TLP **15** (4–5): 696–710, 2015. © Cambridge University Press 2015
doi:10.1017/S1471068415000320

Taming primary key violations to query large inconsistent data via ASP

MARCO MANNA, FRANCESCO RICCA and GIORGIO TERRACINA
Department of Mathematics and Computer Science,
University of Calabria, Italy
(*e-mail:* {`manna,ricca,terracina`}`@mat.unical.it`)

submitted 29 April 2015; revised 3 July 2015; accepted 15 July 2015

Abstract

Consistent query answering over a database that violates primary key constraints is a classical hard problem in database research that has been traditionally dealt with logic programming. However, the applicability of existing logic-based solutions is restricted to data sets of moderate size. This paper presents a novel decomposition and pruning strategy that reduces, in polynomial time, the problem of computing the consistent answer to a conjunctive query over a database subject to primary key constraints to a collection of smaller problems of the same sort that can be solved independently. The new strategy is naturally modeled and implemented using Answer Set Programming (ASP). An experiment run on benchmarks from the database world prove the effectiveness and efficiency of our ASP-based approach also on large data sets.

KEYWORDS: Inconsistent Databases, Consistent Query Answering, 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 (Abiteboul *et al.* 1995). 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 (Bertossi *et al.* 2005), 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 (Elmagarmid *et al.* 2007) 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" (Arenas *et al.* 1999; Bertossi 2011).

In this paper, we adopt the second approach – which has been proposed by Arenas *et al.* (1999) 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 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 (Arenas *et al.* 2003), 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 (Arenas *et al.* 1999; Fuxman *et al.* 2005; Fuxman and Miller 2007; Wijsen 2009; Wijsen 2012). Actually, in the unrestricted case, CQA has been traditionally dealt with logic programming (Greco *et al.* 2001; Arenas *et al.* 2003; Barceló and Bertossi 2003; Eiter *et al.* 2003; Greco *et al.* 2003; Manna *et al.* 2013). However, it has been argued (Kolaitis *et al.* 2013) 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 (Kolaitis *et al.* 2013) has revealed good performances on large databases (featuring up to one million tuples per relation) with primary key violations.

In this paper, we demonstrate that logic programming can still be effectively used for computing consistent answers over large relational databases. We design a novel 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) (Gelfond and Lifschitz 1991; Brewka *et al.* 2011), and we prove empirically the effectiveness of our ASP-based approach on existing benchmarks from the database world. In particular, we compare our approach with some classical (Barceló and Bertossi 2003) and optimized (Manna *et al.* 2013) encodings of CQA in ASP that were presented in the literature. 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 Preliminaries

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 (or sets, with a slight abuse of notation) 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 $n \geqslant 1$. Given a sequence $\mathbf{t} = t_1,\ldots,t_n$ of terms and a set $S = \{p_1,\ldots,p_k\} \subseteq [n]$, $\mathbf{t}|_S$

is the subsequence $t_{p_1}, \dots, t_{p_k}$. For example, if $\mathbf{t} = t_1, t_2, t_3$ and $S = \{1, 3\}$, then $\mathbf{t}|_S = t_1, t_3$.

A (*relational*) *schema* is a triple $\langle \mathscr{R}, \alpha, \kappa \rangle$ where $\mathscr{R}$ is a finite set of *relation symbols* (or *predicates*), $\alpha : \mathscr{R} \rightarrow \mathbb{N}$ is a function associating an *arity* to each predicate, and $\kappa : \mathscr{R} \rightarrow 2^{\mathbb{N}}$ is a function that associates, to each $r \in \mathscr{R}$, a nonempty set of positions from $[\alpha(r)]$, which represents the *primary key* of r. Moreover, for each relation symbol $r \in \mathscr{R}$ and for each position $i \in [\alpha(r)]$, $r[i]$ denotes the i-th *attribute* of r. Throughout, let $\Sigma = \langle \mathscr{R}, \alpha, \kappa \rangle$ denote a relational schema. An *atom* (over Σ) is an expression of the form $r(t_1, \dots, t_n)$, where $r \in \mathscr{R}$, 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 $\mu : \mathbf{C} \cup \mathbf{V} \rightarrow \mathbf{C} \cup \mathbf{V}$ which is the identity on $\mathbf{C}$. Given a set A of atoms, $\mu(A) = \{r(\mu(t_1), \dots, \mu(t_n)) : r(t_1, \dots, t_n) \in A\}$. 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 $X_1, \dots, X_n$, the *answer* to q over a database D, denoted $q(D)$, is the set of all n-tuples $\langle t_1, \dots, t_n \rangle \in \mathbf{C}^n$ 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 $F_1, \dots, F_k$ 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); then we provide complementary techniques to further reduce the number of inconsistencies to be handled for answering the original CQ. (Complete proofs in the online Appendix A.)

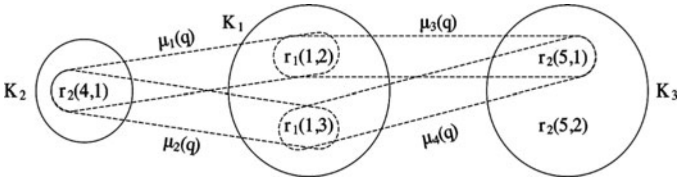


Fig. 1. Conflict-join hypergraph.

3.1 Fragments identification

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, $sub(q,D) = \{\mu|_{var(q)} : \mu$ 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 notions of conflict-hypergraph (Chomicki and Marcinkowski 2005) and conflict-join graph (Kolaitis and Pema 2012), 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 $H_D = \langle D,E\rangle$, 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 $\Sigma = \langle \mathscr{R},\alpha,\kappa\rangle$, where $\mathscr{R} = \{r_1,r_2\}$, $\alpha(r_1) = \alpha(r_2) = 2$, 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 depicted 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.

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 the formal notion of fragment.

Definition 1
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 Definition 1, the set $F = K_1 \cup K_2$ in Example 1 is a fragment of D. The following proposition, states a useful property that holds for any fragment.

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

By Definition 1, 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 Proposition 1. Regarding completeness, we rely on the following result.

Theorem 1
Consider a database D. If $D \models_\Sigma q$, then there is a bunch B of H_D s.t. $B \models_\Sigma q$.

By combining Proposition 1 with Theorem 1 we are able to reduce, in polynomial time, the original problem into a collection of smaller ones of the same sort.

3.2 *The cascade pruning mechanism*

The technique proposed in the previous section 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. Let us first define formally the term redundant.

Definition 2
A key component K of a database D is called *redundant* (w.r.t. q) if the following condition is satisfied: for each fragment F of D, $F \models_\Sigma q$ implies $F \setminus K \models_\Sigma q$.

The above definition states that a key component is redundant independently from the fact that some other key component is redundant or not. Therefore:

Proposition 2
Consider a database D and a set C of redundant components of D. It holds that $D \models_\Sigma q$ iff $\left(D \setminus \bigcup_{K \in C} K\right) \models_\Sigma q$.

In light of Proposition 2, 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:

Proposition 3
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, by Definition 2, 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 2
Consider a database D, and a key component K of D. Let $H_D = \langle D, E\rangle$ be the conflict-join hypergraph of D. If $K \setminus \bigcup_{e\in E_q} e \neq \emptyset$, then K is redundant.

In what follows, a redundant component that can be identified via Theorem 2 is called *strongly redundant*. As discussed just before Theorem 2, 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 in the following definition.

Definition 3
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 3
Consider a database D, and a substitution $\mu \in sub(q,D)$. If there exists a redundant component K of D such that $\mu(q) \cap K \neq \emptyset$, then μ is unfounded.

Clearly, Theorem 3 alone is not helpful since it relies on the identification of redundant components. However, if combined with Theorem 2, it forms the desired cascade pruning mechanism. To this end, we call *strongly unfounded* an unfounded substitution that can be identified by applying Theorem 3 by only considering strongly redundant components. Hereafter, let us denote by $sus(q, D)$ the subset of $sub(q, D)$ containing only strongly unfounded substitutions. Hence, both substitutions μ_3 and μ_4 in Example 1 are strongly unfounded, since K_3 is strongly redundant. Moreover, we reformulate the statement of Theorem 2 by exploiting the notion of strongly unfounded substitution, and the fact that the set $K \setminus \bigcup_{e \in E_q} e$ is nonempty if and only if there exists an atom $\underline{a} \in K$ such that the set $\{\mu \in sub(q, D) : \underline{a} \in \mu(q)\}$ – or equivalently the set $\{e \in E_q : \underline{a} \in e\}$ – is empty. For example, according to Figure 1, the set $K_3 \setminus \bigcup_{e \in E_q} e$ is nonempty since it contains the atom $r_2(5, 2)$. But this atoms makes the set $\{\mu \in sub(q, D) : r_2(5, 2) \in \mu(q)\}$ empty since no substitution of $sub(q, D)$ (or no hyperedge of E_q) involves $r_2(5, 2)$.

Proposition 4

A key component K of D is strongly redundant if there is an atom $\underline{a} \in K$ such that one of the two following conditions is satisfied: (1) $\{\mu \in sub(q, D) : \underline{a} \in \mu(q)\} = \emptyset$, or (2) $\{\mu \in sub(q, D) : \underline{a} \in \mu(q)\} = \{\mu \in sus(q, D) : \underline{a} \in \mu(q)\}$.

By combining Theorem 3 and Proposition 4, we have a declarative (yet inductive) specification of all the strongly redundant components of D. Importantly, the process of identifying strongly redundant components and strongly unfounded substitutions by exhaustively applying Theorem 3 and Proposition 4 is monotone and reaches a fixed-point (after no more than $|comp(D, \Sigma)|$ steps) when no more key component can be marked as strongly redundant.

3.3 Idle attributes

Previously, we have described a technique to reduce inconsistencies by progressively eliminating key components that are involved in query substitutions but are redundant. In the following, we show how to reduce inconsistencies by reducing the cardinality of conflicting components, which may be even treated as safe ones.

The act of *removing* an attribute $r[i]$ from a triple $\langle q, D, \Sigma \rangle$ consists of reducing the arity of r by one, cutting down the i-th term of each r-atom of D and q, and adapting the positions of the primary key of r accordingly. Moreover, let $attrs(\Sigma) = \{r[i] | r \in \mathscr{R} \text{ and } i \in [\alpha(r)]\}$, let $B \subseteq attrs(\Sigma)$, and let $A = attrs(\Sigma) \setminus B$. The *projection* of $\langle q, D, \Sigma \rangle$ on A, denoted by $\Pi_A(q, D, \Sigma)$, is the triple that is obtained from $\langle q, D, \Sigma \rangle$ by removing all the attributes of B. Consider a CQ q and a predicate $r \in \mathscr{R}$. The attribute $r[i]$ is *relevant* (w.r.t. q) if q contains an atom of the form $r(t_1, \ldots, t_{\alpha(r)})$ such that at least one of the following conditions is satisfied: (1) $i \in \kappa(r)$; or (2) t_i is a constant; or (3) t_i is a variable that occurs more than once in q; or (4) t_i is a free variable of q. An attribute which is not relevant is *idle* (w.r.t. q). (An example in the online Appendix B.) The following theorem states that the consistent answer to a CQ does not change after removing the idle attributes.

Theorem 4
Consider a CQ q, the set $R = \{r[i] \in \mathit{attrs}(\Sigma) | r[i] \text{ is relevant w.r.t. } q\}$, and a database D. It holds that $\mathit{ans}(q, D, \Sigma) = \mathit{ans}(\Pi_R(q, D, \Sigma))$.

3.4 Conjunctive queries and safe answers

Let Σ be a relational schema, D be a database, and $q = \exists \mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y})$ be a CQ, where we assume that Σ contains only relevant attributes w.r.t. q (idle attributes, if any, have been already removed). Since $\mathit{ans}(q, D, \Sigma) \subseteq q(D)$, for each candidate answer $\mathbf{t}_c \in q(D)$, one should evaluate whether the BCQ $\bar{q} = \varphi(\mathbf{t}_c, \mathbf{Y})$ is (or is not) consistently true in D. Before constructing the conflict-join hypergraph of D (w.r.t. $\bar{q}$ and Σ), however, one could check whether there is a substitution μ that maps $\bar{q}$ to D with the following property: for each $\underline{a} \in \mu(\bar{q})$, the singleton $\{\underline{a}\}$ is a safe component of D. And, if so, it is possible to conclude immediately that $\mathbf{t}_c \in \mathit{ans}(q, D, \Sigma)$. Intuitively, whenever the above property is satisfied, we say that $\mathbf{t}_c$ is a *safe answer* to q because, for each $R \in \mathit{rep}(D, \Sigma)$, it is guaranteed that $\mu(\bar{q}) \subseteq R$. The next result follows.

Theorem 5
Consider a CQ $q = \exists \mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y})$, and a tuple $\mathbf{t}_c$ of $q(D)$. If there is a substitution μ s.t. each atom of $\mu(\varphi(\mathbf{t}_c, \mathbf{Y}))$ forms a safe component of D, then $\mathbf{t}_c \in \mathit{ans}(q, D, \Sigma)$.

4 The encoding in ASP

In this section, we propose an ASP-based encoding to CQA that implements the techniques described in Section 3, and that is able to deal directly with CQs, instead of evaluating separately the associated BCQs. Hereafter, we assume the reader is familiar with Answer Set Programming (Gelfond and Lifschitz 1991; Brewka *et al.* 2011; Baral 2003) and with the standard ASP Core 2.0 syntax of ASP competitions (Calimeri *et al.* 2014; Calimeri *et al.* 2013). Given a relational schema $\Sigma = \langle \mathscr{R}, \alpha, \kappa \rangle$, a database D, and a CQ $q = \exists \mathbf{Y}\, \varphi(\mathbf{X}, \mathbf{Y})$, we construct a program $P(q, \Sigma)$ s.t. a tuple $\mathbf{t} \in q(D)$ belongs to $\mathit{ans}(q, D, \Sigma)$ iff each answer set of $D \cup P(q, \Sigma)$ contains an atom of the form $q^*(c, \mathbf{t})$, for some constant c. Importantly, a large part of $P(q, \Sigma)$ does not depend on q or Σ. To lighten the presentation, we provide a simplified version of the encoding that has been used in our experiments. In fact, for efficiency reasons, idle attributes should be "ignored on-the-fly" without materializing the projection of $\langle q, D, \Sigma \rangle$ on the relevant attributes; but this makes the encoding a little more heavy. Hence, we first provide a naive way to consider only the relevant attributes, and them we will assume that Σ contains no idle attribute. Let R collect all the attributes of Σ that are relevant w.r.t. q. For each $r \in \mathscr{R}$ that occurs in q, let $\mathbf{W}$ be a sequence of $\alpha(r)$ different variables and $S = \{i | r[i] \in R\}$, the terms of the r-atoms of D that are associated to idle attributes can be removed via the rule $r'(\mathbf{W}|_S) :- r(\mathbf{W})$. Hereafter, assume that Σ contains no idle attribute, and $\mathbf{Z} = \mathbf{X} \cup \mathbf{Y}$. Program $P(q, \Sigma)$ is depicted in Figure 2.

`% Computation of the safe answer.`
1 $sub(\mathbf{Z})$:– $\varphi(\mathbf{Z})$.
2 $involvedAtom(\mathtt{k}_r(\hat{\mathbf{t}}), \mathtt{nk}_r(\check{\mathbf{t}}))$:– $sub(\mathbf{Z}), r(\mathbf{t})$. $\quad \forall r(\mathbf{t}) \in q$
3 $confComp(\mathsf{K})$:– $involvedAtom(\mathsf{K}, \mathsf{NK}_1),\ involvedAtom(\mathsf{K}, \mathsf{NK}_2),\ \mathsf{NK}_1 \neq \mathsf{NK}_2$.
4 $safeAns(\mathbf{X})$:– $sub(\mathbf{Z})$, not $confComp(\mathtt{k}_{r_1}(\hat{\mathbf{t}}_1)),\ \ldots,$ not $confComp(\mathtt{k}_{r_n}(\hat{\mathbf{t}}_n))$.

`% Hypergraph Construction.`
5 $subEq(\mathtt{sID}(\mathbf{Z}), \mathtt{ans}(\mathbf{X}))$:– $sub(\mathbf{Z})$, not $safeAns(\mathbf{X})$.
6 $compEk(\mathtt{k}_r(\hat{\mathbf{t}}), \mathsf{Ans})$:– $subEq(\mathtt{sID}(\mathbf{Z}), \mathsf{Ans})$. $\quad \forall r(\mathbf{t}) \in q$
7 $inSubEq(\mathtt{atom}_r(\mathbf{t}), \mathtt{sID}(\mathbf{Z}))$:– $subEq(\mathtt{sID}(\mathbf{Z}), \_)$. $\quad \forall r(\mathbf{t}) \in q$
8 $inCompEk(\mathtt{atom}_r(\mathbf{t}), \mathtt{k}_r(\hat{\mathbf{t}}))$:– $compEk(\mathtt{k}_r(\hat{\mathbf{t}}), \_),\ involvedAtom(\mathtt{k}_r(\hat{\mathbf{t}}), \mathtt{nk}_r(\check{\mathbf{t}}))$. $\quad \forall r(\mathbf{t}) \in q$

`% Pruning.`
9 $redComp(\mathsf{K}, \mathsf{Ans})$:– $compEk(\mathsf{K}, \mathsf{Ans}),\ inCompEk(\mathsf{A}, \mathsf{K})$,
 $\#\mathrm{count}\{\mathsf{S} : inSubEq(\mathsf{A}, \mathsf{S}),\ subEq(\mathsf{S}, \mathsf{Ans})\} = 0$.
10 $unfSub(\mathsf{S}, \mathsf{Ans})$:– $subEq(\mathsf{S}, \mathsf{Ans}),\ inSubEq(\mathsf{A}, \mathsf{S}),\ inCompEk(\mathsf{A}, \mathsf{K}), redComp(\mathsf{K}, \mathsf{Ans})$.
11 $redComp(\mathsf{K}, \mathsf{Ans})$:– $compEk(\mathsf{K}, \mathsf{Ans}),\ inCompEk(\mathsf{A}, \mathsf{K})$,
 $\#\mathrm{count}\{\mathsf{S} : inSubEq(\mathsf{A}, \mathsf{S}), subEq(\mathsf{S}, \mathsf{Ans})\} = \#\mathrm{count}\{\mathsf{S} : inSubEq(\mathsf{A}, \mathsf{S}), unfSub(\mathsf{S}, \mathsf{Ans})\}$.
12 $residualSub(\mathsf{S}, \mathsf{Ans})$:– $subEq(\mathsf{S}, \mathsf{Ans})$, not $unfSub(\mathsf{S}, \mathsf{Ans})$.

`% Fragments identification.`
13 $shareSub(\mathsf{K}_1, \mathsf{K}_2, varsAns)$:– $residualSub(\mathsf{S}, \mathsf{Ans}),\ inSubEq(\mathsf{A}_1, \mathsf{S}),\ inSubEq(\mathsf{A}_2, \mathsf{S})$,
 $\mathsf{A}_1 \neq \mathsf{A}_2,\ inCompEk(\mathsf{A}_1, \mathsf{K}_1),\ inCompEk(\mathsf{A}_2, \mathsf{K}_2), \mathsf{K}_1 \neq \mathsf{K}_2$.
14 $ancestorOf(\mathsf{K}_1, \mathsf{K}_2, \mathsf{Ans})$:– $shareSub(\mathsf{K}_1, \mathsf{K}_2, \mathsf{Ans}),\ \mathsf{K}_1 < \mathsf{K}_2$.
15 $ancestorOf(\mathsf{K}_1, \mathsf{K}_3, \mathsf{Ans})$:– $ancestorOf(\mathsf{K}_1, \mathsf{K}_2, \mathsf{Ans}),\ shareSub(\mathsf{K}_2, \mathsf{K}_3, \mathsf{Ans}), \mathsf{K}_1 < \mathsf{K}_3$.
16 $child(\mathsf{K}, \mathsf{Ans})$:– $ancestorOf(\_, \mathsf{K}, \mathsf{Ans})$.
17 $keyCompInFrag(\mathsf{K}_1, \mathtt{fID}(\mathsf{K}_1, \mathsf{Ans}))$:– $ancestorOf(\mathsf{K}_1, \_, \mathsf{Ans})$, not $child(\mathsf{K}_1, \mathsf{Ans})$.
18 $keyCompInFrag(\mathsf{K}_2, \mathtt{fID}(\mathsf{K}_1, \mathsf{Ans}))$:– $ancestorOf(\mathsf{K}_1, \mathsf{K}_2, \mathsf{Ans})$, not $child(\mathsf{K}_1, \mathsf{Ans})$.
19 $subInFrag(\mathsf{S}, \mathtt{fID}(\mathsf{KF}, \mathsf{Ans}))$:– $residualSub(\mathsf{S}, \mathsf{Ans}),\ inSubEq(\mathsf{A}, \mathsf{S}),\ inCompEk(\mathsf{A}, \mathsf{K})$,
 $keyCompInFrag(\mathsf{K}, \mathtt{fID}(\mathsf{KF}, \mathsf{Ans}))$.
20 $frag(\mathtt{fID}(\mathsf{K}, \mathsf{Ans}), \mathsf{Ans})$:– $keyCompInFrag(\_, \mathtt{fID}(\mathsf{K}, \mathsf{Ans}))$.

`% Repair construction.`
21 $1 \leqslant \{activeFrag(\mathsf{F}) : frag(\mathsf{F}, \mathsf{Ans})\} \leqslant 1$:– $frag(\_, \_)$.
22 $1 \leqslant \{activeAtom(\mathsf{A}) : inCompEk(\mathsf{A}, \mathsf{K})\} \leqslant 1$:– $activeFrag(\mathsf{F}), keyCompInFrag(\mathsf{K}, \mathsf{F})$.
23 $ignoredSub(\mathsf{S})$:– $activeFrag(\mathsf{F}),\ subInFrag(\mathsf{S}, \mathsf{F}),\ inSubEq(\mathsf{A}, \mathsf{S})$, not $activeAtom(\mathsf{A})$.

`% New query.`
24 $q^*(\mathtt{s}, \mathsf{X}_1, \ldots, \mathsf{X}_n)$:– $safeAns(\mathsf{X}_1, \ldots, \mathsf{X}_n)$.
25 $q^*(\mathsf{F}, \mathsf{X}_1, \ldots, \mathsf{X}_n)$:– $frag(\mathsf{F}, \mathtt{ans}(\mathsf{X}_1, \ldots, \mathsf{X}_n))$, not $activeFrag(\mathsf{F})$.
26 $q^*(\mathsf{F}, \mathsf{X}_1, \ldots, \mathsf{X}_n)$:– $activeFrag(\mathsf{F}),\ subInFrag(\mathsf{S}, \mathsf{F})$, not $ignoredSub(\mathsf{S})$,
 $frag(\mathsf{F}, \mathtt{ans}(\mathsf{X}_1, \ldots, \mathsf{X}_n))$.

Fig. 2. The Encoding in ASP.

Computation of the safe answer. Via rule 1, we identify the set $\mathscr{M} = \{\mu|_{\mathbf{Z}} : \mu$ is a substitution and $\mu(\varphi(\mathbf{X}, \mathbf{Y})) \subseteq D\}$. It is now possible (rule 2) to identify the atoms of D that are involved in some substitution. Here, for each atom $r(\mathbf{t}) \in q$, we recall that $\hat{\mathbf{t}}$ is the subsequence of $\mathbf{t}$ containing the terms in the positions of the primary key of r, and we assume that $\check{\mathbf{t}}$ are the terms of $\mathbf{t}$ in the remaining positions. In particular, we use two function symbols, $\mathtt{k}_r$ and $\mathtt{nk}_r$, to group the terms in the key of r and the remaining ones, respectively. It is now easy (rule 3) to identify the conflicting components involved in some substitution. Let $\varphi(\mathbf{X}, \mathbf{Y}) = r_1(\mathbf{t}_1), \ldots, r_n(\mathbf{t}_n)$. We now compute (rule 4) the safe answers.

Hypergraph construction. For each candidate answer $\mathbf{t}_c \in q(D)$ that has not been already recognized as safe, we construct the hypergraph $H_D(\mathbf{t}_c) = \langle D, E \rangle$ associated to the BCQ $\varphi(\mathbf{t}_c, \mathbf{Y})$, where $E = E_q \cup E_\kappa$, as usual. Hypergraph $H_D(\mathbf{t}_c)$ is identified by the functional term $\mathtt{ans}(\mathbf{t}_c)$, the substitutions of E_q (collected via rule 5) are identified by the set $\{\mathtt{sID}(\mu(\mathbf{Z})) \mid \mu \in \mathscr{M}$ and $\mu(\mathbf{X}) = \mathbf{t}_c\}$ of functional terms, while the key components of E_κ (collected via rule 6) are identified by the set $\{\mathtt{k}_r(\mu(\hat{\mathbf{t}})) \mid \mu \in \mathscr{M}$ and $\mu(\mathbf{X}) = \mathbf{t}_c$ and $r(\mathbf{t}) \in q\}$ of functional terms.

Pruning. We are now ready to identify (rules $9 - 11$) the strongly redundant components and the strongly unfounded substitutions (as described in Section 3) to implement our cascade pruning mechanism. Hence, it is not difficult to collect (rule 12) the substitutions that are not unfounded, that we call *residual*.

Fragments identification. Key components involving at least a residual substitution (i.e., not redundant ones), can be aggregated in fragments (rules $13 - 20$) by using the notion of bunch introduced in Section 3.1. In particular, any given fragment F – associated to a candidate answer $\mathbf{t}_c \in q(D)$, and collecting the key components $K_1, \ldots, K_m$ – is identified by the functional term $\mathtt{fID}(K_i, \mathbf{t}_c)$ where, for each $j \in \{1, \ldots, m\} \setminus \{i\}$, the functional term associated to K_i lexicographically precedes the functional term associated to K_j.

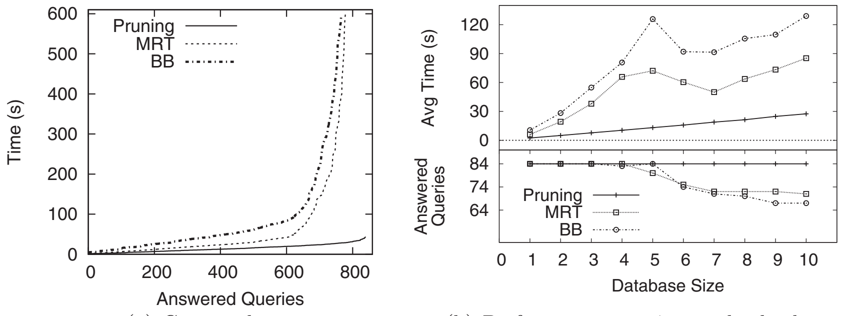
Repair construction. Rules 1–20 can be evaluated in polynomial time and have only one answer set, while the remaining part of the program cannot in general. In particular, rules 21–23 generate the search space. Actually, each answer set M of $P(q, \Sigma)$ is associated (rule 21) with only one fragment, say F, that we call *active* in M. Moreover, for each key component K of F, answer set M is also associated (rule 22) with only one atom of K, that we also call *active* in M. Consequently, each substitution which involves atoms of F but also at least one atom which is not active, must be ignored in M (rule 23).

New query. Finally, we compute the atoms of the form $q^*(c, \mathbf{t})$ via rules 24–26.

5 Experimental evaluation

The experiment for assessing the effectiveness of our approach is described in the following. We first describe the benchmark setup and, then, we analyze the results.

Benchmark Setup. The assessment of our approach was done using a benchmark employed in (Kolaitis *et al.* 2013) 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 (Wijsen 2009) (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 (Barceló and Bertossi 2003), and an optimized technique that was introduced more recently in (Manna *et al.* 2013); 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



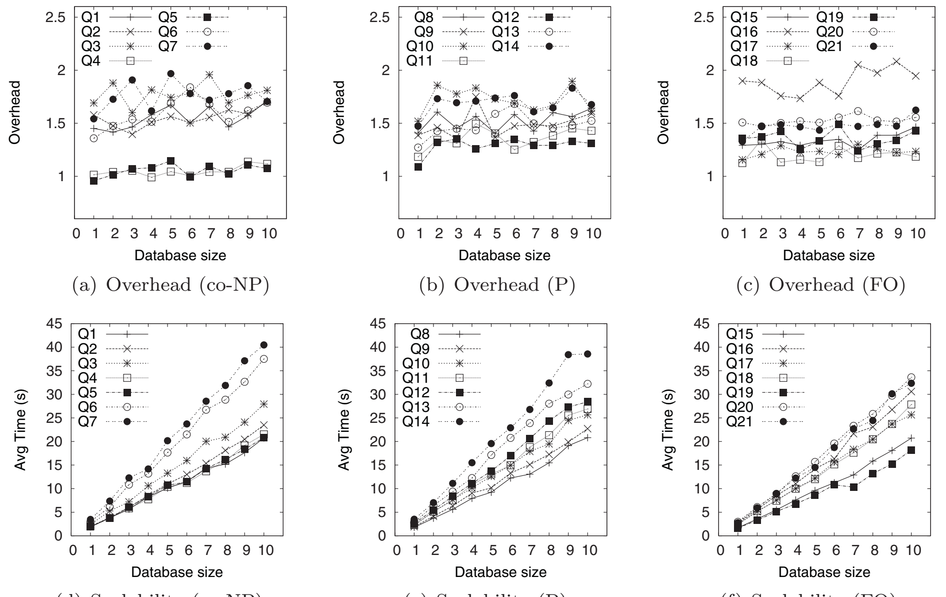
(a) Cactus plot. (b) Performance avg time and solved.

Fig. 3. Comparison with alternative encodings: answered queries and execution time.

three alternative encodings for all 21 queries of the benchmark, and we run them on the ASP solver WASP 2.0 (Alviano *et al.* 2014b), configured with the iterative coherence testing algorithm (Alviano *et al.* 2014a), coupled with the grounder Gringo ver. 4.4.0 (Gebser *et al.* 2011). For completeness we have also run clasp ver. 3.1.1 (Gebser *et al.* 2013) obtaining similar results. WASP performed better in terms of number of solved instances on *MRT* and *BB*. The experiment was run on a Debian server with Xeon E5-4610 CPUs and 128GB of RAM. Resource usage was limited to 600 seconds and 16GB of RAM in each execution. Execution times include both grounding and solving. (ASP programs and solver binaries can be downloaded from `www.mat.unical.it/ricca/downloads/mrtICLP2015.zip`.)

Analysis of the results. Concerning the capability of providing an answer to a query within the time limit, 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 cactus plot in Figure 3(a) provides an aggregate view of the performance of the compared methods. Recall that a cactus plot reports for each method the number of answered queries (solved instances) in a given time. We observe that the line corresponding to *Pruning* in Figure 3(a) is always below the ones of *MRT* and *BB*. In more detail, *Pruning* execution times grow almost linearly with the number of answered queries, whereas *MRT* and *BB* show an exponential behavior. We also note that *MRT* behaves better than *BB*, and this is due to the optimizations done in *MRT* that reduce the search space.

The performance of the approaches w.r.t. the size of the database is studied in Figure 3(b). The x-axis reports the number of tuples per relation in tenth of thousands, in the upper plot is reported the number of queries answered in 600s, and in the lower plot is reported the corresponding the average running time. We observe that all the approaches can answer all 84 queries (21 queries per 4 databases) up to the size of 300k tuples, then the number of answered queries by both *BB* and *MRT* starts decreasing. Indeed, they can answer respectively 74 and 75 queries of size 600k tuples, and only 67 and 71 queries on the largest databases (1M tuples). Instead, *Pruning* is able to solve all the queries in the data set. The average time elapsed by running *Pruning* grows linearly from 2.4s up to



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

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

Fig. 4. Scalability and overhead of consistent query answering with Pruning encoding.

27.4s. *MRT* and *BB* average times show a non-linear growth and peak at 128.9s and 85.2s, respectively. (Average is computed on queries answered in 600s, this explains why it apparently decreases when a method cannot answer some instance within 600s.)

The scalability of *Pruning* is studied in detail for each query in Figures 4(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* is able to 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 (Kolaitis *et al.* 2013) for measuring the performance of CQA systems. Given a query Q the overhead is given by $\frac{t_{cqa}}{t_{plain}}$, where t_{cqa} 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 (Kolaitis *et al.* 2013), 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 4(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 (Kolaitis *et al.* 2013) has overheads that range between 5 and 2.8 on the very same dataset

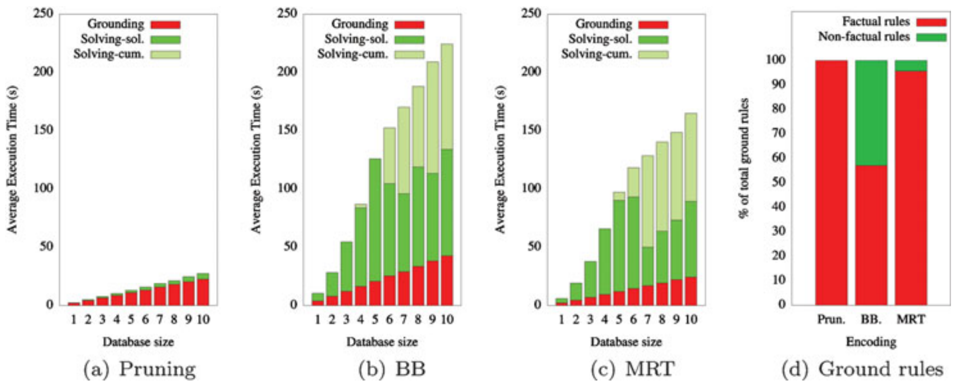


Fig. 5. Average execution times per evaluation step.

(more details in the online appendix (Appendix C)). 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 (detailed plots in the online appendix (Appendix C)). 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. The first three histograms in Figure 5 report the average running time spent for answering queries in databases of growing size for *Pruning* (Fig. 5(a)), *BB* (Fig. 5(b)), and *MRT* (Fig. 5(c)). In each bar different colors distinguish the average time spent for grounding and solving. In particular, the average solving time over queries *answered within the timeout* is labeled Solving-sol, and each bar extends up to the average cumulative execution time computed over all instances, where each timed out execution counts 600s. Recall that, roughly speaking, the grounder solves stratified normal programs, and the hard part of the computation is performed by the solver on the residual non-stratified program; thus, we additionally report in Figure 5(d) the average number of facts (knowledge inferred by grounding) and of non-factual rules (to be evaluated by the solver) in percentage of the total for the three compared approaches. The data in Figure 5 confirm that most of the computation is done with *Pruning* during the grounding, whereas this is not the case for *MRT* and *BB*. Figure 5(d) shows that for *Pruning* the grounder produces a few non-factual rules (below 1% in average), whereas *MRT* and *BB* produce 5% and 63% of non-factual rules, respectively. Roughly, this corresponds to about 23K non-factual rules (resp., 375K non-factual rules) every 100K tuples per relation for *MRT* (resp., *BB*), whereas our approach produces no more than 650 non-factual rules every 100K tuples per relation.

6 Conclusion

Logic programming approaches to CQA were recently considered not competitive (Kolaitis *et al.* 2013) on large databases affected by primary key violations. In this paper, we proposed a new strategy based on a cascade pruning mechanism that dramatically reduces the number of 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

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

Alviano, M., Dodaro, C., and Ricca, F. 2014a. Anytime computation of cautious consequences in answer set programming. *TPLP 14,* 4-5, 755–770.

Alviano, M., Dodaro, C., and Ricca, F. 2014b. Preliminary report on WASP 2.0. *CoRR abs/1404.6999.*

Arenas, M., Bertossi, L. E., and Chomicki, J. 1999. Consistent query answers in inconsistent databases. In *Proceedings of PODS '99.* 68–79.

Arenas, M., Bertossi, L. E., and Chomicki, J. 2003. Answer sets for consistent query answering in inconsistent databases. *TPLP 3,* 4-5, 393–424.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press.

Barceló, P. and Bertossi, L. E. 2003. Logic programs for querying inconsistent databases. In *Proceedings of PADL'03.* LNCS, vol. 2562. Springer, 208–222.

Bertossi, L. E. 2011. *Database Repairing and Consistent Query Answering.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers.

Bertossi, L. E., Hunter, A., and Schaub, T., Eds. 2005. *Inconsistency Tolerance.* LNCS, vol. 3300. Springer, Berlin / Heidelberg.

Brewka, G., Eiter, T., and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., and Schaub, T. 2013. Asp-core-2 input language format. Available at `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf`.

Calimeri, F., Ianni, G., and Ricca, F. 2014. The third open answer set programming competition. *TPLP 14,* 1, 117–135.

Chomicki, J. and Marcinkowski, J. 2005. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput. 197,* 1-2, 90–121.

Eiter, T., Fink, M., Greco, G., and Lembo, D. 2003. Efficient evaluation of logic programs for querying data integration systems. In *Proceedings of ICLP'03.* LNCS, vol. 2916. Springer, 163–177.

Elmagarmid, A. K., Ipeirotis, P. G., and Verykios, V. S. 2007. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng. 19,* 1, 1–16.

Fuxman, A., Fazli, E., and Miller, R. J. 2005. Conquer: Efficient management of inconsistent databases. In *Proceedings of SIGMOD'05.* ACM, 155–166.

Fuxman, A. and Miller, R. J. 2007. First-order query rewriting for inconsistent databases. *J. Comput. Syst. Sci. 73,* 4, 610–635.

Gebser, M., Kaminski, R., König, A., and Schaub, T. 2011. Advances in *gringo* series 3. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, J. P. Delgrande and W. Faber, Eds. Lecture Notes in Computer Science, vol. 6645. Springer, 345–351.

Gebser, M., Kaufmann, B., and Schaub, T. 2013. Advanced conflict-driven disjunctive answer set solving. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, F. Rossi, Ed. IJCAI/AAAI.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Comput. 9,* 3/4, 365–386.

Greco, G., Greco, S., and Zumpano, E. 2001. A logic programming approach to the integration, repairing and querying of inconsistent databases. In *Proceedings of ICLP'01.* LNCS, vol. 2237. Springer, 348–364.

Greco, G., Greco, S., and Zumpano, E. 2003. A logical framework for querying and repairing inconsistent databases. *IEEE Trans. Knowl. Data Eng. 15,* 6, 1389–1408.

Kolaitis, P. G. and Pema, E. 2012. A dichotomy in the complexity of consistent query answering for queries with two atoms. *Inf. Process. Lett. 112,* 3, 77–85.

Kolaitis, P. G., Pema, E., and Tan, W.-C. 2013. Efficient querying of inconsistent databases with binary integer programming. *PVLDB 6,* 6, 397–408.

Manna, M., Ricca, F., and Terracina, G. 2013. Consistent query answering via asp from different perspectives: Theory and practice. *TPLP 13,* 2, 227–252.

Wijsen, J. 2009. On the consistent rewriting of conjunctive queries under primary key constraints. *Inf. Syst. 34,* 7, 578–601.

Wijsen, J. 2012. Certain conjunctive query answering in first-order logic. *ACM Trans. Database Syst. 37,* 2, 9.