Optimization of Answer Set Programs for Consistent Query Answering by Means of First-Order Rewriting

Aziz Amezian El Khalfioui
Jonathan Joertz
Dorian Labeeuw
Gaëtan Staquet
Jef Wijsen
jef.wijsen@umons.ac.be
University of Mons
Mons, Belgium

ABSTRACT

Consistent Query Answering (CQA) with respect to primary keys is the following problem. Given a database instance that is possibly inconsistent with respect to its primary key constraints, define a repair as an inclusion-maximal consistent subinstance. Given a Boolean query q, the problem CERTAINTY(q) takes a database instance as input, and asks whether q is true in every repair. For every Boolean conjunctive query q, the complement of CERTAINTY(q) can be straightforwardly implemented in Answer Set Programming (ASP) by means of a generate-and-test approach: first generate a repair, and then test whether it falsifies the query.

Theoretical research has recently revealed that for every self-join-free Boolean conjunctive query q, the complexity class of CERTAINTY(q) is one of **FO**, **L**-complete, or **coNP**-complete. Faced with this complexity trichotomy, one can hypothesize that in practice, the full power of generate-and-test is a computational overkill when CERTAINTY(q) is in the low complexity classes **FO** or **L**. We investigate part of this hypothesis within the context of ASP, by asking the following question: whenever CERTAINTY(q) is in **FO**, does a dedicated first-order algorithm exhibit significant performance gains compared to a generic generate-and-test implementation? We first elaborate on the construction of such dedicated first-order algorithms in ASP, and then empirically address this question.

CCS CONCEPTS

• **Information systems → Relational database query languages**;
• **Theory of computation → Constraint and logic programming**.

KEYWORDS

answer set programming; conjunctive queries; consistent query answering; database repairing; first-order rewriting; primary keys

CIKM '20, October 19–23, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-6859-9/20/10…$15.00
https://doi.org/10.1145/3340531.3411911

ACM Reference Format:
Aziz Amezian El Khalfioui, Jonathan Joertz, Dorian Labeeuw, Gaëtan Staquet, and Jef Wijsen. 2020. Optimization of Answer Set Programs for Consistent Query Answering by Means of First-Order Rewriting. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20), October 19–23, 2020, Virtual Event, Ireland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3340531.3411911

1 MOTIVATION

Consistent query answering (CQA) is a principled approach for dealing with inconsistent databases that has gained considerable research attention in the past twenty years [3, 22]. Inconsistency in databases can be caused, among others, by data integration. For a running example, consider the database instance of Fig. 1. The table r stores conference locations and should ideally satisfy PRIMARY KEY(*Conf*, *Year*) because no conference series is organized twice in a same year. This primary key is currently violated, because there are two distinct locations for CIKM 2021.

r

Conf	*Year*	*City*
CIKM	2020	Galway
CIKM	2021	Perth
CIKM	2021	Sydney

s

City	*Country*
Perth	Australia
Sydney	Australia
Galway	Ireland

Figure 1: Running Example

This paper focuses on violations of primary key constraints. A *repair* of a (possibly inconsistent) database instance is a maximal consistent subinstance. The database instance of Fig. 1 has two repairs, because there are two possible choices for the city of CIKM 2021.

A significant problem is how to answer queries on inconsistent databases that have multiple repairs. The CQA approach uses a cautious semantics: the *consistent answer* to a query is defined as the intersection of the query answers on all repairs. For example, the consistent answer to the query *"In which city will CIKM 2021 be organized?"* is empty, because the two repairs yield disjoint answers. On the other hand, *Australia* is a consistent answer to the query *"In which country will CIKM 2021 be organized?"*.

The computational complexity of CQA with respect to primary keys has been studied in considerable depth. In these studies, one usually focuses on Boolean queries, which return either "yes" or "no."

```
{ r_repair(Conf,Year,V) : r(Conf,Year,V) } == 1 :- r(Conf,Year,_).
         { s_repair(City,W) : s(City,W) } == 1 :- s(City,_).
                                               :- r_repair(X,Y,Z), s_repair(Z,X).
```

Figure 2: ASP Program for CERTAINTY(q_0).

The consistent answer to a Boolean query q is "yes" on a database instance if q is true on every repair; otherwise the consistent answer is "no." For example, "yes" is the consistent answer to the query *"Will CIKM 2021 be held in Australia?"*. For a fixed Boolean query q, CERTAINTY(q) is the following decision problem:

Decision problem CERTAINTY(q).

INPUT: A (possibly inconsistent) database instance **db**.
QUESTION: Is q true in every repair?

It is easy to see that CERTAINTY(q) is in **coNP** for all first-order queries q, since a non-deterministic polynomial-time algorithm can guess a subinstance of **db** and verify in polynomial time that it has guessed a repair that falsifies q. It is known since the early days of CQA [7] that there are simple conjunctive queries q for which CERTAINTY(q) is **coNP**-complete. An example of such a query is

$$q_0 := \exists X \exists Y \exists Z \left(r(\underline{X, Y}, Z) \wedge s(\underline{Z}, X) \right),$$

which on our example database is the (weird) question whether there is a conference whose name is that of the organizing country.

The focus of this paper is on solving CERTAINTY(q) by means of *Answer Set Programming (ASP)*, which is a notorious logic programming paradigm tailored toward **NP**-complete problems. Many ASP programs can be easily written by directly encoding the guess-and-check approach that underlies **NP**. In our approach, the program has to guess a repair, and then check whether that repair falsifies the query. In clingo ASP [9, 10], a program for CERTAINTY(q_0) takes only three lines of code, as illustrated by Fig. 2. We briefly explain this program. The first line says that for every primary-key value in `r`, one should select exactly one fact in `r` with that primary-key value. The fresh relation name `r_repair` is used for repairs of `r`. The second line operates analogously on `s`. The last line of the program does the check: it is a rule with an empty head. In ASP, an empty-head rule is an integrity constraint which asserts that the body of the rule must not be satisfied. Thus, the last rule in Fig. 2 asserts that q_0 must be false in the repair guessed by the first two lines. If we run this program in clingo ASP on a database instance, the output is either a repair that falsifies the query q_0, or the message "`UNSATISFIABLE`" telling us that no such repair exists.

The *guess-and-check* solution of Fig. 2 can be used to solve CERTAINTY(q) for any Boolean conjunctive query q. Such programs follow the *generate-and-test* approach that is typical of many ASP programs that solve **NP**-complete problems. The generate-part of a program for CERTAINTY(q) specifies, for every relation name in q, what is a repair of that relation. The test-part checks whether this leads to a repair that falsifies q. An advantage is that these programs are straightforward to write because of their syntactical simplicity. From a theoretical perspective, however, these programs exhibit an overkill in complexity whenever CERTAINTY(q) can be solved in polynomial time, as explained next.

Recent theoretical research has revealed much about the computational complexity of CERTAINTY(q) with q ranging over the negation-free fragment of the relational calculus. In particular, the complexity landscape is by now well understood for queries in sjfBCQ, i.e., the class of self-join-free Boolean conjunctive queries, which will be rigorously defined in Section 3. It is now known that the set {CERTAINTY(q) | $q \in$ sjfBCQ} exhibits a complexity trichotomy between **FO**, **L**-complete, and **coNP**-complete [16, 17]. Here, **FO** denotes the class of decision problems whose input is a relational database instance and that are solvable by a single query in relational calculus, or equivalently, in non-recursive datalog with negation, which is a sublanguage of ASP.

The preceding tells us that whenever CERTAINTY(q) is in **FO**, it can be solved in two different ways in ASP:

(1) by using the generic generate-and-test approach illustrated in Fig. 2; or
(2) by means of a program in non-recursive datalog with negation. Such a program is also called a *[consistent] first-order rewriting* of q.

We will explain later in this paper how to construct the programs referred to in item (2). The important thing to note for now is that (1) and (2) will give us two different (but equivalent) declarative logic programs, both written in ASP. It is then natural to ask:

> *Research question:* Is there a difference in run-time efficiency between "generate-and-test" and "first-order rewriting"?

In theory, programs in non-recursive datalog with negation are less expressive (and therefore, expectedly, faster) than generate-and-test programs that use the full expressive power of ASP. However, there are notorious computational problems, like primality testing or linear programming, where algorithms of theoretically lower complexity do not execute faster in practice. Could it be the same for CERTAINTY(q)?

Some comments are in place here. A main principle in declarative logic programming is that programmers focus on *what* is to be computed, not on *how* it is to be computed, nor on run-time efficiency. In this paper, we will stick to this principle. Our question is therefore *not* whether we can "hack" a logic program to make it run faster. Likewise, our research goal is *not* to develop the "fasted computer program ever" to solve CERTAINTY(q). Instead, using fairly straightforward encodings for consistent first-order rewriting and generate-and-test, we want to empirically compare the performances of both approaches within existing ASP solvers, like clingo or DLV [19].

This paper is organized as follows. Section 2 discusses related work. Section 3 introduces a small number of theoretical concepts that are needed for the remainder of the paper. Section 4 explains how to encode CERTAINTY(q) in non-recursive datalog with negation (and thus in ASP) whenever this is possible. The translation is tricky and has not yet been described in previous works. In Section 5, we ask the question: how many queries admit a first-order

rewriting? We show that there are database schemas where first-order rewritability is more a rule than an exception. In Section 6, we discuss the difference between solving CERTAINTY(q) or its complement, and why algorithms may perform differently on "yes"- and "no"-instances of CERTAINTY(q). At that point, we have all the background that is needed to run some experiments, which are described in Section 7. Most results in this paper concern Boolean queries, i.e., queries that ask for a yes/no (or true/false) answer. In Section 8, we discuss the implications of having non-Boolean queries with free variables. Finally, Section 9 concludes the paper.

Conquesto. This paper is accompanied by software released under a 3-clause BSD license at https://github.com/DocSkellington/Conquesto. This software allows for the construction of first-order rewritings and generate-and-test programs in both clingo and DLV syntax, and for the automated experimentation in both systems.

2 RELATED WORK

Consistent query answering (CQA) started in 1999 with a seminal paper by Arenas, Bertossi and Chomicki [1]. Two decades of research in CQA have recently been surveyed in [3, 22].

The suitability of Answer Set Programming (ASP) and stable model semantics for CQA has been recognized since the early days in theoretical research [2, 11]. In [20], the authors present a prototype system for CQA that is theoretically founded in ASP.

In CQA, the existence of consistent first-order rewritings, for different classes of queries and integrity constraints, is a recurrent research problem. For self-join-free conjunctive queries and primary keys, the problem has been studied in depth since 2005 [7, 8], and was eventually solved in 2015 [13, 14]. Experiments of CQA with respect to primary keys have been conducted on several prototype systems, including ConQuer [6], EQUIP [12], and CAvSAT [5]. The Hippo system [4] can also deal with primary key violations, but disallows quantifiers in queries. The recent study [5], in particular, has revealed that discrepancies may exists between the theoretical computational complexity of CERTAINTY(q) and observed empirical performances. In particular, it was observed that a generic SAT-based approach to CERTAINTY(q) may outperform solutions that use first-order rewriting. The main goal of the current paper is to investigate whether such observations also hold within existing ASP systems.

3 PRELIMINARIES

We assume a set of *relation names*. Every relation name r is associated with a *signature*, which is a pair $[\lambda, k]$ of positive integers with $\lambda \geq k$, where λ is the *arity* and $\{1, \ldots, k\}$ is the *primary key*. We write $r : [\lambda, k]$ to indicate that r has signature $[\lambda, k]$. A *database schema* is a finite set of relation names. From here on, we assume that some database schema has been fixed. For example, the database schema of our running example contains $r : [3, 2]$ and $s : [2, 1]$.

A *term* is a constant or a variable. If $\vec{t}$ is a sequence of terms, then $\text{vars}(\vec{t})$ denotes the set of variables that occur in $\vec{t}$. Whenever a set of variables is used in a context where a list of variables is expected, we assume that the variables are ordered according to some prefixed linear order on the set of all variables.

An r-atom (or simply atom if r is understood) is an expression $r(t_1, \ldots, t_\lambda)$ where each t_i is a term. An atom without variables is called a *fact*. When writing an atom, we will often (but not always) underline the primary-key positions: $r(\underline{t_1, \ldots, t_k}, t_{k+1}, \ldots, t_\lambda)$. A *database instance* **db** is a finite set of facts. A database instance is *consistent* if it does not contain two facts with the same relation name, say $r(\underline{\vec{a}_1}, \vec{b}_1)$ and $r(\underline{\vec{a}_2}, \vec{b}_2)$, such that $\vec{a}_1 = \vec{a}_2$ and $\vec{b}_1 \neq \vec{b}_2$. A *repair* of a database instance is an inclusion-maximal consistent subinstance. A *block* is a maximal set of facts with the same relation name that agree on all positions of the primary key. In Fig. 1, blocks are separated by dashed lines.

A *first-order query* $q(X_1, \ldots, X_n)$ is an expression in (safe) relational calculus with free variables $X_1, \ldots, X_n$. Such a query can be written as q for short if the free variables are clear from the context. Satisfaction of a query by a database instance is defined as usual (and not repeated here). Given a database instance **db**, a tuple $(c_1, \ldots, c_n)$ is a *consistent answer* if every repair of **db** satisfies $q(c_1, \ldots, c_n)$. For example, the query $q(Y) \coloneqq r(\text{"CIKM"}, Y, \text{"Perth"})$ asks for the years that CIKM was hosted by the city of Perth. The database of Fig. 1 satisfies $q(\text{"2021"})$, but 2021 is not a consistent answer, because there is a repair that falsifies $q(\text{"2021"})$. A query without free variables is called *Boolean*. The consistent answer to a Boolean query q is *true* (or "yes") if every repair satisfies q, and is *false* (or "no") otherwise.

A *conjunctive query* is a first-order query of the form

$$\exists \vec{X} \left(r_1(\vec{t}_1) \wedge \cdots \wedge r_n(\vec{t}_n) \right). \tag{1}$$

Note that each $\vec{t}_i$ can contain both constants and variables. Such a query is *self-join-free* if $i \neq j$ implies $r_i \neq r_j$. The class of self-join-free Boolean conjunctive queries is denoted sjfBCQ. When we write a query in sjfBCQ, we often omit quantifiers and replace $\wedge$ with comma (,). We write $\text{vars}(q)$ for the set of variables that occur in q. We write $q_1 \equiv q_2$ if queries q_1 and q_2 are equivalent (i.e., if they yield the same answer on every database instance).

Let q be a query in sjfBCQ of the form (1). Assume that the signature of each r_i is $[\lambda_i, k_i]$. We now show the construction of a generate-and-test program for CERTAINTY(q). In clingo and DLV, variables start with an uppercase letter, and constants with a lowercase letter. We will follow this convention in this paper. Using clingo-specific syntax, we specify repairs by adding the following rules, for every $i \in \{1, \ldots, n\}$:

$$\{ins\_r_i(X_1, \ldots, X_{\lambda_i}) : r_i(X_1, \ldots, X_{\lambda_i})\} == 1 \leftarrow r_i(\underline{X_1, \ldots, X_{k_i}}, Y_{k_i+1}, \ldots, Y_{\lambda_i}).$$

Here, ins_r_i is a fresh relation name of arity λ_i that is used for the facts inserted in a repair. To finish the program, we add a single empty-head rule that constrains the repairs to those that falsify q:

$$\leftarrow ins\_r_1(\vec{t}_1), \ldots, ins\_r_n(\vec{t}_n). \tag{2}$$

This yields a program with $n + 1$ rules.

Finally, we show how repairs can be specified in ASP without using clingo-specific syntax. We encode that every r_i-fact is either inserted (*ins*) in a repair or deleted (*del*). Furthermore, at least one fact from each block must be inserted in a repair. This is encoded

in the following four rules ($1 \leq i \leq n$):

$$\begin{aligned}
&\mathit{del_r_i}(X_1,\ldots,X_{\lambda_i}) &&\leftarrow r_i(X_1,\ldots,X_{\lambda_i}), \text{not } \mathit{ins_r_i}(X_1,\ldots,X_{\lambda_i}).\\
&\mathit{ins_r_i}(X_1,\ldots,X_{\lambda_i}) &&\leftarrow r_i(X_1,\ldots,X_{\lambda_i}), \text{not } \mathit{del_r_i}(X_1,\ldots,X_{\lambda_i}).\\
&\mathit{block_r_i}(X_1,\ldots,X_{k_i}) &&\leftarrow \mathit{ins_r_i}(X_1,\ldots,X_{\lambda_i}).\\
& &&\leftarrow r_i(X_1,\ldots,X_{\lambda_i}), \text{not } \mathit{block_r_i}(X_1,\ldots,X_{k_i}).
\end{aligned}$$

Here, $\mathit{ins\_r_i}$ and $\mathit{del\_r_i}$ are fresh relation names of arity λ_i, and $\mathit{block\_r_i}$ is a fresh relation of arity k_i. Together with (2), this yields a program with $4n+1$ rules. Note incidentally that there is no need to encode that at most one fact from each block must be inserted in a repair, because a Boolean conjunctive query that is false in a database instance, is also false in every subinstance.

4 CONSISTENT FIRST-ORDER REWRITING

In this section, we introduce the approach known as *consistent first-order rewriting*. In Section 4.1, we recall a fundamental result from the literature stating that it is decidable whether a self-join-free conjunctive query has a consistent first-order rewriting. Our novel and nontrivial contribution is to translate such a rewriting into a program that can be executed by an ASP engine. We will first illustrate this translation in Section 4.2, and then provide a formal theoretical treatment in Section 4.3.

4.1 Theoretical Background

A *consistent first-order rewriting of* $q(\vec{X})$ is another query $\Upsilon(\vec{X})$ in relational calculus, with the same free variables as q, such that for every database instance **db** and every tuple $\vec{c}$ of constants (of the same length as $\vec{X}$) the following are equivalent:

(1) $q(\vec{c})$ is true on every repair of **db**.
(2) $\Upsilon(\vec{c})$ is true on **db**.

A special case is where q is Boolean. If a Boolean query q has a consistent first-order rewriting, then, by the definition of **FO**, CERTAINTY(q) is in **FO**. If q has a consistent first-order rewriting, then it can be solved by a single query Υ without the need for computing any database repair. For example, consider the query *"Did/will CIKM ever take place in Perth?"*:

$$q_1 := \exists Y \left(r(\underline{\text{"CIKM"}, Y}, \text{"Perth"}) \right).$$

It is easy to verify that q_1 is true in every repair if for some year, Perth was/is the only host city for CIKM stored in the database. The latter condition can be expressed in relational calculus:

$$\Upsilon_1 := \exists Y \left(\begin{array}{l} r(\underline{\text{"CIKM"}, Y}, \text{"Perth"}) \wedge \\ \neg \exists Z \left(r(\underline{\text{"CIKM"}, Y}, Z) \wedge Z \neq \text{"Perth"} \right) \end{array} \right).$$

On the example database of Fig. 1, the query Υ_1 is false, and therefore there is a repair that falsifies q_1. Clearly, the falsifying repair is the one that chooses Sydney as the host city for CIKM 2021. Since the focus of this paper is on ASP, we still have to convert Υ_1 to ASP, which yields the ASP program of Fig. 3(b). For comparison, we also give the generate-and-test program in Fig. 3(a). Note that these two programs solve the same problem, but are syntactically and algorithmically very different. The applicability of consistent first-order rewriting was settled by the following theorem.

Theorem 4.1 ([14]). *There exists an algorithm for the following problem: Given a self-join-free (not necessarily Boolean) conjunctive query q, return a consistent first-order rewriting of q if it exists; otherwise return the message* "there exists no consistent first-order rewriting."

We will now explain how to construct a consistent first-order rewriting in ASP. Details of this construction in relational calculus and SQL can be found in [14, 21]. After some examples in Section 4.2, we give a formal treatment in Section 4.3.

4.2 Examples

Roughly, a consistent first-order rewriting Υ of a query q is constructed by structural induction on q. The base case is simple: if q is the empty query, then it is always *true*. The induction step partitions q in some head atom $r(\vec{t})$ and a tail $q' := q \setminus \{r(\vec{t})\}$. The induction hypothesis gives us a consistent first-order rewriting Υ' of the tail q'. The formula Υ then combines a rewriting of $r(\vec{t})$ with Υ', as illustrated next for the head atom $r(\underline{X}, X, Y, Y, c)$, where c is a constant. This example contains repeated variables and a constant; it can be easily generalized to queries with more repeated variables and constants. By induction, there is a consistent first-order rewriting $\Upsilon'(X,Y)$ of $q'(X,Y) := q \setminus \{r(\underline{X}, X, Y, Y, c)\}$. A consistent first-order rewriting of q is as follows:

$$\Upsilon := \exists X \left(\begin{array}{l} \exists Y r(\underline{X}, X, Y, Y, c) \wedge \\ \forall X' \forall Y \forall Y' \forall Z \left(r(\underline{X}, X', Y, Y', Z) \rightarrow \begin{pmatrix} X' = X \wedge \\ Y' = Y \wedge \\ Z = c \ \ \wedge \\ \Upsilon'(X,Y) \end{pmatrix} \right) \end{array} \right)$$

Informally, this formula states that there is an r-block (indicated by X) such that every fact $r(\underline{x}, x', y, y', z)$ in this block satisfies $x' = x$, $y' = y$, and $z = c$, and therefore every repair satisfies $r(\underline{X}, X, Y, Y, c)$; the subformula Υ', which exists by induction, expresses that the remainder of the query will also be satisfied by every repair. We now eliminate $\forall$ and $\rightarrow$, and define some subformulas:

$$\Upsilon \equiv \exists X \left(\begin{array}{l} \exists Y r(\underline{X}, X, Y, Y, c) \wedge \\ \underbrace{\neg \exists X' \exists Y \exists Y' \exists Z \left(r(\underline{X}, X', Y, Y', Z) \wedge \neg \overbrace{\begin{pmatrix} X' = X \wedge \\ Y' = Y \wedge \\ Z = c \ \ \wedge \\ \Upsilon'(X,Y) \end{pmatrix}}^{\text{goodfact}(X,X',Y,Y',Z)} \right)}_{\text{badblock}(X)} \end{array} \right)$$

From here on, whenever a block is fixed, the predicate goodfact will hold true for facts in that block that satisfy the query under consideration. The predicate badblock will hold true if goodfact is false for some fact in the fixed block. In ASP programs, goodfact and badblock will be abbreviated as `gf` and `bb` respectively.

By induction, we can assume an ASP program for a predicate `yes'`(X,Y) that evaluates $\Upsilon'(X,Y)$. The formula Υ can now be translated into ASP:

```
yes :- r(X,X,Y,Y,c), not bb(X).
bb(X) :- r(X,X',Y,Y',Z), not gf(X,X',Y,Y',Z).
gf(X,X',Y,Y',Z) :- X'=X, Y'=Y, Z=c, yes'(X,Y),
                   r(X,X',Y,Y',Z).
```

The last atom in the last rule is needed to make that rule safe. Note that since this is the translation of a first-order formula, the resulting ASP program is in non-recursive datalog with negation. It is possible to eliminate the equalities in the last rule, which yields the following:

```
{ r_repair(Conf,Year,V) : r(Conf,Year,V) } == 1 :- r(Conf,Year,_).
                                                :- r_repair("CIKM",Y,"Perth").
```

(a) Generate-and-test

```
                yes :- r("CIKM",Y,"Perth"), not otherCity(Y).
otherCity(Y) :- r("CIKM",Y,Z), Z!="Perth".
```

(b) Consistent first-order rewriting

Figure 3: Two ASP Programs for CERTAINTY(q_1) **with** $q_1 \coloneqq \exists Y\,(r(\text{“CIKM”}, Y, \text{“Perth"}))$**.**

```
  yes :- r(X,X,Y,Y,c), not bb(X).
bb(X) :- r(X,X',Y,Y',Z), not gf(X,X',Y,Y',Z).
gf(X,X,Y,Y,c) :- yes'(X,Y), r(X,X,Y,Y,c).
```

Alternatively, the predicate goodfact can be avoided by equivalently rewriting Υ as follows:

$$\Upsilon \equiv \exists X \left(\begin{array}{l} \exists Y r(\underline{X}, X, Y, Y, c) \wedge \\ \underbrace{\neg\, \exists X' \exists Y \exists Y' \exists Z \left(\begin{array}{l} \big(r(\underline{X}, X', Y, Y', Z) \wedge X' \neq X\big) \\ \vee \big(r(\underline{X}, X', Y, Y', Z) \wedge Y' \neq Y\big) \\ \vee \big(r(\underline{X}, X', Y, Y', Z) \wedge Z \neq c\big) \\ \vee \big(r(\underline{X}, X', Y, Y', Z) \wedge \neg \Upsilon'(X, Y)\big) \end{array} \right)}_{\text{badblock}(X)} \end{array} \right)$$

Now the translation in ASP is:

```
  yes :- r(X,X,Y,Y,c), not bb(X).
bb(X) :- r(X,X',Y,Y',Z), X' != X.
bb(X) :- r(X,X',Y,Y',Z), Y' != Y.
bb(X) :- r(X,X',Y,Y',Z), Z != c.
bb(X) :- r(X,X',Y,Y',Z), not yes'(X,Y).
```

The previous example did not illustrate how to pick the first atom to rewrite, nor how to deal with free variables. The treatment of free variables is easy: free variables must be handled as if they were constants. The choice of the first atom to rewrite is more complicated and requires a difficult theoretical treatment that is developed in [14] and summarized in the next subsection.

4.3 Generalization

The proof of Theorem 4.1 in [14] relies on the *attack graph* of a self-join-free conjunctive query q, which is a (unique) directed graph whose vertices are the atoms of q. For the understanding of the current paper, it is sufficient to know that q has a consistent first-order rewriting if and only if its attack graph is acyclic. Moreover, if the attack graph of q is acyclic, then its atoms must be rewritten in a topological order of this attack graph.

So let $q(\vec{X})$ be a self-join-free conjunctive query with relation names $r_1, r_2, \ldots, r_n$, listed in a topological order of the attack graph of q. For $i \in \{1, \ldots, n\}$, define q_i to be the subquery that contains all (and only) the atoms with relation names $r_i, r_{i+1}, \ldots, r_n$. In particular, $q_1 = q$.

For every $i \in \{1, \ldots, n\}$, we define a predicate yes_i of appropriate arity that rewrites q_i. We define yes_{n+1} as *true*, which will terminate the induction. We now give the consistent first-order rewriting of q_1. Let the r_1-atom of q_1 be $r_1(\vec{u}, \vec{w})$ with $\vec{u} = u_1, \ldots, u_k$ and $\vec{w} = w_1, \ldots, w_\ell$. Let $\vec{V} = V_1, \ldots, V_\ell$ be a sequence of distinct variables such that for every $i \in \{1, \ldots, \ell\}$, if w_i is a variable that does not occur in $\vec{X}\vec{u}w_1w_2 \cdots w_{i-1}$, then let $V_i == w_i$; otherwise V_i is a fresh variable. Here, we use == to denote the syntactic identity of symbols. Thus, we copy the occurrence w_i of a variable, rather than using a fresh variable, if it is the leftmost occurrence of a non-free variable in $r_1(\vec{u}, \vec{w})$. The consistent first-order rewriting in ASP is given next; this rewriting ensures the correct treatment of free variables and the safety of rules (i.e., every variable that occurs in a rule must occur in a non-negated atom of the body).

$$\begin{array}{rcl} \text{yes}_1(\vec{X}) & \leftarrow & \text{saferange}_1(\vec{X}), r_1(\vec{u}, \vec{w}), \\ & & \neg\text{badblock}_1(\vec{X}, \vec{u}). \\ \text{saferange}_1(\vec{X}) & \leftarrow & q_1. \\ \text{badblock}_1(\vec{X}, \vec{u}) & \leftarrow & \text{saferange}_1(\vec{X}), r_1(\vec{u}, \vec{V}), \\ & & \neg\text{goodfact}_1(\vec{X}, \vec{u}, \vec{V}). \\ \text{goodfact}_1(\vec{X}, \vec{u}, \vec{V}) & \leftarrow & \text{saferange}_1(\vec{X}), r_1(\vec{u}, \vec{V}), C^{=}(\vec{X}, \vec{u}, \vec{V}), \\ & & \text{yes}_2(\text{vars}(\vec{X}\vec{u}\vec{w}) \cap \text{vars}(q_2)). \end{array}$$

where $C^{=}(\vec{X}, \vec{u}, \vec{V})$ is a set of equalities defined as follows:

(1) for $i \in \{1, \ldots, \ell\}$, if w_i is either a variable in $\vec{X}\vec{u}$ or a constant, then $C^{=}(\vec{X}, \vec{u}, \vec{V})$ contains $V_i = w_i$. Note that V_i and w_i will never be identical variables in this case; and
(2) for $1 \leq i < j \leq \ell$, if w_i is a variable not in $\vec{X}\vec{u}$ and $w_i == w_j$, then $C^{=}(\vec{X}, \vec{u}, \vec{V})$ contains $V_i = V_j$.

The predicates saferange_1 are needed to make the rules safe. Finally, by induction, the program for the predicate yes_2 is constructed in the same way. The following example illustrates several technical aspects.

Example 4.2. Let $q(X_1, X_2) \coloneqq r_1(\underline{b, U}, U, Y, Y, X_1), r_2(\underline{Y}, X_2, c)$. The rewriting of $q_1(X_1, X_2)$ goes as follows. Note that the first occurrence of Y is preserved in the following rewriting; the second occurrence of Y is replaced with V_3, and leads to the equality $V_3 = Y$. Since the free variable X_2 does not occur in the r_1-atom, the predicate sr_1 is needed to make the first rule safe.

```
yes_1(X1,X2) :- sr_1(X1,X2), r_1(b,U,U,Y,Y,X1),
                not bb_1(X1,X2,b,U).
 sr_1(X1,X2) :- r_1(b,U,U,Y,Y,X1), r_2(Y,X2,c).
bb_1(X1,X2,b,U) :- sr_1(X1,X2), r_1(b,U,V1,Y,V3,V4),
                   not gf_1(X1,X2,b,U,V1,Y,V3,V4).
gf_1(X1,X2,b,U,V1,Y,V3,V4) :- sr_1(X1,X2),
                              r_1(b,U,V1,Y,V3,V4),
                              V1=U, V3=Y, V4=X1,
                              yes_2(X2,Y).
```

We next give the first-order rewriting of $q_2(X_2, Y) \coloneqq r_2(\underline{Y}, X_2, c)$, in which the equalities $V_1 = X_2$ and $V_2 = c$ originate from item (1) in the definition of $C^{=}(\vec{X}, \vec{u}, \vec{V})$.

```
yes_2(X2,Y) :- sr_2(X2,Y), r_2(Y,X2,c),
               not bb_2(X2,Y,Y).
 sr_2(X2,Y) :- r_2(Y,X2,c).
bb_2(X2,Y,Y) :- sr_2(X2,Y), r_2(Y,V1,V2),
                not gf_2(X2,Y,Y,V1,V2).
gf_2(X2,Y,Y,V1,V2) :- sr_2(X2,Y), r_2(Y,V1,V2),
                      V1=X2, V2=c.
```

This terminates Example 4.2. □

The previous rewriting of $r_1(\vec{u}, \vec{w})$ uses 4 rules. It is easy to eliminate the rule for $\mathsf{saferange}_1(\vec{X})$, by substituting q_1 for every occurrence of $\mathsf{saferange}_1(\vec{X})$. We thus obtain the following result.

Proposition 4.3. *For every query q in* sjfBCQ *with n atoms, if* CERTAINTY(q) *is in* **FO**, *then it can be solved by a program with at most $3n$ rules in non-recursive datalog with negation.*

As we have seen in Section 4.2, we can also eliminate the predicate $\mathsf{goodfact}_1$, but this will generally not decrease the number of rules. Technically, the elimination of $\mathsf{goodfact}_1$ proceeds as follows.

$$\begin{aligned}
\mathsf{yes}_1(\vec{X}) \quad &\leftarrow \quad \mathsf{saferange}_1(\vec{X}), r_1(\vec{u}, \vec{w}), \\
&\qquad \neg\mathsf{badblock}_1(\vec{X}, \vec{u}). \\
\mathsf{saferange}_1(\vec{X}) \quad &\leftarrow \quad q_1. \\
\mathsf{badblock}_1(\vec{X}, \vec{u}) \quad &\leftarrow \quad \mathsf{saferange}_1(\vec{X}), r_1(\vec{u}, \vec{V}), \\
&\qquad \neg\mathsf{yes}_2(\mathsf{vars}(\vec{X}\vec{u}\vec{w}) \cap \mathsf{vars}(q_2)).
\end{aligned}$$

In addition, for every equality $V_i = t$ in $C^{=}(\vec{X}, \vec{u}, \vec{V})$, we add the following rule:

$$\mathsf{badblock}_1(\vec{X}, \vec{u}) \quad \leftarrow \quad \mathsf{saferange}_1(\vec{X}), r_1(\vec{u}, \vec{V}), V_i \neq t.$$

The latter rewriting style has been used in the experiments of Section 7.

5 HOW MANY QUERIES ARE REWRITABLE?

Theorem 4.1 does not tell us anything about the number of queries in sjfBCQ that have a consistent first-order rewriting. When studying first-order rewriting as an optimization technique, a significant question is what fraction of queries in sjfBCQ have a consistent first-order rewriting. To make this question meaningful, we must first determine how we count queries in sjfBCQ. It seems natural to count modulo equivalence. For example, $\exists X\,(s(X, \text{“Ireland”}))$ and $\exists Y\,(s(Y, \text{“Ireland”}))$ are syntactically distinct but semantically equivalent, and will therefore be counted only once. It easily follows from the theory of conjunctive queries that two queries in sjfBCQ are equivalent if and only if they are the same up to a renaming of variables.

Next, we will also count modulo a change of constants. For example, even though $\exists X\,(s(X, \text{“Ireland”}))$ and $\exists X\,(s(X, \text{“Australia”}))$ are not equivalent, they will count for only one query. This is motivated by the fact that the existence of a consistent first-order rewriting for a query q in sjfBCQ does not depend on the actual values of the constants, nor on the eventuality of repeated constants. For example, if q is a query in sjfBCQ that does not contain relation name r, then $q \cup \{r(\vec{x}, a, b)\}$ has a consistent first-order rewriting if and only if $q \cup \{r(\vec{x}, c, c)\}$ has a consistent first-order rewriting, where a, b, c are constants. In view of this, we will fix one constant c from here on, and define sjfBCQ$[c]$ as the set of queries in sjfBCQ that use no constant other than c. We write sjfBCQ$[c]/{\equiv}$ for the set sjfBCQ$[c]$ modulo equivalence. Finally, we will count for a fixed database schema, which leads to the following problem.

Database schema	Number of non-equivalent queries in sjfBCQ$[c]$	How many have a consistent first-order rewriting?
$r_1 : [2, 1], r_2 : [2, 1]$	52	50
$r_1 : [3, 2], r_2 : [2, 1]$	203	194

Table 1: Counts for non-equivalent queries in sjfBCQ$[c]$ **for two database schemas. More than** 95% **have a consistent first-order rewriting.**

Counting problem.

INPUT: Database schema $r_1 : [\lambda_1, k_1], \ldots, r_n : [\lambda_n, k_n]$.
QUESTIONS:

(1) What is the number of queries in sjfBCQ$[c]/{\equiv}$ of the form $r_1(\vec{t}_1), \ldots, r_n(\vec{t}_n)$; and
(2) how many of these queries have a consistent first-order rewriting?

Proposition 5.1. *For the preceding counting problem, the answer to question* (1) *is B_m with $m = 1 + \sum_{i=1}^{n} \lambda_i$, where $B_0, B_1, \ldots$ are the Bell numbers.*

The proof of the preceding proposition is easy and omitted. We have no analytic solution for question (2) in the preceding counting problem. Nevertheless, Table 1 gives empirically obtained solutions for two schemas that often occur in theoretical studies because these are the smallest schemas that exhibit the full complexity landscape of CERTAINTY(q). The second schema was also used in our running example (cf. Fig. 1). Interestingly, most queries (more than 95%) turn out to have a consistent first-order rewriting.

6 ASYMMETRY IN YES- AND NO-INSTANCES

The claim that the two programs of Fig. 3 solve the same problem deserves some nuance. Indeed, a generate-and-test program for CERTAINTY(q) can halt as soon as it finds a solution (called a stable model in ASP terminology); such a solution is a repair that falsifies the query. If the clingo system executes an ASP program that has no stable model, it returns the message “UNSATISFIABLE”. So the generate-and-test program actually computes the complement of CERTAINTY(q): if the answer to CERTAINTY(q) is “no,” then a falsifying repair will be returned; if the the answer to CERTAINTY(q) is “yes,” then the message “UNSATISFIABLE” will be returned. On the other hand, the consistent first-order rewriting of Fig. 3(b) derives “yes” if and only if the the answer to CERTAINTY(q) is “yes.”

For the experiments, given a query q in sjfBCQ, it seems therefore significant to distinguish between “yes”-instances and “no”-instances of the problem CERTAINTY(q). Indeed, since an ASP program that uses generate-and-test solves the complement of CERTAINTY(q), it is halted as soon as a repair is found that falsifies q (which will be the first stable model found by the ASP engine). It is to be expected that generate-and-test takes longer on “yes”-instances, which by definition have no repair falsifying q and therefore do not admit early halting. Informally, every possible

repair of a "yes"-instance has to be considered before it can be decided that no repair falsifies the query. For first-order rewriting, the situation is exactly the opposite, in the following sense. A consistent first-order rewriting Υ of a query q is implemented in an ASP program with a "principal" rule (cf. Section 4.3; we assume no free variables here):

$$\text{yes}_1 \quad \leftarrow \quad r_1(\vec{u}, \vec{w}), \neg \text{badblock}_1(\vec{u}).$$

such that the input database instance is a "yes"-instance if (and only if) yes_1 can be inferred. Such an ASP program can halt as soon as yes_1 is derived, which always happens on (and only on) "yes"-instances.

To conclude, "no"-instances are, expectedly, the "easiest" inputs for generate-and-test, and may be the "hardest" inputs for first-order rewriting. Therefore, if first-order rewriting is faster than generate-and-test on "no"-instances, then it is likely to be faster also on "yes"-instances. For this reason, we will focus in the first place on "no"-instances in our experiments.

A caveat is in place here, however: it remains speculative to contemplate on easy and hard instances for CERTAINTY(q). Indeed, programs written in ASP and datalog are *declarative*, which means that the control flow of their execution is not part of the program, but left to the ASP engine that executes them. The time that an ASP program uses to solve an instance of CERTAINTY(q) therefore depends on which amount of optimization is implemented in this engine. Also, efficiency may be gained by downsizing the inputs of CERTAINTY(q), without changing the answer to the problem. For example, it is easily verified that if q contains $r(\underline{X}, \text{"c"})$, then the answer to CERTAINTY(q) does not change if one prunes the input by deleting all r-blocks with cardinality ≥ 2. However, it is outside the aim of the current paper to look into the ASP engine or to help it with some preprocessing.

7 EXPERIMENTS

In this section, we first describe the experimental environment that was designed and implemented based on the theory of the previous sections, and then report on some experiments.

7.1 Queries

In the description of our experiments, we assume queries that use relation names $r_1, r_2, \ldots, r_n$ ($n \geq 1$). The arity of r_i will be denoted by λ_i. We will assume that the atoms of a query q are listed in the following form:

$$q = r_1(t_{11}, \ldots, t_{1\lambda_1}), \ldots, r_n(t_{n1}, \ldots, t_{n\lambda_n}). \tag{3}$$

Thus, t_{ij} is the term that occurs at the jth position of the ith atom. We will also say that t_{ij} occurs in *column* (i, j). For example, if $q = r_1(\underline{x}, y), r_2(\underline{y}, c)$, then y occurs in columns $(1, 2)$ and $(2, 1)$. We also say that in the fact $r_2(\underline{a}, b)$, the constant a occurs in column $(2, 1)$, and the constant b in column $(2, 2)$. Like in Section 5, we will consider queries in sjfBCQ$[c]$, i.e., c is the only constant that can be used in queries. Since all queries are self-join-free, fixing a single constant does not lose generality. Indeed, if t_{ij} would be a different constant, say $t_{ij} == b$, then the problem CERTAINTY(q) does not change if we change t_{ij} into c, and, accordingly, switch b and c in column (i, j) of the database instances that are given as inputs to CERTAINTY(q).

As a special case, we define the *emptiness query* as a query of the form (3) such that every t_{ij} is a variable that occurs only once in the query. For example, $q = r_1(\underline{x}, y), r_2(\underline{z}, w)$. Clearly, the emptiness query is false in some repair of a database instance **db** if and only if for some $i \in \{1, \ldots, n\}$, the set of r_i-facts in **db** is empty.

The experimental setup is to fix $r_1, \ldots, r_n$ and their signatures, and to consider *all* (modulo equivalence) queries in sjfBCQ$[c]$ of the form (3). This is different from experiments in the literature that typically use a limited sample of queries.

7.2 Generating Database Instances

The database generator takes a positive integer parameter, denoted π, called the *parametric database size* from here on. We describe next our non-randomized generation of "no"-instances for CERTAINTY(q), where q is of the form (3). After that, we describe how this generation procedure is changed to yield "yes"-instances and randomized instances. The motivation for distinguishing between "no"- and "yes"-instances was given in Section 6; their generation is deterministic. Randomized instances were added to allow for some non-determinism in the experiments.

Generation of "no"-instances. To generate "no"-instances for the problem CERTAINTY(q), we first associate a domain to every column (i, j), and then exhaustively add all facts $r_i(a_{i1}, \ldots, a_{i\lambda_i})$ that can be formed under the restriction that a_{ij} must belong to the domain associated with column (i, j). The domains are as follows. If t_{ij} is the constant c, then the domain of (i, j) is a singleton $\{b\}$ with $b \neq c$. If t_{ij} is a variable, say X, and t_{ij} is the ℓth occurrence of X in q (counting from the left), then the domain of (i, j) is the set $\{\pi \cdot (\ell - 1) + 1, \ldots, \pi \cdot \ell\}$, whose cardinality is π. Thus, if the same variable X occurs more than once in q, then the columns at which X occurs have disjoint domains. It can be easily verified that if q is not the emptiness query (which was defined in Section 7.1), then all generated database instances are "no"-instances for CERTAINTY(q). For the emptiness query, the "no"-instance used in the experiments is the empty database instance. From here on, for parametric database size π and query q, we write size(π, q) for the number of database facts generated by the method just described. Note that for a given value of π, the value of size(π, q) varies in function of the number of occurrences of the constant c in q. In particular, for a query of the form (3) that is not the emptiness query, we have that

$$\text{size}(\pi, q) = \sum_{i=1}^{n} \left(\prod_{j=1}^{\lambda_i} \pi^{f(i,j)} \right), \tag{4}$$

where $f(i, j) = 1$ if t_{ij} is a variable, and $f(i, j) = 0$ otherwise.

Generation of "yes"-instances. "Yes"-instances are generated according to the same lines as "no"-instances, but using different domains. If t_{ij} is the constant c, then the domain associated to column (i, j) is $\{c\}$. If t_{ij} is a variable and (i, j) is a primary-key position, then the values in column (i, j) are restricted to the domain $\{1, \ldots, \pi\}$. Finally, if t_{ij} is a variable and (i, j) is a non-primary-key position, then the values in column (i, j) are restricted to the singleton domain $\{1\}$. By exhaustively inserting all facts that can be constructed under these domain restrictions, we obviously obtain a "yes"-instance for CERTAINTY(q). This "yes"-instance is then extended, until its number of facts is equal to size(π, q), by

Algorithm 1 Experimental Run

Require: database schema $r_1 : [\lambda_1, k_1], \ldots, r_n : [\lambda_n, k_n]$
Require: time-out time
Require: upper value $\pi_{\max}$ for parametric database size
  $count \leftarrow 0$
  for each $q \in$ sjfBCQ$[c]/\equiv$ of the form $r_1(\vec{t_1}), \ldots, r_n(\vec{t_n})$ **do**
    if q has a consistent first-order rewriting **then**
      $count \leftarrow count + 1$
      $\Upsilon \leftarrow$ consistent first-order rewriting of q
      $P \leftarrow$ generate-and-test program for CERTAINTY(q)
      for parametric database size π from 1 to $\pi_{\max}$ **do**
        generate a "no"-instance of size size(π, q)
          execute Υ until it finishes or is timed-out
          execute P until it finishes or is timed-out
        generate a "yes"-instance of size size(π, q)
          execute Υ until it finishes or is timed-out
          execute P until it finishes or is timed-out
        generate a randomized instance of size size(π, q)
          execute Υ until it finishes or is timed-out
          execute P until it finishes or is timed-out
      end for
    end if
  end for

allowing the values in primary-key columns to take also values in $\{\pi + 1, \pi + 2, \ldots\}$. Thus, for a given parametric database size π and query q, the generated "yes"- and "no"-instances have the same number of facts; this number is equal to size(π, q).

Generation of "randomized" instances. For a given parametric database size π and query q of the form (3), let $N := \text{size}(\pi, q)$. We first generate n positive integers $N_1, \ldots, N_n$, each one close to N/n, that sum up to N. Then, for every $i \in \{1, \ldots, n\}$, we generate N_i facts of the form $r_i(a_{i1}, \ldots, a_{i\lambda_i})$. If the constant c occurs in q, then each a_{ij} is chosen by a coin flip between c and a value selected uniformly at random in $\{1, 2, \ldots, N\}$. Otherwise a_{ij} is a value selected uniformly at random in $\{1, 2, \ldots, N\}$. For our settings, we get the following desirable property: unless r_i is unary, there is only a small probability of generating a same r_i-fact twice, whereas many r_i-facts will agree on some position. As can be expected, most database instances generated in this way are "no"-instances for the problem CERTAINTY(q).

7.3 Experimental Run

Our experimental environment allows executing experimental runs described in Algorithm 1. For every execution of Υ or P in an experimental run, CPU times are recorded. The main algorithmic difficulty is in testing whether a query has a consistent first-order rewriting, and in constructing such a rewriting in ASP if it exists (cf. Section 4.3).

7.4 Empirical Findings

Our software system, released under a 3-clause BSD license at https://github.com/DocSkellington/Conquesto, allows for the automated experimentation with both clingo [9, 10] and DLV [19]. For reasons of space limitation, we will focus on the clingo experiments here; experiments with DLV can be found at the preceding URL.

We will discuss here our findings for an experimental run with schema $r_1 : [3, 2], r_2 : [2, 1]$, whose length is $3 + 2 = 5$. This schema often appears in theoretical studies because it is a small schema that nevertheless exhibits the full complexity landscape of CERTAINTY(q). It is also the schema of our running example in Fig. 1. Even though our experimental environment described in Algorithm 1 can take as input any database schema, one should remind from Proposition 5.1 that the number of non-equivalent queries is exponential in the length of the schema. Consequently, experimental runs are unfeasible for schemas of great length. Furthermore, unlike most existing studies, we did not want to sample the query space sjfBCQ$[c]/\equiv$, because it is not clear what would be a representative sample.

Mean and median execution times are taken over all queries in sjfBCQ$[c]/\equiv$ of the form $r_1(\underline{t_{11}, t_{12}}, t_{13}), r_2(\underline{t_{21}}, t_{22})$ that have a consistent-first-order rewriting. As mentioned in Table 1, there are 203 non-equivalent queries of this form, of which 194 have a consistent first-order rewriting (the final value of *count* in Algorithm 1). All mean and median values are thus computed over 194 non-equivalent queries, which is a high number compared to related past studies (for comparison, the experiments in [5, 12] consider 7 first-order rewritable queries).

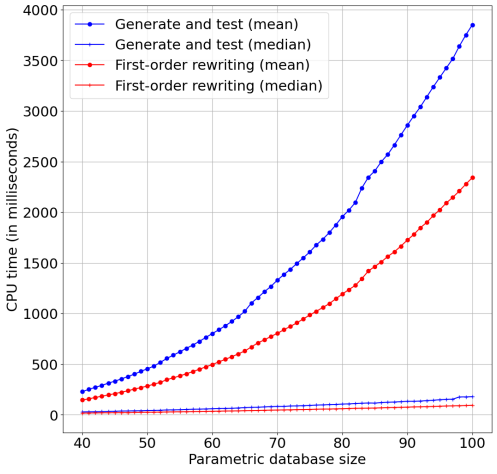
The experimental results for clingo on "no"-, "yes"-, and randomized instances are shown, respectively, in Fig. 4(a), 4(b), and 4(c). Note that these figures use a different scale for the ordinate axis. The time-out time was set to 10^5ms = 100s. In the computation of the mean execution time, we used this time of 100s whenever the first-order rewriting or the generate-and-test program was timed-out. The legend *"lower bound for the mean,"* rather than *"mean,"* is used in figures to indicate that time-out times were used in the computation of the mean. The median is not affected by time-outs, since on all instances, less than 50% of the queries were timed-out. Some time-outs were due to memory issues when running clingo. Concerning the actual database size, Equation (4) tells us that the r_1-relation contains 10^6 tuples when the parametric database size equals 100 (for comparison, Dixit and Kolaitis [5] also report on first-order rewriting experiments on databases with 10^6 tuples per relation).

The following findings hold for all instances, no matter whether they are "yes"-, "no"-, or randomized instances.

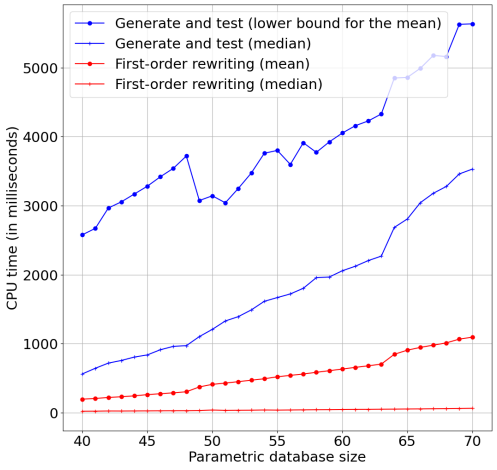
- The median values (+-line) are systematically below the mean values (•-line), indicating a right-skewed distribution (i.e., skewed to higher values). Informally, a majority of the 194 programs finish in a short time, while some programs take a considerably longer time (or are even timed-out).
- The mean values for first-order rewriting (red •-line) are systematically below the mean values for generate-and-test (blue •-line). The median values for first-order rewriting (red +-line) are systematically below the median values for generate-and-test (blue +-line). This means that the optimization by means of first-order rewriting is effective.

When we compare "yes"- and "no"-instances, we observe the following.

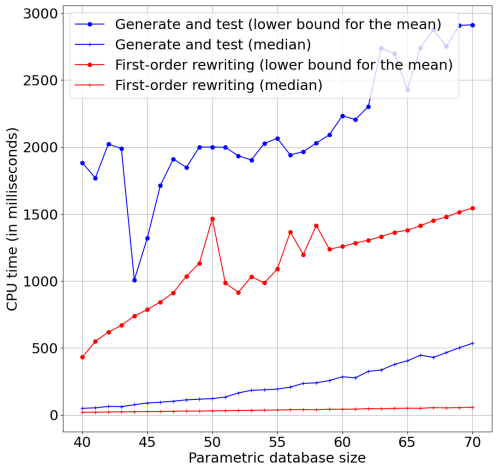
- The experiments confirm our hypothesis that the speedup of first-order rewriting compared to generate-and-test is more pronounced on "yes"-instances than on "no"-instances. On



(a) "No"-instances



(b) "Yes"-instances



(c) Randomized instances

Figure 4: Experimental results for clingo

"yes"-instances the mean values for first-order rewriting (red •-line) are below the median values for generate-and-test (blue +-line). This is no longer true for "no"-instances, nor for randomized instances (which are mostly "no"-instances).
- Generate-and-test takes considerably more time on "yes"-instances than on "no"-instances, which was to be expected.

Overall, our experiments systematically show performance gains of first-order rewriting compared to generate-and-test. Nonetheless, in view of the big gap between **FO** and **NP**-complete, one could have expected that these gains would have been more pronounced than what is empirically observed.

Finally, we note that the generate-and-test results in Fig. 4 were obtained by using clingo-specific syntax with cardinality constraints in the head, illustrated in Fig. 2. The conclusions, however, did not change when we used the standard ASP encoding given in Section 3. Experiments with DLV also showed systematic performance gains of first-order rewriting compared to generate-and-test.

8 FREE VARIABLES

Our experiments have focused on consistent query answering for Boolean queries. We now examine how to move to queries with free variables, which can return sets of tuples. We will argue that this extension is obvious in consistent first-order rewriting, but needs some work in generate-and-test.

In Section 4.3, we showed the construction of a consistent first-order rewriting in the presence of free variables. For example, for the query $q(Y) := r(\text{"CIKM"}, Y, \text{"Perth"})$, in which Y is free, a consistent first-order rewriting is obtained by changing, in the program of Fig. 3(b), the head of the first rule from `yes` into `yes(Y)` (but more work would be needed if Y occurred at a non-primary-key position).

The situation is different for the generate-and-test program of Fig. 3(a). Indeed, the last rule of this program requires an empty head, which means that the variable `Y` is inherently not free. The solution we envisage is to create, for every possible value for `Y`, a new program in which this value is substituted for `Y`. For our example database of Fig. 1, there will be two such programs: one ending with the rule `:- r_repair("CIKM","2020","Perth")`, and the other with the rule `:- r_repair("CIKM","2021","Perth")`. In general, consider that we ask for all consistent answers to $q(\vec{X})$ on a database instance **db**. We can first compute $\{\vec{c} \mid \textbf{db} \models q(\vec{c})\}$, which is the standard answer to $q(\vec{X})$ on **db**. From here on, assume that this answer is equal to $\{\vec{c}_1, \ldots, \vec{c}_m\}$. Since it is easily verified that every consistent answer must belong to this set, it suffices now to compute which $\vec{c}_i$ is a consistent answer. To this end, for each $i \in \{1, \ldots, m\}$, we solve CERTAINTY($q_{[\vec{X} \mapsto \vec{c}_i]}$), where $q_{[\vec{X} \mapsto \vec{c}_i]}$ is the Boolean query obtained from $q(\vec{X})$ by substituting $\vec{c}_i$ for $\vec{X}$. This boils down to executing m slightly distinct generate-and-test programs. Although m is polynomial in the size of **db**, this may significantly slow down generate-and-test compared to consistent first-order rewriting. Indeed, as explained before, in the consistent first-order rewriting approach, the rewriting $\Upsilon(\vec{X})$ is computed once, does not depend on **db**, yet returns consistent answers on any database instance **db**.

9 CONCLUDING REMARKS

We studied consistent query answering for self-join-free conjunctive queries with respect to primary keys, within the paradigm of Answer Set Programming. In summary, contributions of our work are as follows:

- We showed how to construct programs in non-recursive datalog with negation for problems CERTAINTY(q) that are in **FO**. These programs have lower complexity upper bounds than ASP programs that use the generate-and-test approach.
- Experiments in clingo ASP show that these theoretical differences in complexity also reveal themselves, at least moderately, in practice. As we argued in the previous section, even a constant factor speedup on Boolean queries will give us a polynomial speedup on queries with free variables.
- Interestingly, we showed that CERTAINTY(q) is in **FO** for 95% of the queries over two common database schemas.

Therefore, we conclude that first-order rewriting is an advantageous alternative to generate-and-test.

We conclude by listing three issues for future research. First, the **FO**-boundary of CERTAINTY(q) has recently been revealed for queries q with negation [15], and for multiple keys per relation [18]. It is interesting to extend the results of the current paper to these more general settings. Second, although ASP is declarative by design, there exist nevertheless some good practices and tricks for achieving performance gains. The current paper stayed away from such tricks. In the future, we aim to investigate the optimization of ASP programs obtained from first-order rewriting. For example, if a query q in sjfBCQ can be split into two subqueries, say q_1 and q_2, that have no variables in common, then CERTAINTY(q) can be solved by solving CERTAINTY(q_1) and CERTAINTY(q_2) independently. Some preliminary experiments (not reported here) indicate that such splitting is an effective optimization step. Third, our experiments supported our hypothesis that "yes"- and "no"-instances pose different challenges to generate-and-test and first-order rewriting. For either approach, it remains an open question to more precisely identify and generate the hardest instances of CERTAINTY(q). Another perspective is to look at "typical" instances in particular real-world settings. In database systems environments, for example, almost consistent database instances (i.e., "almost yes"-instances) should be more common than highly inconsistent instances.

REFERENCES

[1] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. 1999. Consistent Query Answers in Inconsistent Databases. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania, USA*, Victor Vianu and Christos H. Papadimitriou (Eds.). ACM Press, 68–79. https://doi.org/10.1145/303976.303983

[2] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. 2003. Answer sets for consistent query answering in inconsistent databases. *Theory Pract. Log. Program.* 3, 4-5 (2003), 393–424. https://doi.org/10.1017/S1471068403001832

[3] Leopoldo E. Bertossi. 2019. Database Repairs and Consistent Query Answering: Origins and Further Developments. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Dan Suciu, Sebastian Skritek, and Christoph Koch (Eds.). ACM, 48–58. https://doi.org/10.1145/3294052.3322190

[4] Jan Chomicki, Jerzy Marcinkowski, and Slawomir Staworko. 2004. Hippo: A System for Computing Consistent Answers to a Class of SQL Queries. In *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings (Lecture Notes in Computer Science)*, Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari (Eds.), Vol. 2992. Springer, 841–844. https://doi.org/10.1007/978-3-540-24741-8_53

[5] Akhil A. Dixit and Phokion G. Kolaitis. 2019. A SAT-Based System for Consistent Query Answering. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings (Lecture Notes in Computer Science)*, Mikolás Janota and Inês Lynce (Eds.), Vol. 11628. Springer, 117–135. https://doi.org/10.1007/978-3-030-24258-9_8

[6] Ariel Fuxman, Elham Fazli, and Renée J. Miller. 2005. ConQuer: Efficient Management of Inconsistent Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, Fatma Özcan (Ed.). ACM, 155–166. https://doi.org/10.1145/1066157.1066176

[7] Ariel Fuxman and Renée J. Miller. 2005. First-Order Query Rewriting for Inconsistent Databases. In *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings (Lecture Notes in Computer Science)*, Thomas Eiter and Leonid Libkin (Eds.), Vol. 3363. Springer, 337–351. https://doi.org/10.1007/978-3-540-30570-5_23

[8] Ariel Fuxman and Renée J. Miller. 2007. First-order query rewriting for inconsistent databases. *J. Comput. Syst. Sci.* 73, 4 (2007), 610–635. https://doi.org/10.1016/j.jcss.2006.10.013

[9] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2014. Clingo = ASP + Control: Preliminary Report. *CoRR* abs/1405.3694 (2014). arXiv:1405.3694 http://arxiv.org/abs/1405.3694

[10] Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. 2011. Advances in *gringo* Series 3. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings (Lecture Notes in Computer Science)*, James P. Delgrande and Wolfgang Faber (Eds.), Vol. 6645. Springer, 345–351. https://doi.org/10.1007/978-3-642-20895-9_39

[11] Gianluigi Greco, Sergio Greco, and Ester Zumpano. 2003. A Logical Framework for Querying and Repairing Inconsistent Databases. *IEEE Trans. Knowl. Data Eng.* 15, 6 (2003), 1389–1408. https://doi.org/10.1109/TKDE.2003.1245280

[12] Phokion G. Kolaitis, Enela Pema, and Wang-Chiew Tan. 2013. Efficient Querying of Inconsistent Databases with Binary Integer Programming. *PVLDB* 6, 6 (2013), 397–408. https://doi.org/10.14778/2536336.2536341

[13] Paraschos Koutris and Jef Wijsen. 2015. The Data Complexity of Consistent Query Answering for Self-Join-Free Conjunctive Queries Under Primary Key Constraints. In *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Tova Milo and Diego Calvanese (Eds.). ACM, 17–29. https://doi.org/10.1145/2745754.2745769

[14] Paraschos Koutris and Jef Wijsen. 2017. Consistent Query Answering for Self-Join-Free Conjunctive Queries Under Primary Key Constraints. *ACM Trans. Database Syst.* 42, 2 (2017), 9:1–9:45. https://doi.org/10.1145/3068334

[15] Paraschos Koutris and Jef Wijsen. 2018. Consistent Query Answering for Primary Keys and Conjunctive Queries with Negated Atoms. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, Jan Van den Bussche and Marcelo Arenas (Eds.). ACM, 209–224. https://doi.org/10.1145/3196959.3196982

[16] Paraschos Koutris and Jef Wijsen. 2019. Consistent Query Answering for Primary Keys in Logspace. In *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal (LIPIcs)*, Pablo Barceló and Marco Calautti (Eds.), Vol. 127. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 23:1–23:19. https://doi.org/10.4230/LIPIcs.ICDT.2019.23

[17] Paraschos Koutris and Jef Wijsen. 2020. Consistent Query Answering for Primary Keys in Datalog. *Theory of Computing Systems* (2020), 1–57. https://doi.org/10.1007/s00224-020-09985-6

[18] Paraschos Koutris and Jef Wijsen. 2020. First-Order Rewritability in Consistent Query Answering with Respect to Multiple Keys. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2020, Portland, OR, USA, June 14-19, 2020*, Dan Suciu, Yufei Tao, and Zhewei Wei (Eds.). ACM, 113–129. https://doi.org/10.1145/3375395.3387654

[19] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.* 7, 3 (2006), 499–562. https://doi.org/10.1145/1149114.1149117

[20] Mónica Caniupán Marileo and Leopoldo E. Bertossi. 2010. The consistency extractor system: Answer set programs for consistent query answering in databases. *Data Knowl. Eng.* 69, 6 (2010), 545–572. https://doi.org/10.1016/j.datak.2010.01.005

[21] Jef Wijsen. 2012. Certain conjunctive query answering in first-order logic. *ACM Trans. Database Syst.* 37, 2 (2012), 9:1–9:35. https://doi.org/10.1145/2188349.2188351

[22] Jef Wijsen. 2019. Foundations of Query Answering on Inconsistent Databases. *SIGMOD Rec.* 48, 3 (2019), 6–16. https://doi.org/10.1145/3377391.3377393