Decomposing and Pruning Primary Key Violations from Large Data Sets*
(discussion paper)

Marco Manna, Francesco Ricca, and Giorgio Terracina

DeMaCS, University of Calabria, Italy
{manna,ricca,terracina}@mat.unical.it

Abstract. The problem of computing the certain answer to a conjunctive query over a relational instance subject to primary key constraints is a classical hard problem in database research. On the theoretical side, we present a decomposition and pruning strategy that reduces, in polynomial time, the original problem to a collection of smaller problems of the same sort that can be solved independently. From a practical perspective, we discuss an experiment on large data sets that shows the effectiveness of the overall technique and its implementation in ASP.

1 Introduction

Integrity constraints provide means for ensuring that database evolution does not result in a loss of consistency or in a discrepancy with the intended model of the application domain. A relational database that do not satisfy some of these constraints is said to be inconsistent. In practice it is not unusual that one has to deal with inconsistent data [5], and when a conjunctive query (CQ) is posed to an inconsistent database, a natural problem arises that can be formulated as: *How to deal with inconsistencies to answer the input query in a consistent way?* This is a classical problem in database research and different approaches have been proposed in the literature. One possibility is to clean the database [7] and work on one of the possible coherent states; another possibility is to be tolerant of inconsistencies by leaving intact the database and computing answers that are "consistent with the integrity constraints" [2].

In this paper, we adopt the second approach – which has been proposed by [2] under the name of *consistent query answering* (CQA) – and focus on the relevant class of *primary key* constraints. Formally, in our setting: (1) a database D is *inconsistent* if there are at least two tuples of the same relation that agree on their primary key; (2) a *repair* of D is any maximal consistent subset of D; and (3) a tuple $\mathbf{t}$ of constants is in the *consistent answer* to a CQ q over D

* The paper —overviewing the results presented in [13]— has been partially supported by the Italian Ministry for Economic Development (MISE) under project "PIUCultura – Paradigmi Innovativi per l'Utilizzo della Cultura" (n. F/020016/01-02/X27), and under project "Smarter Solutions in the Big Data World (S2BDW)".

if and only if, for each repair R of D, tuple $\mathbf{t}$ is in the (classical) answer to q over R. Intuitively, the original database is (virtually) repaired by applying a minimal number of corrections (deletion of tuples with the same primary key), while the consistent answer collects the tuples that can be retrieved in every repaired instance.

CQA under primary keys is coNP-complete in data complexity [3], when both the relational schema and the query are considered fixed. Due to its complex nature, traditional RDBMs are inadequate to solve the problem alone via SQL without focusing on restricted classes of CQs [2, 8, 14, 15, 11]. Actually, in the unrestricted case, CQA has been traditionally dealt with logic programming [3, 4, 9, 12]. However, it has been argued [10] that the practical applicability of logic-based approaches is restricted to data sets of moderate size. Only recently, an approach based on Binary Integer Programming [10] has revealed good performances on large databases (featuring up to one million tuples per relation) with primary key violations.

In this paper, we show that logic programming can still be effectively used for computing consistent answers over large relational databases. We describe a decomposition strategy that reduces (in polynomial time) the computation of the consistent answer to a CQ over a database subject to primary key constraints into a collection of smaller problems of the same sort. At the core of the strategy is a cascade pruning mechanism that dramatically reduces the number of key violations that have to be handled to answer the query. Moreover, we implement the new strategy using Answer Set Programming (ASP) [6], and we prove empirically the effectiveness of our ASP-based approach on existing benchmarks from the database world. The experiment empirically demonstrate that our approach is efficient on large data sets, and can even perform better than state-of-the-art methods.

2 The Framework

We are given two disjoint countably infinite sets of *terms* denoted by $\mathbf{C}$ and $\mathbf{V}$ and called *constants* and *variables*, respectively. We denote by $\mathbf{X}$ sequences of variables $X_1, \ldots, X_n$, and by $\mathbf{t}$ sequences of terms $t_1, \ldots, t_n$. We also denote by $[n]$ the set $\{1, \ldots, n\}$, for any $n \geqslant 1$.

A (*relational*) *schema* is a triple $\langle \mathcal{R}, \alpha, \kappa \rangle$ where $\mathcal{R}$ is a finite set of *relation symbols* (or *predicates*), $\alpha : \mathcal{R} \rightarrow \mathbb{N}$ is a function associating an *arity* to each predicate, and $\kappa : \mathcal{R} \rightarrow 2^{\mathbb{N}}$ is a function that associates, to each $r \in \mathcal{R}$, a nonempty set of positions from $[\alpha(r)]$, which represents the *primary key* of r. Moreover, for each relation symbol $r \in \mathcal{R}$ and for each position $i \in [\alpha(r)]$, $r[i]$ denotes the i-th *attribute* of r. Throughout, let $\Sigma = \langle \mathcal{R}, \alpha, \kappa \rangle$ denote a relational schema. An *atom* (over Σ) is an expression of the form $r(t_1, \ldots, t_n)$, where $r \in \mathcal{R}$, and $n = \alpha(r)$. An atom is called a *fact* if all of its terms are constants of $\mathbf{C}$. Conjunctions of atoms are often identified with the sets of their atoms. For a set A of atoms, the variables occurring in A are denoted by $var(A)$. A *database* D (over Σ) is a finite set of facts over Σ. Given an atom $r(\mathbf{t}) \in D$, we denote

by $\hat{\mathbf{t}}$ the sequence $\mathbf{t}|_{\kappa(r)}$. We say that D is *inconsistent* (w.r.t. Σ) if it contains two different atoms of the form $r(\mathbf{t}_1)$ and $r(\mathbf{t}_2)$ such that $\hat{\mathbf{t}}_1 = \hat{\mathbf{t}}_2$. Otherwise, it is *consistent*. A *repair* R of D (w.r.t. Σ) is any maximal consistent subset of D. The set of all the repairs of D is denoted by $rep(D, \Sigma)$. A *substitution* is a mapping $\mu : \mathbf{C} \cup \mathbf{V} \to \mathbf{C} \cup \mathbf{V}$ which is the identity on $\mathbf{C}$. Given a set A of atoms, $\mu(A) = \{r(\mu(t_1), \ldots, \mu(t_n)) \ : \ r(t_1, \ldots, t_n) \in A\}$. The restriction of μ to a set $S \subseteq \mathbf{C} \cup \mathbf{V}$, is denoted by $\mu|_S$. A *conjunctive query* (CQ) q (over Σ) is an expression of the form $\exists \mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y})$, where $\mathbf{X} \cup \mathbf{Y}$ are variables of $\mathbf{V}$, and φ is a conjunction of atoms (possibly with constants) over Σ. To highlight the free variables of q, we often write $q(\mathbf{X})$ instead of q. If $\mathbf{X}$ is empty, then q is called a *Boolean conjunctive query* (BCQ). Assuming that $\mathbf{X}$ is the sequence $X_1, \ldots, X_n$, the *answer* to q over a database D, denoted $q(D)$, is the set of all n-tuples $\langle t_1, \ldots, t_n \rangle \in \mathbf{C}^n$ for which there exists a substitution μ such that $\mu(\varphi(\mathbf{X}, \mathbf{Y})) \subseteq D$ and $\mu(X_i) = t_i$, for each $i \in [n]$. A BCQ is *true* in D, denoted $D \models q$, if $\langle\rangle \in q(D)$. The *consistent answer* to a CQ $q(\mathbf{X})$ over a database D (w.r.t. Σ), denoted $ans(q, D, \Sigma)$, is the set of tuples $\bigcap_{R \in rep(D,\Sigma)} q(R)$. Clearly, $ans(q, D, \Sigma) \subseteq q(D)$ holds. A BCQ q is *consistently true* in a database D (w.r.t. Σ), denoted $D \models_\Sigma q$, if $\langle\rangle \in ans(q, D, \Sigma)$.

3 Dealing with Large Datasets

We present a strategy suitable for computing the consistent answer to a CQ over an inconsistent database subject to primary key constraints. The new strategy reduces in polynomial time that problem to a collection of smaller ones of the same sort. Given a database D over a schema Σ, and a BCQ q, we identify a set $F_1, \ldots, F_k$ of pairwise disjoint subsets of D, called fragments, such that: $D \models_\Sigma q$ *iff there is* $i \in [k]$ *such that* $F_i \models_\Sigma q$. At the core of our strategy we have: (1) a cascade pruning mechanism to reduce the number of "crucial" inconsistencies, and (2) a technique to identify a suitable set of fragments from any (possibly unpruned) database. For the sake of presentation, we start with principle (2). (Proofs are given in [13].)

Given a database D, a *key component* K of D is any maximal subset of D such that if $r_1(\mathbf{t}_1)$ and $r_2(\mathbf{t}_2)$ are in K, then both $r_1 = r_2$ and $\hat{\mathbf{t}}_1 = \hat{\mathbf{t}}_2$ hold. Namely, K collects only atoms that agree on their primary key. Hence, the set of all key components of D, denoted by $comp(D, \Sigma)$, forms a partition of D. If a key component is a singleton, then it is called *safe*; otherwise it is *conflicting*. Let $comp(D, \Sigma) = \{K_1, \ldots, K_n\}$. It can be verified that $rep(D, \Sigma) = \{\{\underline{a}_1, \ldots, \underline{a}_n\} \ : \ \underline{a}_1 \in K_1, \ldots, \underline{a}_n \in K_n\}$. Let us now fix throughout this section a BCQ q over Σ. For a repair $R \in rep(D, \Sigma)$, if q is true in R, then there is a substitution μ such that $\mu(q) \subseteq R$. But since $R \subseteq D$, it also holds that $\mu(q) \subseteq D$. Hence, $sub(q, D) = \{\mu|_{var(q)} \ : \ \mu \text{ is a substitution and } \mu(q) \subseteq D\}$ is an overestimation of the substitutions that map q to the repairs of D.

Inspired by the well-known notions of conflict-hypergraph and conflict-join graph, we now introduce the notion of conflict-join hypergraph. Given a database D, the *conflict-join hypergraph* of D (w.r.t. q and Σ) is denoted by $H_D =$

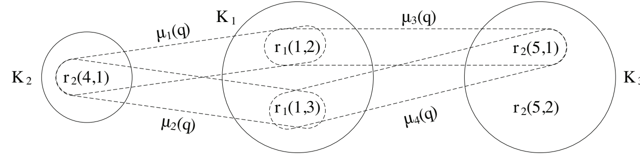


Fig. 1. Conflict-join hypergraph.

$\langle D, E\rangle$, where D are the vertices, and E are the hyperedges partitioned in $E_q = \{\mu(q) : \mu \in sub(q, D)\}$ and $E_\kappa = \{K : K \in comp(D, \Sigma)\}$. A *bunch* B of vertices of H_D is any minimal nonempty subset of D such that, for each $e \in E$, either $e \subseteq B$ or $e \cap B = \emptyset$ holds. Intuitively, every edge of H_D collects the atoms in a key component of D or the atoms in $\mu(q)$, for some $\mu \in sub(q, D)$. Moreover, each bunch collects the vertices of some connected component of H_D. An example follows to fix these preliminary notions.

Example 1. Consider the schema $\Sigma = \langle \mathcal{R}, \alpha, \kappa\rangle$, where $\mathcal{R} = \{r_1, r_2\}$, $\alpha(r_1) = \alpha(r_2) = 2$, and $\kappa(r_1) = \kappa(r_2) = \{1\}$. Consider also the database $D = \{r_1(1,2), r_1(1,3), r_2(4,1), r_2(5,1), r_2(5,2)\}$, and the BCQ $q = r_1(X, Y), r_2(Z, X)$. The conflicting components of D are $K_1 = \{r_1(1,2), r_1(1,3)\}$ and $K_3 = \{r_2(5,1), r_2(5,2)\}$, while its safe component is $K_2 = \{r_2(4,1)\}$. The repairs of D are $R_1 = \{r_1(1,2), r_2(4,1), r_2(5,1)\}$, $R_2 = \{r_1(1,2), r_2(4,1), r_2(5,2)\}$, $R_3 = \{r_1(1,3), r_2(4,1), r_2(5,1)\}$, and $R_4 = \{r_1(1,3), r_2(4,1), r_2(5,2)\}$. Moreover, $sub(q, D)$ contains the substitutions: $\mu_1 = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 4\}$, $\mu_2 = \{X \mapsto 1, Y \mapsto 3, Z \mapsto 4\}$, $\mu_3 = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 5\}$, and $\mu_4 = \{X \mapsto 1, Y \mapsto 3, Z \mapsto 5\}$. The conflict-join hypergraph $H_D = \langle D, E\rangle$ is as in Figure 1. Solid (resp., dashed) edges form the set E_κ (resp., E_q). Since μ_1 maps q to R_1 and R_2, and μ_2 maps q to R_3 and R_4, we conclude that $D \models_\Sigma q$. Finally, D is the only bunch of H_D. $\square$

In Example 1 we observe that K_3 can be safely ignored in the evaluation of q. In fact, even if both $\mu_3(q)$ and $\mu_4(q)$ contain an atom of K_3, μ_1 and μ_2 are sufficient to prove that q is consistently true. This might suggest to focus only on the set $F = K_1 \cup K_2$, and on its repairs $\{r_1(1,2), r_2(4,1)\}$ and $\{r_1(1,3), r_2(4,1)\}$. Also, since $F \models_\Sigma q$, F represents the "small" fragment of D that we need to evaluate q. The practical advantage of considering F instead of D should be already clear: (1) the repairs of F are smaller than the repairs of D; and (2) F has less repairs than D. We are now ready to introduce the notion of fragment. Consider a database D. For any set $C \subseteq comp(D, \Sigma)$ of key components of D, we say that the set $F = \bigcup_{K \in C} K$ is a (*well-defined*) *fragment* of D. According to this notion, the set $F = K_1 \cup K_2$ in Example 1 is a fragment of D. The following theorem, states a useful property that holds for any fragment.

Theorem 1. *Consider a database D, and two fragments $F_1 \subseteq F_2$ of D. If $F_1 \models_\Sigma q$, then $F_2 \models_\Sigma q$.*

Note that D is indeed a fragment of itself. Hence, if q is consistently true, then there is always the fragment $F = D$ such that $F \models_\Sigma q$. But now the

question is: *How can we identify a convenient set of fragments of D?* The naive way would be to use as fragments the bunches of H_D. Soundness is guaranteed by Theorem 1. Regarding completeness, we rely on the following result.

Theorem 2. *Consider a database D. If $D \models_\Sigma q$, then there is a bunch B of H_D such that $B \models_\Sigma q$.*

By combining Theorems 1 and 2 we are able to reduce, in polynomial time, the original problem into a collection of smaller ones of the same sort. However, this technique alone is not sufficient to deal with large data sets. Indeed, it involves the entire database by considering all the bunches of the conflict-join hypergraph. We now introduce an algorithm that can realize that K_3 is "redundant" in Example 1. Formally, a key component K of a database D is *redundant* (w.r.t. q) if *for each fragment F of D, $F \models_\Sigma q$ implies $F \setminus K \models_\Sigma q$*. In practice, a key component is redundant independently from the fact that some other key component is redundant or not. More formally, given a database D and a set C of redundant components of D, it holds that $D \models_\Sigma q$ iff $\left(D \setminus \bigcup_{K \in C} K\right) \models_\Sigma q$. Therefore, if we can identify all the redundant components of D, then after removing from D all these components, what remains is either: (1) a nonempty set of (minimal) bunches, each of which entails consistently q whenever $D \models_\Sigma q$; or (2) the empty set, whenever $D \not\models_\Sigma q$. More formally: given a database D, each key component of D is redundant iff $D \not\models_\Sigma q$.

However, assuming that PTIME $\neq$ NP, any algorithm for the identification of all the redundant components of D cannot be polynomial because, otherwise, we would have a polynomial procedure for solving the original problem. Our goal is therefore to identify sufficient conditions to design a pruning mechanism that detects in polynomial time as many redundant conflicting components as possible. To give an intuition of our pruning mechanism, we look again at Example 1. Actually, K_3 is redundant because it contains an atom, namely $r_2(5, 2)$, that is not involved in any substitution (see Figure 1). Assume now that this is the criterion that we use to identify redundant components. Since we know that $D \models_\Sigma q$ iff $D \setminus K_3 \models_\Sigma q$, this means that we can now forget about D and consider only $D' = K_1 \cup K_2$. But once we focus on $sub(q, D')$, we realize that it contains only μ_1 and μ_2. Then, a smaller number of substitutions in $sub(q, D')$ w.r.t. those in $sub(q, D)$ motivates us to reapply our criterion. Indeed, there could also be some atom in D' not involved in any of the substitutions of $sub(q, D')$. This is not the case in our example since the atoms in D' are covered by $\mu_1(q)$ or $\mu_2(q)$. However, in general, in one or more steps, we can identify more and more redundant components. We can now state the main result of this section.

Theorem 3. *Consider some conflict-join hypergraph $H_D = \langle D, E \rangle$, and a key component K of D. If $K \setminus \bigcup_{e \in E_q} e \neq \emptyset$, then K is redundant.*

As discussed just before Theorem 3, an indirect effect of removing a redundant component K from D is that all the substitutions in the set $S = \{\mu \in sub(q, D) \ : \ \mu(q) \cap K \neq \emptyset\}$ can be in a sense ignored. In fact, $sub(q, D \setminus K) = sub(q, D) \setminus S$. Whenever a substitution can be safely ignored, we say that it is

unfounded. Let us formalize this new notion. Consider a database D. A substitution μ of $sub(q, D)$ is *unfounded* if: for each fragment F of D, $F \models_{\Sigma} q$ implies that, for each repair $R \in rep(F, \Sigma)$, there exists a substitution $\mu' \in sub(q, R)$ different from μ such that $\mu'(q) \subseteq R$. We now show how to detect as many unfounded substitutions as possible.

Theorem 4. *Consider a database D, and some $\mu \in sub(q, D)$. If there exists a redundant component K of D such that $\mu(q) \cap K \neq \emptyset$, then μ is unfounded.*

Clearly, Theorem 4 alone is not helpful since it relies on the identification of redundant components. However, if combined with Theorem 3, it forms the desired cascade pruning mechanism. For example, both substitutions μ_3 and μ_4 in Example 1 are unfounded, since K_3 is redundant.

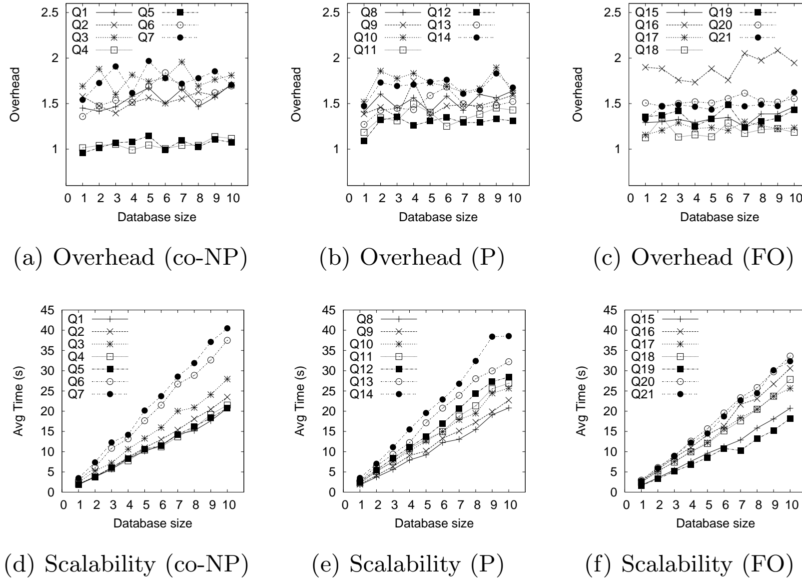
4 Experimental Evaluation

Benchmark Setup. The assessment of our approach was done using a benchmark employed in [10] for testing CQA systems on large inconsistent databases. It comprises 40 instances of a database schema with 10 tables, organized in four families of 10 instances each of which contains tables of size varying from 100k to 1M tuples; also it includes 21 queries of different structural features split into three groups depending on whether CQA complexity is coNP-complete (queries $Q_1, \cdots, Q_7$), PTIME but not FO-rewritable [14] (queries $Q_8, \cdots, Q_{14}$), and FO-rewritable (queries $Q_{15}, \cdots, Q_{21}$). We compare our approach, named *Pruning*, with two alternative ASP-based approaches. In particular, we considered one of the first encoding of CQA in ASP that was introduced in [4], and an optimized technique that was introduced more recently in [12]; these are named *BB* and *MRT*, respectively. *BB* and *MRT* can handle a larger class of integrity constrains than *Pruning*, and only *MRT* features specific optimization that apply also to primary key violations handling. We constructed the three alternative encodings for all 21 queries of the benchmark, and we run them on the ASP solver WASP 2.0 [1], configured with the iterative coherence testing algorithm.

Analysis of the results. Concerning the capability of providing an answer to a query within the time limit of 600 seconds, we report that *Pruning* was able to answer the queries in all the 840 runs in the benchmark with an average time of 14.6s. *MRT*, and *BB* solved only 778, and 768 instances within 600 seconds, with an average of 80.5s and 52.3s, respectively.

The scalability of *Pruning* is studied in detail for each query in Figures 2(d-f), each plotting the average execution times per group of queries of the same theoretical complexity. It is worth noting that *Pruning* scales almost linearly in all queries, and independently from the complexity class of the query. This is because *Pruning* can identify and deal efficiently with the conflicting fragments.

We now analyze the performance of *Pruning* from the perspective of a measure called *overhead*, which was employed in [10] for measuring the performance of CQA systems. Given a query Q the overhead is given by $\frac{t_{cqa}}{t_{plain}}$, where t_{cqa}



(a) Overhead (co-NP) (b) Overhead (P) (c) Overhead (FO)

(d) Scalability (co-NP) (e) Scalability (P) (f) Scalability (FO)

Fig. 2. Scalability and overhead of consistent query answering with Pruning encoding.

is time needed for computing the consistent answer of Q, and t_{plain} is the time needed for a plain execution of Q where the violation of integrity constraints are ignored. Note that the overhead measure is independent of the hardware and the software employed, since it relates the computation of CQA to the execution of a plain query on the same system. Thus it allows for a direct comparison of *Pruning* with other methods having known overheads. Following what was done in [10] (a comparison with [10] can be found in [13]), we computed the average overhead measured varying the database size for each query, and we report the results by grouping queries per complexity class in Figures 2(a-c). The overheads of *Pruning* is always below 2.1, and the majority of queries has overheads of around 1.5. The behavior is basically ideal for query Q5 and Q4 (overhead is about 1). The state of the art approach described in [10] has overheads that range between 5 and 2.8 on the very same dataset. Thus, our approach allows to obtain a very effective implementation of CQA in ASP with an overhead that is often more than two times smaller than the one of state-of-the-art approaches. We complemented this analysis by measuring also the overhead of *Pruning* w.r.t. the computation of safe answers, which provide an underestimate of consistent answers that can be computed efficiently (in polynomial time) by means of stratified ASP programs. We report that the computation of the consistent answer with *Pruning* requires only at most 1.5 times more in average than computing the safe answer. This further outlines that *Pruning* is able to maintain reasonable the impact of the hard-to-evaluate component of CQA. Finally, we have analyzed the impact of our technique in the various solving steps of the evaluation. We report that

for *Pruning* the solver analyzes a few non-factual rules (below 1% in average), whereas *MRT* and *BB* have 5% and 63% of non-factual rules, respectively. Since the hard part of the computation is performed by the solver on non-factual rules, this also outlines the benefits of the pruning technique.

5 Conclusion

Logic programming approaches to CQA were recently considered not competitive [10] on large databases affected by primary key violations. In this paper, we overview a strategy that dramatically reduces the primary key violations to be handled to answer the query. The strategy is encoded naturally in ASP, and an experiment on benchmarks already employed in the literature demonstrates that our ASP-based approach is efficient on large datasets, and performs better than state-of-the-art methods in terms of overhead. As far as future work is concerned, we plan to extend the *Pruning* method for handling inclusion dependencies, and other tractable classes of tuple-generating dependencies.

References

1. Alviano, M., Dodaro, C., Ricca, F.: Anytime computation of cautious consequences in answer set programming. TPLP 14(4-5), 755–770 (2014)
2. Arenas, M., Bertossi, L.E., Chomicki, J.: Consistent query answers in inconsistent databases. In: Proc. of PODS '99. pp. 68–79 (1999)
3. Arenas, M., Bertossi, L.E., Chomicki, J.: Answer sets for consistent query answering in inconsistent databases. TPLP 3(4-5), 393–424 (2003)
4. Barceló, P., Bertossi, L.E.: Logic programs for querying inconsistent databases. In: Proc. of PADL'03. LNCS, vol. 2562, pp. 208–222 (2003)
5. Bertossi, L.E., Hunter, A., Schaub, T. (eds.): Inconsistency Tolerance, LNCS, vol. 3300. Springer, Berlin / Heidelberg (2005)
6. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM 54(12), 92–103 (2011)
7. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate record detection: A survey. IEEE Trans. Knowl. Data Eng. 19(1), 1–16 (2007)
8. Fuxman, A., Miller, R.J.: First-order query rewriting for inconsistent databases. J. Comput. Syst. Sci. 73(4), 610–635 (2007)
9. Greco, G., Greco, S., Zumpano, E.: A logical framework for querying and repairing inconsistent databases. IEEE Trans. Knowl. Data Eng. 15(6), 1389–1408 (2003)
10. Kolaitis, P.G., Pema, E., Tan, W.C.: Efficient querying of inconsistent databases with binary integer programming. PVLDB 6(6), 397–408 (2013)
11. Koutris, P., Wijsen, J.: Consistent query answering for primary keys. SIGMOD Record 45(1), 15–22 (2016)
12. Manna, M., Ricca, F., Terracina, G.: Consistent query answering via asp from different perspectives: Theory and practice. TPLP 13(2), 227–252 (2013)
13. Manna, M., Ricca, F., Terracina, G.: Taming primary key violations to query large inconsistent data via ASP. TPLP 15(4-5), 696–710 (2015)
14. Wijsen, J.: On the consistent rewriting of conjunctive queries under primary key constraints. Inf. Syst. 34(7), 578–601 (2009)
15. Wijsen, J.: Certain conjunctive query answering in first-order logic. ACM Trans. Database Syst. 37(2), 9:1–9:35 (2012)