attribute $Author[Bl\#]$ feeds the LHS of φ_2. Now, enforcing φ_2 on t_4, t_6, t_1, t_2 in D_2 makes the tuples t_4, t_6 get the same value for attribute $Paper[Bl\#]$, as shown in instance D_3 (cf. Figure 6.3).

Author	*Name*	*Aff*	*PID*	*Bl#*
t_1	n_1	a_1	120	251
t_2	n_2	a_2	121	251
t_3	n_3	a_3	122	252

Paper	*PID*	*Title*	*Key*	*Bl#*
t_4	120	$title_1$	k_1	**304**
t_5	122	$title_2$	k_2	303
t_6	121	$title_3$	k_3	**304**

Figure 6.3: Instance D_3

At this stage we have broken the equality of $t_4[Bl\#], t_5[Bl\#]$ we had in D_2, as shown underlined in Figure 6.3. This is a crucial point: φ_3 is still applicable on D_3 with t_4, t_5, because there are no MDs with attribute $Paper[Title]$ or $Paper[Key]$ in their RHSs that could destroy the initial similarities that held in D_0, in particular $t_4^{D_3}[Title] = title_1 \approx title_2 = t_5^{D_3}[Title]$ and $t_4^{D_3}[Key] = k_1 \approx k_2 = t_5^{D_3}[Key]$: φ_3 is applicable with t_4, t_5 along the enforcement path. So, enforcing φ_3 makes $t_4[Bl\#], t_5[Bl\#]$ identical again, as shown in instance D_4 (cf. Figure 6.4).

Notice that the initial similarities of attributes values we had in the initial instance are not destroyed at any time later along a chase sequence.

Next, applying φ_1 on t_1, t_3, t_4, t_5 in D_4 makes the tuples t_1, t_3 get the same value for attribute $Author[Bl\#]$, as shown in instance D_5 (cf. Figure 6.5).

At this stage, we have broken the equality for $t_1[Bl\#], t_2[Bl\#]$ we had in D_4, as

Author	*Name*	*Aff*	*PID*	*Bl#*
t_1	n_1	a_1	120	251
t_2	n_2	a_2	121	251
t_3	n_3	a_3	122	252

Paper	*PID*	*Title*	*Key*	*Bl#*
t_4	120	$title_1$	k_1	**304**
t_5	122	$title_2$	k_2	**304**
t_6	121	$title_3$	k_3	304

Figure 6.4: Instance D_4

Author	*Name*	*Aff*	*PID*	*Bl#*
t_1	n_1	a_1	120	**252**
t_2	n_2	a_2	121	251
t_3	n_3	a_3	122	**252**

Paper	*PID*	*Title*	*Key*	*Bl#*
t_4	120	$title_1$	k_1	304
t_5	122	$title_2$	k_2	304
t_6	121	$title_3$	k_3	304

Figure 6.5: Instance D_5

shown underlined in Figure 6.5. φ_4 is still applicable on D_5 with t_1, t_2. So, enforcing φ_4 makes $t_1[Bl\#]$, $t_2[Bl\#]$ identical again, as shown in instance D_6. No further applications of MDs are possible, and we have reached a stable instance.

Author	*Name*	*Aff*	*PID*	*Bl#*
t_1	n_1	a_1	120	**252**
t_2	n_2	a_2	121	**252**
t_3	n_3	a_3	122	252

Paper	*PID*	*Title*	*Key*	*Bl#*
t_4	120	$title_1$	k_1	303
t_5	122	$title_2$	k_2	303
t_6	121	$title_3$	k_3	303

Figure 6.6: Instance D_6

Actually, D_6 is the only instance that can be reached through any chase sequence. For example, we will now show another chase sequence leading to the same clean instance D_6.

In the above chase sequence, we applied φ_3 on t_4, t_6 in D_1. We could have enforced φ_2 on t_4, t_6, t_1, t_2 in D_1. This makes the tuples t_4, t_6 get the same value for attribute $Paper[Bl\#]$, as shown in instance D_2'' (cf. Figure 6.7).

Next, enforcing φ_3 on t_4, t_5 in D_2'' results in instance D_3'', where t_4, t_5 have identical values for attribute $Paper[Bl\#]$, as shown in Figure 6.8.

Now, applying φ_1 on t_1, t_3, t_4, t_5 in D_3'', makes the tuples t_1, t_3 get the same value for attribute $Author[Bl\#]$, as shown in instance D_4'' (cf. Figure 6.9). Again, we have broken the equality of $t_1[Bl\#], t_2[Bl\#]$ we had in D_3'', as shown underlined in Figure 6.9. MD φ_4 is still applicable on D_4'' with t_1, t_2. Enforcing φ_4 on t_1, t_2 in D_4'' results

Author	*Name*	*Aff*	*PID*	*Bl#*
t_1	n_1	a_1	120	251
t_2	n_2	a_2	121	251
t_3	n_3	a_3	122	252

Paper	*PID*	*Title*	*Key*	*Bl#*
t_4	120	$title_1$	k_1	**304**
t_5	122	$title_2$	k_2	303
t_6	121	$title_3$	k_3	**304**

Figure 6.7: Instance D_2''

Author	*Name*	*Aff*	*PID*	*Bl#*
t_1	n_1	a_1	120	251
t_2	n_2	a_2	121	251
t_3	n_3	a_3	122	252

Paper	*PID*	*Title*	*Key*	*Bl#*
t_4	120	$title_1$	k_1	**304**
t_5	122	$title_2$	k_2	**304**
t_6	121	$title_3$	k_3	304

Figure 6.8: Instance D_3''

in instance D_6 which we had obtained before through a different chase sequence.

Author	*Name*	*Aff*	*PID*	*Bl#*
t_1	n_1	a_1	120	**252**
t_2	n_2	a_2	121	251
t_3	n_3	a_3	122	**252**

Paper	*PID*	*Title*	*Key*	*Bl#*
t_4	120	$title_1$	k_1	304
t_5	122	$title_2$	k_2	304
t_6	121	$title_3$	k_3	304

Figure 6.9: Instance D_4''

The MD φ may become applicable with $\bar{t}$ along a chase sequence, which is not applicable on the initial instance. In a chase sequence, the similarities of attributes values, which do not hold in the initial instance, may be broken. As a result, φ may be not applicable with $\bar{t}$ on an instance in the chase sequence. But φ will be applicable with $\bar{t}$ later on in the same and the other chase sequences. In this example, φ_2 is applicable on D_2 with t_4, t_6, t_1, t_2, but is not applicable on D_5 with t_4, t_6, t_1, t_2. Again, φ_2 is applicable on D_6 with t_4, t_6, t_1, t_2.

In other words, two tuples with similar attribute values, not in the initial instance D_0-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.

Actually, no matter in what order the MDs are enforced in this case, the final, clean instance will be D_6, which is due to a specific property of the blocking combination (Σ, D_0). □

Example 6.1.12 (ex. 6.1.9 cont.) Let's assume that the matching function m_B acts as follows: $m_B(b_1, b_2) = b_3$, $m_B(b_1, b_3) = b_{13}$, $m_B(b_2, b_3) = b_{23}$, $m_B(b_3, b_4) = b_4$. In this case, the condition (b) is not satisfied by (Σ, D_0). Then, (Σ, D_0) is not a blocking. Enforcing Σ on D_0 leads to two clean instances:

$R(D_1)$	A	B	C
t_1	a_1	b_3	c_{123}
t_2	a_2	b_3	c_{123}
t_3	a_3	b_4	c_{1234}
t_4	a_4	b_4	c_{1234}

$R(D_2)$	A	B	C
t_1	a_1	b_3	c_{12}
t_2	a_2	b_3	c_{12}
t_3	a_3	b_4	c_{34}
t_4	a_4	b_4	c_{34}

□

In general, different orders of MD enforcements may result in different clean instances, because similarities between attributes values of tuples may be broken during the chase with interacting MDs and non-similarity-preserving MFs, without reappearing again [Bertossi et al., 2013].

Proposition 6.2.1 will establish that blocking combinations lead to single clean instances. Before addressing this, let's address the blocking class checking problem.

In the general relational case of MDs, one would wonder how difficult is checking the blocking class. Notice that checking the condition (a) or the condition (c) in Definition 6.1.6 is decidable because a finite set of MDs and matching functions have to be checked. In this way, only checking the condition (b) in Definition 6.1.6 is crucial. Notice that only the active domain, $Adom(D_0)$, of the initial instance D_0 matters for the condition (b) in Definition 6.1.6, because attributes values are checked in D_0. Actually, checking the condition (b) is also decidable because a finite set of MDs has to be checked for interaction, and D_0 is finite.

Actually, checking the condition (b) in Definition 6.1.6 can be performed in polynomial time in the size of D_0 (i.e. *in data*) by posing Boolean conjunctive queries (BCQs) to D_0. More precisely, for each pair φ_1, φ_2 in Σ, one has to issue one BCQ for each attribute in the intersection of $ARHS(\varphi_1)$ and $ALHS(\varphi_2)$, and another BCQ for each attribute in the intersection of $ARHS(\varphi_2)$ and $ALHS(\varphi_1)$. If one of those queries gets the value *true* in D_0, the condition (b) in Definition 6.1.6 does not hold.

Example 6.1.13 (ex. 6.1.9 cont.) There is only one case of interaction between φ_1 and φ_2. The condition (b) for D_0 with φ_1, φ_2 for the intersection $ARHS(\varphi_1) \cap ALHS(\varphi_2) = \{R[B]\}$ can be checked by posing the following Boolean conjunctive query to D_0:

$$\exists \bar{t}\ \exists \bar{x}\ \exists \bar{y}\ \exists \bar{z}\ (R(t_1, x_1, y_1, z_1)\ \wedge\ R(t_2, x_2, y_2, z_2))\ \wedge\ y_1 = y_2).$$

In fact, for checking if tuples t, t' in D_0 with $t[B] = t'[B]$ exist, the query contains two relational atoms.

The above query gets the value *false* in D_0. Then, the condition (b) in Definition 6.1.6 holds for (Σ, D_0), which was also verified by other means in Example 6.1.9. □

Lemma 6.1.1 Let D_0 an initial instance, and φ_1, φ_2 be relational MDs with attribute $R[A] \in ARHS(\varphi_1) \cap ALHS(\varphi_2)$. Let $R[A], R'[A']$ share the same domain, and there exists an equality involving variables associated to $R[A], R'[A']$ in $LHS(\varphi_2)$. There is a BCQ, $\mathcal{Q}^{R[A]}_{\varphi_1,\varphi_2}$, that evaluates to *true* in D_0 iff there are two tuples $t \in R(D_0), t' \in R'(D_0)$, such that $t[A] = t'[A']$.

Proof: Assume that $R[A], R'[A']$ share the same domain, and there exists an equality involving variables associated to $R[A], R'[A']$ in $LHS(\varphi_2)$. For $R[A] \in ARHS(\varphi_1) \cap ALHS(\varphi_2)$, checking the condition (b) in Definition 6.1.6 requires checking if there are two tuples $t \in R(D_0), t' \in R'(D_0)$ with $t[A] = t'[A']$. For this reason, the query contains two relational atoms. $\mathcal{Q}^{R[A]}_{\varphi_1,\varphi_2}$ is the following BCQ:

$$\mathcal{Q}^{R[A]}_{\varphi_1,\varphi_2}: \qquad \exists \bar{t}\ \exists \bar{x}\ \exists \bar{y}\ \ (R(t_1, y_1, \bar{x}_1) \wedge R'(t_2, y_2, \bar{x}_2) \wedge z_1 = z_2). \tag{6.6}$$

where $z_i \in x_i \cup \{y_i\}$ with $i \in \{1, 2\}$, and z_1 and z_2 are associated to the attributes $R[A]$ and $R'[A']$, resp.

Next, we prove that the query $\mathcal{Q}^{R[A]}_{\varphi_1,\varphi_2}$ is *true* in D_0 iff there are two tuples $t \in R(D_0), t' \in R'(D_0)$, such that $t[A] = t'[A']$.

($\Rightarrow$): Assume that the query $\mathcal{Q}^{R[A]}_{\varphi_1,\varphi_2}$, as in (6.6), is *true* in D_0. By construction of $\mathcal{Q}^{R[A]}_{\varphi_1,\varphi_2}$, evaluating $\mathcal{Q}^{R[A]}_{\varphi_1,\varphi_2}$ to *true* in D_0 means that there are two tuples $t \in R(D_0), t' \in$

$R'(D_0)$, such that $t[A] = t'[A']$.

($\Leftarrow$): Assume that there are two tuples $t \in R(D_0), t' \in R'(D_0)$, such that $t[A] = t'[A']$. By the construction of $\mathcal{Q}^{R[A]}_{\varphi_1,\varphi_2}$, if we substitute variables in $\mathcal{Q}^{R[A]}_{\varphi_1,\varphi_2}$ by attributes values from tuples $t \in R(D_0), t' \in R'(D_0)$, then $\mathcal{Q}^{R[A]}_{\varphi_1,\varphi_2}$ evaluates to *true* in D_0. □

From the fact that BCQ can be evaluated in polynomial time in data (see, e.g., [Abiteboul et al., 1995]), we obtain the following.

Corollary 1. Checking the condition (b) in Definition 6.1.6 on an instance D_0 can be done in polynomial time in data complexity, i.e. in the size of D_0. □

Let $R[A], R'[A'] \in ARHS(\varphi_1) \cap ALHS(\varphi_2)$. By the construction of the query in the proof of Lemma 6.1.1, for checking the non-satisfaction of the condition (b) for D_0 with φ_1, φ_2, $\mathcal{Q}^{R[A]}_{\varphi_1,\varphi_2}$, and $\mathcal{Q}^{R'[A']}_{\varphi_1,\varphi_2}$ are the same queries. This is because checking the non-satisfaction of the condition (b) for D_0 with φ_1, φ_2 requires checking if there are two tuples $t \in R(D_0), t' \in R'(D_0)$, such that $t[A] = t'[A']$. Checking this gives rise to only one query for both $R[A] \in ARHS(\varphi_1) \cap ALHS(\varphi_2)$ and $R'[A'] \in ARHS(\varphi_1) \cap ALHS(\varphi_2)$.

Example 6.1.14 (ex. 6.1.6 cont.) For the interaction case $ARHS(\varphi_1) \cap ALHS(\varphi_2) = \{R_1[B], R_2[B]\}$, we consider the BCQs $\mathcal{Q}^{R_1[B]}_{\varphi_1,\varphi_2}$ and $\mathcal{Q}^{R_2[B]}_{\varphi_1,\varphi_2}$, respectively:

$$\mathcal{Q}^{R_1[B]}_{\varphi_1,\varphi_2}: \quad \exists \bar{t}\ \exists \bar{x}\ \exists \bar{y} \exists \bar{z}\ (R_1(t_1, x_1, y_1, z_1)\ \wedge R_2(t_2, x_2, y_2, z_2)\ \ \wedge y_1 = y_2,$$

$$\mathcal{Q}^{R_2[B]}_{\varphi_1,\varphi_2}: \quad \exists \bar{t}\ \exists \bar{x}\ \exists \bar{y}\ \exists \bar{z}\ (R_1(t_1, x_1, y_1, z_1)\ \wedge R_2(t_2, x_2, y_2, z_2)\ \ \wedge y_1 = y_2.$$

These queries are the same. Thus, in this example, checking the condition (b) for (Σ, D_0) gives rise to only one query. These queries evaluate to *false* in D_0. Then, (Σ, D_0) is blocking. □

Example 6.1.15 (ex. 6.1.8 cont.) For the first case of interaction between the MDs, $ARHS(\varphi_1) \cap ALHS(\varphi_2) = \{Author[Bl\#]\}$, the following BCQ is posed to D_0:

$$\mathcal{Q}^{Author}_{\varphi_1,\varphi_2}: \ \exists \bar{t}\ \exists \bar{x}\ \exists \bar{y}\ \exists \bar{p}\ \exists \bar{bl}\ \exists \bar{z}\ \exists \bar{w}(Author(t_1, x_1, y_1, p_3, bl_1) \wedge Author(t_2, x_2, y_2, p_4, bl_2) \wedge \ bl_1 = bl_2).$$

$\mathcal{Q}^{Author}_{\varphi_1,\varphi_2}$ takes the value *false* in D_0, then this case (case (1) in Example 6.1.8) does not lead to a violation of the condition (b). Interestingly, $\mathcal{Q}^{Author}_{\varphi_1,\varphi_2}$, $\mathcal{Q}^{Author}_{\varphi_3,\varphi_2}$ are the same queries. This is because checking the non-satisfaction of the condition (b) for D_0 with φ_1, φ_2 and with φ_1, φ_2 require checking if there are two tuples $t \in Author(D_0), t' \in Author(D_0)$, such that $t[Bl\#] = t'[Bl\#]$.

$\mathcal{Q}^{Paper}_{\varphi_2,\varphi_1}$ and $\mathcal{Q}^{Paper}_{\varphi_4,\varphi_1}$ are the same queries since for cases (2),(4) in Example 6.1.8, we check if there are *Paper*-tuples in D_0 with same blocking numbers. For these cases, we consider the following BCQ:

$$\exists \bar{t}\ \exists \bar{x}\ \exists \bar{y}\ \exists \bar{p}\ \exists \bar{bl}\ \exists \bar{z}\ \exists \bar{w}(Paper(t_1, x_1, y_1, p_3, bl_1) \wedge Paper(t_2, x_2, y_2, p_4, bl_2) \wedge \ bl_1 = bl_2).$$

This query also takes the value *false* in D_0. Then, (Σ, D_0) is blocking. □

6.2 The UCI Classes

In this section, we identify and prove a general property, called *Tuple-Similarity Preservation* (TSP), for the blocking class. This property is used to establish that if (Σ, D_0) is blocking, enforcing Σ on D_0 results in a single clean instance. As a consequence, the three classes of relational MDs: IF, with similarity preserving MFs, and blocking class have the UCI property, defined in the beginning of this chapter.

We also prove that the TSP property holds for the two other classes of relational MDs with the UCI property. This property is used in Section 6.3 to specialize the general cleaning programs, developed in Section 5.1, for the three classes of MDs with

the UCI property.

As shown in Example 6.1.11, for a blocking combination (Σ, D_0), the tuple similarities held in D_0 for attributes that appear in $ALHS$ of the MDs are not destroyed during the chase sequence: These similarities keep holding along the enforcement path.

Lemma 6.2.1 Let (Σ, D_0) be a blocking combination. Let $D_0, \ldots, D_k$ be a chase sequence with D_k stable, such that for every $i \in [1, k]$, $(D_{i-1}, D_i)_{[\bar{t}^i]} \models \varphi_i$, for some $\varphi_i \in \Sigma$ and a list of tuple identifiers $\bar{t}^i$. Then, $t^{D_i}[A] = t^{D_0}[A]$ and $t'^{D_i}[A'] = t'^{D_0}[A']$, for every $i \in [0, k]$, every two tuple identifiers t, t', and every MD $\varphi_i \in \Sigma$, where A, A' are two comparable attributes in $ALHS(\varphi_i)$.

Proof: We prove it by an induction on i. For $i = 0$, it clearly holds.

Let $LHS(\varphi)$ be true in D_0 with $\bar{t}$. Now suppose that $LHS(\varphi)$ is true in D_i with $\bar{t}$ for $i < j$, and it does not hold for $i = j$: $LHS(\varphi)$ is true in D_0 with $\bar{t}$, but $LHS(\varphi)$ is false in D_j with $\bar{t}$. Since it holds for every $i < j$, for $\varphi \in \Sigma$, and for the list of tuple identifiers $\bar{t}$, there are some tuple identifier $t' \in \bar{t}$, and some attribute $R_1[X_1] \in ALHS(\varphi)$, such that $t'^{D_j}[X_1] \neq t'^{D_{j-1}}[X_1]$. Therefore, there should be an MD $\varphi_j : R_1(t_1, \bar{y}_1, x_1) \wedge R_2(t_2, \bar{y}_2, x_2) \wedge \psi(\bar{t}_3, \bar{y}_3) \rightarrow x_1 \doteq x_2$ in Σ, with variables x_1, x_2 corresponding to attributes $R_1[X_1], R_2[X_2]$, respectively, and a tuple identifier t'', such that D_j is the immediate result of enforcing φ_j on $\bar{t'''}$ with $t', t'' \in \bar{t'''}$ in D_{j-1}. That is, $LHS(\varphi_j)$ is true in D_{j-1} with $\bar{t'''}$, $t'^{D_{j-1}}[X_1] \neq t''^{D_{j-1}}[X_2]$, and $t'^{D_j}[X_1] = t''^{D_j}[X_2] = m_X(t'^{Dj-1}[X_1], t''^{D_{j-1}}[X_2])$. Since (Σ, D_0) is blocking and $R_1[X_1] \in ARHS(\varphi_j) \cup ALHS(\varphi)$, φ is not applicable in D_0. This means that $LHS(\varphi)$ is false in D_0 with $\bar{t}$, which leads to a contradiction. □

Next, we identify and prove a general property, called *Tuple-Similarity Preservation* (TSP), for the blocking combination class.

The MD φ may become applicable with $\bar{t}$ along a chase sequence, which is not applicable on the initial instance. In a chase sequence, the similarities of attributes values, which do not hold in the initial instance, may be broken, and, as a result, φ may be not applicable on an instance in the chase sequence with $\bar{t}$. Interestingly, when

(Σ, D_0) is blocking, φ will be applicable with $\bar{t}$ later on in the same and the other chase sequences. In this case, we say that (Σ, D_0) has *Tuple-Similarity Preservation* property. As in Example 6.1.11, φ_2 is applicable on D_2 with t_4, t_6, t_1, t_2, but is not applicable on D_5 with t_4, t_6, t_1, t_2. Again, φ_2 is applicable on D_6 with t_4, t_6, t_1, t_2.

Definition 6.2.1 Let Σ be a set of MDs, D_0 an initial instance, and $D_0, D_1, \ldots, D_k$ a chase sequence with D_k stable, and for every $i \in [1, k]$, $(D_{i-1}, D_i)_{[\bar{t}^i]} \models \varphi_i$, for some $\varphi_i \in \Sigma$ and a list of tuple identifiers $\bar{t}^i$. Let D be a clean instance for D_0 and Σ (not necessarily equal to D_k). (Σ, D_0) is *Tuple-Similarity Preservation* (we say that (Σ, D_0) has the TSP property) if $LHS(\varphi)$ is true in D_i with $\bar{t}$, then $LHS(\varphi)$ is true in D with $\bar{t}$, for every $i \in [0, k]$, for every $\varphi \in \Sigma$, and for every list of tuple identifiers $\bar{t}$. □

Lemma 6.2.2 Let (Σ, D_0) be a blocking combination. Then, (Σ, D_0) has TSP property.

Proof: For the proof of the lemma, we need Definition 6.2.2 and Lemma 6.2.3. In the following definition, we make a directed *link-graph* for every chase sequence of D_0 with Σ. Intuitively, the graph shows how each attribute value $t^{D_i}[A] \neq t^{D_0}[A]$, with $i \in [1, k]$, is obtained by a sequence of MD enforcements on tuples of chase instances.

Definition 6.2.2 Let Σ be a set of MDs, D_0 an initial instance, and $D_0, D_1, \ldots, D_k$ a chase sequence with D_k stable, and for every $i \in [1, k]$, $(D_{i-1}, D_i)_{[\bar{t}^i]} \models \varphi_i$, for some $\varphi_i \in \Sigma$ and a list of tuple identifiers $\bar{t}^i$. (a) For Σ and the chase sequence $\bar{S} : D_0, \ldots, D_k$, their directed *link-graph*, $LG(\Sigma, \bar{S})$, is defined as follows:

There is a node for every attribute value $t^{D_i}[A]$:

- If $(D_i, D_{i+1})_{[\bar{t}^{i+1}]} \models \varphi_{i+1}$, $t \in \bar{t}^{i+1}$, $R[A] \in ALHS(\varphi_{i+1})$, or
- If $(D_{i-1}, D_i)_{[\bar{t}^i]} \models \varphi_i$, $t \in \bar{t}^i$, $R[A] \in ARHS(\varphi_i)$.

Edges in $LG(\Sigma, \bar{S})$ are added as follows:

- For every $t^{D_{i-1}}[A]$, $t'^{D_i}[B]$, such that $(D_{i-1}, D_i)_{[\bar{t}^i]} \models \varphi_i$, $t, t' \in \bar{t}^i$, $R[A] \in ALHS(\varphi_i)$, and $R[B] \in ARHS(\varphi_i)$, there is an edge from $t^{D_{i-1}}[A]$ to $t'^{D_i}[B]$.

- For every $t^{D_i}[A]$, $t^{D_j}[A]$ with $i < j$ and $t^{D_i}[A] = t^{D_j}[A]$, there is an edge from $t^{D_i}[A]$ to $t^{D_j}[A]$.

(b) A node $t^{D_i}[A]$ in $LG(\Sigma, \bar{S})$ is called *initial* if $t^{D_i}[A]= t^{D_0}[A]$, and there is no MD φ in Σ with $R[A] \in ARHS(\varphi)$. □

In the following lemma, we prove that there exists a path from some initial node to every node in $LG(\Sigma, \bar{S})$ for blocking (Σ, D_0).

Lemma 6.2.3 Let (Σ, D_0) be a blocking combination. Then, for every node $t^{D_i}[A]$ with $i \in [1, k]$ in $LG(\Sigma, \bar{S})$, there is a path from some initial node to $t^{D_i}[A]$.

Proof of Lemma 6.2.3: We prove it by an induction on i. For $i = 1$, it holds because if $(D_0, D_1)_{[\bar{t}]} \models \varphi$, $t \in \bar{t}$, then there is an edge from some initial node to $t^{D_1}[A]$. If $t \notin \bar{t}$, then $t^{D_0}[A] = t^{D_1}[A]$. Thus, by construction of $LG(\Sigma, \bar{S})$, there is an edge from initial node $t^{D_0}[A]$ to $t^{D_1}[A]$.

Now suppose that for every node $t^{D_i}[A]$ with $i < j$, there is a path from some initial node to $t^{D_i}[A]$, and it does not hold for $i = j$: there exists node $t'^{D_j}[B]$ such that there is no path from an initial node to $t'^{D_j}[B]$. Here, we distinguish two cases for attribute $R[B]$:

(a) If $R[B] \in ARHS(\varphi_j)$ with $\varphi_j \in \Sigma$, then $(D_{j-1}, D_j)_{[\bar{t}^j]} \models \varphi_j$ with $t' \in \bar{t}^j$. By induction hypothesis, there is a path in $LG(\Sigma, \bar{S})$ from some initial node to $t^{D_i}[A]$, such that $i < j$, $t \in t'^{D_j}[B]$, and $R'[A] \in ALHS(\varphi_j)$. By construction of $LG(\Sigma, \bar{S})$, there is an edge from $t^{D_i}[A]$ to $t'^{D_j}[B]$. Then, there is a path from some initial node to $t'^{D_j}[B]$, which leads to a contradiction.

(b) If $R[B] \in ALHS(\varphi_{j+1})$ with $\varphi_{j+1} \in \Sigma$, then $t'^{D_j}[B] = t''^{D_j}[B']$ holds where $(D_j, D_{j+1})_{[\bar{t}^{j+1}]} \models \varphi_{j+1}$, $t', t'' \in \bar{t}^{j+1}$ and $R'[B'] \in ALHS(\varphi_{j+1})$. There is no path from an initial node to $t'^{D_j}[B]$. Then, there is MD φ' in Σ with $R[B]$ or $R'[B']$ in $ARHS(\varphi')$. Thus, $R[B] \in ARHS(\varphi') \cap ALHS(\varphi_{j+1})$ or $R'[B'] \in ARHS(\varphi') \cap ALHS(\varphi_{j+1})$. If both $R[B] \in ARHS(\varphi') \cap ALHS(\varphi_{j+1})$ and $R'[B'] \in ARHS(\varphi') \cap ALHS(\varphi_{j+1})$ hold, then case (a) happens. Thus, there

is a path from some initial node to $t'^{D_j}[B]$, which leads to a contradiction. If only one of $R[B] \in ARHS(\varphi') \cap ALHS(\varphi_{j+1})$ and $R'[B'] \in ARHS(\varphi') \cap ALHS(\varphi_{j+1})$ holds, then there are two tuples $t \in R(D_0), t' \in R'(D_0)$ with $t[B] = t'[B']$ or there is not a matching function for attribute $R[B]$ ($R'[B']$), such that for each pair of values a, a', $m_B(a, a')$ takes one of the values a or a'. These contradict with the fact that (Σ, D_0) is blocking. □

We continue proving Lemma 6.2.2. It is immediately follows from Lemma 6.2.1 and Lemma 6.2.3. □

Proposition 6.2.1 Let (Σ, D_0) be a blocking combination. Then, there is a unique (D_0, Σ)-clean instance D.

Proof: For the proof of the proposition, we need the following lemma.

Lemma 6.2.4 Let (Σ, D_0) be a blocking combination. Also let $D_0, D_1, \ldots, D_k$ be a sequence of instances such that D_k is stable, and for every $i \in [1, k]$, $(D_{i-1}, D_i)_{[\bar{t}^i]} \models \varphi_i$, for some $\varphi_i \in \Sigma$ and a list of tuple identifiers $\bar{t}^i$. Let D be a clean instance for D_0 (Σ not necessarily equal to D_k). Then, for every $i \in [0, k]$, every tuple identifier t and attribute X, $t^{D_i}[X] \preceq t^D[X]$ hold.

Proof of Lemma 6.2.4: We prove it by an induction on i. For $i = 0$, we clearly have $t^{D_0}[X_1] \preceq t^D[X_1]$ since D is a clean instance for D_0 and Σ.

Now suppose $t^{D_i}[X_1] \preceq t^D[X_1]$ holds for $i < j$, and it does not hold for $i = j$: $t^{D_j}[X_1] \not\preceq t^D[X_1]$. Since it holds for every $i < j$, the value of $t^{D_j}[X_1]$ should be different from $t^{D_{j-1}}[X_1]$. Therefore, there should be an MD $\varphi_j : R_1(t_1, \bar{y}_1, x_1) \wedge R_2(t_2, \bar{y}_2, x_2) \wedge \psi(\bar{t}_3, \bar{y}_3) \rightarrow x_1 \doteq x_2$ in Σ, with variable x_1 corresponding to attribute $R_1[X_1]$, and a tuple identifier t', such that D_j is the immediate result of enforcing φ_j on $\bar{t}''$ with $t, t' \in \bar{t}''$ in D_{j-1}. That is, $LHS(\varphi_j)$ is true in D_{j-1} with $\bar{t}''$, $t^{D_{j-1}}[X_1] \neq t'^{D_{j-1}}[X_2]$, and $t^{D_j}[X_1] = t'^{D_j}[X_2] = m_X(t^{Dj-1}[X_1], t'^{D_{j-1}}[X_2])$. Since $LHS(\varphi_j)$ is true in D_{j-1} with $\bar{t}''$, by Lemma 6.2.2, we have $LHS(\varphi_j)$ is true in D with $\bar{t}''$, and thus $t^D[X_1] = t'^D[X_2]$,

because D is a stable instance. Again by induction hypothesis, $t^{D_{j-1}}[X_1] \preceq t^{D}[X_1]$ and $t'^{D_{j-1}}[X_2] \preceq t'^{D}[X_2] = t^{D}[X_1]$. Therefore, $t^{D_j}[X_1] = m_X(t^{D_{j-1}}[X_1], t'^{D_{j-1}}[X_2]) \preceq t^{D}[X_1]$ since m_X takes the least upper bound, which leads to a contradiction. □

We continue proving Proposition 6.2.1. Let D, D' be two clean instances for D_0, Σ. Notice that, from Lemma 6.2.4, we obtain $t^{D}[X] \preceq t^{D'}[X]$ and $t^{D'}[X] \preceq t^{D}[X]$ for every tuple identifier t and every attribute X. Thus, the two clean instances D, D' should be identical. □

By Proposition 6.2.1, if (Σ, D_0) is blocking, then it has the UCI property. The class of IF relational MDs is trivially blocking for every initial instance D_0. Thus, (Σ, D_0), with a set of IF relational MDs Σ, has the TSP property.

Lemma 6.2.5 Let Σ be a set of relational MDs with similarity preserving MFs, and D_0 an initial instance. Then, (Σ, D_0) has TSP property.

Proof: Let Σ be a set of classical MDs, D_0 an initial instance, and $D_0, D_1, \ldots, D_k$ be a sequence of instances such that D_k is stable, and for every $i \in [1, k]$, $(D_{i-1}, D_i)_{[t_1^i, t_2^i]} \models \varphi_i$, for some $\varphi_i \in \Sigma$, and tuple identifiers t_1^i, t_2^i. Let D be a clean instance for D_0 (Σ not necessarily equal to D_k). It has been proved in [Bertossi et al., 2013] that if $t^{D_i}[A_1] \approx t'^{D_i}[A_2]$, then $t^{D}[A_1] \approx t'^{D}[A_2]$, for every $i \in [0, k]$, for every two tuple identifiers t, t' and two comparable attributes A_1, A_2. This follows that (Σ, D_0) has TSP property. The proof in [Bertossi et al., 2013] can be used for establishing that (Σ, D_0) has TSP property for a set of relation MDs Σ. □

6.3 Specialized Cleaning Programs for UCI Class

In this section we specialize the cleaning programs $\Pi(D_0, \Sigma)$, developed in Section 5.1, to obtain the *unique clean instance-programs* $\Pi^{UCI}(D_0, \Sigma)$ for enforcing sets of MDs with the UCI property. We establish that the resulting residual programs $\Pi^{UCI}(D_0, \Sigma)$ belong to a computationally well-behaved extension of plain Datalog.

Let D_0 be a given, possibly dirty initial instance w.r.t. a set Σ of classical MDs. As explained in Section 5.1, the cleaning program $\Pi(D_0, \Sigma)$ contains the rules in **1.-9.** to implicitly simulate the chase sequences, each one represented by a model of the program. In this direction, $\Pi(D_0, \Sigma)$ provides the freedom to match or not to match tuples, to obtain different chase sequences. In particular, the disjunctive rule in **2.**, with predicates $Match_{\varphi_j}$ and $NotMatch_{\varphi_j}$ in the head, is used to capture possible matchings when similarities hold for two tuples. Moreover, $\Pi(D_0, \Sigma)$ explicitly eliminates, using program constraints, instances that are the result of illegal applications of MDs. A set of MDs applications is illegal if there are some MDs enforcements that use old versions of tuples that have been replaced by new versions (cf. Section 5.1).

Since the chase-based semantics developed for classical MDs can be applied to relational MDs without any relevant change, the cleaning programs $\Pi(D_0, \Sigma)$, with a set Σ of relational MDs, can be developed without any relevant change. Actually, for relational MDs, we only have additional atoms in the body of the rule in **2.**. In this way, a set of tuples satisfy the body of the rule.

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 et al., 2013].

When (Σ, D_0), with a set Σ of relational MDs, has the TSP property, two tuples with similar attribute values, not in the initial instance D_0-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.

For (Σ, D_0) with the TSP property, if an MD application uses the older versions of tuples in $\bar{t}$ in a chase sequence, the MD would be applicable again with the newer versions of tuples in $\bar{t}$ later on in the same and the other chase sequences. As a result, with the TSP property, there is no need to have rules in the cleaning program to provide the freedom to match or not to match tuples. Consequently, there is no need to record the relative order of the two matchings using the auxiliary relation, *Prec*, in $\Pi(D_0, \Sigma)$. Therefore, we could remove the predicate $NotMatch_{\varphi_j}$ from the head of

the rule in **2.** resulting a non-disjunctive rule. We could also get rid of the program constraints and rules containing the relation *Prec* in $\Pi(D_0, \Sigma)$.

Notice that a pruning process based on the domination partial order between tuples has to be applied to collect the latest version of each tuple that is used to form the clean instance. Therefore, $\Pi^{UCI}(D_0, \Sigma)$ contains the rule in 4. of $\Pi(D_0, \Sigma)$.

For a given instance D_0 and set of relational MDs Σ, the program $\Pi^{UCI}(D_0, \Sigma)$ contains the following rules:

1. For every tuple (id) $t^{D_0} = R_j(\bar{a})$, the fact $R_j(t, \bar{a})$. We also need facts for the MFs as tables and similarity relations.

2. For each MD φ_j: $R_1(t_1, \bar{x}_1, y_1) \wedge R_2(t_2, \bar{x}_2, y_2) \wedge \psi(\bar{t}_3, \bar{z}) \longrightarrow y_1 \doteq y_2$, the program rule:

$$\begin{aligned} \mathit{Match}_{\varphi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2) \leftarrow\ & R_1(T_1, \bar{X}_1, Y_1),\ R_2(T_2, \bar{X}_2, Y_2), \\ & \psi(\bar{T}_3, \bar{Z}),\ Y_1 \neq Y_2. \end{aligned}$$

3.

$$\mathit{OldVersion}_i(T_1, \bar{Z}_1) \leftarrow R_i(T_1, \bar{Z}_1),\ R_i(T_1, \bar{Z}'_1), \bar{Z}_1 \preceq \bar{Z}'_1,\ \bar{Z}_1 \neq \bar{Z}'_1.$$

4.

$$\begin{aligned} R_1(T_1, \bar{X}_1, Y_3) &\leftarrow \mathit{Match}_{\varphi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2),\ M_j(Y_1, Y_2, Y_3). \\ R_2(T_2, \bar{X}_2, Y_3) &\leftarrow \mathit{Match}_{\varphi_j}(T_1, \bar{X}_1, Y_1, T_2, \bar{X}_2, Y_2),\ M_j(Y_1, Y_2, Y_3). \end{aligned}$$

5.

$$R_i^c(T_1, \bar{Z}_1) \leftarrow R_i(T_1, \bar{Z}_1),\ \mathit{not}\ \mathit{OldVersion}_i(T_1, \bar{Z}_1).$$

Theorem 6.3.1 Let (Σ, D_0) be a blocking combination. (a) The (D_0, Σ)-clean instance D is exactly the restrictions of the elements of the single stable model of the unique clean instance-program $\Pi^{UCI}(D_0, \Sigma)$ to schema $\mathcal{R}^c$. (b) The unique clean instance-program $\Pi^{UCI}(D_0, \Sigma)$ belongs to the class $\mathit{Datalog}^{\,not,s}$. □

Proof: Part (a) immediately follows from Lemma 6.2.2. Part (b) follows from the fact that the rules of the program $\Pi^{UCI}(D_0, \Sigma)$ are from the cleaning program $\Pi(D_0, \Sigma)$, belonging to the class $Datalog^{\vee, not, s}$ (cf. Proposition 5.3.1), and the fact that there is no disjunctive rule in $\Pi^{UCI}(D_0, \Sigma)$. □

Example 6.3.1 (ex. 6.1.11 cont.) The unique clean instance-program $\Pi^{UCI}(D_0, \Sigma)$ has the following rules (facts are not shown):

2.

$$Match_{\varphi_4}(T_1, X_1, Y_1, Z_1, Bl_1, T_2, X_2, Y_2, Z_2, Bl_2) \leftarrow Author(T_1, X_1, Y_1, Z_1, Bl_1), \\ Author(T_2, X_2, Y_2, Z_2, Bl_2),\ X_1 \approx X_2,\ Y_1 \approx Y_2,\ Bl_1 \neq Bl_2.$$

$$Match_{\varphi_3}(T_1, X_1, Y_1, Z_1, Bl_1, T_2, X_2, Y_2, Z_2, Bl_2) \leftarrow Paper(T_1, X_1, Y_1, Z_1, Bl_1), \\ Paper(T_2, X_2, Y_2, Z_2, Bl_2),\ Y_1 \approx Y_2,\ Z_1 \approx Z_2,\ Bl_1 \neq Bl_2.$$

$$Match_{\varphi_2}(T_3, X_1', Y_1', Z_1, Bl_1, T_4, X_2', Y_2', Z_2, Bl_2) \leftarrow Author(T_1, X_1, Y_1, Z_1, Bl_3), \\ Author(T_2, X_2, Y_2, Z_2, Bl_3), Paper(T_3, X_1', Y_1', Z_1, Bl_1), \\ Paper(T_4, X_2', Y_2', Z_2, Bl_2),\ X_1' \approx X_2',\ Bl_1 \neq Bl_2.$$

$$Match_{\varphi_1}(T_1, X_1, Y_1, Z_1, Bl_1, T_2, X_2, Y_2, Z_2, Bl_2) \leftarrow Author(T_1, X_1, Y_1, Z_1, Bl_1), \\ Author(T_2, X_2, Y_2, Z_2, Bl_2), Paper(T_3, X_1', Y_1', Z_1, Bl_3), \\ Paper(T_4, X_2', Y_2', Z_2, Bl_3),\ X_1 \approx X_2,\ Bl_1 \neq Bl_2.$$

3.

$$AOldVersion(T_1, \bar{W}_1) \leftarrow Author(T_1, \bar{W}_1),\ Author(T_1, \bar{W}_1'),\ \bar{W}_1 \preceq \bar{W}_1',\ \bar{W}_1 \neq \bar{W}_1'.$$

$$POldVersion(T_1, \bar{W}_1) \leftarrow Paper(T_1, \bar{W}_1),\ Paper(T_1, \bar{W}_1'),\ \bar{W}_1 \preceq \bar{W}_1',\ \bar{W}_1 \neq \bar{W}_1'.$$

4.

$$\begin{aligned}
Author(T_1, X_1, Y_3, Z_1, Bl_3) \;&\leftarrow\; Match_{\varphi_4}(T_1, X_1, Y_1, Z_1, Bl_1, T_2, X_2, Y_2, Z_2, Bl_2),\\
&\qquad M_B(Bl_1, Bl_2, Bl_3).\\
Paper(T_1, X_1, Y_1, Z_3, Bl_3) \;&\leftarrow\; Match_{\varphi_3}(T_3, X_1', Y_1', Z_1, Bl_1, T_4, X_2', Y_2', Z_2, Bl_2),\\
&\qquad M_B(Bl_1, Bl_2, Bl_3).\\
Paper(T_1, X_1, Y_1, Z_3, Bl_3) \;&\leftarrow\; Match_{\varphi_2}(T_3, X_1', Y_1', Z_1, Bl_1, T_4, X_2', Y_2', Z_2, Bl_2),\\
&\qquad M_B(Bl_1, Bl_2, Bl_3).\\
Author(T_1, X_1, Y_3, Z_1, Bl_3) \;&\leftarrow\; Match_{\varphi_1}(T_1, X_1, Y_1, Z_1, Bl_1, T_2, X_2, Y_2, Z_2, Bl_2),\\
&\qquad M_B(Bl_1, Bl_2, Bl_3).
\end{aligned}$$

5.

$$\begin{aligned}
&Author^c(T_1, \bar{W}_1) \;\leftarrow Author(T_1, \bar{W}_1),\; not\; AOldVersion(T_1, \bar{W}_1).\\
&Paper^c(T_1, \bar{W}_1) \;\leftarrow Paper(T_1, \bar{W}_1),\; not\; POldVersion(T_1, \bar{W}_1).
\end{aligned}$$

□

From the fact that the single clean instance of a program belonging to $Datalog^{not,s}$ can be computed in polynomial time in the size of the initial instance [Abiteboul et al., 1995], we obtain the following.

Corollary 2. For blocking (Σ, D_0), the single clean instance can be computed with the program $\Pi^{UCI}(D_0, \Sigma)$ in polynomial time in the size of the instance D_0. □

6.4 Conclusions

In this chapter, we have identified a class of MDs and initial instances that has good properties in terms of the number of models (a single one, in our case), and computable in polynomial time. We present a uniform methodology to specialize the

developed cleaning programs to obtain residual programs for enforcing MDs in classes with the unique clean instance property, containing MDs with similarity preserving MFs, interaction-free MDs and blocking combinations of MDs and initial instances. As a result, the residual programs belong to computationally well-behaved extension of plain Datalog.

The results presented in this chapter have been published in [Bahmani et al., 2017].

Chapter 7

ERBlox: Combining MDs with Machine Learning for ER

In this chapter, we describe our *ERBlox* system, and show the process and the benefits of integrating four components of ER in *ERBlox* system: (a) Building a classifier for duplicate/non-duplicate record pairs built using machine learning (ML) techniques; (b) Use of MDs for supporting the blocking phase of ML; (c) Record merging on the basis of the classifier results; and (d) The use of the declarative language *LogiQL* -an extended form of Datalog supported by the *LogicBlox* platform- for all activities related to data processing, and the specification and enforcement of MDs.

7.1 Overview of *ERBlox*

A high-level description of the components and workflow of *ERBlox* is given in Figure 7.1. In the rest of this section, numbers in boldface refer to the edges in that figure. *ERBlox*'s main four components are: 1. MD-based collective blocking (path $\mathbf{1}, \mathbf{3}, \mathbf{5}, \{\mathbf{6}, \mathbf{8}\}$), 2. Classification-model construction (all the tasks up to **12**, inclusive), 3. Duplicate detection (continues with edge **13**), and 4. MD-based merging (previous path extended with $\mathbf{14}, \mathbf{15}$). All the tasks in the figure, except for the classification model construction (that applying the SVM algorithm), are supported by *LogiQL*.[1]

The initial input data is stored in structured text files, which are initially standardized and free of misspellings, etc.. However, there may be duplicates. The general *LogiQL* program supporting the above workflow contains rules for importing data from the files into the extensions of relational predicates (tables). This is edge **1**. This results in a relational database instance T containing the training data (edge **2**), and instance D to be subject to ER (edge **3**).

[1]The implementation of in-house developed ML algorithms as components of the LogicBlox platform is ongoing work.

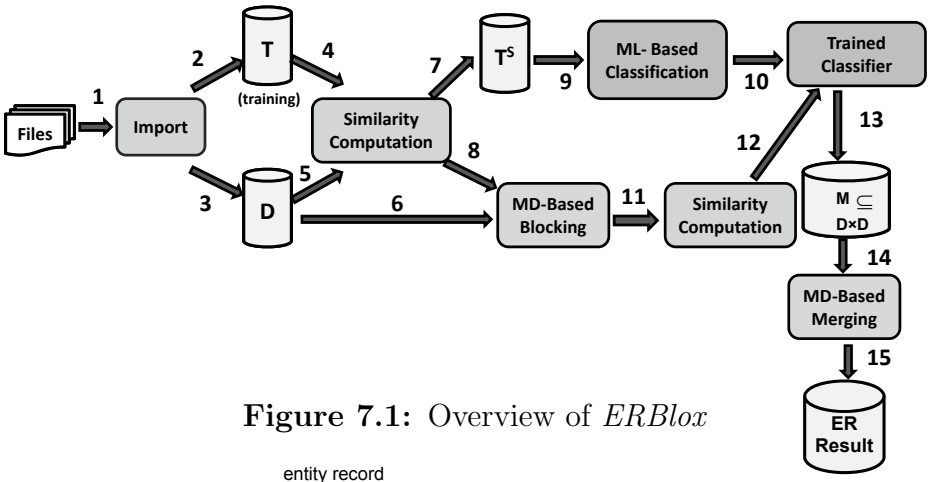


Figure 7.1: Overview of *ERBlox*

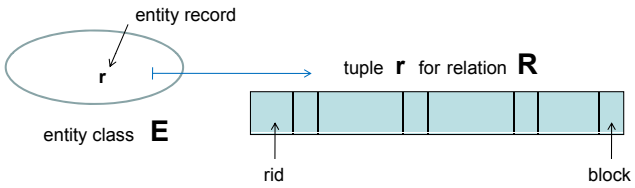


Figure 7.2: Records

Entity records are represented as relational tuples as shown in Figure 7.2. However, we will keep referring to them as records, and will be generally denoted with $r_1, r_2, ...$

The next tasks require similarity computation of pairs of records $\langle r_1, r_2 \rangle$ in T and (separately) in D (edges **4** and **5**). Similarity computation is based on two-argument similarity functions on the domain of a record attribute, say $f_i \colon Dom(A_i) \times Dom(A_i) \to [0, 1]$, each of which assigns a numerical value to (the comparison of) two values for attribute A_i, in two different records.

These similarity functions, being real-valued functions of the objects under classification, correspond to *features* in the general context of machine learning. They are considered only for a pre-chosen subset of record attributes. Weight-vectors $w(r_1, r_2) = \langle \cdots, w_i(f_i(r_1[A_i], r_2[A_i])), \cdots \rangle$ are formed by applying predefined weights, w_i, on real-valued similarity functions, f_i, on pair of of values for attributes A_i (edges **4** and **5**), as in Figure 7.3. (For more details on similarity computation see Section 7.2.)

Some record-pairs in the training dataset T are considered as duplicates and others as non-duplicates, which results (according to path $\mathbf{4}, \mathbf{7}$) in a "similarity-enhanced"

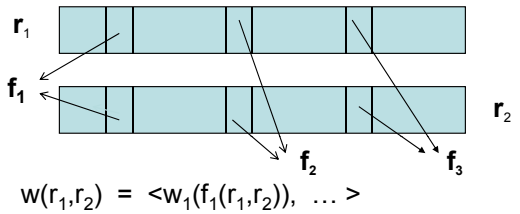


Figure 7.3: Feature-based similarity

training database T^s of tuples of the form $\langle r_1, r_2, w(r_1, r_2), L\rangle$, with label $L \in \{0, 1\}$. Label L indicates if the two records are duplicates ($L = 1$) or not ($L = 0$). These labels are consistent with the corresponding weight vectors. The classifier is trained using T^s, leading, through the application of the SVM algorithm, to the classification model (edges $\mathbf{9}, \mathbf{10}$) to be used for ER.

Blocking is applied to instance D, pre-classifying records into blocks, so that only records in a same block will form input pairs for the trained classification model. Accordingly, two records in a same block may end up as duplicates (of each other) or not, but two records in different blocks will never be duplicates.

We assume each record $r \in D$ has two extra, auxiliary attributes: a unique and global (numerical) *record identifier* (*rid*) whose value is originally assigned and never changes; and a *block number* that initially takes the *rid* as value. This block number is subject to changes.

For the records in D, similarity measures are used for blocking (see sub-path $\mathbf{5}, \mathbf{8}$). To decide if two records, r_1, r_2, go into the same block, the weight-vector $w(r_1, r_2)$ can be used: it can be read off from it if their values for certain attributes are similar enough or not. However, the similarity computations required for blocking may be different from those involved in the computation of the weight-vectors $w(r_1, r_2)$, which are related to the classification model. Either way, this similarity information is used by the *blocking-matching dependencies*, which are pre-declared and domain-dependent.

Blocking-MDs specify and enforce (through their RHSs) that the blocks (block numbers) of two records have to be made identical. This happens when certain similarities between pairs of attribute values appearing in the LHSs of the relational

MDs hold.

Example 7.1.1 (ex. 1.6.1 cont.) We could use the following MD for blocking Author records. In it there are similarity comparisons involving attributes for both entities Author and Paper:

$$Author(x_1, y_1, bl_1) \quad \wedge \quad Paper(y_1, z_1, bl_4) \wedge Author(x_2, y_2, bl_2) \wedge \\ Paper(y_2, z_2, bl_4) \wedge x_1 \approx_1 x_2 \wedge z_1 \approx_2 z_2 \longrightarrow bl_1 \doteq bl_2. \tag{7.1}$$

It specifies that when the Author record similarities on the LHS hold, and corresponding papers are in the same block, then blocks bl_1, bl_2 have to be made identical. □

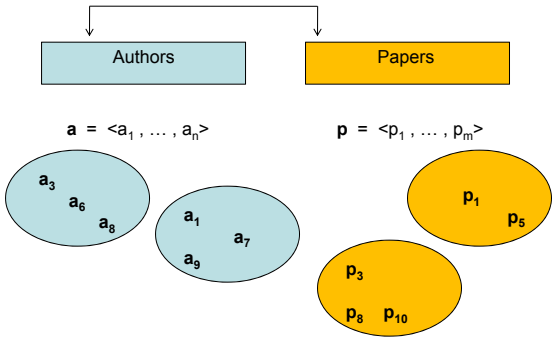


Figure 7.4: Collective blocking

We can see from (7.1) that information about classifications in blocks of records for the entity at hand (Author in this case) and for others entities (Paper in this case) may simultaneously appear as conditions in the LHSs of blocking-MDs. Furthermore, blocking-MDs may involve in their LHSs similarity conditions about attribute values in records for entities that are different ($z_1 \approx_2 z_2$ for Paper in (7.1)) from that under blocking (i.e. Author). All this is the basis for our "semantically-enhanced" collective blocking process. Cf. Example 1.6.1 and Figure 7.4.

The *MD-based collective blocking* stage (steps $\mathbf{5}, \mathbf{8}, \mathbf{6}$) consists in the enforcement of the blocking-MDs on D, which results in database D enhanced with information about the blocks to which the records are assigned. Pairs of records with the same block form *candidate duplicate record pairs*.

We make notice that blocking-MDs are more general than those of the form (2.1) introduced in [Fan et al., 2009] or Section 2.2: In their LHSs, they may contain regular database atoms, from more that one relation, that are used to give context to the similarity atoms in the MD.

A unique assignment of blocks is obtained after the enforcement of the blocking-MDs. Uniqueness is guaranteed by the properties of the classes of relational MDs we use for blocking: the blocking-MDs with D turn out to belong to the blocking class. About this, see the brief discussion in Section 7.3, and Chapter 6 for more details.

After the records have been assigned to blocks, record-pairs $\langle r_1, r_2\rangle$, with r_1, r_2 in the same block, are considered for the duplicate test. As this point, we proceed as we did for the training data: the weight-vectors $w(r_1, r_2)$, which represent the record-pairs in the "feature vector space", are computed and passed over to the classifier (edges **11**, **12**).[2]

The result of applying the trained ML-based classifier to the record-pairs is a set of triples $\langle r_1, r_2, 1\rangle$ containing records that come from the same block and are considered to be duplicates. Equivalently, the output is a set $M \subseteq D \times D$ containing pairs of duplicate records (edge **13**). The records in pairs in M are merged by enforcing an application-dependent set of (merge-)MDs (edge **14**). This set of MDs is classical which is different from that used for blocking.

Since records have kept their *rids*, we define a "similarity" predicate $\approx_{id}$ on the domain of *rids* as follows: $r_1[rid] \approx_{id} r_2[rid]$ iff $\langle r_1, r_2\rangle \in M$, i.e. iff the corresponding records are considered to be duplicates by the classifier. We informally denote $r_1[rid] \approx_{id} r_2[rid]$ by $r_1 \approx r_2$. Using this notation, the merge-MDs are usually and informally written in the form: $r_1 \approx r_2 \rightarrow r_1 \doteq r_2$. Here, the RHS is a shorthand for $r_1[A_1] \doteq r_2[A_1] \wedge \cdots \wedge r_1[A_m] \doteq r_2[A_m]$, where $A_1, \ldots, A_m$ are all the record attributes, excluding the first and last, i.e. ignoring the identifier and the block number (cf. Figure 7.2). Putting all together, merge-MDs take the official form:

$$r_1[rid] \approx_{id} r_2[rid] \longrightarrow r_1[A_1] \doteq r_2[A_1] \wedge \cdots \wedge r_1[A_m] \doteq r_2[A_m]. \tag{7.2}$$

[2]Similarity computations are kept in appropriate program predicates. So, similarity values computed before blocking can be reused at this stage, or whenever needed.

Merging at the attribute level, as required by the RHS, uses the predefined and domain-dependent matching functions m_{A_i}.

After applying the merge-MDs, a single duplicate-free instance is obtained from D (edge **15**). Uniqueness is guaranteed by the fact that the classes of merge-MDs that we use in our generic approach turn out to be interaction-free. More details are given in Section 7.5. (See also the brief discussion in Section 2.2.)

More details about the *ERBlox* system and our approach to ER are found in the subsequent sections.

7.2 Data Sets and Similarity Computation

We now describe some aspects of the MAS dataset that are relevant for the description of the *ERBlox* system components,[3] and the way the initial data is processed and created for their use with the *LogiQL* language of *LogicBlox*.

7.2.1 Data files and relational data

In the initial, structured data files, entries (non-relational records) for entity Author relation contains authors names and their affiliations. The entries for entity Paper contain: paper titles, years of publication, conference IDs, journal IDs, and keywords. Entries for the PaperAuthor relationship contain: papers IDs, authors IDs, authors names, and their affiliations. The entries for the Journal and Conference entities contain both short names of the publication venue, their full names, and their home pages.

the affiliation of the authors is often missing in the publications dataset of Microsoft Academic Search.

There are missing values in the MAS dataset. For example, the journal IDs or conference IDs of the papers are often missing in the publications of MAS. Additionally, there are non-word characters in the MAS dataset, e.g. in the affiliations of the authors. These make the MAS dataset noisy. The dataset is preprocessed by means of Python scripts, in preparation for proper *ERBlox* tasks. This is necessary because

[3]We also independently experimented with the DBLP and Cora Citation data sets, but we will concentrate on MAS.

the data gathering methods in general, and for the MAS dataset in particular, are often loosely controlled, resulting in out-of-range values, impossible data combinations, missing values, etc. For example, non-word characters are replaced by blanks, some strings are converted into lower case, etc. Not solving this problems may lead to later execution problems and, in the end, to misleading ER results. This preprocessing produces updated structured data files. As expected, there is no ER at this stage, and in the new files there may be many authors who publish under several variations of their names; also the same paper may appear under slightly different titles, etc. This kind of cleaning is that that will be performed with *ERBlox*.

Next, from the data in (the preprocessed) structured files, relational predicates and extensions (contents) for them extensions are created and computed, by means of a generic Datalog program in *LogiQL* [Aref et al., 2015,Halpin and Rugaber, 2015]. For example, these rules are part of the program:

$$\_file\_in(x1, x2, x3) \longrightarrow string(x1), string(x2), string(x3). \tag{7.3}$$

$$lang : physical : filePath[\text{`}\_file\_in] = \text{"}author.csv\text{"}. \tag{7.4}$$

$$+author(id1, x2, x3) \leftarrow \_file\_in(x1, x2, x3), string : int64 : convert[x1] = id1. \tag{7.5}$$

Here, (7.3) is a predicate schema declaration, in this case of the "_file_in" predicate with three string-valued attributes. It is used to automatically store the contents extracted from the source file "author.csv", as specified in (7.4). In *LogiQL* in general, metadata declarations use "→". (In *LogiQL*, each predicate's schema has to be declared, unless it can be inferred from the rest of the program.) Derivation rules, such as (7.5), use "←", as usual in Datalog. It defines the author predicate, and the "+" in the rule head inserts the data into the predicate extension. The rules also makes the first attribute a tuple identifier.

Figure 7.5 shows three relational predicates that are created and populated in this way: $Author(AID, Name, Affiliation, Bl\#)$, $Paper(PID, Title, Year, CID, JID, Keyword, Bl\#)$, $PaperAuthor(PID, AID, Name, Affiliation)$. The (partial) tables show that there may be missing attributes values.

Author	*AID*	*Name*	*Affiliation*	*Bl#*
	659	*Jean-Pierre Olivier de*	*Ecole des Hautes*	659
	2546	*Olivier de Sardan*	*Recherche Scientifique*	2546
	612	*Matthias Roeckl*	*German Aerospace Center*	612
	4994	*Matthias Roeckl*	*Institute of Communications*	4994

Paper	*PID*	*Title*	*Year*	*CID*	*JID*	...
	123	*Illness entities in West Africa*	1998	179		
	205	*Illness entities in Africa*	1998	179		
	769	*DLR Simulation Environment m3*	2007	146		
	195	*DLR Simulation Environment*	2007	146		...

...	*Keyword*	*Bl#*
	West Africa, Illness	123
	Africa, Illness	205
	Simulation m3	769
...	*Simulation*	195

PaperAuthor	*PID*	*AID*	*Name*	*Affiliation*
	123	659	*Jean-Pierre Olivier de*	*Ecole des Hautes*
	205	2546	*Olivier de Sardan*	*Recherche Scientifique*
	769	612	*Matthias Roeckl*	*German Aerospace Center*
	195	4994	*Matthias Roeckl*	*Institute of Communications*

Figure 7.5: Relation extensions from MAS using LogiQL rules

7.2.2 Features and similarity computation

Form the general description of our methodology in Section 7.1, a crucial component is *similarity computation*. It is needed for: (a) blocking, and (b) building the classification model. Similarity measures are related to *features*, which are numerical functions of the data, more precisely of the values of some specially chosen attributes. Feature selection is a fundamental task in machine learning [Dash and Liu, 1997,Tang et al., 2015]. Going in detail into this subject is beyond the scope of this work. Example 7.2.1 shows some specific aspects of this task as related to our dataset.

In relation to blocking, in order to decide if two records, r_1, r_2 in D, go into the same block, similarity of values for certain attributes are computed, those that are appear in similarity conditions in the LHSs of blocking-MDs. All is needed is whether they are similar enough or not, which is determined by predefined numerical thresholds.

For model building, similarity values are computed to build the weight-vectors, $w(r_1, r_2)$, for records r_1, r_2 from the training data in T. The numerical values in those vectors depend on the values taken by some selected record attributes (cf. Figure 7.3).

Example 7.2.1 (ex. 1.6.1 cont.) Bibliographic datasets, such as MAS, have been commonly used for evaluation of machine learning techniques, in particular, classification for ER. In our case, the features chosen in our work for the classification of records for entities Paper and Author from the MAS dataset (and the other datasets) correspond to those previously used in [Torvik and Smalheiser, 2009, Christen, 2008]. Experiments in [Kopcke and Rahm, 2010] show that the chosen features enhance generalization power of the classification model, by reducing over-fitting.

In the case of Paper-records, if the "journal ID" values are null in both records, but not their "conference ID" values, "journal ID" is not considered for feature computation, because it does not contribute to the recall or precision of the classifier under construction. Similarly, when the "conference ID" values are null. However, the values for "journal ID" and "conference ID" are replaced by "journal full name" and "conference full name" values that are found in Conference- and Journal-records, resp. Attributes *Title*, *Year*, *ConfFullName* or *JourFullName*, and *Keyword* are chosen for feature computation.

For feature computation in the case of Author-records, the *Name* attribute is split in two, the *Fname* and *Lname* attributes, to increase recall and precision of the classifier under construction. Accordingly, features are computed for attributes *Fname*, *Lname* and *Affiliation*. □

Once the classifier has been build, also weight-vectors, $w(r_1, r_2)$ are computed as inputs for the classifier, but this time for records from the data under classification (in D).[4]

Notice that numerical values, associated to similarities, in a weight-vector $w(r_1, r_2)$ for r_1, r_2 under classification, could be used as similarity information for blocking. However, the attributes and features used for blocking may be different from those used for weight-vectors. For example, in our experiments with the MAS dataset, the classification of Author-records is based on attributes *Fname*, *Lname*, and *Affiliation*.

[4]In our experiments, we did not care about null values in records under classification. Learning, inference, and prediction in the presence of missing values are pervasive problems in machine learning and statistical data analysis. Dealing with missing values is beyond the scope of this work.

For blocking, the latter is reused as such (cf. MD (7.9) below), but also the combination of *Fname* and *Lname* is reused, as attribute *Name* in MDs (cf. MDs (7.9) and (7.11) below).

There is a class of well-known and widely applied similarity functions that are used in data cleaning and machine learning [Cohen et al., 2003]. For our application with *ERBlox* we used three of them, depending on the attribute domains for the MAS dataset. Long-text-valued attributes, in our case, e.g. for the *Affiliation* attribute, their values are represented as lists of strings. For computing similarities between these kind of attribute values, the "TF-IDF cosine" measure was used [Salton and Buckley, 1988]. It assigns low weights to frequent strings and high weights to rare strings. For example, affiliation values usually contain multiple strings, e.g. "Carleton University, School of Computer Science". Among them, some are frequent, e.g. "School", and others are rare, e.g. "Carleton".

For attributes with "short" string values, such as author names, "Jaro-Winkler" similarity was used [Jaro, 1995,Winkler, 1999]. This measure counts the characters in common in two strings, even if they are misplaced by a short distance. For example, this measure gives a high similarity value to the pair of first names "Zeinab" and "Zienab". In the MAS dataset, there are many author first names and last names presenting this kind of misspellings.

For numerical attributes, such as publication year, the "Levenshtein distance" was used [Navarro, 2001]. The similarity of two numbers is based on the minimum number of operations required to transform one into the other.

As already mentioned in Section 7.1, these similarity measures are used, but differently, for blocking and the creation and application of the classification algorithm. In the former case, similarity values related to LHSs of blocking-MDs are compared with user-defined thresholds, in essence, making them boolean variables. In the latter case, they are used for computing the similarity vectors, which contain numerical values (in $\mathbb{R}$). Notice that similarity measures are *not* used beyond the output of the classification algorithm, in particular, not for MD-based record merging.

Similarity computation for *ERBlox* is done through *LogiQL*-rules that define the similarity functions. In particular, similarity computations are kept in extensions of

program-defined predicates. For example, if the similarity value for the pair of values, a_1, a_2, for attribute *Title* is above the threshold, a tuple *Title-Sim*(a_1, a_2) is created by the program.

It is worth noting that the datasets used in this work are balanced, meaning there are approximately the same number of duplicate and non-duplicate record-pairs [Roy et al., 2013].

7.3 MD-Based Collective Blocking

As described in Section 7.1, the *Block* attribute, Bl, in records takes integer numerical values; and before the blocking process starts (or blocking-MDs are enforced), each record in the instance D has a unique block number that coincides with its *rid*. Blocking policies are specified by blocking-MDs, all of which use the same matching function for identity enforcement, given by:

$$\text{For } i, j \in \mathbb{N}, \text{ with } j \leq i, \quad m_{Bl}(i, j) := i. \tag{7.6}$$

A blocking MD that identifies block numbers (i.e. makes them identical) in two records (tuples) for database relation R (cf. Figure 7.2) takes the form:

$$R(\bar{x}_1, bl_1) \ \wedge \ R(\bar{x}_2, bl_2) \ \wedge \ \psi(\bar{x}_3) \ \longrightarrow \ bl_1 \doteq bl_2. \tag{7.7}$$

Here, bl_1, bl_2 are variables for block numbers, R is a database predicate (representing an entity), the lists of variables $\bar{x}_1, \bar{x}_2$ stand for all the attributes in R but $Bl\#$, for which variables bl_i are used. The MD in (7.7) is *relational* when formula ψ in it is a conjunction of relational atoms plus comparison atoms via similarity predicates; including implicit equalities of block numbers (but not $\approx$-similarities between block numbers). The variables in $\psi(\bar{x}_3)$ may appear among those in $\bar{x}_1, \bar{x}_2$ (in R) or in another database predicate or in a similarity atom. We assume that $(\bar{x}_1 \cup \bar{x}_2) \cap \bar{x}_3 \neq \emptyset$. (Cf. 6.1 for more details on relational MDs.)

An example is the MD in (7.1), where the leading R_1, R_2-atoms are Author tuples, the extra conjunction contains Paper atoms, non-block-similarities, and an implicit equality of blocks through the shared use of variable bl_4. There, ψ is *Paper*$(t_3, y_1', z_1,$

$bl_4) \ \wedge y_1 \approx y_1' \ \wedge \ Paper(t_4, y_2', z_2, bl_4) \ \wedge \ y_2 \approx y_2' \ \wedge \ x_1 \approx x_2 \ \wedge \ y_1 \approx y_2.$

Example 7.3.1 These are some of the blocking-MDs used with the MAS dataset. The first two are *classical* blocking-MDs, and the last two are properly *relational* blocking-MDs:

$$Paper(pid_1, x_1, y_1, z_1, w_1, v_1, bl_1) \ \wedge \ Paper(pid_2, x_2, y_2, z_2, w_2, v_2, bl_2) \ \wedge \quad (7.8)$$
$$x_1 \approx_{Title} x_2 \ \wedge \ y_1 = y_2 \ \wedge z_1 = z_2 \ \longrightarrow \ bl_1 \doteq bl_2.$$
$$Author(aid_1, x_1, y_1, bl_1) \ \wedge \ Author(aid_2, x_2, y_2, bl_2) \ \wedge \quad (7.9)$$
$$x_1 \approx_{Name} x_2 \ \wedge \ y_1 \approx_{Aff} y_2 \ \longrightarrow \ bl_1 \doteq bl_2.$$

$$Paper(pid_1, x_1, y_1, z_1, w_1, v_1, bl_1) \ \wedge \ Paper(pid_2, x_2, y_2, z_2, w_2, v_2, bl_2) \wedge \quad (7.10)$$
$$PaperAuthor(pid_1, aid_1, x_1', y_1') \ \wedge PaperAuthor(pid_2, aid_2, x_2', y_2') \ \wedge$$
$$Author(aid_1, x_1', y_1', bl_3) \ \wedge Author(aid_2, x_2', y_2', bl_3) \wedge x_1 \approx_{Title} x_2$$
$$\longrightarrow \ bl_1 \doteq bl_2.$$

$$Author(aid_1, x_1, y_1, bl_1) \ \wedge Author(aid_2, x_2, y_2, bl_2) \ \wedge \ x_1 \approx_{Name} x_2 \wedge \quad (7.11)$$
$$PaperAuthor(pid_1, aid_1, x_1, y_1) \ \wedge PaperAuthor(pid_2, aid_2, x_2, y_2) \wedge$$
$$Paper(pid_1, x_1', y_1', z_1', w_1', v_1', bl_3) \wedge Paper(pid_2, x_2', y_2', z_2', w_2', v_2', bl_3)$$
$$\longrightarrow \ bl_1 \doteq bl_2.$$

In informal terms, (7.8) requires that, for every two Paper entities $\mathbf{p}_1, \mathbf{p}_2$ for which the values for attribute *Title* are similar, and with the same publication year and conference ID, the values for attribute *Bl#* must be made identical. According to (7.9), whenever there are similar values for name and affiliation in Author, the corresponding authors should go into the same block.

The relational blocking-MDs in (7.10) and (7.11) *collectively block* Paper and Author entities. According to (7.10), a blocking-MD for Paper, if two authors are in the same block, their papers $\mathbf{p}_1$, $\mathbf{p}_2$ having similar titles must be in the same block too. Notice that if papers $\mathbf{p}_1$ and $\mathbf{p}_2$ have similar titles, but they do not have same publication

year or conference ID, we cannot block them together using (7.8) alone. The blocking-MD (7.11) for Author is similar to that discussed in Example 6.1.2. □

For the application-dependent set, Σ^{Bl}, of blocking-MDs we adopt the chase-based semantics [Bertossi et al., 2013], which may lead, in general, to several, alternative final instances. In each of them, every record is assigned to a unique block, but now records may share block numbers, which is interpreted as belonging to the same block. In principle, there might be two final instances where the same pair of records is put in the same block in one of them, but not in the other one. However, with a set of the relational blocking-MDs of the form (7.7) acting on an initial instance D (created with *LogicBlox* as described above), the chase-based enforcement of the MDs results in a single, final instance, D^{Bl}. This is because the combination of the blocking-MDs with the initial instance D turns out to belong to the blocking class, which has the UCI property (cf. Section 6.1.1 and 6.2).

That the initial instance and the blocking-MDs form a blocking combination is easy to see. In fact, initially the block numbers in tuples (or records) are all different, they are the same as their tids. Now, the only relevant attributes in records (for blocking class membership) are "block attributes", those appearing in RHSs of blocking-MDs (cf. (7.7)). In the LHSs of blocking-MDs they may appear only in implicit equality atoms. Since all initial block numbers in D are different, no relevant equality holds in D.

Due to the UCI property of blocking-MDs in combination with the initial instance, MD enforcement leads to a single instance that can be computed in polynomial time in data, which gives us the hope to use a computationally well-behaved extension of plain Datalog for MD enforcement (and blocking). It turns out that the representation and enforcement of these MDs can be done by means of Datalog with stratified negation [Ceri et al., 1989, Abiteboul et al., 1995], which is supported by *LogiQL*. Stratified Datalog programs have a unique stable model, which can be computed in a bottom-up manner in polynomial time in the size of the extensional database.

General sets of MDs are specified and enforced by means of disjunctive, stratified

answer set programs, with the possibly multiple resolved instances corresponding to the stable models of the program (cf. Section 5.1). These programs are specialized, via an automated rewriting mechanism, for the blocking class, obtaining residual programs in Datalog with stratified negation (cf. Section 6.3).

In *LogiQL*, blocking-MDs take the form as Datalog rules:

$$R[\bar{X}_1]=Bl_2,\quad R[\bar{X}_2]=Bl_2 \quad\longleftarrow\quad R[\bar{X}_1]=Bl_1,\quad R[\bar{X}_2]=Bl_2,\quad \psi(\bar{X}_3),\quad Bl_1<Bl_2, \tag{7.12}$$

subject to the same conditions as for (7.7). The condition $Bl_1 < Bl_2$ in the rule body corresponds to the use of the MF m_{Bl} in (7.6).

An atom of the form $R[\bar{X}]{=}Bl$ not only declares Bl as an attribute value for R, but also that predicate R is functional on $\bar{X}$ [Aref et al., 2015]: Each record in R can have only one block number.

In addition to the blocking-MDs, we need some auxiliary rules (cf. Section 6.3), which we discuss next. Given an initial instance D and a set of blocking-MDs Σ^{Bl}, the *LogiQL*-program $\Pi^{Bl}(D)$ that specifies MD-based collective blocking contains the following rules:

1. For every atom $R(rid, \bar{x}, bl) \in D$, the fact $R[rid, \bar{x}] = bl$. That is, initially, the block number, bl, is functionally assigned the value rid.

2. Facts of the form $A\text{-}Sim(a_1, a_2)$, where $a_1, a_2 \in Dom(A)$, the finite attribute domain of an attribute A. They state that the two values are similar, which is determined by similarity computation. (Cf. Section 7.2.2 for more on similarity computation.)

3. Rules for the blocking-MDs, as in (7.12).

4. Rules specifying older versions of entity records (in relation R) after MD-enforcement:

$$R\text{-}OldVer(r, \bar{x}, bl_1) \quad\longleftarrow\quad R[r, \bar{x}] = bl_1,\ R[r, \bar{x}] = bl_2,\ bl_1 < bl_2.$$

Here, variable r stands for the *rid*. Since for each *rid*, r, there could be several atoms of the form $R[r, \bar{x}] = bl$, corresponding to the evolution of the record identified by r through an MD-based chase sequence, the rule specifies as old those versions of the record with a block number that is smaller than the last one obtained for it.

5. Rules that collect the records' latest versions, to form blocks:

$$R\text{-}Block[r, \bar{x}] = bl \quad \longleftarrow \quad R[r, \bar{x}] = bl, \ not\ R\text{-}OldVer(r, \bar{x}, bl).$$

The rule collects R-records that are not old versions.[5]

Program $\Pi^{Bl}(D)$ as above is a Datalog program with stratified negation (there is no recursion through negation). In computational terms, this means that the program computes old version of records (using negation), and next definitive blocks are computed. As expected from the UCI property of blocking-MDs in combination with the initial instance, the program has and computes a single model, in polynomial time in the size of the initial instance. From it, the final block numbers of records can be read off.

Example 7.3.2 (ex. 7.3.1 cont.) We consider only blocking-MDs (7.8) and (7.10). The portion of $\Pi^{Bl}(D)$ that does the blocking of records for the Paper entity has the following rules (we follow the numbering used in the generic program):

2. Facts such as:

Title-Sim("*Illness entities in West Africa*", "*Illness entities in Africa*").

Title-Sim("*DLR Simulation Environment m3*", "*DLR Simulation Environment*").

3. $Paper[pid_1, x_1, y_1, z_1, w_1, v_1] = bl_2, Paper[pid_2, x_2, y_2, z_2, w_2, v_2] = bl_2 \ \leftarrow$

$$Paper[pid_1, x_1, y_1, z_1, w_1, v_1] = bl_1, Paper[pid_2, x_2, y_2, z_2, w_2, v_2] = bl_2,$$
$$Title\text{-}Sim(x_1, x_2), y_1 = y_2, z_1 = z_2, bl_1 < bl_2.$$

$Paper[pid_1, x_1, y_1, z_1, w_1, v_1] = bl_2, Paper[pid_2, x_2, y_2, z_2, w_2, v_2] = bl_2 \ \leftarrow$

[5] *LogiQL*, uses "!" instead of *not* for Datalog negation [Aref et al., 2015].

$$Paper[pid_1, x_1, y_1, z_1, w_1, v_1] = bl_1, Paper[pid_2, x_2, y_2, z_2, w_2, v_2] = bl_2,$$
$$Title\text{-}Sim(x_1, x_2), PaperAuthor(pid_1, aid_1, x'_1, y'_1), bl_1 < bl_2,$$
$$PaperAuthor(pid_2, aid_2, x'_2, y'_2), Author[aid_1, x'_1, y'_1] = bl_3,$$
$$Author[aid_2, x'_2, y'_2] = bl_3.$$

4. $Paper\text{-}OldVer(pid, x, y, z, w, v, bl_1) \leftarrow Paper[pid, x, y, z, w, v] = bl_1,$
$$Paper[pid, x, y, z, w, v] = bl_2,\ bl_1 < bl_2.$$

5. $Paper\text{-}Block[pid, x, y, z, w, v] = bl \leftarrow Paper[pid, x, y, z, w, v] = bl,$
$$not\ Paper\text{-}OldVer(pid, x, y, z, w, v, bl).$$

By restricting the model of the program to attributes *PID* and *Block#* of predicate *Paper-Block*, we obtain blocks: $\{123, 205\}, \{195, 769\}, \ldots$. That is, the papers with *pids* 123 and 205 are blocked together; similarly for those with *pids* 195 and 769, etc. □

The execution of the blocking-program $\Pi^{Bl}(D)$ will return in the end, for each entity-relation R a list of subsets of the extension of R in D. These subsets are blocks of R-records. Pairs of records in a same block will be inputs to the classification model, which has to be independently constructed first.

7.4 Classification Model Construction

Both for the classification model construction and duplicate detection with it, weight-vectors for record-pairs have to be computed. The numerical values for these vectors come from features related to similarity comparisons between attribute values for two records r_1, r_2. Only a subset of record attributes are chosen, those attributes that have strong discriminatory power, to achieve maximum classification recall and precision (cf. Section 7.2.2).

Different ML techniques can be used for the classification model. We used the established classification algorithm *support vector machines* (SVM). This algorithm works well on small datasets with high dimensional feature vectors [Vapnik, 2009].

Data in MAS, DBLP and Cora used in this work have the mentioned property. Additionally, SVM delivers a unique solution, in contrast to some ML algorithms which have multiple solutions, e.g. classification with neural networks [Zhang, 2000].

As Figure 7.6 shows, in our case we need to classify *pairs of records*, that is our weight vectors are *record-pairs* of the form $\mathbf{e} = \langle r_1, r_2 \rangle$. If $h(\mathbf{e}) = 1$, the SVM classifier returns as output $\langle r_1, r_2, 1 \rangle$, meaning that the two records are duplicates (of each other). Otherwise, it returns $\langle r_1, r_2, 0 \rangle$, meaning that the records are non-duplicates (of each other).

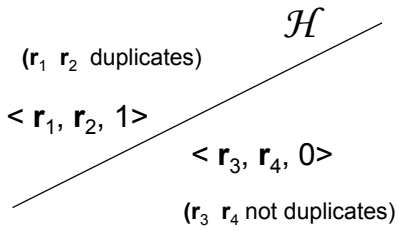


Figure 7.6: SVM classification hyperplane

The input to the SVM algorithm (that will produce the classification model) is a set of tuples of the form $\langle r_1, r_2, w(r_1, r_2), L \rangle$, where r_1, r_2 are records (for the same entity) in the training dataset T, $L \in \{0, 1\}$, and $w(r_1, r_2)$ is the computed weight-vector for the record-pair. In the *LogiQL* program, that input uses two defined predicates. Predicate *TrainLabel* has two arguments: One for pairs of *rids*, $r_1 r_2$, together, which is called "the vector id" for vector $w(r_1, r_2) = \langle w_1, \ldots, w_n \rangle$, and another to represents label L associated to $w(r_1, r_2)$. Predicate *TrainVector* contains one argument for vector ids, and n arguments to represent entries w_i in the weight-vectors $w(r_1, r_2)$.

Several ML techniques are accessible from (or within) the *LogicBlox* platform, through the *BloxMLPack* library that provides a generic Datalog interface. Then, *ERBlox* can call a SVM-based classification model constructor, through the general *LogiQL* program.

In particular, the *BloxMLPack* wraps calls to the machine learning library in a predicate-to-predicate mapping called *mlpack*, and manages marshalling the inputs and outputs to the machine learning library from/to *LogiQL* predicates. This is done via special rules in *LogiQL* that come in two modes: the learning mode (when a model

is being learned, in our case, a SVM classification model), and the evaluation mode (when the model is applied, for record-pair classification in our case) [LOGICBLOX, 2012, Aref et al., 2015]. We do not give here the formal syntax and semantics for these rules, but just the gist by means of an example.

Assume that we want to train a SVM-model for Author-record classification. For invoking SVM from *LogiQL*, a relation $InputMatrix[j, i]$ is needed. It contains tabular data where each column (j) represents a feature of Author-records, while each row i represents a vector id for which the tuple $TrainVector(i, w_1, w_2, w_3)$ exists. So, $InputMatrix[j, i]$ represents the value of the feature j in the weight-vector i. The following rules are used in *LogiQl* to populate relation *InputMatrix*: (They involve predicates $Feature(\text{“}Fname\text{”})$, $Feature(\text{“}Lname\text{”})$, and $Feature(\text{“}Affiliation\text{”})$, associated to the three chosen attributes for Author-records. They appear in quotes, because they are constants, i.e. attribute names.)

$$InputMatrix[\text{“}Fname\text{”}, i] = w_1,\ InputMatrix[\text{“}Lname\text{”}, i] = w_2,$$
$$InputMatrix[\text{“}Affiliation\text{”}, i] = w_3 \longleftarrow TrainVector(i, w_1, w_2, w_3),$$
$$Feature(\text{“}Fname\text{”}),\ Feature(\text{“}Lname\text{”}),\ Feature(\text{“}Affiliation\text{”}).$$

The following learning rule learns a SVM model for Author, and stores the resulting model in the predicate $SVMsModel(model)$:

$$SVMsModel(m) \longleftarrow mlpack \ll m = SVM(\bar{p}), train \gg InputMatrix[j, i] = v,$$
$$Feature(j), TrainLabel(i, l).$$

Here, the head of the rule defines a predicate where the ML algorithm outputs its results, while the body of the rule lists a collection of predicates that supplies data for the ML algorithm. In the above rule, the required parameters $\bar{p}$ for running the SVM algorithm are specified by the user. The above rule is in the training mode.

7.5 Duplicate Detection and MD-Based Merging

The input to the trained classifier is a set of tuples of the form $\langle r_1, r_2, w(r_1, r_2)\rangle$, where r_1, r_2 are record (ids) in a same block for a relation R, and $w(r_1, r_2)$ is the weight-vector for the record-pair $\langle r_1, r_2\rangle$. The output is a set of triples of the form $\langle r_1, r_2, 1\rangle$ or $\langle r_1, r_2, 0\rangle$.

Using *LogiQL* rules, the triples $\langle r_1, r_2, 1\rangle$ form the extension of a defined predicate *R-Duplicate*.

Example 7.5.1 (ex. 7.2.1 and 7.3.2 cont.) Considering the previous Paper-records, the input to the trained classifier consists of: $\langle 123, 205, w(123, 205)\rangle$, with $w(123, 205) = [0.8, 1.0, 1.0, 0.7]$; and $\langle 195, 769, w(195,\ 769)\rangle$, with $w(195, 769) = [0.93, 1.0, 1.0, 0.5]$.

In this case, the SVM-based classifier returns $\langle[0.8, 1.0, 1.0, 0.7], 1\rangle$ and $\langle[0.93, 1.0,$ $1.0, 0.5], 1\rangle$. Accordingly, the tuples *Paper-Duplicate*$(123, 205)$ and *Paper-Duplicate* $(195, 769)$ are created. □

The extensions of predicates *R-Duplicate* will be the input to the merging process.

Record merging is carried out through the enforcement of merge-MDs, as described in Section 7.1, where we showed that they form an *interaction-free* set. Consequently, there is a single instance resulting from their enforcement.

The merge-MDs use application-dependent matching functions (MFs). The generic merge-MDs in (7.2) can be expressed in their Datalog versions by means of the above mentioned *R-Duplicate* predicates. The RHSs of MDs in (7.2) have to be expressed in terms of MFs, m_{A_i}. All these become ingredients of a Datalog merge-program Π^M. Actually, the general cleaning programs for general sets of MDs in Section 5.1 are specialized to the case of interaction-free MDs, obtaining residual programs in Datalog with stratified negation in Section 6.3. Therefore, the representation and enforcement of merge-MDs in Π^M can be done by means of *LogiQL*.

Example 7.5.2 (ex. 7.2.1 cont.) Duplicate Paper-records are merged by enforcing the merge-MD:

$$Paper[pid_1] \approx Paper[pid_2] \;\longrightarrow\; Paper[Title, Year, CID, Keyword] \doteq Paper[Title,\ Year, CID,\ Keyword]. \qquad \square$$

The general *LogiQL* program, Π^M, for MD-based merging contains rules as in **1.-4.** below (cf. Section 6.3 for more details):

1. The ground atoms of the form $R\text{-}Duplicate(r_1, r_2)$ mentioned above, and those representing MFs, of the form $m_A(a_1, a_2) = a_3$.

2. For an MD $R[r_1] \approx R[r_2] \;\longrightarrow\; R[\bar{r}_1] \doteq R[\bar{r}_2]$, the rules:

$$R(r_1, \bar{x}_3),\ R(r_2, \bar{x}_3) \longleftarrow R\text{-}Duplicate(r_1, r_2),\ R(r_1, \bar{x}_1),\ R(r_2, \bar{x}_2), \\ m(\bar{x}_1, \bar{x}_2) = \bar{x}_3,$$

where $\bar{x}_1, \bar{x}_2, \bar{x}_3$ stand for all attributes of relation R, except for the *rid* and the block number (block numbers play no role in merging). $m(\bar{x}_1, \bar{x}_2) = \bar{x}_3$ is just a shorthand to denote the componentwise application of m individual MFs m_{A_i} (cf. (7.2)).

At the end of the iterative application of these rules, there may be several tuples with different *rids* but identical "tails". Only one of those tuples is is kept in the resolved instance.

3. As for the blocking-program $\Pi^{Bl}(D)$ of Section 7.3, we need rules specifying the old versions of a record:

$$R\text{-}OldVer(r_1, \bar{x}_1) \longleftarrow R(r_1, \bar{x}_1),\ R(r_1, \bar{x}_2),\ \bar{x}_1 \prec \bar{x}_2,$$

where $\bar{x}_1$ stands for all attributes other than *rid* and the block number; and $\bar{x}_1 \prec \bar{x}_2$ means componentwise comparison of values according to the partial orders defined by the MFs. (Recall from Section 2.2, that each application of an MF makes us grow in the information lattice: the highest values are the newest values.)

4. Finally, we introduce rules to collect, in a new predicate *R-ER*, the latest version of each record, to build the final resolved instance:

$$R\text{-}ER(r, \bar{x}) \;\; \leftarrow \;\; R(r, \bar{x}), \;\; not \;\; R\text{-}OldVer(r, \bar{x}).$$

This is a stratified Datalog program that computes a single resolved instance in polynomial time in the size of the extensional database, (cf. Section 6.3), in this case formed by the contents of relations *R-Duplicate* and *D*.

In our application to bibliographic datasets, we used as matching functions "the union case" [Benjelloun et al., 2009], which was investigated in detail in [Bertossi et al., 2013] in terms of MDs. The idea is to treat attribute values as objects, i.e. sets of pairs attribute/value. For example, the address "250 Hamilton Str., Peterbrook, K2J5G3" could be represented as the set $\{\langle \mathsf{number}, 250\rangle, \langle \mathsf{stName}, \textit{Hamilton Str.}\rangle, \langle \mathsf{city}, \textit{Peter brook}\rangle, \langle \mathsf{areaCode}, \textit{K2J5G3}\rangle\}$. When two values of this kind are merged, their union is computed. For example, the two strings "250 Hamilton Str., K2J5G3" and "Hamilton Str., Peterbook", represented as objects, are merged into "250 Hamilton Str., Peterbook, K2J5G3" [Bertossi et al., 2013]. This generic merge function has the advantage that information is preserved. It also works fine when two values complete each other. In the case of two alternative values, the two versions will be kept in the union. This may require some sort of domain-dependent postprocessing, essentially making choices and possibly edits. In any case, working with the union case for matching dependencies is good enough for our purposes, namely to compare traditional techniques with ours.

We point out that MD-based merging takes care of "transitive cases" produced by the classifier. More precisely, if it returns $\langle r_1, r_2, 1\rangle$ and $\langle r_2, r_3, 1\rangle$, but not $\langle r_1, r_3, 1\rangle$, we still merge r_1, r_3 (even when $r_1 \approx r_3$ does not hold). Indeed, if MD-enforcement first merges r_1, r_2 into the same record, the similarity between r_2 and r_3 still holds (it was pre-computed and stored, and not destroyed by the updating of attributes values of r_2). Then, the merge-MD will be applied to r_3 and the new version of r_2. Iteratively, r_1, r_2, r_3 will end up having the same attribute values (except for the *rid*).[6]

[6]There is certain similarity with the argument around the blocking case of MDs in Section 7.3. This is not a coincidence: interaction-free MDs form a case of blocking, for any initial instance.

There might be applications where we do *not* want this form of full entity resolution triggered by transitivity. If that is the case, we could use semantic constraints on the ER result (or process). Actually, *negative rules* have been proposed in [Whang et al., 2009a], and discussed in Section 5.4.2 in the context of general answer set programs for MD-based ER. However, the introduction of constraints into Datalog changes the entire picture. Under a common approach, if the intended model of the program does not satisfy the constraint, it is rejected. This is not particularly appealing from the application point of view. An alternative is to transform constraints into non-stratified program rules, which would take us in general to the realm of ASPs [Brewka et al., 2011]. In any case, developing this case in full is outside the scope of this work.

7.6 Experimental Results

In comparison with *standard blocking* (SB) techniques, our experiments with the MAS dataset show that our approach to ER, in particular, through the use of semantically rich matching dependencies for blocking result in lower *reduction ratio* for blocking, and higher *recall* and *precision* for classification. These are positive results that can also be observed in the experimental results with the DBLP and Cora Citation datasets. Cf. Figures 7.7, 7.9, and 7.10 (more details follow below).

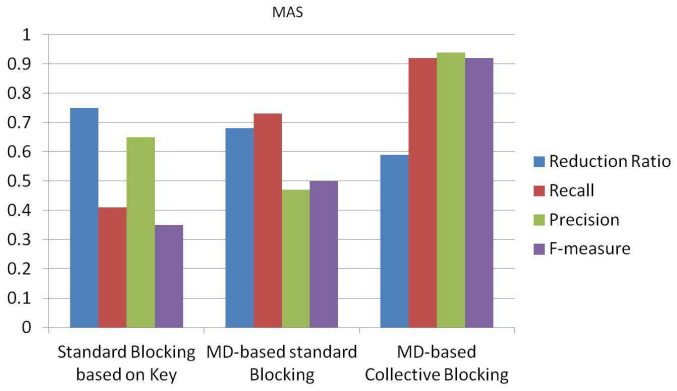


Figure 7.7: The experiments (MAS)

We considered three different blocking techniques, shown, respectively, in the sets of columns in Figure 7.7: (a) *Standard Blocking* (SB), (b) MD-based Standard Blocking (MDSB), and (c) MD-based Collective Blocking (MDCB), which we now describe:

(a) According to SB, records are clustered into a same block when they share the identical values for blocking keys [Jaro, 1989].

(b) MDSB generalizes standard blocking through the use of blocking-MDs that consider on the LHS exactly the same attributes (actually, keys) as in SB. However, for some of the attributes, equality is replaced by similarity, adding more flexibility and the possibility of increasing the number of two-record comparisons.

For example, the following could be an MD directly representing a blocking-key rule:

$$Author(aid_1, x_1, y_1, bl_1) \ \wedge \ Author(aid_2, x_2, y_2, bl_2) \ \wedge \ x_1 = x_2 \ \wedge \\ y_1 = y_2 \ \longrightarrow \ bl_1 \doteq bl_2;$$

and the following could be a relaxed version of it, a single-relation MD where instead of equalities we now have similarities:

$$Author(aid_1, x_1, y_1, bl_1) \ \wedge \ Author(aid_2, x_2, y_2, bl_2) \ \wedge \ x_1 \approx_{Name} x_2 \ \wedge \\ y_1 \approx_{Aff} y_2 \ \longrightarrow \ bl_1 \doteq bl_2.$$

In this case we had as many MDs as blocking keys in SB, and they are each, single entity, such as (7.8) and (7.9) in Example 7.3.1).

(c) MDCB uses, in addition to single-entity blocking-MDs, also multi-relational MDs, such as (7.10) and (7.11) in Example 7.3.1. In this case, the set of MDs contains all those in MDSB plus properly multi-relational ones.

Reduction ratio refers to the record-blocking task of ER, and is defined by $1 - \frac{S}{N}$, where S is the number of candidate duplicate record-pairs produced by the blocking technique, and N is the total number of possible candidate duplicate record-pairs in the entire dataset. If there are n records for an entity, then $N = n \times n$ for that entity.

Reduction ratio is the relative reduction in the number of candidate duplicate record-pairs to be compared. The higher the reduction ratio, the fewer the candidate

record-pairs that are generated, but the quality of the generated candidate record-pairs is not considered [Christen, 2011].

That the reduction ratio decreases from left to right in Figure 7.7 shows that the use of blocking-MDs increasingly captures more potential record-pairs comparisons that would be missed otherwise.

Recall, precision and F-measure refer to the result of the next task, that of classification [Christen, 2011]. *Recall* (R) is defined by $TP/(TP + FN)$, where TP and FN stand for true positives and false negatives, resp. Higher recall means that more true candidate duplicate record-pairs have been actually found. Now, with *precision* (P) defined by $TP/(TP + FP)$, where FP stands for flase positives, higher precision means more of the retrieved candidate duplicate record-pairs are actually true (cf. Figure 7.8).

The F-measure, commonly used in information retrieval [Manning and Schutze, 2008], measures accuracy using precision and recall. It is given by $(2 \times P \times R)/(P + R)$. The F-measure wights recall and precision equally. F-measure tells how many candidate duplicate record-pairs the classifier finds correctly, as well as how robust it is (it does not miss a significant number of candidate duplicates).

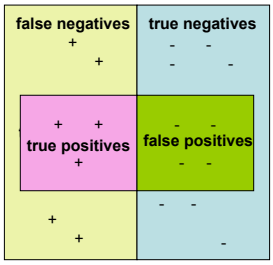


Figure 7.8: Precision and recall

If we want high recall and precision, then, as Figure 7.8 shows, we prefer a blocking technique that generates a small number of candidates for false positives and false negatives.

Our experiments focused mainly on the recall, precision and F-measure of the overall results after classification (and before merging). They indirectly allow for the evaluation of the blocking techniques, as well. Actually, recall measures the effectiveness a blocking technique through non-dismissal of true candidate duplicate

record-pairs. Similarly, a high precision reflects that the blocking technique generates mostly true candidate duplicate record-pairs. Inversely, a low precision shows a large number of non-duplicate record-pairs are also considered, through blocking, as candidate duplicate record-pairs. We can see that it becomes crucial to verify that filtering out record-pairs by a particular blocking technique does not affect the quality of the results obtained after classification.

All the above mentioned measures were computed by training than test methodology, on the basis of the training data. Approximately 70% of the training data was used for training, and the other 30%, for testing.

The SVM parameters have to be tuned to find the optimal separating hyperplane. For example, we have to find C parameter in SVM algorithm which is the degree of correct classification that the algorithm has to meet. The parameters selection affects the results of the classifier. The most common and reliable approach to parameter selection is to do an exhaustive grid search over the parameter space to find the best setting. Multiple rounds of the experiments were performed to keep track of the precision, recall and F-measure while tuning parameters in SVM. At the end, the experiments with the best measures are picked.

The MAS dataset includes 250K authors, 2.5M papers, and a training set. For the authors dataset, the training and test sets contain 3,739 and 2,244 cases (author ids), respectively. Figures 7.7, 7.9 and 7.10 show the comparative performances of *ERBlox* with the three forms of blocking mentioned above, for three different datasets. In all cases, the same SVM technique was applied.

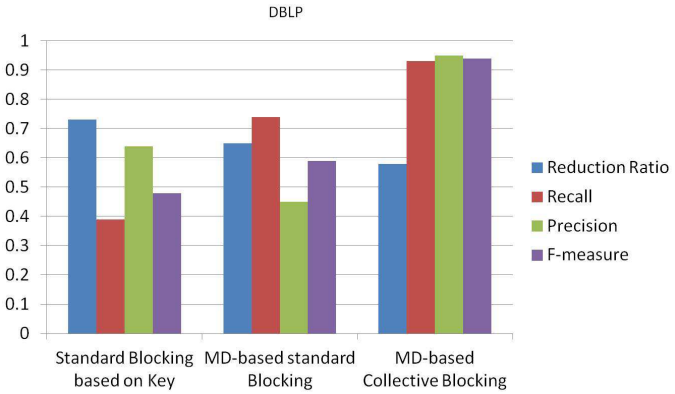


Figure 7.9: The experiments (DBLP)

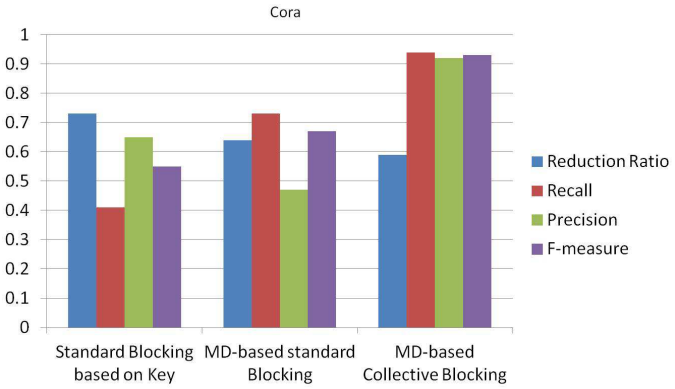


Figure 7.10: The experiments (Cora)

In our concrete application domain, standard blocking based on key-equalities of Paper-records of the MAS dataset used attributes *Title*, *Publication Year*, and *Conference ID*, together, as one blocking key. The MD-version of this key, for MD-based standard blocking and MD-based collective blocking, is the MD (7.8) in Example 7.3.1. According to it, if two records have similar titles, with the same publication year and conference ID, they have the same block numbers. Deciding which attribute equalities become similarities is domain-dependent.

Standard blocking based on key-equalities has higher reduction ratio than MD-based standard blocking, i.e. the former generates fewer candidate duplicate record-pairs. Standard blocking also leads to higher precision than MD-based standard blocking, i.e. we can trust more candidate duplicate record-pairs judgements obtained via standard blocking. However, this standard blocking is very conservative, and has a very low rate of recall, i.e. many of the true candidate duplicate record-pairs are not identified as such. All this makes sense since with standard blocking based on keys we only consider equalities of blocking keys, not similarities.

Precision, recall and F-measure of MD-based collective blocking are higher than those for the two standard blocking techniques. This emphasizes the importance of MDs that supporting collective blocking, and shows that blocking based on string similarity alone fails to capture the semantic interrelationships that naturally hold in the data. On the other side, MD-based collective blocking has lower reduction ratio than standard MD-based blocking, which may lead to better ER results, but may impact computational cost: larger blocks may be produced, and then, more

candidate duplicate record-pairs become inputs for the classifier. In blocking, this is a common tradeoff that needs to be considered [Christen, 2011].

Overall, the quality of MD-based collective blocking dominates standard blocking, both in its key-based and MD-based forms, for the three datasets.

It may be interesting to also compare the MD-based collective blocking against the case where no blocking techniques are applied on records before the classification. However, our goal was to compare MD-based collective blocking with the standard blocking techniques used in literature to conclude that blocking records with additional semantic knowledge, captured by MDs, leads to better result.

7.7 Conclusions

In this chapter, we have shown that matching dependencies, a new class of semantic constraints for data quality and cleaning, can be profitably integrated with traditional ML-methods, in our case for developing classification models for entity resolution. These dependencies play a role not only in their intended goal, that of merging duplicate representations, but also in the record-blocking process that precedes the proper learning task. At that stage they declaratively capture semantic information that can be used to enrich the blocking activity.

The results presented in this chapter have been published in [Bahmani et al., 2015], and in a slightly extended version have been submitted in [Bahmani et al., 2016].

Chapter 8

Related Work

8.1 Related Work

An unsupervised clustering-based approach to collective deduplication is proposed in [Bhattacharya and Getoor, 2007]. While traditional deduplication techniques assume that only similarities between attribute values are available, in relational data the entities are assumed to have additional relational information that can be used to improve the deduplication process. This approach falls in the context of *relational learning* [Getoor and Taskar, 2007]. More precisely, in [Bhattacharya and Getoor, 2007], a relationship graph is built whose nodes are the entities (records), and edges indicate entities which co-occur. The graph supports the propagation of similarity information to related entities. In particular, the similarity between two nodes is calculated as the weighted sum of the attribute-value similarity and their relational similarity (as captured through the graph). Experimental results [Bhattacharya and Getoor, 2007] show that this form of *collective deduplication* outperforms traditional deduplication.

The approach to ER in [Bhattacharya and Getoor, 2007] could be seen as implicitly involving collective blocking, where relationships between entities and similarities between attribute values are used to create the blocks of records. However, this form of collective blocking does not take advantage of a declarative, logic-based semantics. In contrast, a *relationship graph* is used for collective deduplication. In our case, semantic information for this task is captured by matching dependencies. Most importantly, the main focus of our approach to ER is MD-based collective blocking. For this reason, our experiments compare this approach with other blocking techniques. A comparison of our whole approach to ER with other (whole) collective approaches to ER, such as that in [Bhattacharya and Getoor, 2007] has to be left for future research. However, the results of such a comparison may not be very eloquent, because our approach is

based on crucial intermediate techniques, such as the use of SVM for the classification task, which is somehow orthogonal to the blocking approach.

Dedupalog, a declarative approach to collective entity deduplication in the presence of constraints, is proposed in [Arasu et al., 2009]. Constraints are represented by a form of Datalog language. The focus of this work is unsupervised clustering, where constraints are an additional element. Clusters of records make their elements candidates for merging, but blocking *per se* or the actual merging are not main objectives. However, this kind of clustering could be interpreted as a form of blocking. The additional use of constraints could be seen as a form of *collective clustering*. In [Arasu et al., 2009], equality-generating dependencies were used as hard constraints, and clustering-rules as weak constraints.

The success of duplicate detection methods based on supervised machine learning techniques critically depends on being able to provide a set of training pairs, such that the set is representative for the objects to be matched and exhibits the variety and distribution of errors observed in practice [Kopcke and Rahm, 2010]. This is non-trivial because it requires manually searching for various data inconsistencies between any two records spread apart in large data.

A learning-based deduplication system has been proposed in [Sarawagi and Bhamidipaty, 2002] that uses a novel method of interactively discovering suitable training pairs using active learning. Active learning is defined by contrast to the passive model of supervised learning where all the labels for learning are obtained without reference to the learning algorithm. In active learning, the learner interactively chooses which data points to label. The aim of active learning is that interaction can substantially reduce the number of labels required, making solving problems via machine learning more practical.

Our approach can also be seen as a form of relational learning. However, in our case, the semantic relational information (constraints) are, in some sense, implicitly captured through matching dependencies. Their semantics is non-classical (it is chase-based as seen in Section 2.2), and involves directly the blocking or merging processes, as opposed to having higher-level logical constraints "compiled" into them. In our case, the proper learning part of the process, i.e. classification-model learning via

SVM, is supervised,[1] but it does not use any kind of additional relational knowledge. In this regard, it is worth pointing out to quite recent research proposing supervised ML-techniques for classification that involve semantic knowledge in the form of logical formulas in kernels for kernel-based methods (such as SVM) [Diligenti et al., 2012].

Various blocking techniques have been proposed, investigated and applied. See [Baxter et al., 2003, Christen, 2011, Draisbach and Naumann, 2009, Papadakis et al., 2016b, Steorts et al., 2014] for comprehensive surveys and comparative studies. To the best of our knowledge, existing approaches to blocking are inflexible and limited in that they: (a) allow blocking on only single entity types, in isolation from other entity types, or (b) do not take advantage of valuable domain or semantic knowledge. Possible exceptions are [Nin et al., 2007, Rastogi et al., 2011]. Collective blocking in [Nin et al., 2007] disregards blocking keys and creates blocks by considering exclusively the relationships between entities. The relationships correspond to links in a graph connecting entities, and blocks are formed by grouping together entities within neighborhoods with a predefined (path) "diameter". Under this approach, in contrast with ours (cf. Example 1.6.1), relationships are not declarative, and blocking decisions on one entity do not have a direct, explicit impact on blocking decisions to be made on another related entity.

In [Rastogi et al., 2011], similarity of blocking keys and relational relationships are considered for blocking in the context of identification of duplicates (not the merging). However, the semantics of relational relationships (or closeness) between blocking keys and entities is not fully developed.

[1] We refer to [Kopcke and Rahm, 2010] for a discussion on supervised vs. unsupervised approaches.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

There is a lack of fundamental research in data cleaning. We are contributing to fill that vacuum by investigating the logical foundations of some data cleaning activities, in the case of this thesis, of entity resolution based on MDs, a new class of semantic constraints for data quality and cleaning. The semantics of this process, considering matching functions, has been given in terms of intended instances, those that appear after all the MDs have been enforced, through a chase-like procedure in [Bertossi et al., 2013].

In this thesis, we have proposed a general methodology for specifying, by means of logic programs with stable model semantics, the intended clean instances of a dirty instance subject to MDs. The programs provide compact specifications of the classes of clean instances, as their stable models. This approach enables reasoning and clean query answering on top of the virtual results of ER. We have analyzed the programs, obtaining additional complexity results for resolved query answering, query rewriting techniques, and optimizations.

Since clean query answering in the presence of MDs is based on the underlying lattices, the notion of certain answer is not purely set-theoretic. As a consequence, skeptical reasoning from logic programs, which is suitable for a set-theoretic definition of certain answer, had to be adapted for our purposes. In this thesis, we have presented and studied a couple of alternatives. This is an interesting direction that deserves further investigation.

In this work we have provided a fully declarative version of Swoosh's union case [Benjelloun et al., 2009]. It uses some extensions with sets and functional terms of the logic programming paradigm. We have experimented with this approach and the *DLV-Complex* system [Calimeri et al., 2009] that supports such extensions. The

Swoosh approach to ER has been extended with negative rules [Whang et al., 2009a]. We have extended our ASP reconstruction of the Swoosh's union case approach to include them too. This sometimes requires calls/access to external experts. In our approach they will be simulated as a separate program or as calls to external predicates [Eiter et al., 2005].

In this thesis, 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 et al., 2013]. In this thesis we have identified a new class of MDs and initial instances, called blocking, that has good properties in terms of the number of models (a single one, in our case), and computable in polynomial time.

Furthermore, we have presented a uniform methodology to specialize the proposed declarative specification for ER via MDs to obtain residual programs for enforcing the MDs in classes with the unique clean property, containing the three well-behaved cases of MDs with similarity preserving MFs, interaction-free MDs and blocking combinations of MDs and initial instances, which leads to single stable instances. As a result, the residual programs belong to computationally well-behaved extension of plain Datalog.

In this thesis, we have shown that MDs can be profitably integrated with traditional machine learning methods, in our case for developing classification models for entity resolution. These dependencies play a role not only in their intended goal, that of merging duplicate representations, but also in the record-blocking process that precedes the proper learning task. At that stage they declaratively capture semantic information that can be used to enrich the blocking activity.

MDs declaration and enforcement, data cleaning in general, and machine learning can all be integrated using the *LogiQL* language. Actually, all the data extraction, movement and transformation tasks are carried out via L*ogicQL*, a form of extended Datalog supported by the *LogicBlox* platform.

In this regards it is interesting to mention that Datalog has been around since the early 80s, as a declarative and executable rule-based language for relational databases.

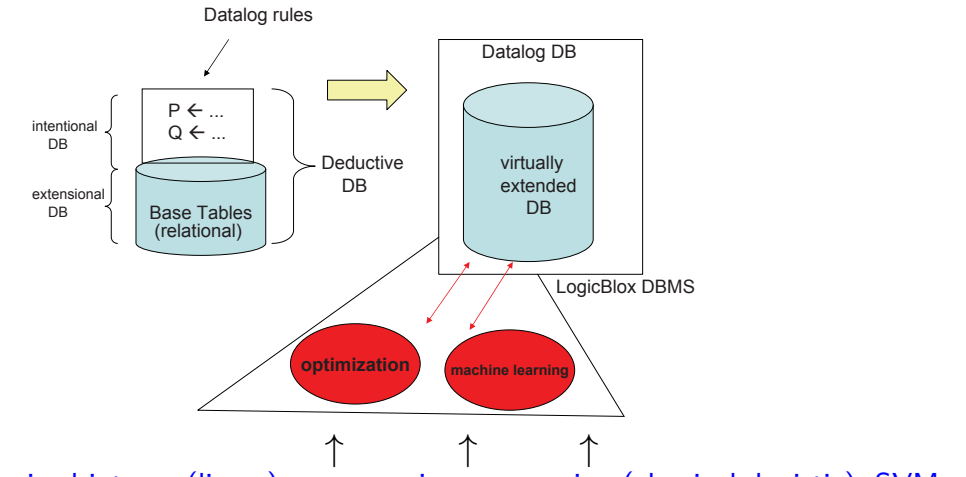


Figure 9.1: *LogicQL* and extended *LogicBlox*

It was used mostly in DB research, until recently. In the last few years Datalog has experienced a revival, and many new applications have been found.

LogicQL, in particular, is being extended in such a way it can smoothly interact with optimization and machine learning algorithms, on top of a single platform. Data for optimization and ML problems stored as "extensions" for a relational database (that is a component of *LogicBlox*), and Datalog predicates. The results of those algorithms can be automatically stored in existing database predicates or or newly defined Datalog predicates, for additional computations or query answering. Currently new ML methods are being implemented as components of the *LogicBlox* system (cf. Figure 9.1).

9.2 Future Work

In this section we point to and present some ideas for interesting extensions to our work.

9.2.1 Optimization of Query Answering via Cleaning Programs

The MF-induced lattice structure on attribute domains [Bertossi et al., 2013] complicates clean query answering via ASP for MD-based ER. We have provided some

first ideas and techniques on how to do clean query answering using the cleaning program in Section 5.2, but they incur in possibly avoidable overhead. The complication resides in the above mentioned lattice-theoretic structure (and minimality) vs. the set-theoretic minimality of ASP models. A more satisfactory solution should come from a deeper investigation of the interaction between these two forms of minimality.

9.2.2 MDs and Database Repairs

The combination and interaction of database repairs, as found in CQA [Bertossi, 2011], and matching dependencies has been initially investigated in [Fan et al., 2011]. The cleaning programs we have presented could be combined with *repair programs*, which are disjunctive programs with stable model semantics that specify -as stable models- the repairs of a database that fails to satisfy a given set of ICs [Arenas et al., 2003, Barcelo et al., 2003, Greco et al., 2003, Caniupan and Bertossi, 2010].

Repair policies, e.g., changes of attribute values and tuple deletions, can also be expressed via repair programs. Repair and cleaning programs could interact in different ways, as it is described in Section 3.5.

We should emphasize that data cleaning and CQA are different problems. For the former, the main goal is to compute a clean instance, as determined by MDs. For the latter, the main goal is obtaining semantically correct query answers. Furthermore, MDs are not (static) ICs. In principle, we could see clean instances as *repairs*, treating MDs similarly to static FDs. However, none of the existing repair semantics captures the matchings based on MDs with MFs.

9.2.3 ER and Virtual Data Integration

Doing entity resolution on a virtual data integration system is a challenging problem. A user may not have access to the data sources, and the matchings can be applied only on-the-fly, at query answering time. Something similar happens with violations of global ICs and database repairs.

Actually, this idea was developed in [Bravo and Bertossi,2003], as follows. First, leaving the ICs aside, the *legal*, intended global instances of a virtual data integration system [Lenzerini, 2002] can be specified as the stable models of an ASP. On top of

it, a repair program, fully combined with the former into a single program, computes the repairs of the global instances. In this way, the consistent answers from the integration system can be computed.

A similar approach could be attempted with ER via global MDs. The cleaning program can be combined with the ASP that specifies the legal instances of the integration system.

9.2.4 ER in Data Exchange

Data exchange is about materializing a target instance using data from a source instance. The source and target instances are related by source-to-target (st) logical mappings that are tuple generating dependencies. Participation of MDs in the data exchange process is worth investigating. More precisely, we may have st-MDs, and possibly target MDs. The former are particularly interesting because the entity resolution process is expected to be applied at data transportation time (much in the spirit of ETL), as opposed to ER applied after a classic data exchange process is applied. This requires producing the model for data exchange under MDs. The interaction of MDs and ICs (or ER and repairs) has been investigated in [Fan et al., 2011], but that is not a data exchange setting.

9.2.5 Improving MD-based Collective Blocking

A most interesting extension to MD-based collective blocking would consider the use of more expressive blocking-MDs than those of the form 7.7. Actually, they could have in their RHSs attributes other than $Bl\#$, the block attributes. As a consequence, blocking-MDs, together with making block numbers identical, would make identical pairs of application-dependent values for some other attributes. Doing this would refine the blocking process itself (modifying the data for the next applications of blocking-MDs), but would also prepare the data for the next task in *ERblox* system, that of classification for entity resolution.

Bibliography

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

[Apt et al., 1988] Apt, K.R., Blair, H.A. and Walker, A. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, (Minker, J., ed.), Morgan Kaufmnn, 1988, pp. 89-148.

[Arasu et al., 2009] Arasu, A., Ré, Ch. and Suciu, D. Large-Scale Deduplication with Constraints using Dedupalog. Proc. of the 25th International Conference on Data Engineering (ICDE), 2009, pp. 952-963.

[Aref et al., 2015] 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. of the 35th ACM International Conference on Management of Data (SIGMOD), 2015, pp. 1371-1382.

[Arenas et al., 1999] Arenas, M., Bertossi, L. and Chomicki, J. Consistent Query Answers in Inconsistent Databases. Proc. of the 19th ACM International Conference on Principles of Database Systems (PODS), 1999, pp. 68-79.

[Arenas et al., 2003] Arenas, M., Bertossi, L. and Chomicki, J. Answer Sets for Consistent Query Answering in Inconsistent Databases. *Theory and Practice of Logic Programming*, 2003, 3(4/5), pp. 393-424.

[Arenas et al., 2010] Arenas, M., Barcelo, P., Libkin, L. and Murlak, F. *Relational and XML Data Exchange*. Synthesis Lectures on Data Management, Morgan & Claypool Publishers, 2010.

[Bahmani et al., 2012] Bahmani, Z., Bertossi, L., Kolahi, S. and Lakshmanan, L. Declarative Entity Resolution via Matching Dependencies and Answer Set Programs. Proc. of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR), 2012, pp. 380-390.

[Bahmani et al., 2015] Bahmani, Z., Bertossi, L. and Vasiloglou, N. ERBlox: Combining Matching Dependencies with Machine Learning for Entity Resolution. Proc. of the 9th International Conference on Scalable Uncertainty Management (SUM), 2015, Springer LNAI 9310, pp. 399-414.

[Bahmani et al., 2016] Bahmani, Z., Bertossi, L. and Vasiloglou, N. ERBlox: Combining Matching Dependencies with Machine Learning for Entity Resolution. Journal submission, extended version of [Bahmani et al., 2015], 2016.

[Bahmani et al., 2016] Bahmani, Z., Bertossi, L., Kolahi, S. and Lakshmanan, L. Declarative Entity Resolution via Matching Dependencies and Answer Set Programs. Extended, journal version of [Bahmani et al., 2012]. In preparation, 2016.

[Bahmani et al., 2017] Bahmani, Z. and Bertossi, L. Enforcing Relational Matching Dependencies with Datalog for Entity Resolution. To appear in Proc. the International Florida Artificial Intelligence Research Society Conference (FLAIRS), 2017.

[Barcelo et al., 2003] Barcelo, P., Bertossi, L. and Bravo, L. Characterizing and Computing Semantically Correct Answers from Databases with Annotated Logic and Answer Sets. In *Semantics in Databases*, 2003, Springer LNCS 2582, pp. 7-33.

[Barcelo, 2009] Barcelo, P. Logical Foundations of Relational Data Exchange. *SIGMOD Record*, 2009, 38(1):49-58.

[Batini and Scannapieco, 2006] Batini, C. and Scannapieco, M. *Data Quality: Concepts, Methodologies and Techniques*. Springer, 2006.

[Baudat and Anouar, 2000] Baudat G. and Anouar, F. Generalized Discriminant Analysis using a Kernel Approach. *Neural Computation*, 2000, 12(3):2385-2404.

[Baxter et al., 2003] Baxter, R., Christen, P. and Churches, T. A Comparison of Fast Blocking Methods for Record Linkage. Proc. of the ACM SIGKDD Workshop on Data Cleaning, Record Linkage, and Object Identification, 2003, pp. 234-256.

[Benedikt et al., 1998] Benedikt, M., Dong, G., Libkin, L. and Wong, L. Relational Expressive Power of Constraint Query Languages. *Journal of the ACM*, 1998, 45(1):1-34.

[Ben-Eliyahu and Dechter, 1994] Ben-Eliyahu, R. and Dechter, R. Propositional Semantics for Disjunctive Logic Programs. *Annals of Mathematics in Artificial Intelligence*, 1994, 12:53-87.

[Benjelloun et al., 2009] Benjelloun, O., Garcia-Molina, H., Menestrina, D., Su, Q., EuijongWhang, S. and Widom, J. Swoosh: a Generic Approach to Entity Eesolution. *VLDB Journal*, 2009, 18(1):255-276.

[Bertossi and Bravo, 2004] Bertossi, L. and Bravo, L. Consistent Query Qnswers in Virtual Data Integration Systems. *Inconsistency Tolerance*, Springer LNCS 3300, 2004, pp. 42-83.

[Bertossi, 2006] Bertossi, L. Consistent Query Answering in Databases. *ACM Sigmod Record*, 2006, 35(2):68-76.

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

[Bertossi et al., 2013] Bertossi, L., Kolahi, S. and Lakshmanan, L. Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. *Th. Comp. Systems*, 2013, 52(3):441-482.

[Bertossi and Bravo, 2013] 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-Verlag, Berlin Heidelberg, 2013, pp. 181-212.

[Bhattacharya and Getoor, 2007] Bhattacharya, I., Getoor, L. Collective Entity Resolution in Relational Data. *ACM Transaction Knowledge Discovery Data*, 2007, 1(1):15-51.

[Bilenko and Mooney, 2003a] Bilenko, M., Mooney, R.J. Adaptive Duplicate Detection using Learnable String Similarity Measures. Proc. of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2003, pp. 39-48.

[Bilenko and Mooney, 2003b] Bilenko, M., Mooney, R.J. On Evaluation and Training-set Construction for Duplicate Detection. Proc. of the ACM SIGKDD Workshop on Data Cleaning, Record Linkage and Object Consolidation, 2003, pp. 712.

[Bizer et al., 2009] Bizer, C., Heath, T. and Berners-Lee, T. Linked data - the Story so Far. *Int. J. Semantic Web Inf. Syst.*, 2009, 5(3):1-22.

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

[Bravo and Bertossi,2003] Bravo, L. and Bertossi, L. Logic Programs for Consistently Querying Data Integration Systems. Proc. of the 18th Internation Joint Conference on Artificail Inteligence(IJCAI), 2003, pp. 10-15.

[Brewka et al., 2011] Brewka, G., Eiter, Th. and Truszczynski, M. Answer Set Programming at a Glance. *Commun. ACM*, 2011, 54(12):92-103.

[Burdick et al., 2015] Burdick, D., Fagin, R., Kolaitis, P.G., Popa, L. and Tan, W. C. A Declarative Framework for Linking Entities. Proc. of the 18th Internation Confernce on Database Theory (ICDT), 2015, pp. 25-43.

[Calimeri et al., 2008] Calimeri, F. Cozza, S. Ianni, G. and Leone, N. Computable Functions in ASP: Theory and Implementation. Proc. of the 24th Internation Confernce on Logic Programming (ICLP), Springer LNCS 5366, 2008, pp. 407-424.

[Calimeri et al., 2009] Calimeri, F., Cozza, S., Ianni, G. and Leone, N. An ASP System with Functions, Lists, and Sets. Proc. of the 7th International Conference on

Logic Programming and Non-monotonic Reasoning (LPNMR), Springer LNCS 5753, 2009, pp. 483-489.

[Caniupan and Bertossi, 2010] Caniupan, M. and Bertossi, L. The Consistency Extractor System: Answer Set Programs for Consistent Query Answering in Databases. *Data & Knowledge Engineering*, 2010, 69(6): pp. 545-572.

[Ceri et al., 1989] Ceri, S., Gottlob, G. and Tanca, L. *Logic Programming and Databases*. Springer, 1989.

[Chandel et al., 2007] Chandel, A., Hassanzadeh,O., Koudas,N., Sadoghi, M. and Srivastava, D. Benchmarking Declarative Approximate Selection Predicates. Proc. of the ACM International Conference on Management of Data (SIGMOD), 2007, pp. 155-165.

[Chandra and Harel, 1985] Chandra, A. and Harel, D. Horn Clauses and Generalizations. *Journal of Logic Programming*, 1985, 2(1):1-15.

[Chomicki and Marcinkowski, 2005] Chomicki, J. and Marcinkowski, J. Minimal-change Integrity Maintenance using Tuple Deletions. *Information and Computation*, 2005, 197(1/2):90-121.

[Chomicki, 2007] Chomicki, J. Consistent Query Answering: Five Easy Pieces. Proc. of the 10th Internation Confernce on Database Theory (ICDT), Springer LNCS 4353, 2007, pp. 1-17.

[Christen and Goiser, 2010] Christen, P. and Goiser, K. Quality and Complexity Measures for Data Linkage and Deduplication. In *Quality Measures in Data Mining*, Guillet, F. and Hamilton, H. (eds.), Springer, 2010, pp. 127-151.

[Christen, 2008] Christen, P. Automatic Record Linkage using Seeded Nearest Neighbour and Support Vector Machine Classification. Proc. of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2008, pp. 151-159.

[Christen, 2011] Christen, P. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE Transactions in Knowledge and Data Engineering*, 2011, 19(1):1-16.

[Christen, 2012] Christen, P. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer, 2012.

[Cohen et al., 2003] Cohen, W. C., Ravikumar, P. D. and Fienberg, S.E. A Comparison of String Metrics for Matching Names and Records. Proc. of the ACM SIGKDD Workshop on Data Cleaning, Record Linkage and Object Consolidation, 2003, pp. 13-18.

[Cover and Hart, 1967] Cover, T.M. and Hart, P.E. Nearest Neighbor Pattern Classification. *IEEE Transactions on Information Theory*, 1967, 13(1): 21-27.

[Cristianini and Shawe-Taylor, 2000] Cristianini, N. and Shawe-Taylor, J. *An Introduction to Support Vector Machines*. Cambridge U. Press, 2000.

[Dash and Liu, 1997] Dash, M. and Liu, H. Feature Selection for Classification. *Intelligent Data Analysis*, 1997, 1(1-4):131-156.

[Dantsin et al., 2001] Dantsin, E., Eiter, T., Gottlob, G. and Voronkov, A. Complexity and Expressive Power of Logic Programming. *ACM Comput. Surv.*, 2001, 33(3):374-425.

[Deutsch and Tannen, 2003] Deutsch, A. and Tannen,V. Reformulation of XML Queries and Constraints. Proc. of the 6th Internation Confernce on Database Theory (ICDT), 2003, pp. 225-241.

[Dey et al., 2010] Dey, D., Mookerjee, V. and Liu, D. Efficient Techniques for Online Record Linkage. *IEEE Transactions on Knowledge and Data Engineering*, 2010, 23(3):373-387.

[Diligenti et al., 2012] Diligenti, M, Gori, M., Maggini, M. and Rigutini, L. Bridging Logic and Kernel Machines. *Machine Learning*, 2012, 86(1):57-88.

[Dong et al., 2012] Dong, X., Halevy, A.Y. and Madhavan, J. Reference Reconciliation in Complex Information Spaces. Proc. of the 25th ACM International Conference on Management of Data (SIGMOD), 2005, pp. 85-96.

[Draisbach and Naumann, 2009] Draisbach, U. and Naumann, F. A Comparison and Generalization of Blocking and Windowing Algorithms for Duplicate Detection. Proc. of VLDB Workshop on Quality in Databases (QDB), 2009, 51-56.

[Eiter and Gottlob, 1995] Eiter, T. and Gottlob, G. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Annals of Mathematics and Artificial Intelligence*, 1995, 15(3-4):289-323.

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

[Eiter et al., 2005] Eiter, T., Ianni, G., Schindlauer, R. and Tompits, V. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. Proc. of the 20th Internation Joint Conference on Artificial Intelligence (IJCAI), 2005, pp. 90-96.

[Eiter et al., 2008] Eiter, T., Fink, M., Greco, G. and Lembo, D. Repair Localization for Query Answering from Inconsistent Databases. *ACM Trans. Database Syst.*, 2008, 33(2).

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

[Faber and Woltran, 2011] Faber, W. and Woltran, S. Manifold Answer-Set Programs and Their Applications. In *Gelfond Festschrift*, Springer LNAI 6565, 2011, pp. 44-63.

[Fagin et al., 2005a] Fagin,R., Kolaitis, P.G., Miller, R.J. and Popa, L. Data Exchange: Semantics and Query Answering. *Theory of Computer Science*, 2005, 336(1):89-124.

[Fagin et al., 2005b] Fagin,R., Kolaitis, P.G., Miller, R.J. and Popa, L. Data Exchange: Getting to the Core. *ACM TODS*, 2005, 30(1):174-210.

[Fellegi and Sunter, 1969] Fellegi, I.P. and Sunter, A.B. A Theory for Record Linkage. *Journal of the American Statistical Society*, 1969, 64:11831210.

[Fan, 2008] Fan, W. Dependencies Revisited for Improving Data Quality. Proc. of the 27th ACM International Conference on Principles of Database Systems (PODS), 2008, pp. 159-170.

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

[Fan et al., 2011] Fan, W., Li, J., Ma, S., Tang, N. and Yu, W. Interaction between Record Matching and Data Repairing. Proc. of the 31th ACM International Conference on Management of Data (SIGMOD), 2011, pp. 469-480.

[Flach, 2014] Flach, P. *Machine Learning*. Cambridge University Press, 2014.

[Franconi et al. 2001] Franconi, E., Laureti Palma, A., Leone, N., Perri, S. and Scarcello, F. Census Data Repair: a Challenging Application of Disjunctive Logic Programming. Proc. of the 8th Inetrnational Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR), Springer LNCS 2250, 2001, pp. 561-578.

[Fuxman and Miller, 2007] Fuxman, A. and Miller, R. First-order Query Rewriting for Inconsistent Databases. *Computer and System Sciences*, 2007, 73(4):610-635.

[Gardezi et al., 2012] Gardezi, J., Bertossi, L. and Kiringa, I. Matching Dependencies: Semantics and Query Answering. *Frontiers of Computer Science* (Springer journal), 2012, 6(3):278-292.

[Gardezi and Bertossi, 2012] Gardezi, J. and Bertossi, L. Query Rewriting using Datalog for Duplicate Resolution. Proc. of the 2nd Workshop on the Resurgence of Datalog in Academia and Industry, Springer LNCS 7494, 2012, pp. 134-145.

[Gelfond and Lifschitz, 1990] Gelfond, G. and Lifschitz, V. Logic Programs with Classical Negation. Proc. of the 6th Internationl Conference on Logic Programming (ICLP), 1990, pp. 579-597.

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

[Gelfond and Leone, 2002] Gelfond, M. and Leone, N. Logic Programming and Knowledge Representation - The A-Prolog Perspective. *Artificial Intelligence*, 2002, 138(1-2):3-38.

[Geerts et al., 2013] Geerts, F., Mecca, G., Papotti, P. and Santoro, D. The LLUNATIC Data-Cleaning Framework. *PVLDB*, 2013, 6(9):625-636.

[Geerts et al., 2014] Geerts, F., Mecca, G., Papotti, P. and Santoro, D. Mapping and Cleaning. Proc. of the 30th International Conference on Data Engineering (ICDE), 2014, pp. 232-243.

[Getoor and Taskar, 2007] Getoor, L. and Taskar, B. (eds.). *Introduction to Statistical Relational Learning*, MIT Press, 2007.

[Greco et al., 2003] Greco, G., Greco, S. and Zumpano, E. A Logical Framework for Querying and Repairing Inconsistent Databases. *IEEE Transactions on Knowledge and Data Engineering*, 2003, 15(6):1389-1408.

[Green et al., 2012] Green, T.J., Aref, M. and Karvounarakis, G. LogicBlox, Platform and Language: A Tutorial. Proc. Datalog 2.0, 2012, LNCS 7494, pp. 1-8.

[Halpin and Rugaber, 2015] Halpin, T. and Rugaber, S. *LogiQL: A Query Language for Smart Databases*. CRC Press, 2015.

[Herzog, 2007] Herzog, T.N., Scheuren,F.J. and Winkler, W.E.

Data Quality and Record Linkage Techniques. Springer, 2007.

[Horwitz and Teitelbaum, 1986] Horwitz, S. and Teitelbaum, T. Generating Editing Envornmnets based on Attributes and Relations. *ACM Transactions on Programming Languages and Systems*, 1986, 8(4):577-608.

[Imielinski and Lipski, 1984] Imielinski, T. and Lipski, W. Jr. Incomplete Information in Relational Databases. *J. ACM*, 1984, 31(4):761-791.

[Ioannou et al., 2001] Ioannou, E., Nejdl, W., Niederee, C. and Velegrakis, Y. On-the-fly Entity-aware Query Processing in the Presence of Linkage. *PVLDB*, 2001, 3(1):138-150.

[Jaro, 1976] Jaro, M.A. UNIMATCH: A Record Linkage System: User's Manual, Technical Report, U.S. Bureau of the Census, 1976.

[Jaro, 1989] Jaro, M. Advances in Record Linkage Methodology as Applied to Matching the 1985 Census of Tampa. *Journal of the American Statistical Society*, 1989, 84(406):414-420.

[Jaro, 1995] Jaro, M.A. Probabilistic Linkage of Large Public Health Data Files. *Journal of Statistics in Medicine*, 1995, 14(1):491-498.

[Kopcke and Rahm, 2010] Kopcke, H. and Rahm, E. Frameworks for Entity Matching: a Comparison. *Journal of Data and Knowledge Engineering*, 2010, 69(2):197-210.

[Koudas et al., 2006] Koudas, N., Sarawagi, S. and Srivastava, D. Record linkage: Similarity Measures and Algorithms. Proc. of the 25th ACM International Conference on Management of Data (SIGMOD), 2006, pp. 802-803.

[Lenzerini, 2002] Lenzerini, M. Data Integration: a Theoretical Perspective. Proc. of the 22th ACM International Conference on Principles of Database Systems (PODS), 2002, pp. 233-246.

[Leone et al., 2006] 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.

[Lloyd, 1987] Lloyd, J. *Foundations of Logic Programming*. Springer, 1987, 2nd. edition.

[Manning and Schutze, 2008] Manning, D. and Schutze, H. *Introduction to Information Retrival*. Cambridge University Press, 2008.

[LOGICBLOX, 2012] LOGICBLOX. Machine Learning Methods in LogicBlox. Release. LogicBlox Inc., September 27, 2012.

[Motro and Anokhin, 2006] Motro, A. and Anokhin, P. Fusionplex: Resolution of Data Inconsistencies in the Integration of Heterogeneous Information Sources. *Information Fusion*, 2006, 7(2): 176-196.

[Navarro, 2001] Navarro, G. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 2001, 33(1):31-88.

[Nin et al., 2007] Nin, J., Muntes, V., Martinez-Bazan, N. and Larriba, J. On the Use of Semantic Blocking Techniques for Data Cleansing and Integration. Proc. of the 11th International Database Engineering and Applications Symposium (IDEAS), 2007, pp. 190-198.

[Papadakis et al., 2014] Papadakis, G., Koutrika, G., Palpanas, T. and Nejdl, W. Meta-blocking: Taking Entity Resolution to the Next Level. *IEEE Trans. Knowl. Data Eng.*, 2014, 26(8):1946-1960.

[Papadakis et al., 2016a] Papadakis, G., Papastefanatos, G., Palpanas, T. and Koubarakis, M. Scaling Entity Resolution to Large, Heterogeneous Data with Enhanced Meta-blocking. Proc. of the 19th International Conference on Extending Database Technology (EDBT), 2016, pp. 221-232.

[Papadakis et al., 2016b] Papadakis, G., Svirsky, J., Gal, A. and Palpanas, T. Comparative Analysis of Approximate Blocking Techniques for Entity Resolution. *PVLDB*, 2016, 9(9): 684-695.

[Popa et al., 2002] Popa, L., Velegrakis, Y., Miller, R.J, Hernandez, M.A. and Fagin, R. Translating Web Data. Proc. of the 28th International Conference on Very Large Data Bases (VLDB), 2002, pp. 598-609.

[Rastogi et al., 2011] Rastogi, V., Dalvi, N.N. and Garofalakis, M.N. Large-scale Collective Entity Matching. *PVLDB*, 2011, 4(4):208-218.

[Roy et al., 2013] Roy, S. B. , De Cock, M., Mandava, V., Savanna, S., Dalessandro, B., Perlich, C., Cukierski, W. and Hamner. B. The Microsoft Academic Search Dataset and KDD Cup 2013. Proc. of the KDD Cup Workshop, 2013, pp. 34-40.

[Sais et al., 2007] Sais, F., Pernelle, N. and Rousset MC. L2R: A logical Method for Reference Reconciliation. Proc. of the 4th AAAI Conference on Artificial Intelligence, 2007, pp. 329-334.

[Salton and Buckley, 1988] Salton, G. and Buckley, C. Term-weighting Approaches in Automatic Text Retrieval. *Information Processing and Management*, 1988, 24(5):513-523.

[Sarawagi and Bhamidipaty, 2002] Sarawagi, S. and Bhamidipaty, A. Interactive Deduplication Using Active Learning. Proc. of the 8th ACM Conferece on Knowledge Discovery and Data Mining, 2002, pp. 253-343.

[Singla and Domingos, 2006] Singla, P. and Domingos, P. Entity Resolution with Markov Logic. Proc. of the 10th International Conference on Data Mining (ICDM), 2006, pp. 572-582.

[Sismanis et al., 2009] Sismanis,Y., Wang, L., Fuxman, A., Haas, P.G. and Reinwald, B. Resolution-Aware Query Answering for Business Intelligence. Proc. of the 25th International Conference on Data Engineering (ICDE), 2009, pp. 976-987.

[Song abd Chen, 2009] Song, S. and Chen, L. Efficient Discovery of Similarity Constraints for Matching Dependencies, *Data Knowledge Engineering*, 2009, 14(1):313-323.

[Steorts et al., 2014] Steorts, R., Ventura, S., Sadinle, M. and Fienberg, S. A Comparison of Blocking Methods for Record Linkage. Proc. of the 5th International Conference on Privacy in Statistical Databases (PSD), Springer LNCS 8744, 2014, pp. 253-268.

[Tang et al., 2015] Tang, J., Alelyani,S. and Liu, H. Feature Selection for Classification: A Review. In *Data Classification: Algorithms and Applications*, CRC Press, 2015, pp. 37-64.

[Torvik and Smalheiser, 2009] Torvik, I. and Smalheiser, R. Author Name Disambiguation in Medline. *ACM Transactions on Knowledge Discovery from Data*, 2009, 11(3):1-29.

[Ullman, 1989] Ullman, J.D. *Principles of Database and Knowledge-Base Systems*. volume II edition, Computer Science Press, 1989.

[Vapnik, 2009] Vapnik, V.N. *The Nature of Statistical Learning Theory*. 2nd ed., Springer, 2000.

[Vassiliadis, 2009] Vassiliadis, P. A Survey of Extract-Transform-Load Technology. *International Journal of Data Warehousing & Mining*, 2009, 5(3):1-27.

[Whang et al., 2009a] Whang, S.E., Benjelloun, O. and Garcia-Molina, H. Generic Entity Resolution with Negative Rules. *VLDB Journal*, 2009, 18(6):1261-1277.

[Whang et al., 2009b] Whang, S., Menestrina, D., Koutrika, G., Theobald, M. and Garcia-Molina, H. Entity Resolution with Iterative Blocking. Proc. of the 29th ACM International Conference on Manage- ment of Data (SIGMOD), 2009, pp. 219-232.

[Winkler, 1999] Winkler, W. E. The State of Record Linkage and Current Research Problems. Technical Report, U.S. Census Bureau, 1999.

[Winkler, 2004] Winkler, W.E. Methods for Evaluating and Creating Data Quality. *Information Systems*, 2004, 29(7):531-550.

[Winkler, 2006] Winkler, W.E. Overview of Record Linkage and Current Research Directions. US Bureau of the Census, Technical Report, 2006.

[Zhang, 2000] Zhang, G. Neural Networks for Classification: a Survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 2000, 30(4): 451-462.

Appendix A

DLV Codes

Example A.0.1 (ex. 5.1.1 cont.) The DLV codes are:

```
%Extensional database
r(t1,a1,b1).
r(t2,a2,b2).
r(t3,a3,b3).

%Domain of database
dom(a1). dom(a2). dom(a3). dom(b1).
dom(b2). dom(b3). dom(b12). dom(b23).
dom(b123).

%Existing similarities
att(a1,a2). att(b2,b3).

%Matching functions
ma(b1,b2,b12). ma(b2,b3,b23).
ma(b1,b23,b123).

%Rules related to match functions and similarity relations
attmatch(X,Y)  :- attmatch(Y,X).
ma(Y,X,Z)      :- ma(X,Y,Z).
attmatch(X,X)  :- dom(X).
ma(X,X,X)      :- dom(X).
ma(X,S,V)      :- ma(X,Y,Z), ma(W,Z,V), ma(Y,W,S).
ma(Z,W,V)      :- ma(Y,W,S), ma(X,S,V), ma(X,Y,Z).
```

```
%Rules for obtaining clean solutions

match(T1,X1,Y1,T2,X2,Y2) v notmatch(T1,X1,Y1,T2,X2,Y2) :-
            r(T1,X1,Y1), r(T2,X2,Y2), att(X1,X2), Y1!=Y2, T1!=T2.

:- notmatch(T1,X1,Y1,T2,X2,Y2), not old(T1,X1,Y1), not old(T2,X2,Y2).

old(T1,X1,Y1):- r(T1,X1,Y1), r(T1,X1,Y2), ma(Y1,Y2,Y2), Y2!=Y1.

match(T2,X2,Y2,T1,X1,Y1) :- match(T1,X1,Y1,T2,X2,Y2).

r(T1,X1,Y3) :- match(T1,X1,Y1,T2,X2,Y2), ma(Y1,Y2,Y3).

match(T1,X1,Y1,T2,X2,Y2) v notmatch(T1,X1,Y1,T2,X2,Y2) :-
            r(T1,X1,Y1), r(T2,X2,Y2), att(Y1,Y2), Y1!=Y2,T1!=T2.

:- notmatch(T1,X1,Y1,T2,X2,Y2), not old(T1,X1,Y1), not old(T2,X2,Y2).

old(T1,X1,Y1)  :- r(T1,X1,Y1), r(T1,X1,Y2), ma(Y1,Y2,Y2), Y2!=Y1.

prec(T1,X1,Y1,T2,X2,Y2,T1,X1,Y3,T4,X4,Y4) :- match(T1,X1,Y1,T2,X2,Y2),
            match(T1,X1,Y3,T4,X4,Y4), ma(Y1,Y3,Y3), Y3!=Y1.

prec(T1,X1,Y1,T2,X2,Y2,T1,X1,Y1,T4,X4,Y4) :- match(T1,X1,Y1,T2,X2,Y2),
            match(T1,X1,Y1,T4,X4,Y4), ma(Y1,Y4,Y3),Y1!=Y3.

prec(T1,X1,Y1,T2,X2,Y2,T1,X1,Y1,T2,X2,Y2) :- match(T1,X1,Y1,T2,X2,Y2).

:- prec(T1,X1,Y1,T2,X2,Y2,T1,X1,Y3,T4,X4,Y4),
```

```
   prec(T1,X1,Y3,T4,X4,Y4,T1,X1,Y1,T2,X2,Y2), Y1 !=Y3.

:- prec(T1,X1,Y1,T2,X2,Y2,T1,X1,Y3,T4,X4,Y4),
   prec(T1,X1,Y3,T4,X4,Y4,T1,X1,Y1,T2,X2,Y2), T2!=T4.

:- prec(T1,X1,Y1,T2,X2,Y2,T1,X1,Y3,T4,X4,Y4),
   prec(T1,X1,Y3,T4,X4,Y4,T1,X1,Y1,T2,X2,Y2), X2 != X4.

:- prec(T1,X1,Y1,T2,X2,Y2,T1,X1,Y3,T4,X4,Y4),
   prec(T1,X1,Y3,T4,X4,Y4,T1,X1,Y1,T2,X2,Y2), Y2 !=Y4.

:- prec(T1,X1,Y1,T2,X2,Y2,T1,X1,Y3,T4,X4,Y4),
   prec(T1,X1,Y3,T4,X4,Y4,T1,X1,Y5,T6,X6,Y6),
   not prec(T1,X1,Y1,T2,X2,Y2,T1,X1,Y5,T6,X6,Y6).
match(T2,X2,Y2,T1,X1,Y1) :- match(T1,X1,Y1,T2,X2,Y2).

r(T1,X1,Y3) :- match(T1,X1,Y1,T2,X2,Y2), ma(Y1,Y2,Y3).

match(T1,X1,Y1,T2,X2,Y2) v notmatch(T1,X1,Y1,T2,X2,Y2) :-
            r(T1,X1,Y1), r(T2,X2,Y2), att(Y1,Y2), Y1!=Y2,T1!=T2.

:- notmatch(T1,X1,Y1,T2,X2,Y2), not old(T1,X1,Y1), not old(T2,X2,Y2).

old(T1,X1,Y1)  :- r(T1,X1,Y1), r(T1,X1,Y2), ma(Y1,Y2,Y2), Y2!=Y1.

prec(T1,X1,Y1,T2,X2,Y2,T1,X1,Y3,T4,X4,Y4) :- match(T1,X1,Y1,T2,X2,Y2),
            match(T1,X1,Y3,T4,X4,Y4), ma(Y1,Y3,Y3), Y3!=Y1.


:- prec(T1,X1,Y1,T2,X2,Y2,T1,X1,Y3,T4,X4,Y4),
   prec(T1,X1,Y3,T4,X4,Y4,T1,X1,Y1,T2,X2,Y2), Y2 !=Y4.
```

```
:- prec(T1,X1,Y1,T2,X2,Y2,T1,X1,Y3,T4,X4,Y4),
   prec(T1,X1,Y3,T4,X4,Y4,T1,X1,Y5,T6,X6,Y6),
   not prec(T1,X1,Y1,T2,X2,Y2,T1,X1,Y5,T6,X6,Y6).
match(T2,X2,Y2,T1,X1,Y1) :- match(T1,X1,Y1,T2,X2,Y2).

rc(T1,X1,Y1) :- r(T1,X1,Y1), not old(T1,X1,Y1).
```

□

Example A.0.2 (ex. 5.4.1 cont.) Since in this example we exploit built-in predicates and functions from library of the system, the logic program must contain, in the preamble, a line that tells the system to include the library itself. Other used predicates have the same meanings as mentioned before in the example.

```
#include<ListAndSet>
%Extensional database
r({a1},{b1}). r({a2},{b2}). r({a3},{b3}).

%Existing similarities
match(a1,a2). match(a2,a3). match(a3,a2). match(a2,a1).

%Rules for obtaining resolution instance
r(As3,Bs3) :-  #union(As1,As2,As3), #union(Bs1,Bs2,Bs3), r(As1,Bs1),
               r(As2,Bs2), #member(S1,As1), #member(S2,As2),
               match(S1,S2).
r(As3,Bs3) :-  #union(As1,As2,As3), #union(Bs1,Bs2,Bs3), r(As1,Bs1),
               r(As2,Bs2), #member(S1,As1), #member(S2,As2),
               match(S1,S2).

dominated(As1,Bs1) :- r(As1,Bs1), r(As2,Bs2), #union(As1,As2,As2),
                      As2!= As1, #union(Bs1,Bs2,Bs2).
```

```
dominated(As1,Bs1) :- r(As1,Bs1), r(As2,Bs2), #union(As1,As2,As2),
                      Bs2!= Bs1, #union(Bs1,Bs2,Bs2).

er(As1,Bs1) :- r(As1,Bs1), not dominated(As1,Bs1).
```

□