Generic and Declarative Approaches to Data Quality Management

Leopoldo Bertossi and **Loreto Bravo**

Abstract Data quality assessment and data cleaning tasks have traditionally been addressed through procedural solutions. Most of the time, those solutions have been applicable to specific problems and domains. In the last few years we have seen the emergence of more generic solutions; and also of declarative and rule-based specifications of the intended solutions of data cleaning processes. In this chapter we review some recent developments.

1 Introduction

Data quality assessment and data cleaning have been mostly *ad hoc*, rigid, vertical, and application-dependent activities. There is a need for more general methodologies and solutions. Most of the existing approaches have been also procedural, provided in terms of specific mechanisms. However, their semantics and scope of applicability are not fully understood. Declarative approaches could be attempted in this regard. They specify, usually by means of a logic-based formalism, what is the intended result of a data cleaning process. The semantics of the specification tells us what the result should look like, if there are alternative solutions, and what are the conclusions that can be derived from the process. They also allows us, in principle, to better understand the range of applicability and complexity of the declaratively specified cleaning mechanism.

Considering the large body of literature accumulated in the areas of data quality assessment and cleaning, it is safe to say that there is a lack of fundamental research around the activities of data quality assessment and data cleaning. Fortunately, things are starting to change. Important impulses in this direction have come from two sources.

One of them is related to a new look at classic integrity constraints (ICs). Since they can be (and are) violated in many applications, they can still be used as desirable properties that could be enforced if necessary, cleaning the database from

Leopoldo Bertossi
Carleton University, Ottawa, Canada. e-mail: bertossi@scs.carleton.ca

Loreto Bravo
Universidad de Concepción, Concepción, Chile. e-mail: lbravo@udec.cl

semantic anomalies. However, this enforcement could be only partial, full, or even virtual. It could also be imposed at query answering time, seeing ICs more like constraints on query answers than on database states (cf. Section 3).

The other source is related to the introduction of newer classes of ICs that are intended to capture data quality issues or conditions, and are intended to be used to directly support data cleaning processes. We could call them *data quality constraints*. They have been proposed and investigated in the last few years, and provide generic languages for expressing quality concerns [42]. They may be a suitable basis for declaratively specifying adaptive and generic data quality assessment and cleaning mechanisms (cf. Sections 4.1 and 5.2).

Generic methods for data cleaning may be proposed for possibly solving a single but still general problem in the area, e.g. entity resolution, data editing (i.e. changes of data values in records), incompleteness of data, etc. They are abstract and parameterized approaches or mechanisms, that can be applied to different specific instances of the data cleaning problem at hand, by instantiating parameters, methods and modules as required by the specificity of the problem and application domain. For example, a general entity resolution algorithm may rely on matching functions to do the actual merging of records. In the abstract formulation they are left rather open, except for some general requirements they have to satisfy. However, when the algorithm is applied to a specific domain, those matching functions become domain dependent, as expected.

Earlier generic methods in data cleaning were proposed in [52]. Actually, this is a framework for entity resolution whose aim is to separate the logic of data transformations from their actual implementations. Data transformations, e.g. matching and merging of records, are specified in an extension of SQL. The methodology was implemented and tested in the *AJAX* system [52], and inspired newer developments in the area, e.g. the *Swoosh* generic approach to entity resolution [7] (cf. Section 5.1). Other generic approaches are described in some articles in [39].

Declarative methods in data cleaning, again possibly for a specific and general problem, are expected to be based on a formal, say logic-based, specification of the intended results of the cleaning process. These results are usually represented by (or through) the models of the specification, which requires defining a precise semantics for the formalism at hand.

This chapter gives a survey of some recent research on generic and declarative methods in data cleaning. Considering that this is a vast area of research, we concentrate in more detail only on some problems. In Section 2 we review basics of integrity constraints in relational databases. In Section 3 we concentrate on *database repairs* with respect to classic semantic constraints. In Section 4, we present *conditional dependencies* for data quality assessment and data cleaning. In Section 5, we demonstrate the value of declarative approaches to entity resolution (or deduplication), with emphasis on the use of *matching dependencies*. In Section 6, we make some final remarks and briefly mention some other problems and approaches as related to the previous sections. For more on data quality and data cleaning we refer to the general reference [6], and to the earlier general survey [68].

2 Classic Integrity Constraints

Integrity constraints (ICs) are used to capture semantics of the outside world that is being modeled through the data model and the database. For this reason they are also called semantic constraints. ICs have been around at least since the inception of the relational model of data. Already in the classical and seminal papers in the area [36] it is possible to find the notions of integrity and consistency of a database.

ICs have been studied in general and have wide application in data management. A large body of research has been developed, in particular fundamental research has been carried out. Furthermore, methodologies for dealing with ICs are quite general and have broad applicability.

2.1 The basics of ICs

A database can be seen as a model, i.e. as a simplified, abstract description, of an external reality. In the case of relational databases, one starts by choosing certain predicates of a prescribed arity. The *schema* of the database consists of this set of predicates, possibly *attributes*, which can be seen as names for the arguments of the predicates, together with an indication of the domains where the attributes can take their values. Having chosen the schema, the representation of the external reality is given in terms of relations, which are extensions for the predicates in the schema. This set of relations is called an *instance* of the schema.

For example, relational database for representing information about students of a university might be based on the schema consisting of the predicates $Students(StuNum, StName)$ and $Enrollment(StuName, Course)$. The attribute *StuNum* is expected to take numerical values; *StuName*, character string values; and *Course*, alphanumeric string values. In Figure 1 there is a possible instance for this schema.

Students

StuNum	*StuName*
101	john bell
102	mary stein
104	claire stevens
107	pat norton

Enrollment

StuNum	*Course*
104	comp150
101	comp100
101	comp200
105	comp120

Fig. 1 A database instance

In order to make the database a more accurate model of the university domain (or to be in a more accurate correspondence with it), certain conditions are imposed on the possible instances of the database. Those conditions are intended to capture more meaning from the outside application domain. In consequence, these conditions are called *semantic constraints* or *integrity constraints* (ICs). For example, a condition could be that, in every instance, the student name functionally depends upon the student number, i.e. a student number is assigned to at most one student name. This condition, called a *functional dependency* (FD), is denoted with $StuNumber \rightarrow StuName$, or $Students : StuNumber \rightarrow StuName$, to indicate that this dependency should hold for attributes of relation *Students*. Actually, in this

case, since all the attributes in the relation functionally depend on *StuNum*, the FD is called a *key constraint*.

Students

StuNum	*StuName*
101	john bell
101	joe logan
104	claire stevens
107	pat norton

Enrollment

StuNum	*Course*
104	comp150
101	comp100
101	comp200

Fig. 2 Another instance

Integrity constraints can be declared together with the schema, indicating that the instances for the schema should all satisfy the integrity constraints. For example, if the functional dependency $Students : StuNumber \rightarrow StuName$ is added to the schema, the instance in Figure 1 is consistent, because it satisfies the FD. However, the instance in Figure 2 is *inconsistent*. This is because this instance does not satisfy, or, what is the same, violates the functional dependency (the student number *101* is assigned to two different student names.

It is also possible to consider with the schema a *referential integrity constraint* that requires that every student (number) in the relation *Enrollment* appears, associated with a student name, in relation *Students*, the official "table" of students. This is denoted with $Enrollement[StuNum] \subseteq Students[StuNum]$, and is a form of *inclusion dependency*. If this IC is considered in the schema, the instance in Figure 1 is inconsistent, because student *105* does not appear in relation *Students*. However, if only this referential constraint were associated to the schema, the instance in Figure 2 would be consistent. The combination of the given referential constraint and the functional dependency creates a *foreign key constraint*: The values for attribute *StuNum* in *Enrollment* must appear in relation *Students* as values for its attribute *StuNum*, and this attribute form a *key* for *Students*.

In can be seen that the notion of consistency is relative to a set of integrity constraints. A database instance that satisfies each of the constraints in this set is said to be *consistent*, and *inconsistent* otherwise.

The two particular kinds of integrity constraints presented above and also other forms of ICs can be easily expressed in the language of predicate logic. For example, the FD above can be expressed by the symbolic sentence

$$\forall x \forall y \forall z ((Students(x,y) \wedge Students(x,z)) \longrightarrow y = z), \tag{1}$$

whereas the referential constraint above can be expressed by

$$\forall x \forall y (Enrollment(x,y) \longrightarrow \exists z Students(x,z)). \tag{2}$$

Notice that this language of predicate logic is determined by the database schema, whose predicates are now being used to write down logical formulas. We may also use "built-in" predicates, like the equality predicate. Thus, ICs can be seen as forming a set Σ of sentences written in a language of predicate logic.

A database instance can be seen as an *interpretation structure D* for the language of predicate logic that is used to express ICs. This is because an instance has an underlying domain and (finite) extensions for the predicates in the schema. Having the database instance as an interpretation structure and the set of ICs as a set of symbolic sentences makes it possible to simply apply the notion of satisfaction of a formula by a structure of first-order predicate logic. In this way, the notion of satisfaction of an integrity constraint by a database instance is a precisely defined notion: the database instance D is consistent if and only if it satisfies Σ, which is commonly denoted with $D \models \Sigma$.

Since it is usually assumed that the set of ICs is consistent as a set of logical sentences, in databases the notion of consistency becomes a condition on the database instance. Thus, this use of the term "consistency" differs from its use in logic, where consistency characterizes a set of formulas.

2.2 Checking and enforcing ICs

Inconsistency is an undesirable property for a database. In consequence, one attempts to keep it consistent as it is subject to updates. There are a few ways to achieve this goal. One of them consists in declaring the ICs together with the schema, and the database management system (DBMS) will take care of the *database maintenance*, i.e. of keeping it consistent. This is done by rejecting transactions that may lead to a violation of the ICs. For example, the DBMS should reject the insertion of the tuple $(101, sue\ jones)$ into the instance in Figure 1 if the FD (1) was declared with the schema (as a key constraint). Unfortunately, the classes of ICs for which most commercial DBMSs offer this kind of automated, built-in support are quite restricted [72].

An alternative way of keeping consistency is based on the use of triggers (or active rules) that are stored in the database [32]. The reaction to a potential violation is programmed as the action of the trigger: if a violation is about to be produced or is produced, the trigger automatically reacts, and its action may reject the violating transaction or compensate it with additional updates, to make sure that at the end, consistency is reestablished. Consistency can also be enforced through the application programs that interact with the DBMS. However, the correctness of triggers or application programs with respect to ensuring database consistency is not guaranteed by the DBMS.

Inconsistency of a DB under updates can be checked via *violation views* that transitorily store violating tuples, if any. IC satisfaction corresponds to an empty violation view. Some consistency restoration policy can be applied on the basis of the violation views. No wonder that database maintenance and *view maintenance* [60] are closely related problems. It is possible to apply similar techniques to both problems. For example, to check IC violation and view freshness, i.e. the correspondence with the base tables, inductive incremental techniques can be applied. More precisely, on the basis of the syntactic form of an IC or a view definition, it is possible to identify the updates that are relevant under the assumption that the database is consistent or the view is up-to-date when those updates are executed. Potential IC

violations or changes in view extensions are checked only for those relevant updates [67; 16].

It is the case that, for whatever reasons, databases may become inconsistent, i.e. they may violate certain ICs that are considered to be relevant to maintain for a certain application domain. This can be due to several reasons, e.g. poorly designed or implemented applications that fail to maintain the consistency of the database, or ICs for which a DBMS does not offer any kind of support, or ICs that are not enforced for better performance of application programs or DBMSs, or ICs that are just assumed to be satisfied based on knowledge about the application domain and the kind of updates on the database. It is also possible to have a legacy database on which semantic constraints have to be imposed; or more generally, a database on which imposing new constraints depending on specific needs, e.g. user constraints, becomes necessary.

In the area of data integration the satisfaction of desirable ICs by a database is much more difficult to achieve. One can have different autonomous databases that are separately consistent with respect totheir own, local ICs. However, when their data are integrated into a single database, either material or virtual, certain desirable global ICs may not be satisfied. For example, two university databases may use the same numbers for students. If their data are put together into an integrated database, a student number might be assigned to two different students (cf. Section 3.3).

When trying to use an inconsistent database, the application of some *data cleaning* techniques may be attempted, to cleanse the database from data that participate in the violation of the ICs. This is done sometimes. However, data cleaning is a complex and non-deterministic process; and it may also lead to the loss of information that might be useful. Furthermore, in certain cases like virtual data integration, where the data stay at the autonomous data sources, there is no way to change the data without the ownership of the sources.

One might try to live with an inconsistent database. Actually, most likely one will be forced to keep using it, because there is still useful information in it. It is also likely that most of the information in it is somehow consistent. Thus, the challenge consists in retrieving from the database only information that is consistent. For example, one could pose queries to the database at hand, but expecting to obtain only answers that are semantically correct, i.e. that are consistent with the ICs. This is the problem of *consistent query answering* (CQA), which can be formulated in general terms as the one of characterizing and computing semantically correct answers to queries posed to inconsistent databases [2].

3 Repairs and Consistent Answers

The notion of consistency of a database is a holistic notion, that applies to the entire database, and not to portions of it. In consequence, in order to pursue this idea of retrieving consistent query answers, it becomes necessary to characterize the consistent data in an inconsistent database first. The idea that was proposed in [2] is as follows: the consistent data in an inconsistent data are the data that are invariant under all possible way of restoring the consistency by performing minimal changes

on the initial database. That is, no matter what minimal consistency restoration process is applied to the database, the consistent data stay in the database. Each of the consistent versions of the original instance obtained by minimal changes is called a *minimal repair*, or simply, a *repair*.

It becomes necessary to be more precise about the meaning of minimal change. In between, a few notions have been proposed and studied (cf. [9; 35; 10] for surveys of CQA). Which notion to use may depend on the application. The notion of minimal change can be illustrated using the definition of repair given in [2]. First of all, a database instance D can be seen as a finite set of ground atoms (or database tuples) of the form $P(\bar{c})$, where P is a predicate in the schema, and $\bar{c}$ is a finite sequence of constants in the database domain. For example, $\mathit{Students}(\mathit{101}, \mathit{john\ bell})$ is an atom in the database. Next, it is possible to compare the original database instance D with any other database instance D' (of the same schema) through their symmetric difference $D\,\Delta D' = \{A \mid A \in (D \smallsetminus D') \cup (D' \smallsetminus D)\}$.

Now, a repair of an instance D with respect toa set of ICs Σ is defined as an instance D' that is consistent, i.e. $D' \models \Sigma$, and for which there is no other consistent instance D'' that is closer to D than D', i.e. for which it holds $D\,\Delta D'' \subsetneq D\,\Delta D'$. For example, the database in Figure 2 has two repairs with respect tothe FD (1). They are shown in Figure 3 and are obtained each by deleting one of the two conflicting tuples in relation *Students* (relation *Enrollment* does not change).

Having defined the notion of repair, a *consistent answer* from an instance D to a query $\mathscr{Q}(\bar{x})$ with respect toa set Σ of ICs is defined as an answer $\bar{c}$ to $\mathscr{Q}$ that is obtained from every possible repair of D with respect toΣ. That is, if the query $\mathscr{Q}$ is posed to each of the repairs, $\bar{c}$ will be returned as a usual answer to $\mathscr{Q}$ from each of them.

For example, if the query $\mathscr{Q}_1(x,y)$: $\mathit{Students}(x,y)$, asking for the tuples in relation *Students*, is posed to the instance in Figure 2, then $(\mathit{104}, \mathit{claire\ stevens})$ and $(\mathit{107}, \mathit{pat\ norton})$ should be the only consistent answers with respect tothe FD (1). Those are the tuples that are shared by the extensions of *Students* in the two repairs. Now, for the query $\mathscr{Q}_2(x) : \exists y \mathit{Students}(x,y)$, i.e. the projection on the first attribute of relation *Students*, the consistent answers are $(\mathit{101})$, $(\mathit{104})$ and $(\mathit{107})$.

Students

StuNum	*StuName*
101	john bell
104	claire stevens
107	pat norton

Students

StuNum	*StuName*
101	joe logan
104	claire stevens
107	pat norton

Fig. 3 Two repairs of *Students* in Fig. 2

There might be a large number of repairs for an inconsistent database. In consequence, it is desirable to come up with computational methodologies to retrieve consistent answers that use only the original database, in spite of its inconsistency. Such a methodology, that works for particular syntactic classes of queries and ICs, was proposed in [2]. The idea is to take the original query $\mathscr{Q}$ that expects consistent answers, and syntactically transform it into a new query $\mathscr{Q}'$, such that the *rewritten query* $\mathscr{Q}'$, when posed to the original database, obtains as usual answers the con-

sistent answers to query $\mathscr{Q}$. The essential question is, depending on the language in which $\mathscr{Q}$ is expressed, what kind of language is necessary for expressing the rewriting $\mathscr{Q}'$. The answer to this question should also depend on the kind of ICs being considered.

The idea behind the rewriting approach presented in [2] can be illustrated by means of an example. The consistent answers to the query $\mathscr{Q}_1(x,y) : \mathit{Students}(x,y)$ above with respect to the FD (1) can be obtained by posing the query $\mathscr{Q}'(x,y) : \mathit{Students}(x,y) \wedge \neg\exists z(\mathit{Students}(x,z) \wedge z \neq y)$ to the database. The new query collects as normal answers those tuples where the value of the first attribute is not associated to two different values of the second attribute in the relation. It can be seen that the set of answers to the new query can be computed in polynomial time in the size of the database.

In this example, a query expressed in first-order predicate logic was rewritten into a new query expressed in the same language. It has been established in the literature that, for complexity-theoretic reasons, a more expressive language to do the rewriting of a first-order query may be necessary. For example, it may be necessary to do the rewritings as queries written in expressive extensions of Datalog [3; 59; 5; 40]. See Section 3.1 for more details.

If a database is inconsistent with respect to referential ICs, like the instance in Figure 1 and the constraint in (2), it is natural to restore consistency by deleting tuples or inserting tuples containing *null values* for the existentially quantified variables in the ICs. For example, the tuple $(105, comp120)$ could be deleted from *Enrollment* or the tuple $(105, null)$ could be inserted in relation *Students*. This requires a modification of the notion of repair and a precise semantics for satisfaction of ICs in the presence of null values [21; 30].

Some repair semantics consider changes of attribute values as admissible basic repair actions [51; 74; 18; 12; 50], which is closer to many data cleaning processes. Usually, it is the number of these changes what is minimized. With a change of repair semantics, the problems of complexity and computation of consistent answers, query rewriting, and also of specification of database repairs, have to be reinvestigated.

CQA involves, possibly only implicitly, the whole class of database repairs. However, some research in this area, and much in the spirit of classical data cleaning, has also addressed the problem of computing a single "good" repair [18; 75; 37], or a single universal repair that can act for some tasks as a good representative of the class of repairs [71], or an approximate repair [12; 61].

3.1 Answer set programs for database repairing

ICs like (1) and (2) specify desirable properties of a database. However, when they are not satisfied by a given instance, they do not tell us how to change the instance so that they hold again. This requires a separate specification of the repair process. A declarative specification of it will tell us, in a language with a clear logical semantics, what are the intended results of the repair process, without having to explicitly

express how to go about computing them. The intended models of the specification should correspond to the expected results.

In an ideal situation, the declarative specification can also be used as (the basis for) an *executable specification*, that can be used, for example, for *computing* the models (the repairs in our case) or computing query answers from the specification (the consistent answers in our case).

Actually, it turns out that the class of repairs of an inconsistent database can be specified by means of disjunctive logic programs with stable model semantics, aka. *answer-set programs* [24]. The programs can be modified in order to accommodate computation of single repairs or an approximation to one of them, enabling in this way a form of rule-based data cleaning process, and in this case, of restoration of semantic integrity. In this section we briefly describe the approach proposed in [5; 21] (cf. also [30] for more details).

The idea behind *repair programs* is to represent the process of capturing and resolving inconsistencies through rules, and enforce the minimality of repairs through the minimality of the stable models of the program.

A repair program is produced for a database schema, including a non-necessarily enforced set of ICs. More specifically, for each database predicate P, a new predicate $P_$ is introduced. It corresponds to P augmented with an extra attribute that can take the following values (annotation constants): $\mathbf{t}$, $\mathbf{f}$, $\mathbf{t}^\star$ and $\mathbf{t}^{\star\star}$, whose intended semantics is as follows.

Atom $P_(\bar{t},\mathbf{t})$ (respectively, $P_(\bar{t},\mathbf{f})$) indicates the insertion of tuple $\bar{t}$ into relation P (respectively, the deletion of $\bar{t}$ from P) during the repair process. Atom $P_(\bar{t},\mathbf{t}^\star)$ indicates that tuple $\bar{t}$ was in the original extension of P or is inserted in the repair process. Finally, atom $P_(\bar{t},\mathbf{t}^{\star\star})$ indicates that tuple $\bar{t}$ belongs to the final extension of P. For example, if a stable model of the repair program for the FD $StuNumber \rightarrow StuName$ contains the atoms $Students(101, john\ bell)$, $Students(101, joe\ logan)$, $Students\_(101, john\ bell, \mathbf{f})$ and $Students\_(101, joe\ logan, \mathbf{t}^{\star\star})$, it means that tuple $Students(101, john\ bell)$ was removed in order to solve an inconsistency, and tuple $Students(101, joe\ logan)$ belongs to a repair.

As an example, the repair program for the student database with the FD (1) and the referential constraint (2) contains the original database tuples as program facts, plus:

1. A rule to enforce the FD:

$$Students\_(x,y,\mathbf{f}) \vee Students\_(x,z,\mathbf{f}) \leftarrow Students\_(x,y,\mathbf{t}^\star), Students\_(x,z,\mathbf{t}^\star),\ y \neq z,\\ x \neq null,\ y \neq null,\ z \neq null.$$

The right hand side of the rule checks if there are tuples (that were originally part of the database or that where made true while repairing) that violate the FD. If that is the case, it solves the inconsistency by removing one of the two tuples (the left hand side of the rule). Notice that there can be a violation of the FD in a database with *null* only if x, y and z are not *null*.

2. Rule to enforce the referential constraint:

$Aux(x) \leftarrow Students\_(x,z,\mathbf{t}^\star),\ not\ Students\_(x,z,\mathbf{f}),\ x \neq null,\ z \neq null.$
$Enrollment\_(x,y,\mathbf{f}) \vee Students\_(x,null,\mathbf{t}) \leftarrow Enrollment\_(x,y,\mathbf{t}),\ not\ Aux(x), x \neq null.$

The first rule populates an auxiliary predicate storing all the students numbers stored in the table *Students* that are not deleted in the repair process. The second rule checks through its right hand side if for any enrolled student, his/her number does appear in the table *Students*. If there is a violation, to solve the inconsistency, it forces either to remove the tuple from *Enrollment* or add a tuple to *Students*.

3. Rules defining the annotation semantics:

$$\left.\begin{array}{l} Students\_(x,y,\mathbf{t}^\star) \leftarrow Students(x,y). \\ Students\_(x,y,\mathbf{t}^\star) \leftarrow Students\_(x,y,\mathbf{t}). \\ Students\_(x,y,\mathbf{t}^{\star\star}) \leftarrow Students\_(x,y,\mathbf{t}^\star),\ not\ Students\_(x,y,\mathbf{f}). \\ \leftarrow Students\_(x,y,\mathbf{t}), Students\_(x,y,\mathbf{f}). \end{array}\right\} \text{Similarly for } Enrollment\_$$

The first two rules ensure that tuples with $\mathbf{t}^\star$ are those in the original database or those that are made true through the repair. The third rule collects tuples that belong to a final repair. The last rule is a *program constraint* that discards models were a tuple is made both true and false.

If rules **1.** to **3.** are combined with the database instance in Figure 2, the repair program has two stable models $\mathscr{M}_1$ and $\mathscr{M}_2$, such that in $\mathscr{M}_1$ predicate $Students\_ = \{(101, john\ bell, \mathbf{t}^\star), (101, joe\ logan, \mathbf{t}^\star), (104, claire\ stevens, \mathbf{t}^\star), (101, john\ bell, \mathbf{f}), (101, joe\ logan, \mathbf{t}^{\star\star}), (104, claire\ stevens, \mathbf{t}^{\star\star}), (107, pat\ norton, \mathbf{t}^{\star\star})\}$, whereas in $\mathscr{M}_2$ predicate $Students\_ = \{(101, john\ bell, \mathbf{t}^\star), (101, joe\ logan, \mathbf{t}^\star), (104, claire\ stevens, \mathbf{t}^\star), (101, joe\ logan, \mathbf{f}), (101, john\ bell, \mathbf{t}^{\star\star}), (104, claire\ stevens, \mathbf{t}^{\star\star}), (107, pat\ norton, \mathbf{t}^{\star\star})\}$. These models correspond to the repairs shown in Figure 3.

Given a database D and a set of ICs Σ, if Σ is *RIC-acyclic* [21], i.e. there is no cycle through referential constraints, then there is a one-to-one correspondence between repairs and stable models of the program: the tuples annotated with $\mathbf{t}^{\star\star}$ in a stable model form a repair of D with respect to Σ.

Consistent query answers can be computed directly from the repair program using the *cautious or skeptical semantics*, according to which an atom is true if it belongs to all models of the program. For example, if we want to pose the query $\mathscr{Q}_1(x,y) : Students(x,y)$, or the query $\mathscr{Q}_2(x) : \exists y Students(x,y)$, we would simply add the rule $Ans_{\mathscr{Q}_1}(x,y) \leftarrow Students\_(x,y,\mathbf{t}^{\star\star})$ to the program, or $Ans_{\mathscr{Q}_2}(x) \leftarrow Students\_(x,y,\mathbf{t}^{\star\star})$, respectively. Cautious reasoning from the extended program returns the consistent query answers to the queries.

There are techniques to improve the efficiency of repair programs by concentrating only on what is relevant for the query at hand [30]. Repair programs can also be modified to capture other semantics. For example, with *weak constraints* one can specify repairs that minimize the *number* of tuple insertions/deletions [3].

3.2 Active integrity constraints

When specifying a database that could possibly not satisfy certain ICs, we might want to explicitly indicate how to restore consistency when an IC is violated. In this

way, we specify both the constraint and its enforcement. In this direction, in [31] *active integrity constraints* were proposed. They extend traditional ICs with specifications of the actions to perform in order to restore their satisfaction, declaratively bringing ICs and active rules together.

For example, if we wanted to repair violations of the constraint (2), by only removing tuples from *Enrollment*, we could specify the following active IC:

$$\forall x \forall y [\mathit{Enrollment}(x,y),\ \neg \exists z \mathit{Students}(x,z) \supset -\mathit{Enrollment}(x,y)],$$

with the effect of removing a tuple *Enrollment* when it participates in a violation.

If, instead, we want to repair by either removing or inserting, we could use

$$\forall x \forall y [\mathit{Enrollment}(x,y), \neg \exists z \mathit{Students}(x,z) \supset -\mathit{Enrollment}(x,y) \vee +\mathit{Students}(x,\bot)],$$

where $\bot$ is a constant denoting an *unknown* value.

A *founded repair* is a minimal repair (as defined in Section 3) where each insertion and removal is supported by an active IC. The founded repairs can be computed using logic programs with stable models semantics (cf. [31] for details).

3.3 ICs and virtual data integration

Virtual data integration [62; 11] is about providing a unified view of data stored in different sources through what is usually called a *mediator*. This is achieved by the definition of a global database schema, and mappings between this schema and the source schemas. Queries that are received by the integration system via the mediator are automatically rewritten into a set of sub-queries that are sent to the sources. The answers from the sources are combined to give an answer to the user query. In this setting, specially because of the autonomy of the sources, it is hard or impossible to ensure that data are consistent across the different sources. Actually, it is natural to try to impose *global ICs*, i.e. on the global schema, with the intention to capture the semantics of the integration system as a whole. However, there is nothing like an explicit global database instance on which these ICs can be easily checked or enforced.

Consider, for example, a university that uses a virtual data integration system that integrates the departmental databases, which are maintained by the departments. The university officials can pose queries through a global schema that logically unifies the data, without the need of transforming and moving all the data into a single place.

StuPs

StuNum	*StuName*
101	john bell
104	claire stevens
107	pat norton

StuCS

StuNum	*StuName*	*Admission*
104	alex pelt	2011
121	ana jones	2012
137	lisa reeves	2012

Fig. 4 Two data sources with student information from different departments

For simplicity, assume we have only two data sources, containing information about psychology and computer science students, in tables *StuPs* and *StuCS*, respectively, as shown in Figure 4. The mediator offers a global schema

$$StuUniv(StuNum, StuName, Admission, Department).$$

The relationship between the sources and the global schema can be defined through the following mappings:

$$StuPs(x,y) \leftarrow StuUniv(x,y,z,\text{`}psychology'). \quad (3)$$

$$StuCS(x,y,z) \leftarrow StuUniv(x,y,z,\text{`}comp.\ sci.'). \quad (4)$$

These mappings are defined according to the *local-as-view* (LAV) approach, where the sources are defined as views over the global schema.[1] Usually sources are assumed to be *open* (or *incomplete*), in the sense that these mappings require that potential global instances, i.e. for the global schema, if we decided to built and materialize them, have to contain *at least* the data that are needed to reconstruct the source contents through the view definitions (3) and (4). For example, the first mapping requires that, for every tuple in *StuPs*, there exists a tuple in *StuUniv* with a value for *Admission* (possibly not known) and with the value *psychology* for attribute *Department*.

As a consequence, a global instance satisfies a mapping if the result of applying the mapping to it (through the right hand side of the mapping) produces a superset of the data in the source that is defined by it. If a global instance satisfies all the mappings, it is called a *legal instance* of the integration system.

The semantics of query answering in such a data integration system is the one of *certain answers*. More precisely, an answer to a query expressed in terms of the global schema is a certain answer it is a (usual) answer from each of the possible legal instances. When dealing with monotone queries, say without negation, it is enough to concentrate on the *minimal legal instances*. They are the legal instances that are minimal under set inclusion.

StuUniv

StuNum	*StuName*	*Admission*	*Department*
101	john bell	**X**	psychology
104	claire stevens	**Y**	psychology
107	pat norton	**Z**	psychology
104	alex pelt	2011	comp. sci.
121	ana jones	2012	comp. sci.
137	lisa reves	2012	comp. sci.

Fig. 5 A minimal legal instance with **X**, **Y** and **Z** representing arbitrary values from the domain

In the university database the minimal legal instances can be represented by the instance in Figure 5 where the variables can take any value in the domain. A legal instance will be any instance of the global schema that is a superset of one of these minimal legal instances. The set of certain answers to query $\mathscr{Q}_3(x)$: $\exists y \exists z \exists w Students(x,y,z,w)$ is $\{101, 104, 107, 121, 137\}$ since its elements belong to

[1] cf. [62] for alternative approaches to mapping definitions.

all the minimal legal instances. Query $\mathscr{Q}_4(x,z): \exists y \exists w \mathit{Students}(x,y,z,w)$ will return only $\{(104,2011),(121,2012),(137,2012)\}$ since different minimal legal instances will return different admission years for the students of the Psychology Department.

It is possible to specify this class of legal instances as the models of an answer-set program [11], and certain answers can be computed by cautious reasoning from the program. In the university example, the program that specifies the minimal legal instances contains the source facts, plus the rules:

1. $\mathit{dom}(\mathbf{X})$. (for every value $\mathbf{X}$ in the domain)
2. $\mathit{StuUniv}(x,y,z,\text{`}\mathit{psychology}\text{'}) \leftarrow \mathit{StuPs}(x,y), F_1(x,y,z)$.
   $\mathit{StuUniv}(x,y,z,\text{`}\mathit{comp.\ sci.}\text{'}) \leftarrow \mathit{StuPs}(x,y,z)$.
3. $F_1(x,y,z) \leftarrow \mathit{StuPs}(x,y), \mathit{dom}(z), \mathsf{choice}((x,y),z)$.

The first rule adds all the elements of the finite domain.[2] The next two rules computes the minimal legal instances from the sources. Since the psychology department does not provide information about the admission year of the student, a value of the domain needs to be added to each instance. This is achieved with predicate $F_1(X,Y,Z)$, which satisfies the functional dependency $X,Y \rightarrow Z$ and assigns in each model a different value for Z for each combination of values for X and Y. This requirement for F_1 is enforced with the choice operator in the last rule, whose semantics can be defined by ordinary rules with a stable models semantics [56]. For more details, more complex cases, and examples, see [11].

So far, we haven't consider global ICs. IC violations are likely in integration systems, and dealing with global ICs is a real challenge. In our ongoing example, we want to enforce that the *StuNum* is unique within the University. Both data sources in Figure 4 satisfy this requirement, but when we combine the data, every legal instance will violate this constraint since they will contain students *claire stevens* and *alex pelt* with the same student number. As in the case of traditional relational databases, most of the data are still consistent, and we would like to be able to still provide answers from the consistent data.

StuUniv

StuNum	*StuName*	*Admission*	*Department*
101	john bell	**X**	psychology
104	claire stevens	**Y**	psychology
107	pat norton	**Z**	psychology
121	ana jones	2012	comp. sci.
137	lisa reeves	2012	comp. sci.

StuUniv

StuNum	*StuName*	*Admission*	*Department*
101	john bell	**X**	psychology
107	pat norton	**Z**	psychology
104	alex pelt	2011	comp. sci.
121	ana jones	2012	comp. sci.
137	lisa reeves	2012	comp. sci.

Fig. 6 Repairs of the global integration system

Different approaches have been explored for dealing with global ICs. One of them consists in applying a repair semantics as in Section 3. More precisely, if a minimal legal instance does not satisfy a set of global ICs, it is repaired as before. The consistent answers to global queries are those that can be obtained from every repair from every minimal legal instance [11]. In our example, the (global) re-

[2] For details of how to treat infinite domains see [11].

pairs in Figure 6 are obtained by deleting either $(104, claire\ stevens, \mathbf{Y}, psychology)$ or $(104, alex\ pelt, 2011, comp.\ sci.)$ from every minimal legal instance in Figure 5. Now, the consistent answers to query $\mathcal{Q}_3$ will coincide with the certain answers since *104* is also part of every repair. On the other hand, the consistent answers to query $\mathcal{Q}_4$ now do not contain $(104, 2011)$ since that answer is not obtained from all repairs.

Notice that the program above that specifies the minimal legal instances can be then extended with the rules introduced in Section 3.1, to specify the repairs of the minimal legal instances with respect tothe global FD [11]. Next, in order to retrieve consistent answers from the integration system, the already combined program can be further extended with the query program.

Since the repair process may not respect the *nature of the sources*, in terms of being open, closed, etc. (as it is the case with the global repairs we just showed), we may decide to ignore this aspect [11] or, whenever possible, go for a repair semantics that respects this nature. For example, inserting global tuples may never damage the openness of a source [26; 27]. However, if these tuple insertions are due to the enforcement of inclusion dependencies, the presence of functional dependencies may create a problem since those insertions might violate them. This requires imposing some conditions on the syntactic interaction of FDs and inclusion dependencies [26].

Another possibility is to conceive a mediated system as an ontology that acts as a metadata layer on top of incomplete data sources [63]. The underlying data can be *chased* by means of the mappings and global ICs, producing data at the global level, extending the original extensional data [28].

Under any of these approaches the emphasis is on certain query answering, and a full computation of legal instances, repairs, or chase extensions should be avoided whenever possible.

4 Data Dependencies and Data Quality

In Section 2 we have considered techniques for data quality centered in classical ICs, such as functional dependencies and referential constraints. These constraints, however, are not always expressive enough to represent the relationships among values for different attributes in a table.

Let us consider the example in Figure 7 of a database storing information of the staff working at a newspaper. There are several restrictions on the data that can be represented using classical constraints. For example, in table *Staff* we have two FDs: $(Num \rightarrow Name, Type, Street, City, Country, Zip)$, and $(Zip, Country \rightarrow City)$. We also have the inclusion dependency: $(Journalist[Num] \subseteq Staff[num]$. It is easy to check that the instance in Figure 7 satisfies all these constraints.

The data, however, is not consistent with respect to other dependencies that cannot be expressed using this type of constraints since they apply only to some tuples in the relation. For example, in the UK, the zip code uniquely determines the street. If we try to enforce this requirement using the FD $Zip \rightarrow Street$, we would not obtain what we want since this requirement would be imposed also for tuples correspond-

ing to the US. Also, we cannot express that every journalist in *Staff* should have a role assigned in table *Journalist*, and that every *Num* in *Journalist* belongs to a tuple in table *Staff* with the $SNum = Num$ and $Type =$'*journalist*'. The given instance does not satisfy any of these additional constraints. In order to clean the data, we need to consider those dependencies that apply only under certain conditions.

Staff

Num	*Name*	*Type*	*Street*	*City*	*Country*	*Zip*
01	john	admin	First	Miami	US	33114
02	bill	journalist	Bronson	Miami	US	33114
03	nick	journalist	Glebe	Tampa	US	33605
04	ana	admin	Crow	Glasgow	UK	G11 7HS
05	mary	journalist	Main	Glasgow	UK	G11 7HS

Journalist

SNum	*Role*
01	news
02	news
02	columnist
03	columnist

Fig. 7 The *Newspaper* database

4.1 Conditional dependencies

In [19; 22], *conditional functional dependencies* and *conditional inclusion dependencies* are introduced, to represent data dependencies that apply only to tuples satisfying certain conditions.

A conditional functional dependency (CFD) is a pair $(X \to Y, T_p)$, where $X \to Y$ is a classical functional dependency, and T_p is a *pattern tableau*, showing attributes among those in X and Y. For every attribute A of T_p and every tuple $t_p \in T_p$, it holds that $t_p[A]$ is a constant in the domain of A or an unnamed variable '_'.

For the newspaper database we could define a CFD for relation *Staff* that enforces that for the UK the zip code determines both the city and the street:

$\psi_1 = (Country, Zip \to Street, City, T_1)$, with T_1:

Country	*Zip*	*Street*	*City*
UK	_	_	_

Two data values n_1 and n_2 match, denoted $n_1 \asymp n_2$ if $n_1 = n_2$ or one of n_1, n_2 is '_'. Two tuples match, denoted $t_1 \asymp t_2$, if they match componentwise. Now, a CFD $(X \to Y, T_p)$ is satisfied by a database instance if for every $t_p \in T_p$ and every pair of tuples t_1 and t_2 in the database, if $t_1[X] = t_2[X] \asymp t_p[X]$, then $t_1[Y] = t_2[Y] \asymp t_p[Y]$.

Constraint ψ_1 is violated by the instance in Figure 7 since the last two tuples are from the UK, they share the same zip code but they are associated to different streets.

A conditional inclusion dependency (CIND) is a pair $(R_1[X;X_p] \subseteq R_2[Y;Y_p], T_p)$ where:

(i) R_1 and R_2 are database predicates,
(ii) X, X_p (respectively, Y and Y_p) are disjoint lists of attributes of R_1 (respectively, R_2),[3] and
(iii) T_p is a pattern tableau that contains all the attributes in X, X_p, Y and Y_p and $t_p[X] = t_p[Y]$.

[3] The empty list is denoted by nil.

In this case, the inclusion dependency $R_1[X] \subseteq R_2[Y]$ is said to be embedded in the CIND.

For the newspaper database we could define two CINDs:

$\psi_2 = (\mathit{Staff}[\mathit{Num}; \mathit{Type}] \subseteq \mathit{Journalist}[\mathit{SNum}; \mathsf{nil}], T_2)$, with T_2:

Num	*Type*	*SNum*
_	journalist	_

$\psi_3 = (\mathit{Journalist}[\mathit{SNum}; \mathsf{nil}] \subseteq \mathit{Staff}[\mathit{Num}; \mathit{Type}], T_3)$, with T_3:

SNum	*Num*	*Type*
_	_	journalist

Constraint ψ_2 enforces that every *Num* of a journalist in relation *Staff* has a role assigned to it in relation *Journalist*. Constraint ψ_3 enforces that, for every *SNum* in relation *Staff*, there is a tuple with the same number in *Num* of type *Journalist*.

A CIND $(R_1[X; X_p] \subseteq R_2[Y; Y_p], t_p)$ is satisfied, if for every tuple $t_1 \in R_1$ and every $t_p \in T_p$ if $t_1[X, X_p] \asymp t_p[X, X_p]$ then there exists a tuple $t_2 \in R_2$ such that $t_2[Y] = t_1[Y]$ and $t_2[Y, Y_p] \asymp t_p[Y, Y_p]$. Intuitively, the CIND enforces the inclusion dependency $R_1[X] \subseteq R_2[Y]$ for each tuple that matches the pattern for $[X, X_p]$, and also requires that the tuple in R_2 should match the pattern for $[Y, Y_p]$.

Instance in Figure 7 does not satisfy constraint ψ_2 since there is a journalist with *Num* = *05* in relation *Staff* for which there is no tuple in *Journalist* with *SNum* = *05*. It also violates constraint ψ_3 because for tuple $(\mathit{01}, \mathit{news})$ in relation *Journalist* there is no tuple in *Staff* with *Num* = *01* and *Type* = *journalist*.

Classical functional dependencies are a particular case of CFDs for which the pattern tableau has a single tuple with only unnamed variables. In the same way, classical inclusion dependencies are a particular case of CINDs that can be defined using $X_p = Y_p = \mathsf{nil}$, and a pattern tableau with a single tuple with only unnamed variables.

CFDs have been extended to consider (i) disjunction and inequality [23], (ii) ranges of values [57], and (iii) cardinality and synonym rules [33]. Automatic generation and discovery of CFDs have been studied in [46; 34; 43]. Pattern tableaux have also been used to show the portions of the data that satisfy (or violate) a constraint [57; 58]. These pattern tableaux can be used both to characterize the quality of the data and generate CFDs and CINDs.

We can see that conditional dependencies (CDs) have a classic semantics. Actually, it is easy to express them in first-order predicate logic. However, they have been proposed with the problems of data quality and data cleaning in mind.

4.2 Data cleaning with CDs

CFDs and CINDs were introduced to specify data dependencies that were not captured by classic constraints, and to be used for improving the quality of the data. In [37] a data cleaning framework is provided for conditional functional dependencies. Given a database instance which is inconsistent with respect toa set of CFDs, the main objective is to find a repair through attribute updates that minimizes a cost function. The cost of changing a value v by a value v' is $\mathit{cost}(v, v') = w(t, A) \cdot \mathit{dist}(v, v') / \max(|v|, |v'|)$, where w is a weight assigned to attribute A in tuple t, and *dist* is a distance function. The weights, for example, could be used to include

in the repair process information that we could have about the data, e.g. about their accuracy. The distance function should measure the similarity between values. It could be, for example, the *edit distance*, that counts the number of insertions and deletions of characters to transform from one string into another.

Consider relation S in Figure 8(a), the conditional FDs $\varphi_1 = (X \to Y, T_1)$, $\varphi_2 = (Y \to W, T_2)$, and $\varphi_3 = (Z \to W, T_3)$, with:

T_1:

X	Y
_	_

T_2:

Y	W
b	_

T_3:

Z	W
c	d
k	e

The instance does not satisfy φ_1. Solving the inconsistencies with respect to φ_1 by replacing $t_1[Y]$ by b triggers more inconsistencies with respect to φ_2, that need to be solved by changing more values for attributes Z and W. On the other hand, if we instead replace $t_2[Y]$ and $t_3[Y]$ by c, all inconsistencies are solved. The repairs obtained in these two ways correspond to S_1 and S_2 in Figure 8 (the updated values are in bold). If we consider $w(t,A) = 1$ for all attributes, and use the edit distance, the cost of each change is 2; and therefore, the cost of repairs S_1 and S_2 are 6 and 4, respectively. Thus, S_2 is a better repair than S_1. Another minimal-cost repair, say S_3, is obtained by replacing $t_2[Y]$ by c, and $t_3[X]$ by any constant different from a.

(a) S

	X	Y	Z	W	U
t_1:	a	c	c	d	f
t_2:	a	b	k	e	g
t_3:	a	b	c	d	h

(b) Repair S_1

	X	Y	Z	W	U
t_1:	a	**b**	c	d	f
t_2:	a	b	**c**	**d**	g
t_3:	a	b	c	d	h

(c) Repair S_2

	X	Y	Z	W	U
t_1:	a	c	c	d	f
t_2:	a	**c**	k	e	g
t_3:	a	**c**	c	d	h

Fig. 8 Inconsistent instance S and two possible repairs S_1 and S_2.

There are some situations in which inconsistencies with respect to CFDs cannot be solved by replacing values by constants of the domain. In those cases, repairs are obtained using *null*. A tuple with *null* will not create a new inconsistency with respect to an existing constraint since we assume that $t[X] \not\asymp t_p[X]$ when $t[X]$ contains *null*. For example, assume that we add the constraint $(U \to W, T_4)$, with $T_4 = \{(\_, d), (\_, e)\}$, requiring that every tuple in S should contain d *and* e in attribute W. Enforcing it is, of course, not possible, unless we satisfy the constraint by using *null*.

The problem of finding a minimal-cost repair is *coNP*-complete, but an efficient approximation algorithm, based on equivalent classes, is provided in [37]. The repair process is guided by interaction with the user, to ensure its accuracy. As a way to minimize the required feedback, it is possible to add machine learning capabilities, to learn from previous choices by the user [75]. If the cost of each update depends only on the tuple that is being updated, i.e. $cost(v, v') = w(t)$, it is possible to find a constant factor approximation of a repair when the set of constraints is fixed [61].

5 Applications of Declarative Approaches to Entity Resolution

The problem of *entity resolution* (ER) is about discovering and matching database records that represent the same entity in the application domain, i.e. detecting and solving *duplicates* [17; 41; 66]. ER is a classic, common and difficult problem in data cleaning, for which several *ad hoc* and domain-dependent mechanisms have been proposed.

ER is a fundamental problem in the context of data analysis and decision making in business intelligence. From this perspective, it becomes particularly crucial in data integration [65], and even more difficult in virtual data integration systems (VDIS). As we saw in Section 3.3, logic-based specifications of the intended solutions of a generic VDIS have been proposed, used and investigated [62; 11]. As a consequence, logic-based specifications of ER or generic approaches to ER, that could be combined with the specifications of the integration process, would be particularly relevant.

Notice that in virtual data integration systems, sources are usually not modified through the mediator. As a consequence, physical ER through the integration system is not possible. This forces us to consider as a real alternative some form of on-the-fly ER, performed at query-answering time. The declarative, logic-based approaches to ER are particularly appropriate for their amalgamation with queries and query answering processes via some sort of query rewriting.

5.1 A generic approach: Swoosh

In [7], a generic conceptual framework for entity resolution is introduced, the *Swoosh* approach. It considers a general match relation M and a general merge function, μ. In the main general formulation of Swoosh, the match relation M and the merge function μ are defined at the *record* (or tuple) level (but see [7] for some extensions). That is, when two records in a database instance are matched (found to be similar), they can be merged into a new record. This is iteratively done until the *entity resolution* of the instance is computed. Due to the merge process, some database tuples (or records) may be discarded. More precisely, the number of tuples may decrease during the ER process, because tuples that are *dominated* by others are eliminated (see below).

Swoosh views a database instance I as a finite set of records $I = \{r_1, \ldots, r_n\}$ taken from an infinite domain of records *Rec*. Relation M maps $Rec \times Rec$ into $\{true, false\}$. When two records are similar (and then could be merged), M takes the value *true*. Moreover, μ is a partial function from $Rec \times Rec$ into *Rec*. It produces the merge of two records into a new record, and is defined only when M takes the value *true*.

Given an instance I, the *merge closure* of I is defined as the smallest set of records $\bar{I}$, such that $I \subseteq \bar{I}$, and, for every two records r_1, r_2 for which $M(r_1, r_2) = true$, it holds $\mu(r_1, r_2) \in \bar{I}$. The merge closure of an instance is unique and can be obtained by adding merges of matching records until a fixpoint is reached.

Swoosh considers a general domination relationship between two records r_1, r_2, written as $r_1 \preceq_s r_2$, which means that the information in r_1 is subsumed by the information in r_2. Going one step further, we say that instance I_2 dominates instance I_1, denoted $I_1 \sqsubseteq_s I_2$, whenever every record of I_1 is dominated by some record in I_2.

For an instance I, an *entity resolution* is defined as a subset-minimal set of records I', such that $I' \subseteq \bar{I}$ and $\bar{I} \sqsubseteq_s I'$. It is shown that for every instance I, there is a unique entity resolution I' [7], which can be obtained from the merge closure by removing records that are dominated by other records.

A particularly interesting case of Swoosh occurs when the match relation M is reflexive and symmetric, and the merge function μ is idempotent, commutative, and associative. We then use the domination order imposed by the merge function, which is defined by: $r_1 \preceq_s r_2$ if and only if $\mu(r_1, r_2) = r_2$. Under these assumptions, the merge closure and therefore the entity resolution of every instance are finite [7].[4]

We can see that Swoosh's framework is generic and abstract enough to accommodate different forms of ER in different domains. Now, a still generic but special case of Swoosh, that also captures common forms of ER, is the so-called *union case*, that does do matching, merging, and merge domination at the attribute level [7]. We illustrate this case by means of an example.

Example 1. We can treat records as *objects*, i.e. as sets of attribute/value pairs. In this case, a common way of merging records is via their union, as objects. For example, consider:

$$r_1 = \{\langle \mathit{Name}, \{\text{J. Doe}\}\rangle, \langle \mathit{St.Number}, \{55\}\rangle, \langle \mathit{City}, \{\text{Toronto}\}\rangle\},$$
$$r_2 = \{\langle \mathit{Name}, \{\text{J. Doe}\}\rangle, \langle \mathit{Street}, \{\text{Grenadier}\}\rangle, \langle \mathit{City}, \{\text{Vancouver}\}\rangle\}.$$

If they are considered to be similar, e.g. on the basis of their values for attribute *Name*, they can be merged into:

$$\mu(r_1, r_2) = \{\langle \mathit{Name}, \{\text{J. Doe}\}\rangle, \langle \mathit{St.Number}, \{55\}\rangle, \langle \mathit{Street}, \{\text{Grenadier}\}\rangle, \langle \mathit{City}, \{\text{Toronto}, \text{Vancouver}\}\rangle\}. \quad \square$$

In the union-case, one obtains a single resolved instance (i.e. a single entity resolution).

Swoosh has been extended in [73] with *negative rules*. They are used to avoid inconsistencies (e.g. with respect to semantic constraints) that could be introduced by indiscriminate matching. From this point of view, certain elements of *database repairs* (cf. Section 3) are introduced into the picture (cf. [73, sec. 2.4]). In this direction, the combination of database repairing and ER is studied in [47].

[4] Finiteness is shown for the case when match and merge have the *representativity* property (equivalent to being similarity preserving) in addition to other properties. However, the proof in [7] can be modified so that representativity is not necessary.

5.2 ER with matching dependencies

Matching Dependencies (MDs) are declarative rules that generalize entity resolution (ER) in relational DBs [44; 45]. They specify attribute values that have to be made equal under certain conditions of similarity for other attribute values.

Example 2. Consider the relational schema $R_1(X,Y), R_2(X,Y)$, where X, Y are attributes (or lists of them). The following symbolic expression is a matching dependency:

$$\varphi: \; R_1[\bar{X}_1] \approx R_2[\bar{X}_2] \;\; \longrightarrow \;\; R_1[A_1] \doteq R_2[A_2]. \tag{5}$$

It says that "When in two tuples the values for attribute(s) $\bar{X}$ in R_1, R_2 are similar, the values in them for attribute(s) A must be matched/merged, i.e. made equal". □

In an MD like this R_1 and R_2 can be the same predicate if we are partially merging tuples of a same relation. The similarity relation, $\approx$, is application-dependent, associated to a particular attribute domain. It is assumed to be reflexive and symmetric.[5]

Example 3. Now consider the MD φ telling us that *"similar name and phone number $\rightarrow$ identical address"*. we apply it to the initial instance D_0 as follows:

D_0

name	*phone*	*address*
John Doe	(613)123 4567	Main St., Ottawa
J. Doe	123 4567	25 Main St.

$\Longrightarrow$

D_1

name	*phone*	*address*
John Doe	(613)123 4567	25 Main St., Ottawa
J. Doe	123 4567	25 Main St., Ottawa

We can see that, in contrast to classical and conditional dependencies, that have a "static semantics", an MD has a *dynamic semantics*, that requires a pair of databases: $(D_0, D_1) \models \varphi$. That is, when the left-hand side (LHS) of φ is satisfied in D_0, the RHS (the matching) is made true in a second instance D_1.

In this example we are using a very particular matching function (MF) that implicitly treats attribute values as objects, i.e. sets of pairs $\langle Attribute, Value\rangle$, e.g. the first value for *address* in D_0 can be seen as $\{\langle StreetName, MainSt.\rangle, \langle City, Ottawa\rangle, \langle HouseNumber, \varepsilon\rangle\}$. The MF produces the *union* of the two records as sets, and next, for a same attribute, also the union of local values. □

An MD does not tell us how to do the matching. In this regard, two alternatives have been explored in the literature. One of them treats the values in common required by the matching on the RHS essentially as an existential quantification. For example, for (5), that there is a value y for which $R_1[Y_1] = y = R_2[Y_2]$. The initial instance can be "chased" with the MDs, producing a duplicate free instance (as prescribed by the set of MDs). Desirable clean instances could be those that minimize

[5] Notice that the similarity relation in this section corresponds somehow to the match function M of Section 5.1; and matching functions we will consider here to identify two values, to the merge function μ of Section 5.1.

the number of changes of attribute values. We refer to [55; 54; 53] for details on this approach.

The second approach uses *matching functions* to provide a value in common for a matching. We briefly describe it in the following (details can be found in [13; 15]).

Given the MD (5) over a database schema, we say that the pair of instances (D,D') satisfies φ, denoted $(D,D') \models \varphi$, iff every pair of tuples for which D satisfies the antecedent but not the consequent of φ, the consequent is made true (satisfied) in D' (by matching attribute values in those tuples as prescribed by function $\mathfrak{m}$). Formally, $(D,D') \models \varphi$ when for every R_1-tuple t_1 and R_2-tuple t_2: If $t_1[\bar{X}_1] \approx t_2[\bar{X}_2]$, but $t_1[A_1] = a_1 \neq t_2[A_2] = a_2$ in D, then $t_1[A_1] = t_2[A_2] = \mathfrak{m}_A(a_1,a_2)$ in D'.

Above, $\mathfrak{m}_A$ is a binary idempotent, commutative and associative function defined on the domain in common of attributes A_1, A_2. The MF $\mathfrak{m}_A$ induces a finite semi-lattice with partial order defined by: $a \preceq_A a' :\Leftrightarrow \mathfrak{m}_A(a,a') = a'$. That is, a' dominates a when a and a' are matched into a'. Furthermore, the least-upper-bound of two elements in the lattice is obtained by applying function $\mathfrak{m}$ to them: $lub_{\preceq}\{a,a'\} = \mathfrak{m}_A(a,a')$. We usually assume the greatest-lower-bound of any two lattice elements, $glb_{\preceq}\{a,a'\}$, also exists.

MFs can be used to define a semantic domination lattice, related to information contents, as in "domain theory" [25]. The higher in the lattice, the more information contents. Figure 9 shows an information lattice at the domain level.

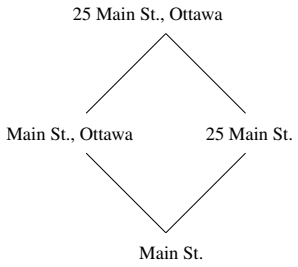

Fig. 9 A semantic-domination lattice

Now, for a set Σ of MDs, an instance D' is *stable* if $(D',D') \models \Sigma$. The idea is to obtain a stable instance by applying the MDs starting from an initial in stance D. This defines a *chase procedure* by which different chase sequences can be generated:

$$\underset{\text{dirty instance}}{D} \Longrightarrow_{\varphi_1} D_1 \Longrightarrow_{\varphi_2} D_2 \Longrightarrow_{\varphi_3} \cdots\cdots \Longrightarrow_{\varphi_n} \underset{\text{stable (clean) instance}}{D'}$$

The final (finitely terminating) results of the chase are the so-called *clean instances*. They form a class that we denote with $Clean(D,\Sigma)$. Since tuples identifiers are used, it is possible to put in correspondence a clean instance with the original instance.

The partial orders $\preceq_A$ at the attribute level can be lifted to a partial order on tuples of a same relation: $t_1 \preceq t_2 :\Longleftrightarrow t_1[A] \preceq_A t_2[A]$, for all A, which in its turn gives rise to a partial order of *semantic domination* at the instance-level (of a same relation): $D_1 \sqsubseteq D_2 :\Longleftrightarrow \forall t_1 \in D_1\ \exists t_2 \in D_2\ t_1 \preceq t_2$.

When constructing the partial order $\sqsubseteq$ we get rid of dominated tuples within a relation. This partial order is useful for comparing sets of query answers as instances of a relation. Actually, we use it to define the set, $\mathit{CleanAns}_D^{\Sigma}(\mathscr{Q})$, of *clean answers* to a query $\mathscr{Q}$ posed to a (possibly dirty) instance D that is subject to a set Σ of MDs. Intuitively, they are the answers that are invariant under the ER process: $\mathit{CleanAns}_D^{\Sigma}(\mathscr{Q}) := \mathit{glb}_{\sqsubseteq}\{\mathscr{Q}(D') \mid D' \in \mathit{Clean}(D,\Sigma)\}$. Notice that this is a lattice-dependent notion of "certain" answer.

Example 4. Consider the MD $\varphi: R[\mathit{name}] \approx R[\mathit{name}] \rightarrow R[\mathit{address}] \doteq R[\mathit{address}]$ applied to instance D_0 below, whose clean instances are D' and D''.

D_0

name	*address*
John Doe	Main St., Ottawa
J. Doe	25 Main St.
Jane Doe	25 Main St., Vancouver

D'

name	*address*
John Doe	25 Main St., Ottawa
J. Doe	25 Main St., Ottawa
Jane Doe	25 Main St., Vancouver

D''

name	*address*
John Doe	Main St., Ottawa
J. Doe	25 Main St., Vancouver
Jane Doe	25 Main St., Vancouver

For the query $\mathscr{Q}$: $\pi_{\mathit{address}}(\sigma_{\mathit{name}=\text{"J. Doe"}}(R))$, it holds: $\mathit{CleanAns}_D^{\Sigma}(\mathscr{Q}) = \{\underline{\text{25 Main St.}}\}$. □

It is possible to prove that every chase sequence terminates in polynomially many steps in the size of the original instance. The result is a clean instance (by definition) that semantically dominates the original instance. Furthermore, computing clean answers is a *coNP*-complete problem (in data complexity) [15].

5.3 Answer-set programs for MD-based ER

A natural research goal is to come up with a general methodology to 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 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.

A simple but important observation about MD-based ER (or ER in general), is that clean query answering becomes a non-monotonic process, in the sense that we can lose clean answers when the database is updated, and has to undergo a new ER process. As a consequence, the specification mentioned above must appeal to some sort of non-monotonic logical formalism. Actually, it is possible to use (disjunctive) *answer set programs* (ASPs), in such a way that the class of stable models of the "cleaning program", say $\Pi(D,\Sigma)$, corresponds to the class $\mathit{Clean}(D,\Sigma)$ of clean instances. On this basis, the clean answers to a query posed to D can be obtained via *cautious reasoning* from the program. In the following we illustrate the approach by means an extended example; details can be found in [4].

The main idea is that program $\Pi(D,\Sigma)$ implicitly simulates the chase sequences, each one represented by a model of the program. For this, $\Pi(D,\Sigma)$ has rules to: (1) Enforce MDs on pairs of tuples satisfying similarities conditions. (2) Create newer versions of those tuples by applying MFs. (c) Make the older versions of the tuples unavailable for other matchings. (d) Make each stable model correspond to *a valid chase sequence*, leading to a clean instance. This is the most intricate part.

In order to give an example of cleaning program, consider the set Σ of MDs, matching function M_B, similarity relation, and initial instance below.

$$\varphi_1\colon\ R[A] \approx R[A] \ \rightarrow\ R[B] \doteq R[B], \quad \varphi_2\colon\ R[B] \approx R[B] \ \rightarrow\ R[B] \doteq R[B].$$

$M_B(b_1,b_2,b_{12})$ $\quad a_1 \approx a_2$
$M_B(b_2,b_3,b_{23})$ $\quad b_1 \approx b_2$
$M_B(b_1,b_{23},b_{123})$

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

Enforcing Σ on D results in two chase sequences:

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

$\Rightarrow_{\varphi_1}$

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

and

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

$\Rightarrow_{\varphi_2}$

D'_1	A	B
t_1	a_1	b_1
t_2	a_2	b_{23}
t_3	a_3	b_{23}

$\Rightarrow_{\varphi_1}$

D'_2	A	B
t_1	a_1	b_{123}
t_2	a_2	b_{123}
t_3	a_3	b_{23}

Program $\Pi(D,\Sigma)$ contains:

1. For every tuple (id) $t^D = R(\bar{c})$, the fact $R'(t,\bar{c})$, i.e. we use explicit tuple IDs.

2. Rules to capture possible matchings when similarities hold for two tuples:

$$\mathit{Match}_{\varphi_1}(T_1,X_1,Y_1,T_2,X_2,Y_2) \vee \mathit{NotMatch}_{\varphi_1}(T_1,X_1,Y_1,T_2,X_2,Y_2) \leftarrow$$
$$R'(T_1,X_1,Y_1),\ R'(T_2,X_2,Y_2),\ X_1 \approx X_2,\ Y_1 \neq Y_2.$$
(similarly for $\mathit{Match}_{\varphi_2}$)

Here, T_i stands for a tuple ID. 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).

3. *Match* does not take place if one of the involved tuples used for another matching, and replaced by newer version:

$$\leftarrow\ \mathit{NotMatch}_{\varphi_1}(T_2,X_2,Y_2,T_1,X_1,Y_1),\ \mathit{not\ OldVersion}(T_1,X_1,Y_1),$$
$$\mathit{not\ OldVersion}(T_2,X_2,Y_2).$$

This is a *program constraint* filtering out models that make the body true (similarly for $\mathit{NotMatch}_{\varphi_2}$).

4. Predicate *OldVersion* is specified as containing different versions of every tuple in relation R' which has been replaced by a newer version. This is captured by *upward lattice navigation*:

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

Up to this point we have no rules to do the actual merging yet.

5. Rules to insert new tuples by merging, creating new versions:

$$R'(T_1,X_1,Y_3) \leftarrow Match_{\varphi_1}(T_1,X_1,Y_1,T_2,X_2,Y_2),\ M_B(Y_1,Y_2,Y_3).$$
$$R'(T_1,X_1,Y_3) \leftarrow Match_{\varphi_2}(T_1,X_1,Y_1,T_2,X_2,Y_2),\ M_B(Y_1,Y_2,Y_3).$$

The rules so far tell us what can be done or not in terms of matching and merging, but not exactly how to combine those possibilities. So, we need additional structure to create valid chase sequences. Since each chase sequence is an ordered sequence of instances within a partial order of instances, these ordered sequences have to be captured by additional conditions, which we do next.

6. Rules specifying a predicate *Prec* that records the relative order of matchings. It applies to two pairs of tuples, to two matchings of two tuples. For MDs $\varphi_j, \varphi_k \in \{\varphi_1, \varphi_2\}$:

$$Prec(T_1,X_1,Y_1,T_2,X_2,Y_2,T_1,X_1,Y_1',T_3,X_3,Y_3) \leftarrow Match_{\varphi_j}(T_1,X_1,Y_1,T_2,X_2,Y_2),\ Match_{\varphi_k}(T_1,X_1,Y_1',T_3,X_3,Y_3), Y_1 \preceq Y_1',\ Y_1 \neq Y_1'.$$

This predicate is better read as $Prec(\langle T_1,X_1,Y_1\rangle, \langle T_2,X_2,Y_2\rangle \mid \langle T_1,X_1,Y_1'\rangle, \langle T_3,X_3,Y_3\rangle)$, saying that the matching of the first two tuples precedes the matching of the last two. For two matchings applicable to two different versions of a tuple, *Prec* records their relative order, with matching applied to $\preceq$-smaller version first

A couple of similar rules are still required (see [4] for more details). With this definition, *Prec* could still not be an order relation, e.g. it could violate antisymmetry. Consequently, program constraints are used to make it an order, and each stable model will have a particular version of that order. That is, different orders correspond to different models, and to different chase sequences. More precisely, additional rules and program constraints are needed to make *Prec* reflexive, antisymmetric and transitive (not given here). They are used to eliminate instances (models) that result from illegal applications of MDs.

In essence, what we have done here is to define a predicate *Prec* in terms of the matching of tuples. By imposing restrictions on this predicate, we implicitly impose conditions on the matchings, that, in their turn, are the basis for the actual merging, i.e. applications of the matching functions.

7. Finally, rules are introduced to collect the latest version of each tuple, to form the clean instance, with new predicates (nicknames):

$$R^c(T_1,X_1,Y_1) \leftarrow R'(T_1,X_1,Y_1),\ not\ OldVersion(T_1,X_1,Y_1).$$

This example illustrates a general and computable methodology to produce cleaning ASPs from an instance D and a set Σ of MDs. With it we obtain a single logical specification of all chase sequences. It is possible to prove that there is a one-to-one correspondence between $Clean(D,\Sigma)$ and the stable models of $\Pi(D,\Sigma)$. More precisely, the clean instances are the restrictions to predicates R^c of the stable models.

We can use the same program $\Pi(D,\Sigma)$ to compute clean answers to different queries $\mathcal{Q}$ posed to D. The query has to be first expressed in terms of the R^c-atoms. For example, the query $\mathcal{Q}(x) : \exists x'y(R(x,y) \wedge R(x',y) \wedge x \neq x'$ becomes the rule (a simple query program) $Ans_{\mathcal{Q}}(X) \leftarrow R^c(T,X,Y),\ R^c(T',X',Y), X \neq X'$, which has to be added to the cleaning program. Clean answers can be obtained by set-theoretic

cautious reasoning from the combined program, i.e. as the intersection of the sets of answers in all stable models [4].

As we can see, the cleaning programs provide a general methodology for clean query answering. However, a natural question is whether these programs are too expressive for this task. In this regard, it is possible to verify that the syntactic structure of the cleaning programs makes them *head-cycle free*. As a consequence, they can be transformed into equivalent non-disjunctive programs, for which cautious query answering is *coNP*-complete in data [38]. This matches the intrinsic complexity of query answering (cf. Section 5.2).

As a final remark, we make notice that the clean answers were not defined via a set-theoretic intersection (as usual in ASP), but via the lattice-theoretic *glb* (cf. Section 5.2). The clean answers can still be computed via ASPs by introducing some additional rules into the query program that capture the *glb* in set-theoretic terms [4].

5.4 MDs and Swoosh

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. On the other side, MD-based ER is initially based on a set of matching dependencies, and the semantics is model-theoretic, as captured by the clean instances. However, the latter have a procedural component. A really declarative specification of MD-based ER is eventually given through the cleaning programs introduced in the previous section.

In [7], algorithms are introduced for different cases of Swoosh. One of them, instead of working at the full record level (cf. Section 5.1), considers doing the matching on the basis of values for *features*, which, consider certain combinations of attributes [7, sec. 4]. This is in some sense close to the spirit of MDs.

In [13; 15], Swoosh's union-case is reconstructed via MDs. This indicates that it is possible to apply the general methodology for writing cleaning programs presented in Section 3.1 for that case of Swoosh. Here we show instead a direct methodology for producing this kind of cleaning programs. In this way, we obtain a declarative and executable version of the Swoosh's union-case.

Actually, for each instance of this case of Swoosh, it is possible to construct a non-disjunctive stratified ASP, $\Pi^{UC}(D)$, that uses function and set terms [29], for set union and set membership. Such a program has a single stable model that corresponds to the unique resolved instance guaranteed by Swoosh, and it can be computed in polynomial time in data. Here we only give an example.

Example 5. Assume the matchings $a_1 \approx_{\underline{A}} a_2$, $a_2 \approx_{\underline{A}} a_3$ hold. Records have attributes A, B, whose values are sets of elements of the underlying domains $\underline{A}$, $\underline{B}$, resp. Here, two records matching in A are fully merged, and two set values match if there are $\underline{A}$-elements in them that match. The following is a resolution process based on the union-case:

$R(D)$

A	B
$\{a_1\}$	$\{b_1\}$
$\{a_2\}$	$\{b_2\}$
$\{a_3\}$	$\{b_3\}$

$\Rightarrow$ $R(D')$

A	B
$\{a_1, a_2\}$	$\{b_1, b_2\}$
$\{a_2, a_3\}$	$\{b_2, b_3\}$

$\Rightarrow$ $ER(D)$

A	B
$\{a_1, a_2, a_3\}$	$\{b_1, b_2, b_3\}$

The cleaning program $\Pi^{UC}(D)$ contains: (1) Facts for the initial instance D, plus $Match_{\underline{A}}(a_1, a_2)$, $a_1, a_2 \in Dom_{\underline{A}}$. (2) Rules for the merge closure of D:

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

Tuple domination is captured via subset relation:

$$Dominated_R(S_1^1, S_2^1) \leftarrow R(S_1^1, S_2^1),\ R(S_1^2, S_2^2),\ (S_1^1, S_2^1) \neq (S_1^2, S_2^2)\ (\#Union(S_1^1, S_1^2), \#Union(S_2^1, S_2^2)) = (S_1^2, S_2^2).$$

Finally, the elements of the ER are collected:

$$R^{Er}(S_1, S_2) \leftarrow R(S_1, S_2),\ not\ Dominated_R(S_1, S_2).$$ □

5.5 Rules and ontologies for duplicate detection

In the previous sections we have mostly concentrated on the merge part of ER. However, identifying similarities and duplicates is also an important and common problem [66]. There are some declarative approaches to duplicate detection. They could be naturally combined with declarative approaches to merging.

A declarative framework for collective entity matching of large data sets using domain-specific soft and hard constraints is proposed in [1]. The constraints specify the matchings. They use a novel Datalog-style language, *Dedupalog*, to write the constraints as rules. The matching 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 could be merged. They do not do the merging or base their work on MDs.

Another declarative approach to ER is presented in [69]. 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, OWL-DL and SWRL 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 [73].

6 Final Remarks

In the previous sections we have presented some recent developments in the area of declarative and generic data cleaning. The proposed solutions are general enough to be applied in different cases. Furthermore, the semantics of those approaches are formal and precise, and also executable on the basis of their logical and symbolic formulations.

In Section 5 we have restricted ourselves to some recent generic and declarative approaches to entity resolution, which is vast subject, with a solid body of research. For more details and a broader perspective of entity resolution we refer the reader to [70].

We have left out of this discussion several problems and approaches in data cleaning that can also be approached from a declarative and generic side. One of them is *data editing* [49], which is particularly crucial in census data, and can also treated with logic-based methods [51; 20]. Master data [8] are used as a reference for different data cleaning tasks, e.g. entity resolution and data editing. The use of master data in combination with data editing and database repairing has been recently investigated in [48].

We are just starting to see the inception of generic and declarative approaches to data quality assessment and data cleaning. An interesting direction to watch is the one of *ontology-based data management* [63], which should naturally lead to *ontology-based data quality*. Ontologies provide both the semantics and the contexts upon which the activities of data quality assessment and data cleaning naturally rely [14; 64].

Acknowledgments: This chapter describes research supported by the NSERC Strategic Network on Business Intelligence (BIN), NSERC/IBM CRDPJ/371084-2008, NSERC Discovery, and Bicentenario Project PSD-57. We are grateful to our research collaborators with whom part of the research described here has been carried out.

References

1. Arasu A, Ré C, Suciu D (2009) Large-scale deduplication with constraints using dedupalog. In: Proceedings of the 2009 IEEE International Conference on Data Engineering, IEEE Computer Society, ICDE '09, pp 952–963
2. Arenas M, Bertossi L, Chomicki J (1999) Consistent query answers in inconsistent databases. In: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of database systems, ACM, PODS '99, pp 68–79
3. Arenas M, Bertossi L, Chomicki J (2003) Answer sets for consistent query answering in inconsistent databases. Theory and Practice of Logic Programming 3(4):393–424
4. Bahmani Z, Bertossi L, Kolahi S, Lakshmanan LVS (2012) Declarative entity resolution via matching dependencies and answer set programs. In: Brewka G, Eiter T, McIlraith SA (eds) Proceedings of the13th International Conference on Principles of Knowledge Representation and Reasoning, AAAI Press, KR' 12, pp 380–390
5. Barceló P, Bertossi L, Bravo L (2001) Characterizing and computing semantically correct answers from databaseswith annotated logic and answer sets. In: Bertossi L, Katona GOH, Schewe KD, Thalheim B (eds) Semantics in Databases, Springer LNCS 2582, pp 7–33
6. Batini C, Scannapieco M (2006) Data Quality: Concepts, Methodologies and Techniques. Data-Centric Systems and Applications, Springer
7. Benjelloun O, Garcia-Molina H, Menestrina D, Su Q, Whang SE, Widom J (2009) Swoosh: a generic approach to entity resolution. VLDB Journal 18(1):255–276
8. Berson A, Dubov L (2010) Master Data Management and Data Governance. McGraw-Hill Osborne Media

9. Bertossi L (2006) Consistent query answering in databases. ACM SIGMOD Record 35(2):68–76
10. Bertossi L (2011) Database Repairing and Consistent Query Answering. Synthesis Lectures on Data Management, Morgan & Claypool Publishers
11. Bertossi L, Bravo L (2005) Consistent query answers in virtual data integration systems. In: Bertossi L, Hunter A, Schaub T (eds) Inconsistency Tolerance, Springer LNCS 3300, pp 42–83
12. Bertossi L, Bravo L, Franconi E, Lopatenko A (2008) The complexity and approximation of fixing numerical attributes in databases under integrity constraints. Information Systems 33(4-5):407–434
13. Bertossi L, Kolahi S, Lakshmanan LVS (2011) Data cleaning and query answering with matching dependencies and matching functions. In: Proceedings of the 14th International Conference on Database Theory, ACM, ICDT '11, pp 268–279
14. Bertossi L, Rizzolo F, Jiang L (2011) Data quality is context dependent. In: Castellanos M, Dayal U, Markl V (eds) Proceedings of the 4th International Workshop on Enabling Real-Time Business Intelligence held at VLDB 2010, Springer LNBIP 84, BIRTE 2010, pp 52–67
15. Bertossi L, Kolahi S, Lakshmanan L (2012) Data cleaning and query answering with matching dependencies and matching functions. Theory of Computing Systems pp 1–42, DOI 10.1007/s00224-012-9402-7
16. Blakeley JA, Coburn N, Larson PA (1989) Updating derived relations: Detecting irrelevant and autonomously computable updates. ACM Transactions on Database Systems 14(3):369
17. Bleiholder J, Naumann F (2008) Data fusion. ACM Computing Surveys 41(1):1–41
18. Bohannon P, Flaster M, Fan W, Rastogi R (2005) A cost-based model and effective heuristic for repairing constraints by value modification. In: Özcan F (ed) Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM, pp 143–154
19. Bohannon P, Fan W, Geerts F, Jia X, Kementsietsidis A (2007) Conditional functional dependencies for data cleaning. In: Chirkova R, Dogac A, Özsu MT, Sellis TK (eds) Proceedings of the International Conference on Data Engineering, IEEE, ICDE 2007, pp 746–755
20. Boskovitz A, Goré R, Hegland M (2003) A logical formalisation of the fellegi-holt method of data cleaning. In: Berthold MR, Lenz HJ, Bradley E, Kruse R, Borgelt C (eds) Proceedings of the 5th International Symposium on Intelligent Data Analysis, Springer LNCS 2810, IDA 2003, pp 554–565
21. Bravo L, Bertossi L (2006) Semantically correct query answers in the presence of null values. In: Proceedings of the EDBT WS on Inconsistency and Incompleteness in Databases, Springer-Verlag, Berlin, Heidelberg, EDBT'06, pp 336–357
22. Bravo L, Fan W, Ma S (2007) Extending dependencies with conditions. In: Koch C, Gehrke J, Garofalakis MN, Srivastava D, Aberer K, Deshpande A, Florescu D, Chan CY, Ganti V, Kanne CC, Klas W, Neuhold EJ (eds) Proceedings of the 33rd International Conference on Very Large Data Bases, ACM, VLDB 2007, pp 243–254
23. Bravo L, Fan W, Geerts F, Ma S (2008) Increasing the expressivity of conditional functional dependencies without extra complexity. In: Alonso G, Blakeley JA, Chen ALP (eds) Proceedings of the 24th International Conference on Data Engineering, IEEE, ICDE 2008, pp 516–525
24. Brewka G, Eiter T, Truszczynski M (2011) Answer set programming at a glance. Communications of the ACM 54(12):92–103
25. Buneman P, Jung A, Ohori A (1991) Using powerdomains to generalize relational databases. Theoretical Computer Science 91(1):23–55
26. Calì A, Lembo D, Rosati R (2003) On the decidability and complexity of query answering over inconsistent and incomplete databases. In: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of database systems, PODS '03, pp 260–271
27. Calì A, Calvanese D, De Giacomo G, Lenzerini M (2004) Data integration under integrity constraints. Information Systems 29:147-163
28. Calì A, Gottlob G, Lukasiewicz T, Marnette B, Pieris A (2010) Datalog+/-: A family of logical knowledge representation and query languages for new applications. In: Proceedings of the

25th Annual IEEE Symposium on Logic in Computer Science, IEEE Computer Society, LICS 2010, pp 228–242
29. Calimeri F, Cozza S, Ianni G, Leone N (2009) An ASP system with functions, lists, and sets. In: Erdem E, Lin F, Schaub T (eds) Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning, Springer LNCS 5753, LPNMR 2009, pp 483–489
30. Caniupan M, Bertossi L (2010) The consistency extractor system: Answer set programs for consistent query answering in databases. Data & Knowledge Engineering 69(6):545–572
31. Caroprese L, Greco S, Zumpano E (2009) Active integrity constraints for database consistency maintenance. IEEE Transactions on Knowledge and Data Engineering 21(7):1042–1058
32. Ceri S, Cochrane R, Widom J (2000) Practical applications of triggers and constraints: Success and lingering issues (10-year award). In: El Abbadi A, Brodie ML, Chakravarthy S, Dayal U, Kamel N, Schlageter G, Whang KY (eds) Proceedings of the 26th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers, VLDB 2000, pp 254–262
33. Chen W, Fan W, Ma S (2009) Incorporating cardinality constraints and synonym rules into conditional functional dependencies. Information Processing Letters 109(14):783–789
34. Chiang F, Miller RJ (2008) Discovering data quality rules. PVLDB 1(1):1166–1177
35. Chomicki J (2007) Consistent query answering: Five easy pieces. In: Schwentick T, Suciu D (eds) Proceedings of the 11th International Conference of Database Theory, Springer LNCS 4353, ICDT 2007, pp 1–17
36. Codd EF (1970) A Relational Model of Data for Large Shared Data Banks. Communications of the ACM 13(6):377–387
37. Cong G, Fan W, Geerts F, Jia X, Ma S (2007) Improving data quality: Consistency and accuracy. In: Koch C, Gehrke J, Garofalakis MN, Srivastava D, Aberer K, Deshpande A, Florescu D, Chan CY, Ganti V, Kanne CC, Klas W, Neuhold EJ (eds) Proceedings of the 33rd International Conference on Very Large Data Bases, ACM, VLDB 2007, pp 315–326
38. Dantsin E, Eiter T, Gottlob G, Voronkov A (2001) Complexity and expressive power of logic programming. ACM Computing Surveys 33(3):374–425
39. Dong XL, Tan WC (2011) Letter from the special issue editors. IEEE Data Engineering Bulletin 34(3):2
40. Eiter T, Fink M, Greco G, Lembo D (2008) Repair localization for query answering from inconsistent databases. ACM Transactions on Database Systems 33(2):10:1–10:51
41. Elmagarmid AK, Ipeirotis PG, Verykios VS (2007) Duplicate record detection: A survey. IEEE Transactions on Knowledge and Data Engineering 19(1):1–16
42. Fan W (2008) Dependencies revisited for improving data quality. In: Lenzerini M, Lembo D (eds) Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM Press, PODS 2008, pp 159–170
43. Fan W, Geerts F, Lakshmanan LVS, Xiong M (2009) Discovering conditional functional dependencies. In: Ioannidis YE, Lee DL, Ng RT (eds) Proceedings of the 25th International Conference on Data Engineering, IEEE, ICDE 2009, pp 1231–1234
44. Fan W, Jia X, Li J, Ma S (2009) Reasoning about record matching rules. PVLDB 2(1):407–418
45. Fan W, Gao H, Jia X, Li J, Ma S (2011) Dynamic constraints for record matching. VLDB Journal 20(4):495–520
46. Fan W, Geerts F, Li J, Xiong M (2011) Discovering conditional functional dependencies. IEEE Transactions on Knowledge and Data Engineering 23(5):683–698
47. Fan W, Li J, Ma S, Tang N, Yu W (2011) Interaction between record matching and data repairing. In: Sellis TK, Miller RJ, Kementsietsidis A, Velegrakis Y (eds) Proceedings of the ACM SIGMOD International Conference on Management of Data, ACM, pp 469–480
48. Fan W, Li J, Ma S, Tang N, Yu W (2012) Towards certain fixes with editing rules and master data. VLDB Journal 21(2):213–238
49. Fellegi IP, Holt D (1976) A systematic approach to automatic edit and imputation. J of the American Statistical Association 71(353):17–35
50. Flesca S, Furfaro F, Parisi F (2010) Querying and repairing inconsistent numerical databases. ACM Transactions on Database Systems 35(2):14:1–14:50

51. Franconi E, Palma AL, Leone N, Perri S, Scarcello F (2001) Census data repair: a challenging application of disjunctive logic programming. In: Nieuwenhuis R, Voronkov A (eds) Proceedings of the 8th International Conference Logic for Programming, Artificial Intelligence and Reasoning, Springer LNCS 2250, LPAR 2001, pp 561–578
52. Galhardas H, Florescu D, Shasha D, Simon E, Saita CA (2001) Declarative data cleaning: Language, model, and algorithms. In: Proceedings of the 27th International Conference on Very Large Data Bases, Morgan Kaufmann, Orlando, VLDB 2001, pp 371–380
53. Gardezi J, Bertossi L (2012) Query rewriting using datalog for duplicate resolution. In: Barceló P, Pichler R (eds) Proceedings of the Second International Workshop on Datalog in Academia and Industry, Springer LNCS 7494, Datalog 2.0, vol 7494, pp 86–98
54. Gardezi J, Bertossi L (2012) Tractable cases of clean query answering under entity resolution via matching dependencies. In: Hüllermeier E, Link S, Fober T, Seeger B (eds) Proceedings of the 6th International Conference Scalable Uncertainty Management, Springer LNCS 7520, SUM 2012, pp 180–193
55. Gardezi J, Bertossi L, Kiringa I (2012) Matching dependencies: semantics and query answering. Frontiers of Computer Science 6(3):278–292
56. Giannotti F, Pedreschi D, Saccà D, Zaniolo C (1991) Non-determinism in deductive databases. In: Delobel C, Kifer M, Masunaga Y (eds) Proceedings of the Second International Conference on Deductive and Object-Oriented Databases, Springer LNCS 566, DOOD 1991, pp 129–146
57. Golab L, Karloff HJ, Korn F, Srivastava D, Yu B (2008) On generating near-optimal tableaux for conditional functional dependencies. PVLDB 1(1):376–390
58. Golab L, Korn F, Srivastava D (2011) Efficient and effective analysis of data quality using pattern tableaux. IEEE Data Engineering Bulletin 34(3):26–33
59. Greco G, Greco S, Zumpano E (2003) A logical framework for querying and repairing inconsistent databases. IEEE Transactions on Knowledge and Data Engineering 15(6):1389–1408
60. Gupta A, Mumick IS (1995) Maintenance of materialized views: Problems, techniques and applications. IEEE Quarterly Bulletin on Data Engineering 18(2):3–18
61. Kolahi S, Lakshmanan LVS (2009) On approximating optimum repairs for functional dependency violations. In: Proceedings of the 12th International Conference on Database Theory, ACM, New York, NY, USA, ICDT '09, pp 53–62
62. Lenzerini M (2002) Data integration: a theoretical perspective. In: Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM Press, PODS 2002, pp 233–246
63. Lenzerini M (2012) Ontology-based data management. In: Freire J, Suciu D (eds) Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management, CEUR Workshop Proceedings, CEUR-WS.org, AMW 2012, vol 866, pp 12–15
64. Malaki A, Bertossi L, Rizzolo F (2012) Multidimensional contexts for data quality assessment. In: Freire J, Suciu D (eds) Proceedings of the 6th Alberto Mendelzon International Workshop on Foundations of Data Management, CEUR Workshop Proceedings, CEUR-WS.org, AMW 2012, vol 866, pp 196–209
65. Motro A, Anokhin P (2006) Fusionplex: resolution of data inconsistencies in the integration of heterogeneous information sources. Information Fusion 7(2):176–196
66. Naumann F, Herschel M (2010) An Introduction to Duplicate Detection. Synthesis Lectures on Data Management, Morgan & Claypool Publishers
67. Nicolas JM (1982) Logic for improving integrity checking in relational data bases. Acta Informatica 18(3):227–253
68. Rahm E, Do HH (2000) Data cleaning: Problems and current approaches. IEEE Data Engineering Bulletin 23(4):3–13
69. Saïs F, Pernelle N, Rousset MC (2007) L2R: A logical method for reference reconciliation. In: Proceedings of the 22nd AAAI Conference on Artificial Intelligence, AAAI Press, AAAI 2007, pp 329–334
70. Talburt J (2013) A practical guide to entity resolution with OYSTER. In: Sadiq S (ed) Handbook on Data Quality Management, Springer, Handbook Series. This volume, chapter ???.

71. ten Cate B, Fontaine G, Kolaitis PG (2012) On the data complexity of consistent query answering. In: Deutsch A (ed) Proceedings of the 15th International Conference on Database Theory, ACM, ICDT 2012, pp 22–33
72. Türker C, Gertz M (2001) Semantic integrity support in SQL:1999 and commercial (object-) relational database management systems. VLDB Journal 10(4):241–269
73. Whang SE, Benjelloun O, Garcia-Molina H (2009) Generic entity resolution with negative rules. VLDB Journal 18(6):1261–1277
74. Wijsen J (2005) Database repairing using updates. ACM Transactions on Database Systems 30(3):722–768
75. Yakout M, Elmagarmid AK, Neville J, Ouzzani M, Ilyas IF (2011) Guided data repair. PVLDB 4(5):279–289