Management of Inconsistencies in Data Integration *

Ekaterini Ioannou[1] **and Sławek Staworko**[2]

1 **Technical University of Crete, Greece**
ioannou@softnet.tuc.gr
2 **Mostrare, INRIA Lille – Nord Europe**
University of Lille 3, France
slawomir.staworko@inria.fr

Abstract

Data integration aims at providing a unified view over data coming from various sources. One of the most challenging tasks for data integration is handling the inconsistencies that appear in the integrated data in an efficient and effective manner. In this chapter, we provide a survey on techniques introduced for handling inconsistencies in data integration, focusing on two groups. The first group contains techniques for computing consistent query answers, and includes mechanisms for the compact representation of repairs, query rewriting, and logic programs. The second group contains techniques focusing on the resolution of inconsistencies. This includes methodologies for computing similarity between atomic values as well as similarity between groups of data, collective techniques, scaling to large datasets, and dealing with uncertainty that is related to inconsistencies.

1998 ACM Subject Classification H.2.m [Database Management]: Miscellaneous

Keywords and phrases Data integration, Consistent query answers, Resolution of inconsistencies

Digital Object Identifier 10.4230/DFU.Vol5.10452.217

1 Introduction

Data integration aims at providing a unified view over data coming from various sources, for example data from different applications, collections, or databases [55]. Providing efficient data integration has received considerable attention by the database community and a variety of approaches have been suggested, spanning from integrating relational databases with the same schema to integrating unstructured, highly heterogeneous data collections. One of the most challenging tasks that existing techniques for data integration focused on is the efficient handling of inconsistencies that appear in the integrated data. The focus of this survey is to present and discuss existing techniques that are able to manage/handle inconsistencies in an efficient and effective manner.

Inconsistencies in data integration can appear for various reasons. One of the most common sources is the use of different schemata and formats in the data that must be integrated. As an example, consider a scenario where we need to integrate three databases

* This research has been co-financed by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the Contrat de Projets Etat Region (CPER) 2007–2013, Codex project ANR-08-DEFIS-004, the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) – Research Funding Program: Thalis. Investing in knowledge society through the European Social Fund.

Data Exchange, Integration, and Streams. *Dagstuhl Follow-Ups*, Volume 5, ISBN 978-3-939897-61-3.
Editors: Phokion G. Kolaitis, Maurizio Lenzerini, and Nicole Schweikardt; pp. 217–235

Dagstuhl Publishing
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany

providing basic information about Muppets, i.e., the CBS trivia, the Vanity Fair magazine, and the DMV database. A fraction of the data from these databases is as follows:

CBS

Name	Job	DoB
Kermit	Manager	14.03.1965
J. Statler	Old Man	12.04.1946
Miss Piggy	Diva	21.06.1976
Gonzo	Stunman	01.03.1982

VF

Name	Job	DoB
Kermit	Manager	14 May 1965
J. Statler	Old Man	18 June 1942
Mlle Piggy	Star	1 April 1936
Gonso	Stunman	1 March 1982

DMV

Name	Job	DoB
Kermit	Manager	03/14/65
J. Statler	Old Man	06/18/42
Ms. Piggy	Diva	01/09/90
Gonzo	Daredevil	03/01/82

We can easily observe that integrating the data of these three databases causes inconsistencies. For instance, inconsistencies arise from the use of different formats that represent the dates (i.e., the DoB attributes), and the existence of spelling mistakes (i.e., in the name of Gonzo). Two additional reasons of inconsistencies are the use of variance, such as for representing "Miss Piggy", and the use of close synonyms, such as "Diva" with "Star", and "Stunman" with "Daredevil".

Modern systems, e.g., Web 2.0 applications, have introduced new challenges to handling inconsistencies, which include the use of unstructured data, and higher levels of heterogeneity. As also illustrated in the previous example, to effectively handle inconsistencies we need to consider text variations, i.e., using similar strings for the same objects. Variations in text can appear due to introduced spelling mistakes, or due to the use of acronyms (e.g., "ICDE" for 'International Conference on Data Engineering"), or abbreviations (e.g., "J. Web Sem." for "Journal of Web Semantics"). Another important source of data inconsistencies is the evolving nature of the data. In essence, as time passed, data is added, removed, or modified [69]. For example, the famous ex-lady of US was born as "Jacqueline Lee Bouvier" but this was later changed to "Jackie Kennedy" and then to "Jackie Onassis". In addition, each source providing data for integration will provide data in a way most adequate for its purpose. For instance, a publication will describe a person using the full name and affiliation, whereas an email will use the email address. This is also amplified by the lack of a global coordination for identifier assignment that forces each source to create and use its own identifiers.

In this chapter, we provide a survey on techniques introduced for handling inconsistencies in data integration, as for example the ones discussed in the previous paragraphs. More specifically, we present and discuss two group of techniques. The first group focuses on techniques for computing consistent query answers, and the second group focuses on the resolution of inconsistencies.

For the first group of techniques, we assume that the user specifies additionally a set of integrity constraints on the global schema. Because integrity constraints play an important role in the way the user formulates queries, it is essential that this information is incorporated into the processing. One easy methodology to do this is to remove from consideration any solutions that do not satisfy the integrity constraints. This naive approach may, however, easily lead to trivialization because even in very simple data integration setting, such as data merging, there is no consistent solution. Consequently, we focus on techniques for consistent

query answers that adjust the semantics of queries to alleviate the possible impact of the inconsistencies on the query answers.

The second group of techniques focuses on the resolution of inconsistencies, and in particular on detecting and merging data fragments that describe the same real-world object. In its simplest form, this involves computing the similarity and resemblance between data fragments, and then merging the data fragments that have a similarity value exceeding a predefined threshold. The whole process is performed offline, and thus at run-time, query answering is performed over the resulted merged data. A significant amount of research proposals focusing on efficiently and effectively addressing this challenge already exist. They can be found in the literature under different names, such as merge-purge [46], deduplication [71], entity identification [59], reference reconciliation [30], or entity resolution [76].

The remaining chapter is organized as follows. Section 2 presents and discusses techniques related to consistent query answers, including mechanisms for the compact representation of repairs, query rewriting, and logic programs. Section 3 techniques related to the resolution of inconsistencies, and more specifically methods for computing atomic similarity, computing similarity between groups of data, collective techniques, scaling to large datasets, and dealing with uncertainty that is related to inconsistencies. Finally, Section 4 provides conclusions.

2 Consistent query answers

In this section we discuss the framework of consistent query answers introduced by Arenas et al. in [8] to alleviate the impact of inconsistencies in a database on the quality of query answers. We begin by recalling standard database notions (Section 2.1) and the framework of consistent query answers (Section 2.2). Next, we discuss exists methods of computing consistent query answers and outline complexity results that indicate inherent challenges laying in this task (Section 2.3).

2.1 Basic notions

We recall the standard notions of relational databases [1]. We assume a fixed *database schema* $\mathcal{S}$, which is a set of relation names of fixed arity. Every relation attribute is typed but for simplicity we assume two domains only: strings and rational numbers. We define in the standard fashion the *first-order language* $\mathcal{L}$ of formulas over $\mathcal{S}$ and the usual build-in comparison predicates ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$ with their natural interpretation). A formula is: *closed* if it has no free variables, *ground* if it has no variables whatsoever, and *atomic* if it consists of one predicate only (other than the built-in predicates). In the sequel, we will denote: relation symbols by R, R_1, R_2,..., atomic formulas by A_1, A_2,..., tuples of constant by t, t_1, t_2,..., tuples of variables by $\bar{x}$, $\bar{y}$,..., and Boolean combinations of built-in predicates by φ.

A *database instance* I is a structure over $\mathcal{S}$ but often we will view I as a finite set of facts. An *integrity constraint* is any closed formula in $\mathcal{L}$. A database instance I is *consistent* with a set of integrity constraints Σ iff $I \models \Sigma$ in the standard model-theoretic way; otherwise I is *inconsistent*. We identify the following basic classes of constraints (all are closed formulas):

- *Universal constraints*: $\forall \bar{x} A_1 \wedge \ldots A_k \wedge \varphi \rightarrow A_{k+1} \vee \ldots \vee A_n$.
- *Tuple-generating dependencies*: $\forall \bar{x} A_1 \wedge \ldots A_k \wedge \varphi \rightarrow \exists \bar{y} A$. The dependency is *full* when there are no existentially quantified variables.
- *Denial constraints*: $\forall \bar{x} A_1 \wedge \ldots A_k \wedge \varphi \rightarrow \textbf{false}$.

- *Functional dependencies* (FDs): $\forall \bar{x}, \bar{y}, \bar{y}', \bar{z}, \bar{z}'.\ R(\bar{x}, \bar{y}, \bar{z}) \wedge R(\bar{x}, \bar{z}, \bar{z}') \rightarrow \bar{y} = \bar{z}$ with a more common formulation $R : X \rightarrow Y$, where X and Y are the sets of attributes corresponding respectively to $\bar{x}$ and $\bar{y}$ (and $\bar{z}$).
- *Key constraints*, a special subclass of functional dependencies: $R : X \rightarrow Y$ is a key constraint if $X \cup Y$ is the set of all attributes of R. Key constraint $R : X \rightarrow Y$ is *primary* if it the sole constraint imposed on R.
- *Inclusion dependencies* (INDs): $\forall \bar{x}, \bar{y}.\ \exists \bar{z}. R(\bar{x}, \bar{y}) \rightarrow P(\bar{y}, \bar{z})$ with a common formulation $R[Y] \subseteq P[Y']$, where Y and Y' are the sets of attributes of respectively R and P that correspond to $\bar{y}$.

A *query* is a formula of $\mathcal{L}$ and we distinguish the class of *conjunctive queries* i.e., formulas of the form $\exists \bar{x} A_1 \wedge \ldots \wedge A_k$. A tuple t is an *answer* to query q in an instance I iff $I \models q(t)$. In the sequel, we do not treat separately closed (i.e., Boolean) queries, but simply, we define **true** to be the answer of a closed query to be synonymous to the empty tuple () being the only answer to the query.

2.2 The framework of consistent query answers

The framework of consistent query answers is based on the notion of a repair of a (possibly) inconsistent database, which is essentially a consistent database instance minimally different from the original database instance. The original definition used the notion of symmetric difference between database instances to define acceptable repairs. Formally, the *symmetric difference* between two database instances I and I' is $\Delta(I, I') = (I \setminus I') \cup (I' \setminus I)$. Essentially, $\Delta(I, I')$ is the set of all facts that need to be either deleted or inserted to obtain I' from I. Now, given database instance I and two possible candidate repairs I' and I'', we use the symmetric difference to identify the candidate repair that is easier to obtain from I: essentially, I'' is closer to I than I' iff $\Delta(I, I'') \subset \Delta(I, I')$.

▶ **Definition 1.** Given a set of integrity constraints Σ and two database instances I and I', we say that I' is a *repair* of I w.r.t. Σ iff $I' \models \Sigma$ and there is no database instance I'' consistent with Σ and such that $\Delta(I, I'') \subset \Delta(I, I')$. By $Repairs_\Sigma(I)$ we denote the set of all repairs of I w.r.t. Σ. ◀

▶ **Example 2.** Take a simplified Muppet schema $Muppet(Name, Age)$ with one key constraint $\Sigma_0 = \{Muppet : Name \rightarrow Age\}$. Consider an inconsistent database

$$I_0 = \{Muppet(\text{Miss Piggy}, 36), Muppet(\text{Miss Piggy}, 86), Muppet(\text{Miss Piggy}, 26),\\ Muppet(\text{J. Statler}, 73), Muppet(\text{J. Statler}, 83), Muppet(\text{Kermit}, 43)\}.$$

I has 6 repairs w.r.t. Σ_0 that follow:

$$I_1 = \{Muppet(\text{Miss Piggy}, 36), Muppet(\text{J. Statler}, 73), Muppet(\text{Kermit}, 43)\},$$
$$I_2 = \{Muppet(\text{Miss Piggy}, 86), Muppet(\text{J. Statler}, 73), Muppet(\text{Kermit}, 43)\},$$
$$I_3 = \{Muppet(\text{Miss Piggy}, 26), Muppet(\text{J. Statler}, 73), Muppet(\text{Kermit}, 43)\},$$
$$I_4 = \{Muppet(\text{Miss Piggy}, 36), Muppet(\text{J. Statler}, 83), Muppet(\text{Kermit}, 43)\},$$
$$I_5 = \{Muppet(\text{Miss Piggy}, 86), Muppet(\text{J. Statler}, 83), Muppet(\text{Kermit}, 43)\},$$
$$I_6 = \{Muppet(\text{Miss Piggy}, 26), Muppet(\text{J. Statler}, 83), Muppet(\text{Kermit}, 43)\}.$$

◀

Intuitively, repairs represent (all) possible ways that the inconsistent database may be repaired. A consistent answer to a query is an answer that is present in every such possibility.

▶ **Definition 3.** Given an instance I, a set of integrity constraints Σ, and a query q, we say that a tuple t is a *consistent answer* to a query q in I w.r.t. Σ iff t is the answer to q in every repair of I w.r.t. Σ. ◀

Hence, if we take the query

$$q_0(x) = \exists y.\ \mathit{Muppet}(x, y) \wedge y \geq 65$$

asking for all Muppets eligible for senior discount, only J. Statler is the consistent answer to q_0 in I_0 w.r.t. Σ_0. On the other hand, *Miss Piggy* is not a consistent answer because of the repair I_1.

2.3 Computing consistent query answers

The main challenge in using the framework of consistent query answers lies in the fact that an inconsistent database may have an exponential number of repairs even for very simple sets of integrity constraints.

▶ **Example 4.** Fix $n \geq 0$ and consider a database instance over the schema $R(A, B)$:

$$I_n = \{R(1,0), R(1,1), \ldots, R(n,0), R(n,1)\}.$$

In the presence of a single key constraint $R : A \to AB$, the instance I_n has 2^n repairs. ◀

Consequently, a significant amount of research has been put into finding methods aiming to use the framework without materialization of all repairs. To identify classes of queries and integrity constraints for which this aim can be attained two basic decision problems have been proposed and their complexity studied: *consistent query answering* and *repair checking*. In virtually all research, the measure of *data complexity* has been adopted. This measure, widely adopted for relational databases [75], expresses the complexity of a problem in terms of the database size only, while the query and the integrity constraints are assumed to be fixed. The first decision problem allows to identify for which classes of queries and integrity constraints computing consistent query answers is tractable.

Consistent query answering Check whether *true* is the consistent answer to a given closed query in a given database w.r.t. to a given set of integrity constraints i.e., the complexity of the following set

$$\mathcal{D}_{\Sigma,Q} = \{I \mid \forall I' \in \mathit{Repairs}_\Sigma(I).\ I' \models Q\}.$$

We point out that the restriction to closed (Boolean) queries only does not make $\mathcal{D}_{\Sigma,Q}$ a special, simpler case of the more general problem of computing consistent query answers. Along the lines of [10] and [20], the treatment of an open query $q(\bar{x})$ can be reduced to a series of checks for closed query $q(t)$ with t ranging over some set of candidate tuples obtained by evaluating a simple derivative of $q(\bar{x})$. The second problem aims at identifying the complexity inherent to integrity maintenance.

Repair checking Check whether a database instance is a repair of a given database instance w.r.t. the given set of integrity constraints i.e., the complexity of the following set

$$\mathcal{B}_\Sigma = \{(I, I') : I' \in \mathit{Repairs}(I, \Sigma)\}.$$

This problem is a natural formulation of model checking for repairs and negative results highlight limitation of integrity enforcement mechanisms [2]. Another reason for the interest

in this problem is its close connections to the data cleaning task. Finally, if the class of integrity constraints includes inclusions dependencies, then repair checking is know to be logspace-reducible to the complement of consistent query answers [19], which makes it an alternative tool for characterizing the complexity of consistent query answering.

Several different methods for computing consistent query answers have been proposed. They can be divided into three categories: *query rewriting*, *compact representation of all repairs*, and *logic programs*. We begin by presenting the first two approaches as they yield computing consistent query answers, and the aforementioned decisions problems, tractable for applicable classes of queries and integrity constraints. Next, we summarize a number of intractability results, which essentially precludes the use of approaches from the first two categories. The solutions in the third category use logic programming, a framework know to be capable of solving even problems complete for Π_2^p, and therefore more suited for handling difficult cases of consistent query answers.

2.3.1 Compact representation of all repairs

While approaches based on compact representation of all repairs has not been historically the first one, we begin with this direction because it allows to present some useful notions and tools. The most popular approach belonging to this category is based on the notion of the conflict graph (for FDs only). First, we define the notion of a conflict: two facts $R(t_1)$ and $R(t_2)$ are *mutually conflicting* w.r.t. a functional dependency $R : X \to Y$ iff $t_1[X] = t_2[X]$ and $t_1[Y] \neq t_2[Y]$.

▶ **Definition 5** ([10])**.** Given a database instance I and a set of functional dependencies Σ, the *conflict graph* of I w.r.t. Σ is a graph $G(I, \Sigma)$ whose set of nodes is I and edges connect pairs of mutually conflicting facts in I. ◀

The conflict graph corresponding for the instance from Example 2 is presented in Figure 1.

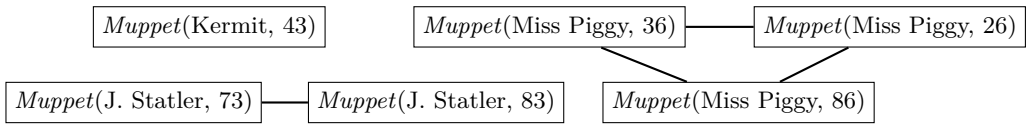

Figure 1 Conflict graph for the instance from Example 2.

The main reason for using conflict graphs lays in the simple observation that any maximal independent set of $G(I, \Sigma)$ is a repair of I w.r.t. Σ and vice versa. Let us recall that a maximal independent set of a graph is any maximal set of nodes containing no edge and note that any independent set can be extended to a maximal independent set.

The main use of conflict graph, and its variants, is to perform a repair existence check: given two sets of facts, required facts $\{A_1, \ldots, A_k\}$ and forbidden facts $\{A_{k+1}, \ldots, A_m\}$, check whether there is a repair that contains all required facts and none of the forbidden ones, i.e., a repair that satisfies the query $\Omega = A_1 \wedge A_k \wedge \neg A_{k+1} \wedge \ldots \wedge \neg A_m$. This test attempts to construct an independent set of nodes consisting of the required facts $A_1, \ldots, A_k$ and facts $B_{k+1}, \ldots, B_m$ blocking addition of facts $A_{k+1}, \ldots, A_m$, respectively. A fact B *blocks* addition of A if $\{A, B\}$ is an edge (i.e., A and B are conflicting) and thus the presence of B precludes the presence of A in the constructed instance. The test is performed by exhaustive enumeration of all combinations of edges adjacent to the forbidden facts. The test succeeds if an independent set is found, which implies the existence of a repair that satisfies Ω and

consequently does not satisfy the following (disjunctive) Boolean query:

$$\Psi = \neg\Omega = \neg A_1 \vee \ldots \vee \neg A_k \vee A_{k+1} \vee \ldots \vee A_m.$$

This implies that **true** is not a consistent answer to Ψ. This check allows to compute consistent query answers to arbitrary Boolean quantifier-free queries: if we take a Boolean quantifier-free query in CNF $\Phi = \Psi_1 \wedge \ldots \wedge \Psi_n$, then **true** is not the consistent query answer to Φ if and only if there is some Ψ_i such that **true** is not consistent query answer.

This approach has been proposed by Chomicki and Marcinkowski [19] to handle denial constraints that requires a generalization of conflict graphs to conflict hypergraps. This algorithm is the basis of the Hippo system allowing to compute consistent answers to the class of projection-free SQL queries [21, 20]. The conflict hypergraph has been further extended to handle conflicts created in the presence of universal constraints. This work has been the basis of a polynomial time repair check algorithm for sets of denial constraints, join dependencies, and acyclic sets of full tuple-generating dependencies [72].

Another compact representation of all repairs is *nucleus* [77, 78]. In this approach all repairs are represented by a tableau (a table with free variables), and queries are evaluated in the standard way (answers with variables are discarded). We note that for some classes of constraints, constructing the nucleus may, however, require time exponential in the size of the input database.

2.3.2 Query rewriting

Query rewriting was the original approach proposed to compute consistent query answers, and, in principle, it functions as follows. Given a query $q \in \mathcal{Q}$ and a set of integrity constraints Σ, we construct a query $q' \in \mathcal{Q}'$ such that for any database I evaluating q' over I yields the consistent query answers to q in I w.r.t. Σ. This approach is parametrized by the class of integrity constraints (containing Σ) and the class of queries $\mathcal{Q}$ the user can use to formulate her queries but also the class of target language for the rewritten queries. Typically, $\mathcal{Q}'$ is richer and more expressive than $\mathcal{Q}$ but the query rewriting aims at using classes of target languages that enjoy efficient query evaluation (in terms of data-complexity), and consequently, the query rewriting yields efficient means of computing consistent query. Note that the *rewritten query* q', called often the *rewritten query*, is constructed independently of the database instance.

▶ **Example 6.** Recall from Example 2 the schema $\mathit{Muppet}(\mathit{Name}, \mathit{Age})$ and the key constraint $\mathit{Muppet} : \mathit{Name} \to \mathit{Age}$, and consider the query $q_0(x) = \exists y.\ \mathit{Muppet}(x, y) \wedge y \geq 65$. Note that the key constraint written as logic formula has the following form

$$\nexists x, y, y'.\ \mathit{Muppet}(x, y) \wedge \mathit{Muppet}(x, y') \wedge y \neq y'.$$

This formulation allows to identify for a fact $\mathit{Muppet}(x, y)$ the facts, $\mathit{Muppet}(x, y') \wedge y \neq y'$, that are conflicting with $\mathit{Muppet}(x, y)$ and may be present in a repair instead of $\mathit{Muppet}(x, y)$. Consequently, we wish to know if $\mathit{Muppet}(x, y)$ satisfying the query may be replaced in some repair by a fact $\mathit{Muppet}(x, y')$ that does not satisfy the query, i.e., $\mathit{Muppet}(x, y') \wedge y \neq y' \wedge y' < 65$. Together, we obtain the rewritten query

$$q_0'(x) = \exists y.\ \mathit{Muppet}(x, y) \wedge y \geq 65 \wedge \neg(\exists y'.\ \mathit{Muppet}(x, y') \wedge y \neq y' \wedge y' < 65).$$

◀

The fact that the rewriting is constructed independently of the database instance has its strong and weak points. On the one hand, this approach has no overhead in the architecture when adapting existing applications: it suffices to replace its queries by the rewritten versions. On the other hand, rewriting introduces a next level of complexity to the queries, which may have a negative impact on the performance of the system. It is also known that there exists relational queries that are not rewritable within the class of relational queries while computing their consistent answers is tractable.

Query rewriting was the first approach proposed to compute consistent query answers [8]. It uses the notion of *residues* obtained from constraints to identify potential impact of integrity violations on the query results. The residues are used to construct rewriting rules for the atoms used in the query. This approach has been shown to be applicable to quantifier-free conjunction of literals in the presence of binary universal constraints.

Chomicki and Marcinkowski [19] observed that if the set of constraints contains one FD per relation only, the conflict graph is a disjoint union of full multipartie graphs. This simple structure allows to construct rewriting for conjunctive queries without repeated relation names and no variable sharing. They also show that relaxing the conditions imposed on the queries and constraints leads to intractability: consistent query answering becomes coNP-complete.

The result of Chomicki and Marcinkowski has been further generalized by Fuxman and Miller [37] to allow restricted variable sharing (joins) in the conjunctive queries. The class C_{forest} of allowed queries is defined using the notion of *join graph* of a query whose vertices are the literals used in the query and an edge runs from a literal R_i to literal R_j if there is a variable that occurs on a non-key attribute of R_i and any attribute of R_j (both occurrences have to be different if $i = j$). The class C_{forest} consist of queries whose join graph is a forest, the joins are full and the join conditions are non-key to key.

Fuxman et al. [36], presented the ConQuer system that computes consistent answers to queries from C_{forest}. The queries can also use aggregates, and then *range*-consistent answers are computed [10]: minimal intervals containing the set of values of the aggregate obtained over the repairs. This allows the system to compute consistent answers to 20 out of 22 queries of the TCP-H decision support benchmark. The experimental evaluation of the system shows that the system performs reasonably well and is scalable w.r.t. both the size of the database and the number of conflicts in the database.

2.3.3 Complexity results

The rewriting scheme presented in [8] renders consistent query answering polynomial for quantifier-free conjunctive queries with negative atoms in the presence of binary universal constraints, which include functional dependencies and full inclusion dependencies. In a followup work, Cali et al. [17] showed that allowing arbitrary inclusion dependencies, together with functional dependencies, leads to undecidability. This large increase in complexity comes from the fact that a violation of non-full inclusion dependencies, caused by absence of a tuple, can be repaired by inserting a tuple chosen among a possibly infinite set of tuples. Furthermore, if the set of constraints has cycles, a cascading effect can occur.

▶ **Example 7.** Consider schema consisting of one relation symbol $R(A, B)$ and one (cyclic) inclusion dependency $R[B] \subseteq R[A]$, which written as a formula is $\forall x, y.\ R(x, y) \rightarrow \exists z.\ R(y, z)$. Now, take this inconsistent instance $I_0 = \{R(0, 1)\}$. The empty instance $I'_0 = \varnothing$ is one of repairs of I_0 but also for any $n \geq 1$ so is the instance $I'_n = \{R(0, 1), R(1, 2), \ldots, R(n - 1, n), R(n, n)\}$. Hence, not only does I_0 have an infinite number of repairs but also there is no bound on their size. ◀

One way to tackle the problem of infinite choice is to consider repairs obtained by deleting facts only, a setting studied in [19]. In the previous example, this yields only the empty repair $I'_0 = \varnothing$. In this setting, the complexity of consistent query answering becomes Π^2_p-complete. Another approach proposed in [16] by Bravo and Bertossi uses a *null* value to instantiate the existentially quantified attributes in the facts to be inserted. The semantics of constraint satisfaction is adapted to the null value so that the presence of tuple with null value may satisfy the constraints but not violate it. For instance, the repairs of I_0 from the previous example obtained this way in this setting are the empty repair $I'_0 = \varnothing$ and the repair $I''_0 = \{R(0,1), R(1,null)\}$. On the one hand, the presence of $R(0,1)$ requires the presence of a fact of the form $R(1,y)$ and the fact $R(1,null)$ fits the role perfectly. On the other hand, the presence of $R(1,null)$ does not require the presence of any other fact.

There are two natural classes of constraints, universal dependencies and full tuple-generating dependencies, that similarly to full inclusion dependencies, may be violated by the absence of some tuples but repairing a violation requires choosing a tuple to insert from a finite set. A recent study by Staworko and Chomicki [72] showed that consistent query answering is Π^p_2-complete for arbitrary universal dependencies, coNP-complete for denial constraints and arbitrary full tuple-generating dependencies, and in PTIME for denial constraints, join dependencies, and acyclic full tuple-generating dependencies.

Establishing the computational complexity of consistent query answering has also served to determine the boundaries of query rewriting for consistent query answering. The data complexity of computing answers to relational queries is known to be in AC^0, a complexity class properly contained in P, and therefore, it is impossible for a relational query to express a coNP-hard problem. For instance, Chomicki and Marcinkowski have shown in [19] coNP-completeness of consistent answering to a conjunctive query in the presence of primary key constraints (i.e., one key constraint per relation), which precludes the applicability of rewriting for the full class of conjunctive queries. Because the class of conjunctive queries and the class of primary key constraints is most commonly found in practice, a considerable amount of effort has been put into finding a subclasses allowing tractable consistent query answering, e.g., Fuxman and Miller have proposed in [37] a practical subclass C_{forest} of conjunctive queries with tractable consistent query answering. This direction of research goes often together with an attempt of establishing a dichotomy for consistent query answers: essentially, finding a subclass of (conjunctive) queries containing only queries for which consistent query answering is either intractable or can be accomplished with query rewriting. An extension C^* of the class C_{forest} was believed to have this property, until very recently Wijsen has found otherwise [80]. Wijsen has also characterized sufficient and necessary conditions for first-order rewritability for a subclass acyclic conjunctive queries [79]. An interesting approach to the dichotomy question, based on structural properties of conflict graphs, is currently pursed by Pema [66].

As for repair checking, while the repair characterization based on the conflict (hyper)graph gives a PTIME repair checking for the class of denial constraints [19], adding arbitrary inclusion dependencies leads to intractability, and under the subset repair semantics (deletions only) repair checking is shown to be coNP-complete for functional dependencies and arbitrary inclusion dependencies. Various restrictions allow to bring the complexity back to PTIME, e.g., the class of functional dependencies and acyclic inclusion dependencies [19], the class of denial constraints and full tuple-generating dependencies [72], the class of weekly acyclic LAV depenencies [2], and semi-LAV dependencies [39]. Repair checking is also coNP-complete for the class of universal constraints [72].

2.3.4 Logic programs

Several different approaches have been developed to compute consistent query answers using logic programs with disjunction and classical negation [9, 11, 33, 41, 42, 74]. Approaches based on logic programs can be seen as a special case of query rewriting: essentially, we incorporate in the program that defines the original query, a special program that defines repairs. The main difference lays in the fact that evaluation of disjunctive logic programs is known to be Π_2^p-complete while query rewritting uses a target language with tractable query evaluation.

Virtually all approaches falling into the category of logic programs use disjunctive rules to model the process of repairing violations of constraints and stable models of the program correspond to the repairs of the inconsistent database. A query evaluated under the *cautious* semantics returns the answers present in every model, which naturally yields the consistent query answers.

▶ **Example 8.** Consider the schema $Muppet(Name, Age)$ from Example 2 with the key constraint $Muppet : Name \rightarrow Age$. The repairing logic program consists of the following rules:

- *Triggering* rule which identifies conflicts and specify the possible repairing actions

$$\neg Muppet'(X, Y) \vee \neg Muppet'(X, Y') \leftarrow Muppet(X, Y) \wedge Muppet(X, Y') \wedge Y \neq Y'.$$

- *Stabilizing* rule which ensures that the constructed instance is consistent

$$\neg Muppet'(X, Y) \leftarrow Muppet'(X, Y) \wedge Muppet'(X, Y') \wedge Y \neq Y'.$$

- *Persistence* rule which copies facts from the original instance unless the fact has been banned by the repairing process

$$Muppet'(X, Y) \leftarrow Muppet(X, Y) \wedge \mathbf{not}\ \neg Muppet'(X, Y).$$

Note that this program uses the classical negation $\neg$ and the negation as failure **not**. Essentially, $\neg A$ means that it is known that A is not true while **not** A captures the assertion that it is not know whether A is true (or the failure of proving that A is true).

The program above is evaluated together with the facts present in the instance and the predicates used in the query need to be interpreted accordingly, e.g., the query $q_0(x)$ becomes

$$Q_0(X) \leftarrow Muppet'(X, Y) \wedge Y \geq 65.$$

There is an one-to-one correspondence between the stable models of this program and the repairs. For instance, the stable model corresponding to the repair I_1 of the instance I_0 (Example 2) is

$$\begin{aligned}\mathcal{M}_1 = \{&Muppet(\text{Miss Piggy}, 36), Muppet(\text{Miss Piggy}, 86), Muppet(\text{Miss Piggy}, 26),\\ &Muppet(\text{J. Statler}, 73), Muppet(\text{J. Statler}, 83), Muppet(\text{Kermit}, 43),\\ &Muppet'(\text{Miss Piggy}, 36), \neg Muppet'(\text{Miss Piggy}, 86), \neg Muppet'(\text{Miss Piggy}, 26),\\ &Muppet'(\text{J. Statler}, 73), \neg Muppet'(\text{J. Statler}, 83), Muppet'(\text{Kermit}, 43),\\ &Q_0(\text{J. Statler})\}.\end{aligned}$$

◀

The main advantage of using logic programs is the generality of this approach: typically arbitrary first-order (or even Datalog$^\neg$) queries are handled in the presence of universal

constraints. Also, the repairing programs can be easily evaluated with existing logic program environments like Smodels or `dlv` [32]. We note, however, that the systems computing answers to logic programs usually perform grounding, which may be cost prohibitive if we wish to work with large databases. Another disadvantage of this approach is the fact that the class of disjunctive logic programs is known to be Π_p^2-complete.

These difficulties are addressed in the INFOMIX system [33] with several optimizations geared toward effective execution of repairing programs. One is *localization* of conflicts with identification of the *affected database* that consists of all tuples involved in constraint violations and all syntactically propagated *conflict-bound* tuples. Another optimization involves using bit-vectors to encode tuple membership to each repair and subsequent use of bitwise aggregate function to find tuples present in every repair. This optimization, however, may be insufficient to handle databases with large numbers of conflicts because typically the number of repairs is exponential in the number of conflicts.

Recently, this deficiency has been addressed with *repair factorization* [34]. Essentially, the affected database is decomposed into parts that are conflict-disjoint (no two mutually conflicting tuples are in separate parts). When computing consistent answers to a query only parts that are simultaneously spanned by the query are considered at a time. The presented experimental results validate this approach: the system computes consistent query answers in a reasonable time and is scalable w.r.t. the size of the database and the number of conflicts. Tests with up to 200^{1000} conflicts are reported.

3 Resolution of Inconsistencies

In this section, we present and discuss techniques that can be used for the resolution of inconsistencies. More specifically, we focus on inconsistencies arising from the use of different **representations** for describing the same real-world object, for example the same conference, person, or location. The techniques we present here aim at detecting such representations. Once detected, the representations with a similarity higher than a predefined threshold are merged together. The final results are used for replacing the original representations in the integrated data, and thus, query processing is performed over the merged data.

The following paragraphs present the techniques for resolution of inconsistencies grouped into five categories according to the data included in the representation (that are used during the processing): (i) atomic similarity techniques for comparing representations that are strings (Section 3.1); (ii) similarity techniques for comparing representations corresponding to groups of data (Section 3.2); (iii) collective techniques that also use inner-relationships between representations (Section 3.3); (iv) techniques for scaling the processing to datasets of large sizes (Section 3.4); and (v) dealing with the uncertainty that is related to the inconsistencies (Section 3.5).

Additional information related to existing techniques in this domain, can be found in surveys [28, 38, 35] and tutorials [54, 44].

3.1 Atomic Similarity Techniques

This category includes techniques that compute similarity when the representations are either a single word, or a small sequence of words. Few examples of representations for this category are: r_1="John D. Smith", r_2="J. D. Smith", r_3="Transactions on Knowledge and Data Engineering", and r_4= "IEEE Trans. Knowl. Data Eng.". As already discussed in Section 1, such differences in representations (i.e., single words or sequence of words) are a common situation that is typically resulted from misspellings, or naming variants due to the use of

abbreviations, acronyms, etc. The merging of two such representations (e.g., "John D. Smith" with "J. D. Smith") is performed when the technique detects high resemblance between the text values composing the representations.

The first group of techniques that belong to the category of atomic similarity techniques are based on the characters composing the string. These techniques compute the similarity between two representations (i.e., strings) as a cost that indicates the total number of the operations needed to convert the string of the first representation to the string of the second representation. The basic method of edit distance, named Levenshtein distance [56], counts the number of character deletions, additions, or modifications that are required for converting the first to the second string. The variations of this technique extends it with additional aspects, such as operation cost depending on the character's location, consideration of additional operations, including open gap, and extend gap [60]. Jaro [49] computes similarity by considering the overlapping characters in the two strings along with their locations. It suitable to small strings, for instance first and last names. An extension of this technique is the Jaro-Winkler [81]. This technique gives higher weight to the prefix (i.e., first characters) of the string, and thus it increases the applicability of this approach to person names.

A second group of techniques are the ones that compute the similarity between collections of words. The basic techniques from this group are the Jaccard similarity coefficient, and the TF/IDF similarity [70]. Fuzzy matching similarity [18] is another technique of this category. It is a generalized edit distance similarity that combines transformation operations with edit distance techniques. Another method is the Soundex similarity. The Soundex method converts each word into a phonetic encoding by assigning the same code to the string parts that sound the same. The similarity between two words is then calculated as the difference between the corresponding phonetic encodings of these words. Finally, [23] and [15] describe and discuss an experimental comparison of various basic similarity techniques used for matching names.

Although, the existing techniques are successful in identifying similar representations, the idea of merging representations based on their string similarity is only partly correct, since the objects to which the context of these representations refer is totally ignored. For example, consider two representations for people with the exact same name. Using a similarity technique from this category would result in incorrectly merging the representation of these people. For this reason, these representations are typically used only as part of the initial steps of more sophisticated representations, in order to identify *potential merges*, which can be then further processed.

3.2 Computing Similarity between Groups of Data

In contrast to the previous category, the techniques of this category focus on dealing with representations that are composed by a group of data. Few examples of representations for this category are: r_1={"John D. Smith", "male", "United States of America"}, and r_2= {"J. D. Smith", "male", "USA"}. They extend techniques of the previous category since they combine basic string similarity with more complicated methodologies.

The first group of techniques for this category are those that consider the data of each tuple (i.e., record) as the representation. The approaches suggested in [53] and [22] concatenate all data composing each tuple and create a string. These strings are then compare using one of the string similarity techniques (Section 3.1). One of the most known techniques of this category is the merge-purge [46], aiming in identifying whether two relational records refer to the same real-world object. Merge-purge considers every database relation (i.e., record) as a representation. This approach first sorts the relations using the different available column

names, and then uses the sorting to easy compare between similar information. The merging of records is performed according to the found resemblances.

The techniques proposed in [73] and [29] aim at matching representations by discovering possible mappings from one representation to another representation. More specifically, in [73] a mapping is identified by applying a collection of *transformations*, such as abbreviation, stemming, and initials. For the same purpose, Doan et al. [29] apply *profilers*, which are described as predefined rules with knowledge about specific representations. Profilers are created by various sources, such as domain experts, learned from training data, or constructed from external data.

Cohen et al. [24] use techniques for string similarity (presented in the previous category) to create techniques to adaptively modify the document similarity metrics. Li et al. [57] also focus in handling multiple types of representations, addressing the problem as this appears in the context of the text documents.

3.3 Collective Techniques

This category includes techniques that identify matches between two representations by using not only the information available in the specific representations but also related information from other representations. In particular, these techniques discover and exploit the inner-relationships that exist among all representations of the given data collection. These inner-relationships can be seen as links, or associations, between the representations and parts of the representation data. As an example consider co-authorship in publications, which is widely used by collective approaches. By knowing that a publication has α, β, and γ as authors, and another publication has β', and γ as authors, we can increase our belief that β describes the same author as β'. Thus, we now have two sources for computing the belief we have that authors β and β' describe the same real-world object: the first is that their strings are similar (computed using a technique for Sections 3.1-3.2), and the second is that both authors have a publication with γ author.

To capture the inner-relationships found inside a data collection, the techniques of this category model the collection into an intermediary structure. For instance, the technique in [6] uses dimensional hierarchies, and the techniques introduced in [13] and [52] use graphs. Ananthakrishna et al. [6] exploit dimensional hierarchies to detect fuzzy duplicates in dimensional tables. The hierarchies are build by following the links between the data from one table to data other tables. Representations are matched when the information along these generated hierarchies is found similar. Getoor et al. [13, 14] model the metadata as a graph structure. The nodes in this graph correspond to the information describing the representations, and edges are the inner-relationships between representations. The technique uses the edges from the graphs to cluster the nodes, and the clusters detected are then used to identify the common representations.

In [52, 51], the data collection is also modeled as a graph following a similar methodology as the previous methods. These techniques also generate other possible relationships to represent the candidate matches between representations. The additional relationships became edges that enhance the generated graph. Then, graph theoretic techniques are applied for analyzing the relationships in the graph and deciding the possible matches between representations. Other techniques follow a different methodology to create their internal supportive structures. In [65], the nodes represent the possible matches between two representations (and not one node representing one representation) and the edges the inner-relationships between the possible representation matches. The relationships from the structure are then used to decide the existence of nodes (matches between representations),

and information encapsulated in identified matches is propagated to the rest of the structure.

Some of the proposed techniques of this category are from the area of metadata management. The TAP system [43] uses a process named *Semantic Negotiation* to identify common representations (if any) between the different resources. These common representations are used to create a unified view of the data. Benjelloun et al. [12] identify the different properties on which the efficiency of such a technique depends on, and introduce different techniques to address the possible combinations of the found properties.

Another well-know technique is the *Reference Reconciliation* [30]. Here, the authors begin their computation by identifying possible associations between representations by comparing their corresponding data. The information encoded in the found associations is propagated to the rest of the representations in order to enrich their information and improve the quality of final results. The approach in [5] is a modified version of the reference reconciliation algorithm that is focused on detecting conflict of interests in paper reviewing processes. The approach introduced in [48] models the resolution-related information a Bayesian network, and uses probabilistic inference for computing the probabilities of representation matches and for propagating the information between matching.

3.4 Scaling to Large Datasets

As noted in [35], applying processing to datasets of a large size can be achieved through data blocking, i.e., instead of comparing each representation with all other representations, the representations are separated into blocks, and only the representations of the same block are compared. The challenge is to create blocks of representations that are most likely to refer to the same real-world objects. The majority of the proposed techniques typically associate each representation with a *Blocking Key Value* (BKV) summarizing the values of selected attributes and then operate exclusively based on the BKVs.

For instance, the Sorted Neighborhood technique [45], sorts blocks according to their BKV and then slides a window of fixed size over them, comparing the representations it contains. The StringMap techniques [50] maps the BKV of each representation to a multi-dimensional Euclidean space, and employs suitable data structures for efficiently identifying pairs of similar representations. Alternatively, the q-grams based blocking presented in [40] builds overlapping clusters of representations that share at least one q-gram (i.e., sub-string of length q) of their BKV. Canopy clustering [58] employs a cheap string similarity metric for building high-dimensional overlapping blocks, whereas the Suffix Arrays technique, coined in [4] and enhanced in [27], considers the suffixes of the BKV instead. The technique in [62] introduces a mechanism for eliminating the redundancy of blocking methods by removing superfluous comparisons.

More recently introduced techniques based on blocking focused not only on scaling the resolution process to large datasets, but also on capturing additional issues related to resolution. Papadakis et al. [61, 63, 64] investigated how to apply the blocking mechanism on heterogeneous semi-structured data with loose schema binding. Among other, the authors introduce an attribute-agnostic mechanism for generating the blocks, and explain how efficiency can be improved through scheduling the order of block processing and identifying when to stop the processing. The approach introduced by Whang et al. [76], iteratively processes blocks in order to use the results of one block when processing other blocks, and thus include the advantages illustrated by collective approaches (i.e., discussed in Section 3.3). The idea of iteratively block processing was also studied in [67], which provided a principled framework with message passing algorithms for generating a global solution for the resolution over the complete collection.

3.5 Dealing with Uncertainty related to Inconsistencies

Uncertain data management approaches deal with a variation of inconsistency resolution. More specifically, they consider the existence of probabilities that model the belief related to the inconsistencies. For example, [68, 26] considers the existence of more than one representations (modeled as relational relations) for the same real-world object. Thus, for each real-world object the database contains a small set of possible-alternative representations, with each representation accompanied by a probability that indicates the belief we have that this is the correct representation.

The approach suggested by the Trio system [3] focuses on creating a database that support uncertainty along with inconsistency and lineage, while also dealing with duplicate tuples, i.e., representations. Dalvi and Suciu [25] follow the "possible worlds" semantics to introduce query processing for independent probabilistic data that model alternative matches between representations, and introduced a methodology for efficiently evaluating queries.

Dong et al. [31] investigate the use of the probabilistic mappings between the attributes of the contributing sources with a mediated schema. Applying this method on representations would have considered the possible mappings between the attribute names as given by contributing sources with a mediated schema S. This means that an attribute of representations α, β, and γ is mapped to an attribute from S with a probability to show the uncertainty of each mapping. The authors explain how answering queries over the mediated schema S can be performed using these mappings.

Andritsos et al. [7] do not focus on the schema information, as the approach presented in [31], but on the actual data. The authors assume that the duplicate tuples for each representation are given, for example as the results computed by a technique from Sections 3.1-3.3. Thus, all tuples describing alternative representations of the same representation have the same identifier. The tuples of the alternative representations are considered as disjoined, which means that only one tuple for each identifier can be part of the final resulted representation. The approach in [47] addresses more challenges of heterogeneous data. In particular, this approach does not assume that the alternative representations of representations are known, but that an representation collection comes with a set of possible linkages between representations. Each linkage represents a possible match between two representations and is accompanied with a probability that indicates the belief we have that the specific representations are for the same real-world object. Representations are compiled on-the-fly, by effectively processing the incoming query over representations and linkages, and thus, query answers reflect the most probable solution for the specific query.

4 Conclusions

In this chapter we elaborated on the management of inconsistencies in data integration. More specifically, we presented and discussed two group of techniques: (i) computing consistent query answers, focusing on mechanisms for the compact representation of repairs, query rewriting, and logic programs; and (ii) resolution of inconsistencies, focusing on methods for computing similarity between atomic values or groups of data, collective techniques, scaling to large datasets, and dealing with uncertainty that is related to inconsistencies.

References

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

2 F. Afrati and P. Kolaitis. Repair checking in inconsistent databases: Algorithms and complexity. In *ICDT*, pages 31–41, 2009.

3 P. Agrawal, O. Benjelloun, A. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, pages 1151–1154, 2006.

4 A. Aizawa and K. Oyama. A fast linkage detection scheme for multi-source information integration. In *WIRI*, pages 30–39, 2005.

5 B. Aleman-Meza, M. Nagarajan, C. Ramakrishnan, L. Ding, P. Kolari, A. Sheth, I. Arpinar, A. Joshi, and T. Finin. Semantic analytics on social networks: Experiences in addressing the problem of conflict of interest detection. In *WWW*, pages 407–416, 2006.

6 R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB*, pages 586–597, 2002.

7 P. Andritsos, A. Fuxman, and R. Miller. Clean answers over dirty databases: A probabilistic approach. In *ICDE*, 2006.

8 M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79, 1999.

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

10 M. Arenas, L. Bertossi, J. Chomicki, X. He, V. Raghavan, and J. Spinrad. Scalar aggregation in inconsistent databases. *Theoretical Computer Science (TCS)*, 296(3):405–434, 2003.

11 P. Barcelo and L. Bertossi. Logic programs for querying inconsistent databases. In *PADL*, pages 208–222, 2003.

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

13 I. Bhattacharya and L. Getoor. Deduplication and group detection using links. In *LinkKDD*, 2004.

14 I. Bhattacharya and L. Getoor. Iterative record linkage for cleaning and integration. In *DMKD*, pages 11–18, 2004.

15 M. Bilenko, R. Mooney, W. Cohen, P. Ravikumar, and S. Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Systems*, 18(5):16–23, 2003.

16 L. Bravo and L. E. Bertoss. Semantically correct query answers in the presence of null values. In *IIDB Workshop co-located with EDBT*, pages 336–357, 2006.

17 A Cali, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *PODS*, pages 260–271, 2003.

18 S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, pages 313–324, 2003.

19 J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197(1-2):90–121, February 2005.

20 J. Chomicki, J. Marcinkowski, and S. Staworko. Computing consistent query answers using conflict hypergraphs. In *CIKM*, pages 417–426, 2004.

21 J. Chomicki, J. Marcinkowski, and S. Staworko. Hippo: A system for computing consistent answers to a class of SQL queries. In *EDBT*, pages 841–844, 2004.

22 W. Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Transactions on Information Systems (TOIS)*, 18(3):288–321, 2000.

23 W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string distance metrics for name-matching tasks. In *IIWeb co-located with IJCAI*, pages 73–78, 2003.

24 W. Cohen and J. Richman. Learning to match and cluster entity names. In *MF/IR Workshop co-located with SIGIR*, 2001.

25 N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB Journal*, 16(4):523–544, 2007.

26 N. Dalvi and D. Suciu. Management of probabilistic data: foundations and challenges. In *PODS*, pages 1–12, 2007.

27 T. de Vries, H. Ke, S. Chawla, and P. Christen. Robust record linkage blocking using suffix arrays. In *CIKM*, pages 305–314, 2009.

28 A. Doan and A. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine*, 26(1):83–94, 2005.

29 A. Doan, Y. Lu, Y. Lee, and J. Han. Object matching for information integration: A profiler-based approach. In *IIWeb co-located with IJCAI*, pages 53–58, 2003.

30 X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, pages 85–96, 2005.

31 X. Dong, A. Halevy, and C. Yu. Data integration with uncertainty. In *VLDB*, pages 687–698, 2007.

32 T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative problem-solving in dlv. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Springer, 2001.

33 T. Eiter, M. Fink, G. Greco, and D. Lembo. Efficient evaluation of logic programs for querying data integration systems. In *ICLP*, pages 163–177, 2003.

34 T. Eiter, M. Fink, G. Greco, and D. Lembo. Repair localization for query answering from inconsistent databases. Technical Report 1843-07-01, Institut Fur Informationssysteme, Technische Universitat Wien, 2007.

35 A. Elmagarmid, P. Ipeirotis, and V. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(1):1–16, 2007.

36 A. Fuxman, E. Fazli, and R. J. Miller. Conquer: Efficient management of inconsistent databases. In *SIGMOD*, pages 155–166, 2005.

37 A. Fuxman and R. J. Miller. First-order query rewriting for inconsistent databases. In *ICDT*, pages 335–349, 2005.

38 L. Getoor and C. Diehl. Link mining: a survey. *SIGKDD Explorations*, 7(2):3–12, 2005.

39 G. Grahne and A. Onet. Data correspondence, exchange and repair. In *ICDT*, pages 219–230, 2010.

40 L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

41 G. Greco, S. Greco, and E. Zumpano. A logic programming approach to the integration, repairing and querying of inconsistent databases. In *ICLP*, pages 348–364, 2001.

42 G. Greco, S. Greco, and E. Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(6):1389–1408, 2003.

43 R. Guha and R. McCool. TAP: a semantic web platform. *Computer Networks*, 42(5):557–577, 2003.

44 O. Hassanzadeh, A. Kementsietsidis, and Y. Velegrakis. Data management issues on the semantic web. In *ICDE*, pages 1204–1206, 2012.

45 M. Hernández and S. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, pages 127–138, 1995.

46 M. Hernández and S. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

47 E. Ioannou, W. Nejdl, C. Niederée, and Y. Velegrakis. On-the-fly entity-aware query processing in the presence of linkage. *PVLDB*, 3(1):429–438, 2010.

48 E. Ioannou, C. Niederée, and W. Nejdl. Probabilistic entity linkage for heterogeneous information spaces. In *CAiSE*, pages 556–570, 2008.

49 M. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *American Statistical Association*, 84, 1989.

50 L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. In *DASFAA*, 2003.

51 D. Kalashnikov and S. Mehrotra. Domain-independent data cleaning via analysis of entity-relationship graph. *ACM Transactions on Database Systems (TODS)*, 31(2):716–767, 2006.

52 D. Kalashnikov, S. Mehrotra, and Z. Chen. Exploiting relationships for domain-independent data cleaning. In *SIAM SDM*, 2005.

53 N. Koudas, A. Marathe, and D. Srivastava. Flexible string matching against large databases in practice. In *VLDB*, pages 1078–1086, 2004.

54 N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, pages 802–803, 2006.

55 M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

56 V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

57 X. Li, P. Morie, and D. Roth. Semantic integration in text: From ambiguous names to identifiable entities. *AI Magazine*, 26(1):45–58, 2005.

58 A. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *KDD*, pages 169–178, 2000.

59 A. Morris, Y. Velegrakis, and P. Bouquet. Entity identification on the semantic web. In *SWAP*, 2008.

60 G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

61 G. Papadakis, E. Ioannou, C. Niederée, and P. Fankhauser. Efficient entity resolution for large heterogeneous information spaces. In *WSDM*, pages 535–544, 2011.

62 G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, and W. Nejdl. Eliminating the redundancy in blocking-based entity resolution methods. In *JCDL*, pages 85–94, 2011.

63 G. Papadakis, E. Ioannou, C. Niederée, T. Palpanas, and W. Nejdl. Beyond 100 million entities: large-scale blocking-based resolution for heterogeneous data. In *WSDM*, pages 53–62, 2012.

64 G. Papadakis, E. Ioannou, T. Palpanas, C. Niederée, and W. Nejdl. A blocking framework for entity resolution in highly heterogeneous information spaces. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, (to appear).

65 Parag and P. Domingos. Multi-relational record linkage. In *MRDM Workshop co-located with KDD*, pages 31–48, 2004.

66 E. Pema. On the tractability ond intractability of consistent conjunctive query answering. In *Ph.D. Workshop co-located with EDBT/ICDT*, 2011.

67 V. Rastogi, N. Dalvi, and M. Garofalakis. Large-scale collective entity matching. *PVLDB*, 4(4):208–218, 2011.

68 C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, pages 886–895, 2007.

69 F. Rizzolo, Y. Velegrakis, J. Mylopoulos, and S. Bykau. Modeling concept evolution: A historical perspective. In *ER*, pages 331–345, 2009.

70 G. Salton and M. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc., New York, NY, USA, 1986.

71 S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *KDD*, pages 269–278, 2002.

72 S. Staworko and J. Chomicki. Consistent query answers in the presence of universal constraints. *Information Systems*, 35(1):1–22, 2010.

73 S. Tejada, C. Knoblock, and S. Minton. Learning domain-independent string transformation weights for high accuracy object identification. In *KDD*, pages 350–359, 2002.

74 D. Van Nieuwenborgh and D. Vermeir. Preferred answer sets for ordered logic programs. In *JELIA*, pages 432–443, 2002.

75 M. Vardi. The complexity of relational query languages. In *STOC*, pages 137–146, 1982.

76 S. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD*, pages 219–232, 2009.

77 J. Wijsen. Condensed representation of database repairs for consistent query answering. In *ICDT*, pages 378–393, 2003.

78 J. Wijsen. Database repairing using updates. *TODS*, 30(3):722–768, 2005.

79 J. Wijsen. On the first-order expressibility of computing certain answers to conjunctive queries over uncertain databases. In *PODS*, pages 179–190, 2010.

80 J. Wijsen. A remark on the complexity of consistent conjunctive query answering under primary key violations. *Information Processing Letters*, 110(21):950–955, 2010.

81 W. Winkler. The state of record linkage and current research problems, 1999.