The Consistency Extractor System: Answer Set Programs for Consistent Query Answering in Databases

Monica Caniupán[a,*], Leopoldo Bertossi[b]

[a] *Universidad del Bío-Bío, Depto. Sistemas de Información, Concepción, Chile.*
[b] *Carleton University, School of Computer Science, Ottawa, Canada.*

Abstract

We describe the *Consistency Extractor System* (*ConsEx*) that computes consistent answers to Datalog queries with negation posed to relational databases that may be inconsistent with respect to certain integrity constraints. In order to solve this task, *ConsEx* uses answers set programming. More precisely, *ConsEx* uses disjunctive logic programs with stable models semantics to specify and reason with the *repairs*, i.e. with the consistent virtual instances that minimally depart from the original database. The consistent information is invariant under all repairs. *ConsEx* achieves efficient query evaluation by implementing *magic sets* techniques. We describe the general methodology, its optimizations for query answering, and the architecture of the system. We also present encouraging experimental results.

Key words: Databases, Integrity Constraints, Inconsistency, Answer Set Programs

1. Introduction

Integrity constraints (ICs) capture the semantics of data with respect to the external reality that the database is expected to model. Databases should satisfy their ICs, but in practice, databases may become inconsistent with respect to them [10]. Nevertheless, in most cases only a small portion of the data is inconsistent (i.e. participates in inconsistency with respect to the ICs). In consequence, an inconsistent database can still give us useful and semantically correct information. The process of characterizing and obtaining consistent answers to queries is called *Consistent Query Answering* (CQA) [12].

Consistent query answering makes sense if, to make thing worse, it becomes impossible, undesirable or too difficult to restore the consistency of the database by applying some form of materialized data cleaning. Conventional data cleaning might be a non-deterministic and expensive process, that also leads to a loss of potentially useful information. In some cases, we might actually have no permission to modify data or a clear way about how to proceed in this direction. This is the case, for example, when autonomous and independent data sources are virtually integrated.

*Corresponding author

Email addresses: mcaniupa@ubiobio.cl (Monica Caniupán), bertossi@scs.carleton.ca (Leopoldo Bertossi)

Enforcing consistency at query time is an alternative to enforcing consistency at the instance level. This idea is applicable in, among other situations, (a) virtual data integration, (b) in the case of a single database on which, for better performance purposes, some ICs are not enforced, (c) in the materialization of a database whose content is obtained from other sources. In the latter case, the ETL process (Extract, Transform, and Load) can be seen, modeled and implemented as the materialization of a process of CQA.

Before computing consistent answers to queries, they have to be formally characterized, in precise logical terms. This was first done in [4]: Consistent answers to a query are invariant under all the forms of restoring consistency by minimal changes on the database. The alternative consistent versions of the original instance are called *repairs*. So, *consistent answers* are those that can be obtained as usual answers from every repair. The notion of consistent answer is related in spirit to the notion of *certain answer* as found, for example, in virtual data integration [1]: Certain answers are those that are true of all the possible legal instances for the integration system.

More precisely, given a relational database instance D and a set IC of ICs, a (minimal) repair [4] is an instance D' of the same schema, that satisfies IC, and differs from D by a minimal set of whole database tuples under set inclusion. Repairs do not have to be materialized; actually there may be too many of them [10]. In principle, they are virtual instances that are used to give a model-theoretic definition of consistent answer. Mechanism for CQA can and have to be assessed against this semantic definition.

Example 1. Consider the database schema *Student*(*id*, *name*). The functional dependency (FD) $id \rightarrow name$ establishes that each student identifier is associated with a unique name value. The first two tuples of the following database instance D, that can be the result of the integration of two data sources, violate the FD:

Student	
id	name
1	*smith*
1	*peter*
2	*jones*

Consistency can be minimally restored by deleting either tuple *Student*(*1*, *smith*) or tuple *Student* (*1*, *peter*). If we delete both tuples, the resulting database is not a repair since it does not satisfy the minimality requirement. Therefore, there are two database repairs:

Student	
id	name
1	*smith*
2	*jones*

Student	
id	name
1	*peter*
2	*jones*

We can see that certain information persists in the repairs, e.g. tuple *Student*(2, *jones*) is in both of them, reflecting the fact that it does not participate in the violation of the FD. On the other hand, the "inconsistent tuples" *Student*(*1*, *smith*) and *Student*(*1*, *peter*) do not persist in all the repairs. If we want to know the id of student *jones*, we can pose the query *Student*(*x*, *jones*). The answer to this query is $\langle 2 \rangle$ in both repairs, therefore the consistent answer is $\langle 2 \rangle$.

Moreover, for the boolean disjunctive query *Student*(*1*, *smith*) $\vee$ *Student*(*1*, *peter*), the consistent answer is *yes*, since each repair satisfies one of the disjuncts in the query. Notice that if we

had simultaneously deleted all the tuples participating in an inconsistency, we would have lost this kind of information. □

Already in [4] some computational mechanisms were presented that do not use or compute the repairs, but pose a new, rewritten query to the given inconsistent database. The answers to the new query are the consistent answers to the original query. Cf. [12] for a survey containing more recent results of this kind.

The algorithm for CQA proposed in [4] -implemented and slightly extended in [25]- is applicable to limited classes of queries and ICs, e.g. projection-free conjunctive queries, functional dependencies, full inclusion dependencies. In these cases, the first-order (FO) query can be rewritten into a new FO query that posed and answered as usual to given instance, obtains the consistent answers to the original query. Other FO query rewriting methods for CQA were presented in [27, 39, 46]. However, they are still limited in their applicability, which is due to the intrinsic higher data complexity of CQA. Cf. [10, 12] for surveys in this direction, and [60] for more recent results about non-FO rewritability of CQA. The on-the-fly, at query time, resolution of inconsistencies is what makes the FO rewriting for CQA difficult or impossible. This is in contrast to, for example, querying databases through DL-Lite ontologies by FO query rewriting [20], where the ontology basically extends the underlying database without being in logical conflict with it.

As a consequence, languages for query rewriting than are more expressive than FO Logic became necessary. Actually, they first and naturally emerged when logic programs were used to specify the repairs, with the idea of query this compact specification of repairs in order to obtain the consistent answers. In several papers [5, 8, 9, 15, 16, 32, 43], database repairs were specified as the stable models of disjunctive logic programs with stable model semantics [40] (aka. *answer set programs*). The logic programs that specify the repairs are called *repair programs*.

The logic programs introduced in [16] are most general and take into consideration the possible occurrences of null values in the databases. Furthermore, they capture the use of null values for restoring consistency with respect to referential ICs (RICs). Actually, in [9, 16] it was shown that there is a one-to-one correspondence between the stable models of the repair program and the repairs with respect to *RIC*-acyclic sets of ICs, i.e. sets of constraints that do not present cycles involving RICs (cf. Section 2).

The data complexity of CQA [27] matches the data complexity of query evaluation from disjunctive logic programs with stable model semantics, which in the general case is Π^P_2-complete [30]. However, it is possible to identify classes of ICs and queries, for which CQA has (provably) lower data complexity. Among them we find those classes that allow for FO query rewriting, as we mentioned above. For them CQA can be solved in polynomial time in data complexity. Also *coNP*-complete cases have been identified [9, 27, 28, 39].

1.1. Problem Statement

Consistent answers to a query can be computed by evaluating a (properly modified) query against the repair program. This can be done, e.g. with the *DLV* system [47]. However, using repair programs for CQA in a straightforward and naive manner is not very efficient (cf. Section 6). Therefore, it becomes relevant to optimize their evaluation.

Optimizations for CQA have been studied and introduced before in the context of data integration. In [32] techniques to efficiently compute and store database repairs are described. Basically, database facts participating in violations of universal ICs are located and extracted from the database (which does not contain null values). This splits the database in two parts: (a)

The *affected* database, which contains data violating the ICs. (b) The *safe* database, which stores consistent data. That operation makes it possible to speed up the computation of repairs. They are computed for the affected part only, and at a second stage they are combined with the safe part. Even though this methodology reduces the amount of data participating in the evaluation of repair programs, the focus is still on the computation of database repairs. In this paper we do not concentrate on explicit computation of repairs, but on consistent query answering.

1.2. Contributions and Outline

In this work we show that only a subset of the program and of the database facts are needed to compute answers to a specific query. This subset can be captured by using *Magic Sets* (MS) methodologies which transform the combination of the query program and the repair program into a new program that -in essence- contains a subset of the original rules in the repair program, namely those that are relevant to evaluate the query at hand. In particular, with MS only a relevant subset of the database will be used for query evaluation.

Classically, MS optimizes the bottom-up processing of queries in deductive (Datalog) databases by simulating a top-down, query-directed evaluation [6, 7, 26]. Recently, the MS techniques have been extended to logic programs with stable models semantics [35, 29, 42, 44].

In this paper we show how to adopt and adapt those techniques to our repair programs, resulting in a sound and complete MS methodology for CQA from repair programs with program constraints. This MS methodology is the one implemented in the *ConsEx* system. In Section 6, we show that the use of MS in the evaluation of queries considerably improves the execution time.

In *ConsEx* we use, optimize, and implement the repair logic programs introduced in [16]. In consequence, *ConsEx* can be applied to relational databases containing NULL, and used for CQA with respect to arbitrary universal ICs, acyclic sets of referential ICs, and NOT-NULL constraints. The NULL appearing in the database or introduced to restore consistency with respect to RICs is as used in DBMSs that conform to the SQL standard. In [16] a FO semantics for this kind of NULLs was presented. The notion of consistency used in [16] is relative to this semantics, and the same applies to the repair semantics.

The queries supported by *ConsEx* are Datalog queries with negation, which goes beyond first-order queries. Consistent answers to queries are computed by evaluating the magic version of the program that consists of the general (query independent) repair program plus the particular query. The magic program is automatically generated by *ConsEx*, and then it is internally passed as an input to *DLV*, which evaluates it. All this is done in interaction with a IBM DB2 relational DBMS.

ConsEx is the most general implementation of CQA, and the optimizations introduced both on the repair programs and query answering make it an interesting tool for experimenting with and doing CQA on real databases stored in a commercial DBMS. In *ConsEx* the execution time of query programs is considerably faster than the naive execution of query programs against repairs programs (cf. Section 6).

In this paper we also describe the methodologies implemented in *ConsEx*, and the features, functionalities and performance of the system. We also state in precise terms and prove results mentioned in [21]. This paper also provides all the logical foundations for the MS methodology described in [22], and extends the experimental results presented therein.

The rest of the paper is organized as follows: In Section 2, we recall some concepts and terminology. Section 3 describes the repair programs. Section 4 presents the MS methodology

for query evaluation against repair programs. Section 5 describes the architecture and interface of *ConsEx*. Section 6 presents the results of the experimental evaluation of CQA. Section 7 presents conclusions and future work.

2. Preliminaries

In this section we introduce some relevant concepts, classes of integrity constraints, database repairs, and consistent query answering.

2.1. Databases and integrity constraints

A relational database schema $\Sigma = (\mathcal{U}, \mathcal{R}, \mathcal{B})$ consists of a possibly infinite database domain $\mathcal{U}$, with $null \in \mathcal{U}$, a set $\mathcal{R}$ of database predicates on $\mathcal{U}$, each of them with an ordered finite set of attributes, and a set $\mathcal{B}$ of built-in predicates, e.g. $<, >, =, \neq$. $R[i]$ denotes the attribute in position i of predicate $R \in \mathcal{R}$. There is a predicate $IsNull(\cdot)$ on $\mathcal{U}$, such that $IsNull(c)$ is true iff c is *null*.

A database instance for schema Σ is a finite set D of ground atoms of the form $R(c_1, ..., c_n)$, which are called *database tuples*. Here, $R \in \mathcal{R}$ and $(c_1, ..., c_n)$ is a *tuple* of constants in $\mathcal{U}$. The extensions for built-in predicates are fixed, and possibly infinite in every database instance.

Queries are formulas over the first-order language $\mathcal{L}(\Sigma)$ determined by Σ. Given a database instance D, a tuple of constants $\bar{t}$ in $\mathcal{U}$ is an answer to a query $\mathcal{Q}(\bar{x})$ in D iff $D \models \mathcal{Q}(\bar{t})$, i.e. $\mathcal{Q}(\bar{x})$ becomes true in D when the variables in $\bar{x}$ are replaced by the corresponding constants in $\bar{t}$. Sometimes we will also consider queries that are expressed in other logical languages based on the predicates in Σ, e.g. queries in Datalog or in some of its extensions.

There is also a fixed set IC of integrity constraints, that are sentences in the first-order language $\mathcal{L}(\Sigma)$. They are expected to be satisfied by any database instance of Σ, but they may not. More precisely, in this paper an *integrity constraint* is a sentence $\psi \in \mathcal{L}(\Sigma)$ of the form [16]:

$$\forall \bar{x}(\bigwedge_{i=1}^{m} P_i(\bar{x}_i) \longrightarrow \exists \bar{z}(\bigvee_{j=1}^{n} Q_j(\bar{y}_j, \bar{z}_j) \vee \varphi)), \tag{1}$$

where $P_i, Q_j \in \mathcal{R}$, $\bar{x} = \bigcup_{i=1}^{m} \bar{x}_i$, $\bar{z} = \bigcup_{j=1}^{n} \bar{z}_j$, $\bar{y}_j \subseteq \bar{x}$, $\bar{x} \cap \bar{z} = \emptyset$, $\bar{z}_i \cap \bar{z}_j = \emptyset$ for $i \neq j$, and $m \geq 1$.[1] Here φ is a formula containing only disjunctions of built-in atoms from $\mathcal{B}$, whose variables appear in the antecedent of the implication.[2] We will assume that there exists a propositional atom **false** $\in \mathcal{B}$ that is always false in the database. Domain constants different from *null* may appear in a constraint of the form (1). Particular cases of this general form are the universal and the referential ICs.

A *universal integrity constraint* (UIC) is a sentence in $\mathcal{L}(\Sigma)$ that is logically equivalent to a sentence of the form:

$$\forall \bar{x}(\bigwedge_{i=1}^{m} P_i(\bar{x}_i) \rightarrow \bigvee_{j=1}^{n} Q_j(\bar{y}_j) \vee \varphi). \tag{2}$$

A *referential integrity constraint* (RIC) is a sentence of the form:

$$\forall \bar{x}(P(\bar{x}) \rightarrow \exists \bar{z}\, Q(\bar{y}, \bar{z})), \tag{3}$$

with $P, Q \in \mathcal{R}$, $\bar{y} \subseteq \bar{x}$, and $\bar{z}$ non-empty.[3]

---

[1] Note that if $\bar{z}_i \cap \bar{z}_j \neq \emptyset$ the formula can be rewritten as an equivalent formula such that $\bar{z}_i \cap \bar{z}_j = \emptyset$.

[2] The left hand side of a implication is called the antecedent, while that the right hand side is called the *consequent* of the implication.

[3] For simplification purposes, we assume that the existential variables appear in the last attributes of Q, but they may appear anywhere else in Q.

Notice that our RICs contain at most one database atom in the consequent. For instance, tuple-generating joins in the consequent are excluded, and this is due to the fact that RICs will be repaired using *null* (for the existential variables), whose participation in joins is problematic. It would be easy to adapt our repair programs and MS methodology by including joins in the consequents and also the use of arbitrary non-null values in the domain or labeled nulls as used in data exchange [36] to represent existentially quantified variables. However, these latter alternatives open the ground for undecidability of CQA [19] due to the potentially infinite number of repairs that appear when violations of RICs are restored. This is avoided in our case by using the null value *null*, with its SQL semantics, to restore consistency, as proposed in [16].

The class (1) of ICs includes most of the ICs commonly found in database practice, e.g. a *denial constraint* can be expressed as $\bar{\forall}\bar{x}(\bigwedge_{i=1}^{m} P_i(\bar{x}_i) \longrightarrow \mathbf{false})$. Functional dependencies can be expressed by several implications of the form (1), each of them with a single equality in the consequent. Partial inclusion dependencies are RICs, and full inclusion dependencies are UICs. We can also specify unary constraints, also called *check constraints*. They allow to express conditions on each row of a table, and then they can be formulated with one predicate in the antecedent of (1) and only a formula φ in the consequent. For example, $\forall xy(P(x, y) \rightarrow y > 0)$ is a unary constraint.

Moreover, we consider *NOT NULL*-constraints (NNC), which are common in commercial DBMSs. They prevent certain attributes from taking a null value. A *NOT NULL*-constraint is of the form [16]:

$$\forall \bar{x}(P(\bar{x}) \wedge IsNull(x_i) \rightarrow \mathbf{false}), \quad (4)$$

where $x_i \in \bar{x}$ is in the position of the attribute that cannot take the null value. Notice that a NNC is not of the form (1), because it contains the special predicate *IsNull*.

Example 2. Consider the schema $\Sigma = \{Student(id, name), Grad(id, name), Assistant(id, course)\}$. The following are ICs:

(a) The functional dependency (FD) $Student : id \rightarrow name$, expressed in $\mathcal{L}(\Sigma)$ by the UIC $\forall id\ name_1\ name_2(Student(id, name_1) \wedge Student(id,name_2) \rightarrow (name_1 = name_2))$.

(b) The full inclusion dependency (IND) $Grad[id,name] \subseteq Student[id, name]$, expressed by the UIC $\forall id\ name\ (Grad(id, name) \rightarrow Student(id, name))$.

(c) The non-full inclusion dependency $Assistant[id] \subseteq Student[id]$ can be expressed as a RIC $\forall id\ course(Assistant(id, course) \rightarrow \exists name\ Student(id, name))$. Here, $\bar{x} = (id, course)$, $\bar{y} = (id)$, and $\bar{z} = (name)$.

(d) The NNC $\forall id, name(Student(id, name) \wedge IsNull(id) \rightarrow \mathbf{false})$ specifies that the first attribute of relation *Student* cannot take a null value. Notice that the (a) and (d) together specify that *id* is the primary key of relation *Student*. □

We consider a fixed finite set *IC* of ICs of the forms (1) and (4). Notice that sets of constraints of these forms are always logically consistent, in the classical sense, since database instances with empty relations always satisfy them. *RIC*-acyclic sets of constraints will be particularly relevant. We need some auxiliary notions before introducing them.

Definition 1. [21] The directed *dependency graph* $\mathcal{G}(IC)$ for a set *IC* of ICs of the form (1) and (4) is defined as follows: Each database predicate P in $\mathcal{R}$ is a node. There is a directed edge (P_i, P_j) from P_i to P_j iff there exists a constraint $\psi \in IC$, such that P_i appears in the antecedent of ψ and P_j appears in its consequent. In addition, there is an edge (P_i, P_i) from P_i to P_i if P_i appears in the antecedent of a constraint ψ that has only built-in predicates in its consequent or the *IsNull* predicate in its antecedent. Finally, a node of $\mathcal{G}(IC)$ is called a *sink* (*source*) if it has only incoming (outgoing) edges. □

Intuitively, the directed dependency graph shows the interaction between predicates involved in ICs. A *connected component* in a graph is a maximal subgraph such that, for every pair (A, B) of its vertices, there is a path from A to B or from B to A. For a graph $\mathcal{G}$, $C(\mathcal{G}) := \{c \mid c$ is a connected component in $\mathcal{G}\}$; and $\mathcal{V}(\mathcal{G})$ is the set of vertices of $\mathcal{G}$.

Definition 2. [16] Given a set IC of UICs, RICs, and NNCs, IC_U denotes the set of UICs and NNCs in IC. The *contracted dependency graph* of IC, $\mathcal{G}^C(IC)$, is obtained from $\mathcal{G}(IC)$ by replacing, for every $c \in C(\mathcal{G}(IC_U))$,[4] the vertices in $\mathcal{V}(c)$ by a single vertex and deleting all the edges obtained from elements of IC_U. Finally, IC is said to be *RIC-acyclic* if $\mathcal{G}^C(IC)$ has no cycles. □

In other words, the contracted dependency allows us to determine if a set of ICs is RIC-acyclic.

Example 3. Figure 1(a) shows the dependency graph $\mathcal{G}(IC)$ for the set IC of UICs containing $\varphi_1 : \forall x(S(x) \rightarrow Q(x))$ and $\varphi_2 : \forall x(Q(x) \rightarrow R(x))$, the RIC $\varphi_3 : \forall x(Q(x) \rightarrow \exists y T(x, y))$, and the NNC $\forall x(S(x) \wedge IsNull(x) \rightarrow$ **false**).

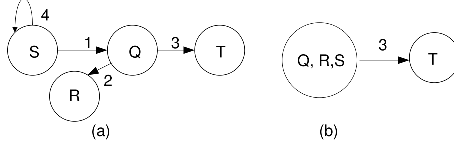


Figure 1: Dependency Graph $\mathcal{G}(IC)$ and Contracted Dependence Graph $\mathcal{G}^C(IC)$

Edges 1 and 2 correspond to the UICs φ_1 and φ_2 resp., edge 3 to the referential IC, and edge 4 to the NNC. Nodes R and T are sink nodes.

Figure 1(b) shows the contracted dependency graph $\mathcal{G}^C(IC)$, which is obtained by replacing in $\mathcal{G}(IC)$ the vertices Q, R, S by a vertex labeled with $\{Q, R, S\}$, and deleting edges 1, 2, and 4. Since there are no loops in $\mathcal{G}^C(IC)$, the set IC is RIC-acyclic. However, if we add a new UIC: $\forall xy(T(x, y) \rightarrow R(y))$ to IC, all the vertices belong to the same connected component. Figure 2 shows $\mathcal{G}(IC)$ and $\mathcal{G}^C(IC)$, respectively. Since there is a self-loop in $\mathcal{G}^C(IC)$, the set of ICs is not RIC-acyclic.

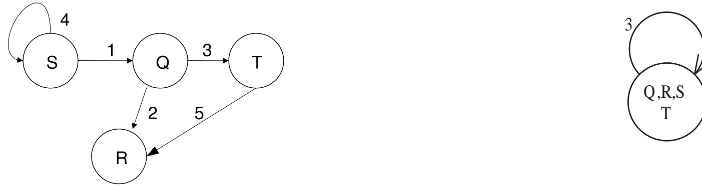


Figure 2: Non-RIC-acyclic Set of ICs

□

A set of UICs is always *RIC*-acyclic, as expected.

[4]Notice that, for every $c \in C(\mathcal{G}(IC_U))$, it holds $c \in C(\mathcal{G}(IC))$.

2.2. Constraint satisfaction and null values

A database instance D is *consistent* if it satisfies the given set IC of ICs. Otherwise, it is *inconsistent* with respect to IC. The semantics of constraint satisfaction in presence of the null value we consider in this paper was defined in [16]. Now we briefly recall it.

Definition 3. [16] (a) For t a term, i.e. a variable or a domain constant, let $pos^R(\psi, t)$ be the set of positions in predicate $R \in \mathcal{R}$ where term t appears in IC ψ. The set of *relevant attributes* for an IC ψ of the form (1) is:
$\mathcal{A}(\psi) = \{R[i] \mid x$ is variable present at least twice in ψ,[5] and $i \in pos^R(\psi, x)\}$
$\cup \{R[i] \mid c$ is a constant in ψ and $i \in pos^R(\psi, c)\}$.[6]
(b) For a set $\mathcal{A}$ of attributes and a predicate $P \in \mathcal{R}$, $P^{\mathcal{A}}$ denotes the predicate P restricted to the attributes in $\mathcal{A}$. $D^{\mathcal{A}}$ denotes the database D with all its database atoms projected onto the attributes in $\mathcal{A}$. $D^{\mathcal{A}}$ has the same underlying domain $\mathcal{U}$ as D. □

In other words, the set of relevant attributes associated to an IC contains the attributes that have to be considered when checking the satisfaction of the constraint, e.g. the attributes in joins, in built-ins, etc. Notice that if the built-ins have variables that are redundant or not needed, this might have the undesirable effect of transforming an attribute in relevant when it does not need to. For example the constraint $\forall xy(P(x,y) \rightarrow y > 3 \vee x = x)$ is equivalent to $\forall xy(P(x,y) \rightarrow y > 3)$, but the first has relevant attributes x and y (more precisely, $P[1], P[2]$) and the second one, only y (or $P[2]$). Possibly unnecessary checks will not produce any logical problem.

Example 4. Consider the UIC ψ : $\forall xyz(P(x,y,z) \rightarrow R(x,y))$ and the database instance $D = \{P(a,b,null), P(b,null,a), R(a,b)\}$. Here variables x and y appear twice in ψ, thus $\mathcal{A}(\psi) = \{P[1], R[1], P[2], R[2]\}$. Since we want to verify if the values in the first two attributes in P also appear in R, the variable z (or attribute $P[3]$) is not relevant to check the satisfaction of the constraint.

$D^{\mathcal{A}(\psi)} = \{P^{\mathcal{A}(\psi)}(a,b), P^{\mathcal{A}(\psi)}(b,null), R^{\mathcal{A}(\psi)}(a,b)\}$ is the database instance restricted to the attributes that are relevant to check ψ. □

Intuitively, a constraint is satisfied if any of the relevant attributes has a *null* or the constraint is satisfied in the traditional way, that is, according to first-order satisfaction and with null values treated as any other constant. In order to indicate that an attribute takes the null value, we use the special predicate $IsNull(\cdot)$. This predicate is needed since using the built-in comparison atom $c = null$ will not work in traditional database management systems, where this equality would be always evaluated as *unknown* (the unique names assumption does not apply to null values [54]).

Definition 4. [16] (a) A constraint ψ of the form (1) is satisfied in the database instance D, denoted $D \models_N \psi$, iff $D^{\mathcal{A}(\psi)} \models \psi^N$, where ψ^N is:

$$\forall \bar{x}(\bigwedge_{i=1}^{m} P_i^{\mathcal{A}(\psi)}(\bar{x}_i) \rightarrow (\bigvee_{v_j \in \mathcal{A}(\psi) \cap \bar{x}} IsNull(v_j) \vee \exists \bar{z}(\bigvee_{j=1}^{n} Q_j^{\mathcal{A}(\psi)}(\bar{y}_j, \bar{z}_j) \vee \varphi))), \tag{5}$$

with $\bar{x} = \cup_{i=1}^m \bar{x}_i$ and $\bar{z} = \cup_{j=1}^n \bar{z}_j$. $D^{\mathcal{A}(\psi)} \models \psi^N$ refers to the classical first-order satisfaction with *null* treated as any other constant in $\mathcal{U}$.
(b) A NNC ψ of the form (4) is satisfied in the database instance D iff $D \models \psi$ in the classical sense, treating *null* as any other constant. □

[5] If a variable appears at least twice in a IC, then it is involved in a join, or it is in the antecedent and in the consequence of the IC, or it is in a built-in atom. In all these cases, the variable (actually, the corresponding attribute) is relevant.

[6] Remember that $R[i]$ denotes the attribute in position i of relation R.

That is, an IC of the form (1) is satisfied whenever: (a) there exists a *null* value in any of the relevant attributes in the antecedent, or (b) if no null appears in the antecedent, then the second disjunction in the consequent of formula (5) is satisfied, which corresponds to the consequent of the original IC restricted to the relevant attributes. This check can be done as usual, treating nulls as any other constant.

Example 5. (example 4 cont.) To check if $D \models_N \psi$ with $\psi: \forall xyz(P(x,y,z) \rightarrow R(x,y))$, we need to verify if $D^{\mathcal{A}(\psi)} \models \forall xy(P^{\mathcal{A}(\psi)}(x,y) \rightarrow (IsNull(x) \vee IsNull(y) \vee R^{\mathcal{A}(\psi)}(x,y)))$.
(a) For $x = a$ and $y = b$, $D^{\mathcal{A}(\psi)} \models P^{\mathcal{A}(\psi)}(a,b)$, since none of them is a null value, i.e. $IsNull(a)$ and $IsNull(b)$ are both false, we need to check if $D^{\mathcal{A}(\psi)} \models R^{\mathcal{A}(\psi)}(a,b)$, which in this case is true.
(b) For $x = b$ and $y = null$, $D^{\mathcal{A}(\psi)} \models P^{\mathcal{A}(\psi)}(b,null)$, since there is a null in a relevant attribute, i.e. $IsNull(null)$ is true, the constraint is trivially satisfied.

As a consequence, and since there are no tuples that violate the IC, the database instance D is consistent regarding IC.

The database instance $D' = \{P(a,b,null), P(b,b,a), R(a,b)\}$ is inconsistent regarding the UIC $\psi: \forall xyz(P(x,y,z) \rightarrow R(x,y))$. This is because, for $x = b$ and $y = b$, $D^{\mathcal{A}(\psi)} \models P^{\mathcal{A}(\psi)}(b,b)$, but since none of them is a null value, we need to check if $D^{\mathcal{A}(\psi)} \models R^{\mathcal{A}(\psi)}(b,b)$, which in this case is false. □

2.3. Database repairs

When inconsistency arises in a database instance D, it can be restored by deleting and/or inserting database tuples. This applies in particular to violations of RICs. Assume, for example, that the RIC is $\psi: \forall x \forall y(P(x,y) \rightarrow \exists z\, R(x,z))$, and that $D = \{P(a,b)\}$, which is inconsistent with respect to ψ. Consistency can be restored by deleting the database tuple $P(a,b)$ or inserting a database tuple of the form $R(a,d)$, where $d \in \mathcal{U}$. We will prefer to restore consistency by taking $d = null$, whenever possible (e.g. an NNC could prevent us from making this choice).

As originally defined in [4], a repair is a new database instance with the same schema as D that now satisfies the ICs, and minimally differs from the D with respect to set inclusion. The notion of repair we will use in this paper departs from the one in [4], where any (not null) value in the database domain could be chosen to satisfy the RIC. In consequence, now we need new notions of distance between database instances and of repair that take in account the preference for a null value when satisfying a RIC.

Definition 5. [4] Let D, D' be database instances over the same schema and domain. The *distance*, $\Delta(D,D')$, between D and D' is the symmetric difference $\Delta(D,D') = (D \smallsetminus D') \cup (D' \smallsetminus D)$. □

We use this distance to define a partial order between database instances.

Definition 6. [16] Let D, D', D'' be database instances over the same schema and domain $\mathcal{U}$. It holds $D' \leq_D D''$ iff:

1. For every atom $P(\bar{a}) \in \Delta(D,D')$, with $\bar{a} \in (\mathcal{U} \smallsetminus \{null\})^n$ where n is the arity of P[7], it holds that $P(\bar{a}) \in \Delta(D,D'')$.
2. For every atom $Q(\bar{a},\overline{null}) \in \Delta(D,D')$,[8] with $\bar{a} \in (\mathcal{U} \smallsetminus \{null\})^n$, there exists a $\bar{b} \in \mathcal{U}$, such that $Q(\bar{a},\bar{b}) \in \Delta(D,D'')$ and $Q(\bar{a},\bar{b}) \notin \Delta(D,D')$. □

[7] That $\bar{a} \in (\mathcal{U} \smallsetminus \{null\})^n$ means that each of the elements in tuple $\bar{a}$ belongs to $(\mathcal{U} \smallsetminus \{null\})^n$.

[8] $\overline{null}$ is a tuple of null values, that for simplification purposes, are placed in the last attributes of Q, but could be anywhere else in Q.

This partial order is used to define the *repairs* of an inconsistent database.

Definition 7. [16] Given a database instance D and a set IC of ICs of the form (1), and NNCs of the form (4), a *repair* of D with respect to IC is a database instance D', such that: (a) It has the same schema as D, (b) $D' \models_N IC$, and (c) D' is $\leq_D$-minimal in the class of database instances that satisfy (a) and (b). The set of repairs of D with respect to IC is denoted by $Rep(D, IC)$. □

In particular, this definition requires from a repair D' that there is no database D'' with similar properties, such that $D'' <_D D'$, where $D'' <_D D'$ means $D'' \leq_D D'$ but not $D' \leq_D D''$. If *null* is absent from $\mathcal{U}$, this definition of repair coincides with the one in [4].

There are good reasons for adopting the repair semantics in [4] when dealing with RICs. First, using the null value of SQL databases is more natural from the point of view of database practice. Second, DBMSs consider only one null value, and labeled nulls cannot be implemented in SQL-complaint DBMSs. Third, under our repair semantics, even with RIC-cyclic ICs, CQA becomes decidable [16]. However, with the repair semantics in [4], either with arbitrary domain elements or arbitrarily many labeled nulls to restore consistency of RIC-cyclic ICs, CQA is undecidable [19]. Having said that, it is important to emphasize that the notion of IC satisfaction has to be properly defined in the presence of the single SQL null. This was done in Definition 4, which is a logical reconstruction of IC satisfaction of ICs under SQL databases (with *null*). For example, according to that semantics (and the practice with SQL databases), the only repair for the database $D = \{P(a)\}$ with respect to the set ICs $IC = \{\forall x(P(x) \rightarrow \exists y Q(x, y, y)), \forall xy(Q(x, y, y) \rightarrow R(y)\}$ would be $D' = \{P(a), Q(a, null, null)\}$, which does satisfy the ICs under the adopted semantics.

The choice of the SQL-null repair semantics is essentially orthogonal to the methodology for query answering being presented in this paper. Our repair programs could be easily adapted to "theoretical" relational databases (without SQL nulls) where inconsistencies with respect to RICs are solved by using arbitrary database domain or a set of labeled nulls. Of course, in this case the repair program would include an domain predicate whose extension is infinite, potentially producing infinitely many repairs. This would lead to the undecidability results mentioned above.

We will assume that the ICs in IC are of the form (1) or NNCs of the form (4), but they are *non-conflicting*, in the sense that there is no NNC on an attribute of a relation that is existentially quantified in an IC of the form (1).[9] Now we can define the notion of consistent answer to queries.

Definition 8. [4] A tuple $\bar{t}$ of elements of $\mathcal{U}$ is a *consistent answer* to a query $Q(\bar{x})$ from D with respect to IC if $\bar{t}$ is an answer to $Q(\bar{x})$ in every $D' \in Rep(D, IC)$. If the query Q is a sentence (or boolean query), the consistent answer is *yes* if Q is true in every repair D' of D; and *no*, otherwise. □

Example 6. Consider the database schema $\Sigma = \{S(id,name), R(id, name), T(id,dept), W(id, dept, since)\}$, the instance $D = \{S(a, c), S(b, c), R(b, c), T(a, null), W(null, b, c)\}$, and $IC = \{\forall xy(S(x, y) \rightarrow R(x, y)), \forall xy (T(x, y) \rightarrow \exists z W(x, y, z)), \forall xyz(W(x, y, z) \wedge IsNull(x) \rightarrow \textbf{false})\}$. The instance D is inconsistent with respect to IC, because:

(a) $S(a, c)$ is in D but $R(a, c)$ is not, therefore the IC $\forall xy(S(x, y) \rightarrow R(x, y))$ is violated. Consistency can be restored by inserting $R(a, c)$ or deleting $S(a, c)$.

[9] It is not difficult to treat the more general case where this restriction is not imposed, but the presentation would be more complex.

(b) Tuple $W(null, b, c)$ is in D violating IC $\forall xyz(W(x, y, z) \wedge IsNull(x) \rightarrow \mathbf{false})$. In this case, consistency may be restored by deleting tuple $W(null, b, c)$.

There are two database repairs. $D_1 = \{S(a, c), S(b, c), R(b, c), R(a, c), T(a, null)\}$, with difference, $\Delta(D, D_1) = \{R(a, c), W(null, b, c)\}$, in terms of whole tuples, with respect to the original database instance D. $D_2 = \{S(b, c), R(b, c), T(a, null)\}$, with $\Delta(D, D_2) = \{S(a, c), W(null, b, c)\}$. The database instance $D_3 = \{\}$ is consistent with respect to IC, but it is not a repair since it does not satisfy minimality. In fact, $\Delta(D, D_3) = \{S(a, c), S(b, c), R(b, c), T(a, null), W(null, b, c)\}$, and $D_2 <_D D_3$.

Consider the instance $D = \{P(a)\}$ and the set of ICs IC : $\{\forall x(P(x) \rightarrow \exists y Q(x, y, y)), \forall xy(Q(x, y, y) \rightarrow R(y))\}$. Instance D is inconsistent with respect to IC since $P(a)$ is in D but there is no tuple in relation Q with a as a first argument. There is one repair $D_1 = \{P(a), Q(a, null, null)\}$, with $\Delta(D, D_1) = \{Q(a, null, null)\}$. Note that the new tuple $Q(a, null, null)$ does not violate the second IC since there is a null in a relevant attribute of Q (the attribute representing by variable y). The database instance $D_2 = \{P(a), Q(a, d, d)\}$, for any $d \in \mathcal{U}$ different from $null$, is not a repair: Since $\Delta(D, D_2) = \{Q(a, d, d)\}$, we have $D_1 <_D D_2$ and, therefore D_2 is not $\leq_D$-minimal. The consistent answer to query $\mathcal{Q}$: $Q(x, y, z)$ is $\langle a, null, null\rangle$. There are no consistent answers to query $\mathcal{Q}$: $R(x)$. If RICs where restored by adding arbitrary values from the domain, then repairs for this instance would be of the form $\{P(a), Q(a, \beta, \beta), R(\beta)\}$ where β is a value from the database domain. □

3. Repair Programs

We can use disjunctive logic programs (DLPs) with a stable models semantics [40] to specify database repairs. The idea is that, given an inconsistent database instance D and a set IC of RIC-acyclic ICs, a repair program $\Pi(D, IC)$ is constructed, in such a way that there is a one-to-one correspondence between the stable models of $\Pi(D, IC)$ and the repairs of D [9, 16].

Repair programs use annotation constants whose role is to enable the definition of atoms that can become true (inserted) or false (deleted) into/from instances in order to satisfy the ICs. The idea is to use logic programming rules to specify what to do when a database violates an IC. Actually, each atom of the form $P(\bar{a})$ (except for those that refer to the original extensional database) receives an annotation constant from those shown in Table 1.

Table 1: Annotation Constants

Annotation	Atom	The tuple $P(\bar{a})$ is ...
$\mathbf{t_a}$	$P_(\bar{a}, \mathbf{t_a})$	$P(\bar{a})$ is advised to be made true.
$\mathbf{f_a}$	$P_(\bar{a}, \mathbf{f_a})$	$P(\bar{a})$ is advised to be made false
$\mathbf{t^\star}$	$P_(\bar{a}, \mathbf{t^\star})$	$P(\bar{a})$ is true or is made true
$\mathbf{t^{\star\star}}$	$P_(\bar{a}, \mathbf{t^{\star\star}})$	$P(\bar{a})$ is true in *the repair*

Since the original database facts do not receive annotations, we also introduce an expanded version $P_$ of each database predicate P. The former has an extra argument to accommodate the annotation constant.

Annotations are introduced and used as follows: First, for each IC we introduce a disjunctive program rule whose body captures the violation of the IC, and whose head describes how to restore the consistency by deleting or inserting tuples (but not both by the minimality of stable

models of the program). The atoms in the disjunctive head use the $\mathbf{t_a}, \mathbf{f_a}$ annotations, that have the intended meaning of “advising to make true (false) the atom”, i.e. inserting (deleting) it into (from) the database. That is, the atom $P_(\bar{a}, \mathbf{t_a})$ suggest the insertion of $P(\bar{a})$; and $P_(\bar{a}, \mathbf{f_a})$, the deletion of $P(\bar{a})$.

For example, for the inclusion dependency in Example 6

$$\forall xy(S(x,y) \ \rightarrow \ R(x,y)), \tag{6}$$

the disjunctive program rule 7 specifies that if $S(x,y)$ is in the database, but $R(x,y)$ is not, i.e. there is a violation of the constraint that is captured by the rule body, then consistency is restored by either deleting $S(x,y)$, which receives constant $\mathbf{f_a}$ in the head of the rule, or by inserting $R(x,y)$, which receives the $\mathbf{t_a}$ constant.

$$S_(x,y,\mathbf{f_a}) \vee R(x,y,\mathbf{t_a}) \leftarrow S(x,y),\ \mathit{not}\ R(x,y), \tag{7}$$

This rule would work well if consistency restoration were only a single step process. However, there might be several, logically interacting ICs, and the process may take more steps. Insertions and deletions done in between have to be taken into account since they could produce new inconsistencies, until the whole process stabilizes. In order to keep track of both original database tuples and also new ones in the database along the repair process, we use the annotation constant $\mathbf{t}^\star$.

For example, if due to an IC other than (6), $S_(c,a,\mathbf{t_a})$ is generated and $R(c,a)$ is not in the database, or $R(c,a)$ has been made false due to $R(c,a,\mathbf{f_a})$, the constraint (6) is violated once again. However, the rule (7) will not be able to restore consistency, because the violation is not due to original tuples in the database. In consequence, the program rule (7) has to be changed to

$$S_(x,y,\mathbf{f_a}) \vee R(x,y,\mathbf{t_a}) \leftarrow S_(x,y,\mathbf{t}^\star),\ \mathit{not}\ R(x,y), \tag{8}$$

where the atom $S_(x,y,\mathbf{t}^\star)$ has to be specified according to its intended meaning of indicating true atoms along the repair process, i.e. as $S(x,y)$ or $S_(x,y,\mathbf{t_a})$ being true. This has to be specified for each database predicate (cf. item 5. in Definition 9 below).

For similar reasons, the rule (9) has to be added to the program, accompanying rule (8). This rule captures the case when $S_(c,a,\mathbf{t_a})$ has been generated, but $R(c,a)$ has been made false (i.e. $R(c,a,\mathbf{f_a})$ has been generated), which again causes the violation of (6):

$$S_(x,y,\mathbf{f_a}) \vee R(x,y,\mathbf{t_a}) \leftarrow S_(x,y,\mathbf{t}^\star), R(x,y,\mathbf{f_a}). \tag{9}$$

A stable model of the program is *coherent* if it does not contain a pair of atoms of the form $P_(\bar{a},\mathbf{t_a}), P_(\bar{a},\mathbf{f_a})$. That is, a database atom is never both inserted and deleted.

Finally, we have to be able to read off the atoms that belong to a repair. They are those that were eventually inserted into the database or were already in the original instance and were never deleted. The annotation constant $\mathbf{t}^{\star\star}$ allows to collect all these atoms, for each database predicate (cf. item 6. in Definition 9). The following general program, introduced, extended and optimized in a series of papers [9, 15, 16, 21], captures all these ideas.

Definition 9. Given a database instance D, a set IC of UICs, RICs, and NNCs, the repair program $\Pi(D, IC)$ contains:

1. $P(\bar{a})$, for each atom $P(\bar{a}) \in D$.

2. For every UIC ψ of the form (2), the set of rules:
   $\bigvee_{i=1}^{m} P_i(\bar{x}_i, \mathbf{f_a}) \vee \bigvee_{j=1}^{n} Q_j(\bar{y}_j, \mathbf{t_a}) \leftarrow \bigwedge_{i=1}^{m} P_i(\bar{x}_i, \mathbf{t^\star}), \bigwedge_{Q_j \in Q'} Q_j(\bar{y}_j, \mathbf{f_a}),$
   $\bigwedge_{Q_k \in Q''} \mathit{not}\ Q_k(\bar{y}_k), \bigwedge_{x_l \in \mathcal{A}(\psi) \cap \bar{x}} x_l \neq \mathit{null}, \bar{\varphi},$
   for every pair of sets Q' and Q'' of atoms appearing in formula (2) such that $Q' \cup Q'' = \bigcup_{j=1}^{n}\{Q_j\}$ and $Q' \cap Q'' = \emptyset$, where $\mathcal{A}(\psi)$ is the set of relevant attributes for ψ, $\bar{x} = \bigcup_{i=1}^{n} x_i$, and $\bar{\varphi}$ is a conjunction of built-ins that is equivalent to the negation of φ.
3. For every RIC ψ of the form (3), the rules:
   $P_(\bar{x}, \mathbf{f_a}) \vee Q_(\bar{y}, \overline{null}, \mathbf{t_a}) \leftarrow P_(\bar{x}, \mathbf{t^\star}),\ \mathit{not}\ aux_\psi(\bar{y}), \bar{y} \neq \mathit{null}.$
   And for every $z_i \in \bar{z}$: $aux_\psi(\bar{y}) \leftarrow Q_(\bar{y}, \bar{z}, \mathbf{t^\star}),\ \mathit{not}\ Q_(\bar{y}, \bar{z}, \mathbf{f_a}), \bar{y} \neq \mathit{null}, z_i \neq \mathit{null}.$
4. For every NNC of the form (4), the rule: $P_(\bar{x}, \mathbf{f_a}) \leftarrow P_(\bar{x}, \mathbf{t^\star}), x_i = \mathit{null}.$
5. For each predicate $P \in R$, the annotation rules:
   $P_(\bar{x}, \mathbf{t^\star}) \leftarrow P(\bar{x}).\quad P_(\bar{x}, \mathbf{t^\star}) \leftarrow P_(\bar{x}, \mathbf{t_a}).$
6. For every predicate $P \in \mathcal{R}$, the interpretation rule:
   $P_(\bar{x}, \mathbf{t^{\star\star}}) \leftarrow P_(\bar{x}, \mathbf{t^\star}),\ \mathit{not}\ P_(\bar{x}, \mathbf{f_a}).$
7. For every predicate $P \in \mathcal{R}$ that is not a sink or a source node in $\mathcal{G}(IC)$, the program constraint: $\leftarrow P_(\bar{x}, \mathbf{t_a}), P_(\bar{x}, \mathbf{f_a}).$ □

We can see that our repair programs may have rules whose heads are disjunctions of positive literals, and their bodies may contain positive or negative literals with weak negation for negative literals.

Example 7. Consider the database schema $\Sigma = \{S(id,name), R(id,name), T(id,dept), W(id,dept,since)\}$, the instance $D = \{S(a,c), S(b,c), R(b,c), T(a,null), W(null,b,c)\}$, and $IC = \{\forall xy(S(x,y) \rightarrow R(x,y)), \forall xy (T(x,y) \rightarrow \exists z W(x,y,z)), \forall xyz(W(x,y,z) \wedge IsNull(x) \rightarrow \mathbf{false})\}$ presented in Example 6. The repair program $\Pi(D, IC)$ contains the following rules:

1. $S(a,c).\ \ S(b,c).\ \ R(b,c).\ \ T(a,null).\ \ W(null,b,c).$
2. $S_(x,y,\mathbf{f_a}) \vee R(x,y,\mathbf{t_a}) \leftarrow S_(x,y,\mathbf{t^\star}), R(x,y,\mathbf{f_a}), x \neq null, y \neq null.$
   $S_(x,y,\mathbf{f_a}) \vee R(x,y,\mathbf{t_a}) \leftarrow S_(x,y,\mathbf{t^\star}),\ not\ R(x,y), x \neq null, y \neq null.$
3. $T_(x,y,\mathbf{f_a}) \vee W_(x,y,null,\mathbf{t_a}) \leftarrow T_(x,y,\mathbf{t^\star}),\ not\ aux(x,y), x \neq null, y \neq null.$
   $aux(x,y) \leftarrow W_(x,y,z,\mathbf{t^\star}),\ not\ W_(x,y,z,\mathbf{f_a}), x \neq null, y \neq null, z \neq null.$
4. $W_(x,y,z,\mathbf{f_a}) \leftarrow W_(x,y,z,\mathbf{t^\star}), x = null.$
5. $S_(x,y,\mathbf{t^\star}) \leftarrow S(x,y).$
   $S_(x,y,\mathbf{t^\star}) \leftarrow S_(x,y,\mathbf{t_a}).$ } (Similarly for R, T, W)
6. $S_(x,y,\mathbf{t^{\star\star}}) \leftarrow S_(x,y,\mathbf{t^\star}),\ not\ S_(x,y,\mathbf{f_a}).$
7. $\leftarrow W_(x,y,z,\mathbf{t_a}), W_(x,y,z,\mathbf{f_a}).$

The rules in 2. establish how to repair the database with respect to the first IC: by making $S(x,y)$ false or $R(x,y)$ true. These rules are constructed by choosing all the possible sets Q' and Q'' such that $Q' \cup Q'' = \{R(x,y)\}$ and $Q' \cap Q'' = \emptyset$. The first rule in 2 considers $Q' = \{R(x,y)\}$ and $Q'' = \emptyset$. The second corresponds to $Q' = \emptyset$ and $Q'' = \{R(x,y)\}$. Note that conditions of the form $x \neq null$ appear for the variables corresponding to relevant attributes only. The rules in 3. specify the form of restoring consistency with respect to the RIC: by deleting $T(x,y)$ or inserting $W(x,y,null)$. Here, only the variables in the antecedent of the RIC cannot take null values. Rule 4. indicates how to restore consistency with respect to the NNC: by eliminating $W(x,y,z)$.

Finally, the *program constraint* 7. filters out possible *non-coherent* stable models of the program, those that have an W-atom annotated with both $\mathbf{t_a}$ and $\mathbf{f_a}$. For the program in this example, given the logical relationship between the ICs, this phenomenon could happen only for predicate W. This is because in the corresponding dependency graph for the ICs W is not either a sink or a source node.

The program has two stable models:[10]

$$
\begin{aligned}
\mathcal{M}_1 &= \{S_(a,c,\mathbf{t}^\star), S_(b,c,\mathbf{t}^\star), R(b,c,\mathbf{t}^\star), T_(a,null,\mathbf{t}^\star), W_(null,b,c,\mathbf{t}^\star),\ W_(null,b,c,\mathbf{f_a}), R(a,c,\mathbf{t_a}),\\
&\quad \underline{S_(a,c,\mathbf{t}^{\star\star})}, \underline{S_(b,c,\mathbf{t}^{\star\star})}, \underline{R(b,c,\mathbf{t}^{\star\star})}, R(a,c,\mathbf{t}^\star), \underline{R(a,c,\mathbf{t}^{\star\star})}, \underline{T_(a,null,\mathbf{t}^{\star\star})}\},\\
\mathcal{M}_2 &= \{S_(a,c,\mathbf{t}^\star), S_(b,c,\mathbf{t}^\star), R(b,c,\mathbf{t}^\star), T_(a,null,\mathbf{t}^\star), W_(null,b,c,\mathbf{t}^\star), W_(null,b,c,\mathbf{f_a}),\ \ S_(a,c,\mathbf{f_a}),\\
&\quad \underline{S_(b,c,\mathbf{t}^{\star\star})}, \underline{R(b,c,\mathbf{t}^{\star\star})}, \underline{T_(a,null,\mathbf{t}^{\star\star})}\}.
\end{aligned}
$$

Thus, consistency is recovered, according to $\mathcal{M}_1$ by inserting atom $R(a,c)$ and deleting atom $W(null,b,c)$ ($\{R(a,c,\mathbf{t_a}),\ W(null,b,c,\mathbf{f_a})\} \in \mathcal{M}_1$); or, according to $\mathcal{M}_2$ by deleting atoms $\{S(a,c),\ W(null,b,c)\}$ ($\{S(a,c,\mathbf{f_a}),\ W(null,b,c,\mathbf{f_a})\} \in \mathcal{M}_2$). Two repairs can be obtained by concentrating on the underlined atoms in the stable models: $\{S(a,c),\ S(b,c), R(b,c),\ R(a,c),\ T(a,null)\}$ and $\{S(b,c), R(b,c),\ T(a,null)\}$, as expected. $\square$

The repair program of Definition 9 is a sound and complete specification of database repairs with respect to *RIC-acyclic* sets of UICs of the form (2), RICs of the form (3), and NNCs of the form (4) [9, 16][11].

In order to use repair programs to compute consistent answers, queries have to be expressed as logic programs. For example, a first-order query can be translated into a logic program, namely as a program in non-recursive Datalog, possibly with negation [49]. Alternatively, the query can be expressed directly in Datalog.[12] Either way, given a query $\mathcal{Q}(\bar{x})$, a query program $\Pi(\mathcal{Q})$ is generated by first expressing $\mathcal{Q}$ as a Datalog program with a query atom $Ans(\bar{x})$, and next replacing every database atom $P(\bar{s})$ in the program by $P(\bar{s},\mathbf{t}^{\star\star})$.

In consequence, in order to get consistent answers, $\Pi(\mathcal{Q})$ is combined (or run) with the repair program $\Pi(D,IC)$. The consistent answers to query $\mathcal{Q}$ are the ground atoms in the intersection of all the extensions of the *Ans* predicate in the different stable models of the program. We can see that CQA becomes a form of *cautious* or *skeptical* reasoning under the stable models semantics [40].

Example 8. (example 7 cont.) For the Datalog query $\mathcal{Q}$: $Ans(x) \leftarrow S(b,x)$, the program $\Pi(\mathcal{Q})$ is $Ans(x) \leftarrow S(b,x,\mathbf{t}^{\star\star})$. The query predicate *Ans* collects the answers to the query. The combined program $\Pi(D,IC,\mathcal{Q}) = \Pi(D,IC) \cup \Pi(\mathcal{Q})$ has two stable models:

$$
\begin{aligned}
\mathcal{M}_1^A &= \{S_(a,c,\mathbf{t}^\star), S_(b,c,\mathbf{t}^\star), R(b,c,\mathbf{t}^\star), T_(a,null,\mathbf{t}^\star), W_(null,b,c,\mathbf{t}^\star), W_(null,b,c,\mathbf{f_a}), R(a,c,\mathbf{t_a}),\\
&\quad \underline{S_(a,c,\mathbf{t}^{\star\star})}, \underline{S_(b,c,\mathbf{t}^{\star\star})}, \underline{R(b,c,\mathbf{t}^{\star\star})}, R(a,c,\mathbf{t}^\star), \underline{R(a,c,\mathbf{t}^{\star\star})}, \underline{T_(a,null,\mathbf{t}^{\star\star})}, \underline{Ans(c)}\},\\
\mathcal{M}_2^A &= \{S_(a,c,\mathbf{t}^\star), S_(b,c,\mathbf{t}^\star), R(b,c,\mathbf{t}^\star), T_(a,null,\mathbf{t}^\star), W_(null,b,c,\mathbf{t}^\star), W_(null,b,c,\mathbf{f_a}), S_(a,c,\mathbf{f_a}),\\
&\quad \underline{S_(b,c,\mathbf{t}^{\star\star})}, \underline{R(b,c,\mathbf{t}^{\star\star})}, \underline{T_(a,null,\mathbf{t}^{\star\star})}, \underline{Ans(c)}\}.
\end{aligned}
$$

In them, the atoms that are potentially relevant to answer a query program are underlined. The answers to the query are the cautious answers, i.e. the ground *Ans*-atoms in the intersection of all stable models. In this case, the stable models have only atom $Ans(c)$ in common. Thus, the only consistent answer to $\mathcal{Q}$ is $\langle c \rangle$. $\square$

---

[10] Stable models are displayed without showing their auxiliary (*aux*) atoms and facts.

[11] For non-*RIC-acyclic* ICs, the repair program provides a provably complete specification, but may have stable models that do not correspond to repairs [15].

[12] From now on, when we refer to Datalog, we mean Datalog itself or any of its extensions with weak negation.

3.1. Stratification and repair programs

In this section we analyze repair programs with respect to the property of stratification. The results we obtain are important since they allow us to use the results presented in [29] and [35] to prove soundness and completeness of our magic set methodology for CQA. This will be done in the next section. The following definition was given in [34] for disjunctive programs with negation but no program constraints.

Definition 10. [34] A disjunctive program Π without program denials is stratified if it is possible to decompose the predicates in Π into pairwise disjoint sets $S_0, S_1, \ldots, S_k$, such that: if a predicate $P \in S_i$ occurs in the head of a rule r of Π and a predicate P' occurs in the same rule, then:
(a) $P' \in S_i$, if P' occurs in the head of r.
(b) $P' \in S_j$ with $j \leq i$, if P' occurs positively in the body of r.
(c) $P' \in S_j$ with $j < i$, if P' occurs negatively in the body of r. □

Any such decomposition is a *stratification* of Π, and it induces a decomposition of the rules of Π into subsets $s_0, \ldots, s_k$, where s_i contains all the rules that define a predicate $P \in S_i$. In a stratified program we find no cyclic interaction of recursion and negation.

If we strictly apply this definition of stratification to the repair programs of Definition 9, they turn out to be non-stratified due to the way recursion and negation are used. However, a closer look at the program reveals that this combination is due to the use of a same expanded database predicate for accommodating different annotation constants. If we used different predicates for different annotations, this problem would disappear. Actually, we will show now that the repair programs, without considering their program constraints, can be translated into stratified DLPs that have essentially the same semantics.

We can also characterize stratification of a disjunctive program by focalizing on its *predicate dependency graph*.

Definition 11. (a) The *predicate dependency graph* $\mathcal{G}(\Pi) = (V, E)$ of a disjunctive program without program constraints Π is a marked directed graph constructed as follows: For each predicate P there is a node P in V. There is a directed edge $(P, P') \in E$, from P to P', if there is a rule r in Π and P and P' occur both in the head of r; or P occurs in the body of r, and P' in its head. This edged is marked, say with $\star$, if P occurs under negation.
(b) An *odd* cycle in $\mathcal{G}(\Pi)$ is a cycle comprising an odd number of marked arcs. □

The notion of *odd cycle* introduced here will be used in Section 4. As for normal programs [3], we obtain the following simple, alternative characterization of stratified programs that is based in the predicate dependency graph of programs. This characterization of stratified programs will be useful for the formulation of Lemmas 2, and 3 in Section 4.

Proposition 1. A disjunctive program Π without program constraints is *stratified* iff its *predicate dependency graph* $\mathcal{G}(\Pi)$ has no cycle with a marked arc.

Proof. The proof is similar to the one in [3] for normal programs. If a disjunctive program Π without program constraints is stratified, then the definition of each relation symbol P is contained in some stratum of Π. Let $s(P)$ be a function that returns the index of the stratum where predicate P is defined. The following holds: (a) If the edge (P, P') is not marked in the predicate

dependency graph $\mathcal{G}(\Pi)$, then $s(P) \leq s(P')$. (b) If (P, P') is a marked edge, then $s(P) < s(P')$. In consequence, there are no cycles in the dependency graph trough a marked edge.

In the other direction, decompose the predicate dependency graph into strongly connected components of maximum cardinality. In each of them, any two nodes are connected in a cycle. We can establish the relation "there is an edge from component A to component B", which is finite and contains no cycles. Now, assign numbers $0, \ldots, n$, for some $n \geq 0$, to the components, in such a way that, if there is an edge from A to B, then the number assigned to A is smaller than the number assigned to B. Let s_i be the subset of program Π consisting of the definitions of all predicates which belong to component i. We claim that $s_0, \ldots, s_n$ is a stratification of Π. Moreover, $\Pi = s_o \cup \cdots \cup s_n$.

In fact, if a predicate $P_1 \in s_i$ is in the head of a rule r and a predicate P_2 is also in the head of r, then P_2 is in stratum s_i, i.e., both predicates have their definitions in s_i. If P_2 is in the body of r, then P_2 is in stratum s_i or in a stratum s_j with $j < i$. In other words, the definition of P_2 is in some s_j with $j \leq i$. If P_2 occurs under negation, then it lies in a stratum s_j with $j < i$, because by assumption there is no cycle trough a marked edge. Therefore, its definition is contained in some subset s_j with $j < i$. $\square$

In the following, we will identify a "natural" stratification of a repair program. In order to do this, we need to introduce a syntactic modification of the program, one that tells apart occurrences of a same predicate with different annotations.

Definition 12. Let Π be a program with program constraints PC:
(a) $\Pi^{-c} := (\Pi \smallsetminus PC)$.
(b) Π_a is the program obtained from Π by replacing each atom of the form $P_(\bar{x}, d)$, with $d \in \{\mathbf{f_a}, \mathbf{t_a}, \mathbf{t^\star}, \mathbf{t^{\star\star}}\}$, by the atom $P_d(\bar{x})$. $\square$

Example 9. (example 7 cont.) $\Pi_a^{-c}(D, IC)$ has, among others, the rules:
$S_{\mathbf{f_a}}(x,y) \vee R_{\mathbf{t_a}}(x,y) \leftarrow S_{\mathbf{t^\star}}(x,y), R_{\mathbf{f_a}}(x,y), x \neq null, y \neq null.$
$S_{\mathbf{f_a}}(x,y) \vee R_{\mathbf{t_a}}(x,y) \leftarrow S_{\mathbf{t^\star}}(x,y),\ not\ R(x,y), x \neq null, y \neq null.$
Figure 3 shows the *predicate dependency graph* $\mathcal{G}(\Pi_a^{-c}(D, IC))$, restricted to predicates S and R. The complete graph would also contain nodes for predicates T, W, and aux.

The following is a possible stratification for program $\Pi_a^{-c}(D, IC)$: s_1: $\{S, R, T, W\}$, s_2: $\{S_{\mathbf{f_a}}, S_{\mathbf{t_a}}, S_{\mathbf{t^\star}}, R_{\mathbf{f_a}}, R_{\mathbf{t_a}}, R_{\mathbf{t^\star}}, T_{\mathbf{t^\star}}, W_{\mathbf{f_a}}, W_{\mathbf{t^\star}}\}$, s_3: $\{aux\}$, s_4: $\{T_{\mathbf{f_a}}, W_{\mathbf{t_a}}\}$, s_5: $\{S_{\mathbf{t^{\star\star}}}, R_{\mathbf{t^{\star\star}}}, T_{\mathbf{t^{\star\star}}}, W_{\mathbf{t^{\star\star}}}\}$. $\square$

Proposition 2. For a repair program $\Pi(D, IC)$, the modified program $\Pi_a^{-c}(D, IC)$ is stratified. $\square$

This proposition follows from the fact that $\mathcal{G}(\Pi_a^{-c}(D, IC))$ does not contain marked cycles. The proposition tells us that a repair program without its program denials is essentially stratified. From the point of view of the model computation, it behaves as a stratified program. However, if we keep the program denials, $\Pi_a(D, IC)$ can be unstratified. This is because, in order to verify the notion of stratification, a denial of the form $\leftarrow P(\bar{x}, \mathbf{t_a}), P(\bar{x}, \mathbf{f_a})$ has to be rewritten into $p \leftarrow P_{\mathbf{t_a}}(\bar{x}), P_{\mathbf{f_a}}(\bar{x}),\ not\ p$, which destroys stratification.

Corollary 1. For every repair program $\Pi(D, IC)$ that does not have program constraints, the program $\Pi_a(D, IC)$ is stratified. $\square$

For example, and according to this result, if IC consists only of denial constraints, e.g. functional dependencies,[13] then the repair program has no program denials, and turns out to be essentially

[13]Functional dependencies can be represented as denial constraints. For instance, the FD $\forall xyz(P(x,y), P(x,z) \rightarrow (y = z))$ is equivalent to the denial constraint (with built-ins) $\forall xyz(P(x,y), P(x,z), y \neq z \rightarrow false)$.

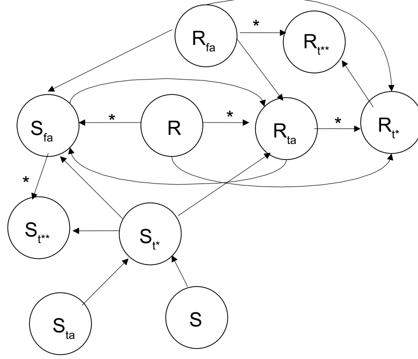


Figure 3: Predicate Dependency Graph $\mathcal{G}(\Pi_a^{-c}(D, IC))$

stratified. For the next section, the following corollary will be useful.

Corollary 2. For every repair program $\Pi(D, IC)$, $\mathcal{G}(\Pi_a^{-c}(D, IC))$ does not have odd cycles. □

The results in this section also hold for sets of ICs that are not *RIC*-acyclic (cf. Definition 2). The following example shows a repair program for a non-*RIC*-acyclic set of ICs, for which $\Pi_a^{-c}(D, IC)$ is stratified.

Example 10. The set $IC = \{\forall x(S(x) \rightarrow Q(x)), \forall x(Q(x) \rightarrow \exists y T(x, y)), \forall xy(T(x, y) \rightarrow Q(y))\}$ is non-*RIC*-acyclic because there is a cycle involving the RIC $\forall x(Q(x) \rightarrow \exists y T(x, y))$. Consider the inconsistent database instance $D = \{S(a), Q(a)\}$, the program $\Pi_a^{-c}(D, IC)$ has the rules:

$S(a).\ \ Q(a).$
$S_{\mathbf{f_a}}(x) \vee Q_{\mathbf{t_a}}(x) \leftarrow S_{\mathbf{t^\star}}(x), Q_{\mathbf{f_a}}(x), x \neq null.$
$S_{\mathbf{f_a}}(x) \vee Q_{\mathbf{t_a}}(x) \leftarrow S_{\mathbf{t^\star}}(x),\ not\ Q(x), x \neq null.$
$Q_{\mathbf{f_a}}(x) \vee T_{\mathbf{t_a}}(x, null) \leftarrow Q_{\mathbf{t^\star}}(x),\ not\ aux(x), x \neq null.$
$aux(x) \leftarrow T_{\mathbf{t^\star}}(x, y),\ not\ T_{\mathbf{f_a}}(x, y), x \neq null, y \neq null.$
$T_{\mathbf{f_a}}(x, y) \vee Q_{\mathbf{t_a}}(y) \leftarrow T_{\mathbf{t^\star}}(x, y), Q_{\mathbf{f_a}}(y), y \neq null.$
$T_{\mathbf{f_a}}(x, y) \vee Q_{\mathbf{t_a}}(y) \leftarrow T_{\mathbf{t^\star}}(x, y),\ not\ Q(y), y \neq null.$

$$\left.\begin{array}{l} S_{\mathbf{t^\star}}(x) \leftarrow S(x). \\ S_{\mathbf{t^\star}}(x) \leftarrow S_{\mathbf{t_a}}(x). \\ S_{\mathbf{t^{\star\star}}}(x) \leftarrow S_{\mathbf{t^\star}}(x),\ not\ S_{\mathbf{f_a}}(x). \end{array}\right\} \text{(Similarly for } T, Q)$$

Program $\Pi_a^{-c}(D, IC)$ does not have recursion via negation, and its *predicate dependency graph*, $\mathcal{G}(\Pi_a^{-c}(D, IC))$, does not have cycles with a marked arc. □

It is worth noticing that the data complexity of cautious query evaluation from disjunctive logic programs with stratified negation is the same as for disjunctive logic programs with unstratified negation and stable model semantics, that is Π_2^P-complete [31, 30, 41].

4. Magic Sets for CQA

Consistent answers are obtained from stable models of the combination of the repair and query programs. Nevertheless, it is usually the case that the repair program, as its stable models,

contains more information than necessary to answer the query. This is because repair programs are built considering all database predicates and database facts. However, query predicates are related to a subset of the database predicates. Moreover, when answering a query, we are not interested in obtaining complete stable models (or repairs), but only in obtaining its (consistent) answers. In consequence, it becomes important to optimize the evaluation of the repair programs by considering only predicates and facts that are relevant to the query.

The magic set (MS) techniques for logic programs with stable model semantics take as an input a logic program -a repair program in our case- and a query expressed as a logic program that has to be evaluated against the repair program. The output is a new logic program, the *magic program*. This program has its own stable models, which can be used to answer the original query more efficiently. Given a query and a program, MS process determines and selects the portion of the program that is relevant to compute the query answers. This is partly achieved by pushing down the constants in the query in order to restrict the tuples involved in the computation.

The stable models of the magic program are those models that contain extensions for the predicates that are relevant to compute the query. Also, they are only partially computed with respect to the models of the original combined program. This means, in essence, each of the stable models of the magic program can be extended to a stable model of the original program [29, 35].

The magic program contains auxiliary rules, the magic rules, that capture the dependency of the query predicates upon certain predicates and rules of the rest of the program, in this case, of the repair program. This avoids unnecessary instantiation of rules and, achieving a faster computation of stable models. In this way, we may obtain less and smaller stable models. The stable models of the magic program provide the same answers to the original query as the models of the program used as input to MS.

Classic MS techniques for Datalog programs [6, 7, 55] have been extended to non-disjunctive logic programs with unstratified negation under stable models semantics [35], and to disjunctive logic programs with stratified negation [42], with an optimized version being implemented in DLV [29]. For this kind of programs, the MS technique is sound and complete, i.e. the method computes all and only correct answers for the query. In [44] a sound but incomplete methodology is presented for disjunctive programs with program constraints of the form $\leftarrow C(\bar{x})$, where $C(\bar{x})$ is a conjunction of literals (i.e. positive or negated atoms). The effect of these program constraints is to discard models of the rest of the program that make true the existential closure of $C(\bar{x})$.

The MS techniques currently implemented in *DLV* cannot be applied to disjunctive programs with program constraints. Furthermore, when the program does not contain program constraints, *DLV* applies MS internally, without giving access to the magic program. As a consequence, the application of MS with *DLV* to repair programs (that contain program constraints) is not straightforward.

Our repair programs are disjunctive, contain non-stratified negation, and have program constraints; the latter with only positive intensional literals in their bodies. In consequence, none of the MS techniques mentioned above can be directly applied to optimize a combination of repair and query programs.

However, in this paper we present a sound and complete MS methodology that can be applied to repair programs (with program constraints). This MS methodology works as follows: First, the program constraints are removed from the repair program. Next, a combination of the MS techniques in [29, 35] is applied to the resulting program. This combination works for repair programs because in them, roughly speaking, negation does not occur in odd cycles. For this kind of programs, soundness and completeness of MS can be obtained from results in [29, 35]

(cf. Theorem 1). Finally, the program constraints are put back into the *magic program* obtained in the previous step, enforcing the magic program to have only coherent models.

4.1. Magic sets: an example

The MS technique sequentially performs three well defined steps on the logic program: *adornment, generation*, and *modification*. They will be illustrated with the repair program in Example 7 together with the query program $Ans(x) \leftarrow S_{_}(b, x, \mathbf{t}^{\star\star})$ of Example 8. As announced in the previous section, the MS technique is applied to the program $\Pi^{-c}(D, IC, Q) := \Pi(D, IC, Q) \smallsetminus PC$, where PC contains the program constraint: $\leftarrow W_{_}(x, y, z, \mathbf{t_a}), W_{_}(x, y, z, \mathbf{f_a})$.

The *adornment* step produces a new, *adorned* program, in which each intensional (defined) predicate P takes the form P^A, where A is a string of letters b, f (for *bound* and *free*, resp.), whose length is equal to the arity of P. Starting from the query, adornments are created and propagated as follows. First $\Pi(Q) : Ans(x) \leftarrow S_{_}(b, x, \mathbf{t}^{\star\star})$ becomes: $Ans^f(x) \leftarrow S_{_}^{bfb}(b, x, \mathbf{t}^{\star\star})$, meaning that the first and third arguments of $S_{_}$ are bound, and the second is a free variable. Constants, in particular annotation constants, are always bound.

The adorned atom $S_{_}^{bfb}(b, x, \mathbf{t}^{\star\star})$ is used to propagate bindings (adornments) onto the rules defining predicate S, i.e. rules in 2., 5., and 6. (in Example 7). As an illustration, the rules in 5. become $S_{_}^{bfb}(x, y, \mathbf{t}^{\star}) \leftarrow S(x, y)$ and $S_{_}^{bfb}(x, y, \mathbf{t}^{\star}) \leftarrow S_{_}^{bfb}(x, y, \mathbf{t_a})$, resp. The extensional (base) atoms, as $S(x, y)$ in the first adorned rule, do not receive any adornments, but they bind variables. As an illustration, suppose we are adorning a program, we generate the adorned atom $P^{fbf}(x, y, z)$, and we have the following rule to be adorned: $P(x, y, z) \leftarrow R(z), M(x, z, y)$, where the only extensional atom is $R(z)$. From atom $P^{fbf}(x, y, z)$ the first and third variables of P, i.e. variables x, z in the head of the rule, are free, while that the second variable y is bound. However, since $R(z)$ is extensional, the variable z becomes bound when adorning the intensional atom $M(x, z, y)$. The adorned rule is: $P^{fbf}(x, y, z) \leftarrow R(z), M^{fbb}(x, z, y)$.

Moreover, returning to our example, the adorned atom $S_{_}^{bfb}(b, x, \mathbf{t}^{\star\star})$ propagates adornments over the disjunctive rules in 2. The adornments are propagated over the literals in the body of the rule (as for non-disjunctive rules), and also to the head literal $R_{_}(x, y, \mathbf{t_a})$. Therefore, this rule becomes:[14] $S_{_}^{bfb}(x, y, \mathbf{f_a}) \vee R_{_}^{bfb}(x, y, \mathbf{t_a}), \leftarrow S_{_}^{bfb}(x, y, \mathbf{t}^{\star}), R_{_}^{bfb}(x, y, \mathbf{f_a})$. Now, the new adorned atoms $R_{_}^{bfb}\ (x, y, \mathbf{t_a}), R_{_}^{bfb}(x, y,\ \mathbf{f_a})$ also have to be processed, producing adornments on rules defining predicate R, and so on. The output of this step is an *adorned program* that contains only adorned rules.

After all the adornments are properly propagated, the adorned program below is generated:

Program 1.

$Ans^f(x) \leftarrow S_{_}^{bfb}(b, x, \mathbf{t}^{\star\star})$.
$S_{_}^{bfb}(x, y, \mathbf{f_a}) \vee R_{_}^{bfb}(x, y, \mathbf{t_a}) \leftarrow S_{_}^{bfb}(x, y, \mathbf{t}^{\star}), R_{_}^{bfb}(x, y, \mathbf{f_a})$.
$S_{_}^{bfb}(x, y, \mathbf{f_a}) \vee R_{_}^{bfb}(x, y, \mathbf{t_a}) \leftarrow S_{_}^{bfb}(x, y, \mathbf{t}^{\star}),\ not\ R(x, y)$.
$S_{_}^{bfb}(x, y, \mathbf{t}^{\star}) \leftarrow S_{_}^{bfb}(x, y, \mathbf{t_a})$. $\quad S_{_}^{bfb}(x, y, \mathbf{t}^{\star}) \leftarrow S(x, y)$.
$S_{_}^{bfb}(x, y, \mathbf{t}^{\star\star}) \leftarrow S_{_}^{bfb}(x, y, \mathbf{t}^{\star}),\ not\ \ S_{_}^{bfb}(x, y, \mathbf{f_a})$.
$R_{_}^{bfb}(x, y, \mathbf{t}^{\star}) \leftarrow R_{_}^{bfb}(x, y, \mathbf{t_a})$. $\quad R_{_}^{bfb}(x, y, \mathbf{t}^{\star}) \leftarrow R(x, y)$.
$R_{_}^{bfb}(x, y, \mathbf{t}^{\star\star}) \leftarrow R_{_}^{bfb}(x, y, \mathbf{t}^{\star}),\ not\ \ R_{_}^{bfb}(x, y, \mathbf{f_a})$. □

The iterative process of passing bindings is called *sideways information passing strategies* (SIPS) [6]. There may be different SIPS strategies, but any SIP strategy has to ensure that all of the body

[14] To simplify the presentation, conditions of the form $x \neq null$ are omitted from the disjunctive rules.

and head atoms are processed. The strategy presented here is the one described in [29], which is implemented in *DLV*. In particular, according to it, negative literals (including extensional) receive adornments by propagation, but do not bind any variable. As an illustration, suppose we are adorning a program, we have the adorned atom $P^{fbf}(x, y, z)$, and the rule:

$$P(x, y, z) \leftarrow S(y, z), T(z, w), not\ R(z, w), M(z, w),$$

where $S(y, z)$ is the only extensional atom. From atom $P^{fbf}(x, y, z)$ variable y is a bound variable. Following our adopted strategy, the adorned rule becomes:

$$P^{fbf}(x, y, z) \leftarrow S(y, z), T^{bf}(z, w), not\ R^{bf}(z, w), M^{bf}(z, w). \tag{10}$$

The extensional atom $S(y, z)$ does not receive adornments, but bounds variable z. Now, variables y, and z are bound. Notice that the literal $not\ R(z, w)$ receives the adornment over variable z, but does not bound variable w, which remains free.

The next step is the *generation of magic rules* from the adorned rules obtained at the previous stage. The former will direct the computation of the stable models of the program consisting of the latter. The generation of magic rules is different for disjunctive and non-disjunctive adorned rules. For each adorned literal (*not*) $P^A(\bar{t})$ in the body of an adorned non-disjunctive rule r, a *magic rule* is generated as follows:

(a) The head of the magic rule is the new atom $magic_P^A(\bar{t}')$, called the magic version of $P^A(\bar{t})$. The tuple of terms $\bar{t}'$ is obtained from $\bar{t}$ by deleting all the variables that are labeled with f according to A.

(b) The atoms in the body of the magic rule are: the atom that is the magic version of the original and adorned head of rule r, and the body atoms (if any) of r that produce bindings on the atom $P^A(\bar{t})$.

For instance, for the adorned atom $S_^{bfb}(x, y, \mathbf{t_a})$ in the body of the adorned rule $S_^{bfb}(x, y, \mathbf{t}^\star) \leftarrow S_^{bfb}(x, y, \mathbf{t_a})$, the magic rule is: $magic_S_^{bfb}(x, \mathbf{t_a}) \leftarrow magic_S_^{bfb}(x, \mathbf{t}^\star)$.

As another example, consider the adorned rule (10). The corresponding magic rule for $T^{bf}(z, w)$ is: $magic_T^{bf}(z) \leftarrow magic_P^{fbf}(y), S(y, z)$. The magic rule contains atom $S(y, z)$ in the body because it bounds variable z. Without this atom the magic rule would be unsafe. The magic rule for the literal $not\ R^{bf}(z, w)$ is: $magic_R^{bf}(z) \leftarrow magic_P^{fbf}(y), S(y, z)$. Finally, the magic rule for atom $M^{bf}(z, w)$ is $magic_M^{bf}(z) \leftarrow magic_P^{fbf}(y), S(y, z)$.

For disjunctive adorned rules, first, intermediate non-disjunctive rules are generated by moving, one at a time, head atoms into the bodies of rules. Next, magic rules are generated as described for non-disjunctive rules. For example, from the rule

$$S_^{bfb}(x, y, \mathbf{f_a}) \vee R_^{bfb}(x, y, \mathbf{t_a}) \leftarrow S_^{bfb}(x, y, \mathbf{t}^\star), R_^{bfb}(x, y, \mathbf{f_a}), \tag{11}$$

we obtain two non-disjunctive rules: (i) $S_^{bfb}(x, y, \mathbf{f_a}) \leftarrow R_^{bfb}(x, y, \mathbf{t_a}), S_^{bfb}(x, y, \mathbf{t}^\star), R_^{bfb}(x, y, \mathbf{f_a})$; and (ii) $R_^{bfb}(x, y, \mathbf{t_a}) \leftarrow S_^{bfb}(x, y, \mathbf{f_a}), S_^{bfb}(x, y, \mathbf{t}^\star), R_^{bfb}(x, y, \mathbf{f_a})$. Three magic rules are generated for rule (i): $magic_R_^{bfb}(x, \mathbf{t_a}) \leftarrow magic_S_^{bfb}(x, \mathbf{f_a})$; $magic_S_^{bfb}(x, \mathbf{t}^\star) \leftarrow magic_S_^{bfb}(x, \mathbf{f_a})$; and $magic_R_^{bfb}(x, \mathbf{f_a}) \leftarrow magic_S_^{bfb}(x, \mathbf{f_a})$.

At this step also the *magic seed atom* is generated, that is the magic version of the *Ans* predicate from the adorned query rule. For example, for the rule: $Ans^f(x) \leftarrow S_^{bfb}(x, y, \mathbf{t}^{\star\star})$, the magic seed atom is $magic_Ans^f$, that becomes a propositional atom. The magic rules for the adorned Program 1 are:

Program 2.

$magic_S_^{bfb}(b, \mathbf{t}^{\star\star}) \leftarrow magic_Ans^f.$
$magic_S_^{bfb}(x, \mathbf{t_a}) \leftarrow magic_S_^{bfb}(x, \mathbf{t}^{\star}).$
$magic_S_^{bfb}(x, \mathbf{t}^{\star}) \leftarrow magic_S_^{bfb}(x, \mathbf{t}^{\star\star}).$
$magic_S_^{bfb}(x, \mathbf{f_a}) \leftarrow magic_S_^{bfb}(x, \mathbf{t}^{\star\star}).$
$magic_R_^{bfb}(x, \mathbf{t_a}) \leftarrow magic_S_^{bfb}(x, \mathbf{f_a}).$
$magic_S_^{bfb}(x, \mathbf{t}^{\star}) \leftarrow magic_S_^{bfb}(x, \mathbf{f_a}).$
$magic_R_^{bfb}(x, \mathbf{f_a}) \leftarrow magic_S_^{bfb}(x, \mathbf{f_a}).$
$magic_S_^{bfb}(x, \mathbf{f_a}) \leftarrow magic_R_^{bfb}(x, \mathbf{t_a}).$
$magic_S_^{bfb}(x, \mathbf{t}^{\star}) \leftarrow magic_R_^{bfb}(x, \mathbf{t_a}).$
$magic_R_^{bfb}(x, \mathbf{f_a}) \leftarrow magic_R_^{bfb}(x, \mathbf{t_a}).$
$magic_R_^{bfb}(x, \mathbf{t_a}) \leftarrow magic_R_^{bfb}(x, \mathbf{t}^{\star}).$
$magic_R_^{bfb}(x, \mathbf{t}^{\star}) \leftarrow magic_R_^{bfb}(x, \mathbf{t}^{\star\star}).$
$magic_R_^{bfb}(x, \mathbf{f_a}) \leftarrow magic_R_^{bfb}(x, \mathbf{t}^{\star\star}).$ □

The last step is the *modification* of the adorned rules: magic atoms constructed at the generation stage are included in the body of adorned rules. That is, for each adorned rule, the magic version of its head is inserted into the body. For instance, the magic versions of the head atoms in (11) are $magic_S_^{bfb}(x, \mathbf{f_a})$ and $magic_R_^{bfb}(x, \mathbf{t_a})$, resp., which are inserted into the body of the adorned rule, generating the modified rule:

$$S_^{bfb}(x, y, \mathbf{f_a}) \vee R_^{bfb}(x, y, \mathbf{t_a}) \leftarrow \quad magic_S_^{bfb}(x, \mathbf{f_a}), magic_R_^{bfb}(x, \mathbf{t_a}), S_^{bfb}(x, y, \mathbf{t}^{\star}), R_^{bfb}(x, y, \mathbf{f_a}). \quad (12)$$

From the modified rules all the adornments, except for those of the magic atoms, are now eliminated. Thus, (12) becomes:

$$S_(x, y, \mathbf{f_a}) \vee R_(x, y, \mathbf{t_a}) \leftarrow magic_S_^{bfb}(x, \mathbf{f_a}), magic_R_^{bfb}(x, \mathbf{t_a}), S_(x, y, \mathbf{t}^{\star}), R_(x, y, \mathbf{f_a}).$$

Therefore, Program 1 is eventually transformed into:

Program 3.
$Ans(x) \leftarrow magic_Ans^f, S_(b, x, \mathbf{t}^{\star\star}).$
$S_(x, y, \mathbf{f_a}) \vee R_(x, y, \mathbf{t_a}) \leftarrow magic_S_^{bfb}(x, \mathbf{f_a}), magic_R_^{bfb}(x, \mathbf{t_a}), S_(x, y, \mathbf{t}^{\star}), R_(x, y, \mathbf{f_a}).$
$S_(x, y, \mathbf{f_a}) \vee R_(x, y, \mathbf{t_a}) \leftarrow magic_S_^{bfb}(x, \mathbf{f_a}), magic_R_^{bfb}(x, \mathbf{t_a}), S_(x, y, \mathbf{t}^{\star}),\ not\ R(x, y).$
$R_(x, y, \mathbf{t_a}) \vee S_(x, y, \mathbf{f_a}) \leftarrow magic_R_^{bfb}(x, \mathbf{t_a}), magic_S_^{bfb}(x, \mathbf{f_a}), S_(x, y, \mathbf{t}^{\star}), R_(x, y, \mathbf{f_a}).$
$R_(x, y, \mathbf{t_a}) \vee S_(x, y, \mathbf{f_a}) \leftarrow magic_R_^{bfb}(x, \mathbf{t_a}), magic_S_^{bfb}(x, \mathbf{f_a}), S_(x, y, \mathbf{t}^{\star}),\ not\ R(x, y).$
$S_(x, y, \mathbf{t}^{\star}) \leftarrow magic_S_^{bfb}(x, \mathbf{t}^{\star}), S_(x, y, \mathbf{t_a}). \qquad S_(x, y, \mathbf{t}^{\star}) \leftarrow magic_S_^{bfb}(x, \mathbf{t}^{\star}), S(x, y).$
$R(x, y, \mathbf{t}^{\star}) \leftarrow magic_R_^{bfb}(x, \mathbf{t}^{\star}), R(x, y, \mathbf{t_a}). \qquad R(x, y, \mathbf{t}^{\star}) \leftarrow magic_R_^{bfb}(x, \mathbf{t}^{\star}), R(x, y).$
$S_(x, y, \mathbf{t}^{\star\star}) \leftarrow magic_S_^{bfb}(x, \mathbf{t}^{\star\star}), S_(x, y, \mathbf{t}^{\star}),\ not\ S_(x, y, \mathbf{f_a}).$
$R(x, y, \mathbf{t}^{\star\star}) \leftarrow magic_R_^{bfb}(x, \mathbf{t}^{\star\star}), R(x, y, \mathbf{t}^{\star}),\ not\ R(x, y, \mathbf{f_a}).$ □

As expected, in the modified rules only magic atoms keep adornments. The final *magic program* corresponding to the program $\Pi^{-c}(D, IC, Q)$ of Examples 7 and 8 is the combination of Program 2 and Program 3 plus the facts of the original program and the magic seed atom.

4.2. Properties of magic methods for CQA

In this section we will assume that the query Q is given directly as a Datalog^{not} program, so that we do not have to mention the associated program $\Pi(Q)$. This program has a query predicate Ans_Q.

Definition 13. (a) For a program Π without program constraints, $\mathcal{MS}(\Pi)$ denotes its magic version (as constructed above). (b) For a program Π possibly containing a set PC of program constraints, its magic version is $\mathcal{MS}^{\leftarrow}(\Pi) = \mathcal{MS}(\Pi^{-c}) \cup PC$. □

We will apply this definition to programs of the form $\Pi(D, IC, Q)$.[15] In this case, $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, Q)) = \mathcal{MS}(\Pi^{-c}(D, IC, Q)) \cup PC$. It consists of the magic rules, the modified rules, the magic

[15] Actually, in this section we show that this magic sets methodology is correct for this kind of programs. However, for other programs it may not provide the expected results (cf. Example 14).

seed atom, and set PC of program constraints. Since in the rewritten program only the magic atoms have adornments, the program constraints can be added without any adornments.

Example 11. (examples 7 and 8 cont.) The magic program $\mathcal{MS}^{\leftarrow}(\ \Pi(D, IC, Q))$ contains the magic rules in Program 2, the modified rules in Program 3, the magic seed atom $magic_Ans^f$, and the program constraint $\leftarrow W_(x,y,z,\mathbf{t_a}), W_(x,y,z,\mathbf{f_a})$ (i.e. rule 7. in Example 7).

In this case, $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, Q))$ contains rules related to predicates S, R, but no rules for predicates T, W, which are not relevant to the query. Therefore the program constraint is trivially satisfied. $\mathcal{MS}^{\leftarrow}(\Pi(\ D, IC, Q))$ has only one stable model (displayed here without the magic atoms): $\mathcal{M} = \{S_(b,c,\mathbf{t}^{\star}), S_(b,c,\mathbf{t}^{\star\star}), Ans(c)\}$, which indicates through its Ans predicate that $\langle c \rangle$ is the consistent answer to the original query, as expected.

We can see that the magic program has only those models that are relevant to compute the query answers. Furthermore, these are partially computed, i.e. they can be extended to stable models of the program $\Pi(D, IC, Q)$. More precisely, except for the magic atoms, model $\mathcal{M}$ is contained in every stable model of the original repair program in Example 7. □

The MS programs of Definition 13 are both sound and complete when applied to (obtained from) programs of the form $\Pi(D, IC, Q)$. That is, for any database D, the latter has the exactly the same cautious answers as its magic version $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, Q))$. Now we make this correspondence precise.

Definition 14. Let Π_1 and Π_2 be any two programs without facts that share a predicate P. Π_1 and Π_2 are P-equivalent, denoted $\Pi_1 \equiv_P \Pi_2$ iff, for any set F of facts, $\bigcap_{M \in SM(\Pi_1 \cup F)} P^M = \bigcap_{M' \in SM(\Pi_2 \cup F)} P^{M'}$.[16] □

Theorem 1. Let D be a database instance, IC a set of UICs, RICs, and NNCs of the forms (2), (3) and (4), respectively, and Q a query. It holds: $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, Q) \smallsetminus D) \equiv_{Ans_Q} (\Pi(D, IC, Q) \smallsetminus D)$. □

The theorem says that for any database contents, the repair plus query programs return the same cautions answers (and then consistent answers) to the query as the magic version.[17]

Now we give a general idea of the proof of Theorem 1. It has two parts: soundness and completeness of MS. The former requires that each query answer returned by the magic program is also a consistent answer (as returned by the original program $\Pi(D, IC, Q)$). The latter, that every consistent answer can be recovered using the magic program. For both directions, we first associate to $\mathcal{MS}^{\leftarrow}(\Pi(D, IC, Q))$ a new program $rel(Q,\ \Pi^{-c}(D, IC, Q))$, whose rules are ground versions of those of $\Pi^{-c}(D, IC, Q)$ that are used to compute query Q (cf. Definition 15 below). In other words, program $rel(Q, \Pi^{-c}(D, IC, Q))$ is a subprogram of $ground\ (\Pi^{-c}(D,\ IC, Q))$ (the ground version of $\Pi^{-c}(D, IC, Q)$) that is sufficient to compute the answers to Q. It is this program that we will compare with the magic program.

Program $rel(Q, \Pi^{-c}(D, IC, Q))$ has to important features: (a) It contains all the rules that are needed to check the program constraints that are relevant to the query (see Definition 16). We establish this in Lemma 1. (b) Programs $rel(Q, \Pi^{-c}(D, IC, Q))$ plus program constraints, and $\Pi(D, IC, Q)$ are query equivalent. We prove this in Proposition 3.

---

[16] $SM(\Pi)$ denotes the set of stable models of program Π and P^M is the extension of predicate P in model M.

[17] A similar result can be obtained for the *brave answers*, i.e. those that are true in some stable model (or repair). However, for CQA the cautions semantics should be used.

After that, we establish completeness by proving that for each stable model $\mathcal{M}$ of program $\mathcal{MS}^{\leftarrow}(\Pi(D,IC,Q))$, there exists a stable model $\mathcal{M}'$ of $rel(Q,\, \Pi^{-c}(D,IC,\, Q))$ that satisfies the program constraints that are relevant to answer the query (cf. Definition 16), such that $\mathcal{M} = \mathcal{M}'$. Of course, this last equality without considering the magic predicates, that appear only in program $\mathcal{MS}^{\leftarrow}(\Pi(D,IC,Q))$ (and in its stable models). This is done in Proposition 4.

To prove soundness, we prove that for each stable model $\mathcal{M}$ of program $rel(Q,\Pi^{-c}(D,IC,Q))$ that satisfies the program constraints, there exists a stable model $\mathcal{M}'$ of $\mathcal{MS}^{\leftarrow}(\Pi(D,IC,Q))$, such that $\mathcal{M} = \mathcal{M}'$. This in done in Proposition 5.

We need some notation and definitions first. For a ground rule r of a Datalog program Π, we denote with $H(r)$ and $B(r)$ the head and body of r (without built-ins) resp., both as sets of database literals. As an illustration, for the ground rule $Ans(a) \leftarrow S_(b,a,\mathbf{t^{\star\star}}),\ not\ R(b,a,\mathbf{t^{\star\star}})$., $H(r)$ is $\{Ans(a)\}$, and $B(r)$ is $\{S_(b,a,\mathbf{t^{\star\star}}),\ not\ \ R_(b,a,\mathbf{t^{\star\star}})\}$.

Given two Datalog programs Π_1 and Π_2, it is possible to generate a program $rel(\Pi_1,\Pi_2)$ by collecting the rules from $ground(\Pi_2)$, the ground version of Π_2, that are useful to compute Π_1 (if any). This is captured in Definition 15, which is based on [29]. Typically, Π_1 will be a query program, and Π_2, the base program on top of which the query is computed. So, the literals in the bodies of the rules of Π_1 are defined in Π_2.

Definition 15. Let Π_1 be a Datalog program without disjunctive heads, possibly with negation, but without program constraints; and Π_2, a disjunctive Datalog program possibly with negation and program constraints.
(a) For $S \subseteq ground(\Pi_2)$, $R(S) := \{r \in ground(\Pi_2) \mid$ there are $r' \in S$ and a predicate P, such that $(not)P(t) \in B(r')\ \cup\ H(r')$ and $P(t) \in H(r)\}$.
(b) $rel(\Pi_1,\Pi_2)$ is the least fixed point of the following sequence $rel_0(\Pi_1,\Pi_2) = \{r \in ground(\Pi_2) \mid$ there is a rule r' in $ground(\Pi_1)$, and a predicate P, such that $(not)P(t) \in B(r')$, and $P(t) \in H(r)\}$, and $rel_{i+1}(\Pi_1,\Pi_2) = R(rel_i(\Pi_1,\Pi_2))$, for $i \geq 0$. □

Example 12. Consider the instance $D = \{S(a,b), S(b,a), R(b,a), T(a,a), W(b,b,b)\}$, $IC = \{\forall xy(S(x,y) \rightarrow R(x,y)), \forall xy(T(x,y) \rightarrow \exists z W(x,y,z)), \forall xyz(W(x,y,z) \wedge IsNull(x) \rightarrow \mathbf{false})\}$, and program $\Pi_1 : Ans(x) \leftarrow T_(a,x,\mathbf{t^{\star\star}})$. Here, $ground(\Pi_1) = \{Ans(a) \leftarrow T_(a,a,\mathbf{t^{\star\star}}), Ans(b) \leftarrow T_(a,b,\mathbf{t^{\star\star}})\}$. Consider Π_2 the ground version of program $\Pi(D,IC)$, which contains the facts: $S(a,b).\ \ S(b,a).\ \ R(b,a).\ \ T(a,a).$ $W(b,b,b)$. It also contains the following rules, among other ground rules for constant b, and combinations of constants a, b:
$S_(a,a,\mathbf{f_a}) \vee R(a,a,\mathbf{t_a}) \leftarrow S_(a,a,\mathbf{t^{\star}}), R(a,a,\mathbf{f_a}), a \neq null.$
$S_(a,a,\mathbf{f_a}) \vee R(a,a,\mathbf{t_a}) \leftarrow S_(a,a,\mathbf{t^{\star}}),\ not\ R(a,a), a \neq null.$
$T_(a,a,\mathbf{f_a}) \vee W_(a,a,null,\mathbf{t_a}) \leftarrow T_(a,a,\mathbf{t^{\star}}),\ not\ aux(a,a), a \neq null.$
$aux(a,a) \leftarrow W_(a,a,a,\mathbf{t^{\star}}),\ not\ W_(a,a,a,\mathbf{f_a}), a \neq null.$
$W_(a,a,a,\mathbf{f_a}) \leftarrow W_(a,a,a,\mathbf{t^{\star}}), a = null.$
$\leftarrow W_(a,a,a,\mathbf{t_a}), W_(a,a,a,\mathbf{f_a}).$
$S_(a,a,\mathbf{t^{\star}}) \leftarrow S(a,a).\quad S_(a,a,\mathbf{t^{\star}}) \leftarrow S_(a,a,\mathbf{t_a}).$
$S_(a,a,\mathbf{t^{\star\star}}) \leftarrow S_(a,a,\mathbf{t^{\star}}),\ not\ S_(a,a,\mathbf{f_a}).$ (Similarly for R, T, W)

The disjunctive program Π_2 has negation and program constraints. $\Pi_1 : Ans(x) \leftarrow T_(a,x,\mathbf{t^{\star\star}})$ contains a unique rule with a body literal that is defined in Π_2.

Program $rel_0(\Pi_1,\Pi_2)$ is:
$T_(a,a,\mathbf{t^{\star\star}}) \leftarrow T_(a,a,\mathbf{t^{\star}}),\ not\ T_(a,a,\mathbf{f_a}).$
$T_(a,b,\mathbf{t^{\star\star}}) \leftarrow T_(a,b,\mathbf{t^{\star}}),\ not\ T_(a,b,\mathbf{f_a}).$

$rel_1(\Pi_1,\Pi_2)$ is $rel_0(\Pi_1,\Pi_2)$ plus:

$T_(a,a,\mathbf{t^\star}) \leftarrow T(a,a).$ $\quad T_(a,a,\mathbf{t^\star}) \leftarrow T_(a,a,\mathbf{t_a}).$

$T_(a,b,\mathbf{t^\star}) \leftarrow T(a,b).$ $\quad T_(a,b,\mathbf{t^\star}) \leftarrow T_(a,b,\mathbf{t_a}).$

$T_(a,a,\mathbf{f_a}) \vee W_(a,a,null,\mathbf{t_a}) \leftarrow T_(a,a,\mathbf{t^\star}),\ not\ aux(a,a), a \neq null.$

$T_(a,b,\mathbf{f_a}) \vee W_(a,b,null,\mathbf{t_a}) \leftarrow T_(a,b,\mathbf{t^\star}),\ not\ aux(a,b), a \neq null, b \neq null.$ $rel_2(\Pi_1,\Pi_2)$ is $rel_1(\Pi_1,\Pi_2)$ plus:

$T(a,a).$ $\quad aux(a,a) \leftarrow W_(a,a,a,\mathbf{t^\star}),\ not\ W_(a,a,a,\mathbf{f_a}), a \neq null.$

$aux(a,a) \leftarrow W_(a,a,b,\mathbf{t^\star}),\ not\ W_(a,a,b,\mathbf{f_a}), a \neq null, b \neq null.$

$aux(a,b) \leftarrow W_(a,b,a,\mathbf{t^\star}),\ not\ W_(a,b,a,\mathbf{f_a}), a \neq null, b \neq null.$

$aux(a,b) \leftarrow W_(a,b,b,\mathbf{t^\star}),\ not\ W_(a,b,b,\mathbf{f_a}), a \neq null, b \neq null.$

$rel_3(\Pi_1,\Pi_2)$ is $rel_2(\Pi_1,\Pi_2)$ plus:

$W_(a,a,a,\mathbf{t^\star}) \leftarrow W(a,a,a).$ $\quad W_(a,a,a,\mathbf{t^\star}) \leftarrow W_(a,a,a,\mathbf{t_a}).$

$W_(a,a,b,\mathbf{t^\star}) \leftarrow W(a,a,b).$ $\quad W_(a,a,b,\mathbf{t^\star}) \leftarrow W_(a,a,b,\mathbf{t_a}).$

$W_(a,b,a,\mathbf{t^\star}) \leftarrow W(a,b,a).$ $\quad W_(a,b,a,\mathbf{t^\star}) \leftarrow W_(a,b,a,\mathbf{t_a}).$

$W_(a,b,b,\mathbf{t^\star}) \leftarrow W(a,b,a).$ $\quad W_(a,b,b,\mathbf{t^\star}) \leftarrow W_(a,b,b,\mathbf{t_a}).$

$W_(a,a,a,\mathbf{f_a}) \leftarrow W_(a,a,a,\mathbf{t^\star}), a = null.$

$W_(a,a,b,\mathbf{f_a}) \leftarrow W_(a,a,b,\mathbf{t^\star}), a = null, b = null.$

$W_(a,b,a,\mathbf{f_a}) \leftarrow W_(a,b,a,\mathbf{t^\star}), a = null, b = null.$

$W_(a,b,b,\mathbf{f_a}) \leftarrow W_(a,b,b,\mathbf{t^\star}), a = null, b = null.$

$rel_4(\Pi_1,\Pi_2) = rel_3(\Pi_1,\Pi_2)$. Thus, program $rel(\Pi_1,\Pi_2)$ contains ground rules defining predicates T and W only. □

For our repair programs, the existence of a finite fix point for the sequence $rel_i(Q,\Pi)$ is guaranteed by the finiteness of the database domain and the number of rules, and the absence of functions symbols in the programs. Notice that, by definition, $rel(\Pi_1,\Pi_2)$ does not contain the rules of Π_1. It does not contain program constraints either. If we wanted to run Π_1, we would combine it with $rel(\Pi_1,\Pi_2)$. The idea is that this would give the same extensions for the predicates defined in Π_1 as running it on top of Π_2.

Now we need to prove that: (a) $rel_i(Q,\Pi)$ contains all the rules that are needed to check the program constraints of $\Pi(D,IC)$ that are relevant to Q (cf. Lemma 1). (b) $rel_i(Q,\Pi)$ with the relevant program constraints is query equivalent to $\Pi(D,IC,Q)$ (cf. Proposition 3). For this purpose, we introduce, for the rest of this section, the following **simplified notation**: (a) Π stands for program $\Pi(D,IC,Q)$, (b) Π^{-c} for program $\Pi^{-c}(D,IC,Q)$, (c) $\mathcal{MS}(\Pi^{-c})$ for the magic version of Π^{-c}, and (d) $\mathcal{MS}^{\leftarrow}(\Pi)$ for $\mathcal{MS}(\Pi^{-c}) \cup PC$. The following concepts are used in Lemma 1.

Definition 16. For the set PC of program constraints of Π: (a) PC_Q denotes the set of program constraints in PC of the form $\leftarrow P_(\bar{x},\mathbf{t_a}), P_(\bar{x},\mathbf{f_a})$, for each predicate P that is connected to a predicate in Q in the dependency graph $\mathcal{G}(IC)$. (b) $PC_Q^\star$ denotes the program consisting of the rules of the form $incoh_P(\bar{x}) \leftarrow P_(\bar{x},\mathbf{t_a}), P_(\bar{x},\mathbf{f_a})$, such that the program constraint $\leftarrow P_(\bar{x},\mathbf{t_a})$, $P_(\bar{x},\mathbf{f_a})$ belongs to PC_Q. □

PC_Q contains the program constraints on the predicates that are relevant to query Q. Program $PC_Q^\star$ contains the program constraints in PC_Q rewritten as rules with an auxiliary head predicate, $incoh_P$. We proceed as follows: First, we compute program $rel(PC_Q^\star,\Pi^{-c})$ that contains rules from $ground(\Pi^{-c})$ that are relevant to compute $PC_Q^\star$, i.e. rules that allow us to check the relevant denials. Second, we show that the rules in $rel(PC_Q^\star,\Pi^{-c})$ are all in $rel(Q,\Pi^{-c})$. All this is first illustrated in the next example.

Example 13. (example 12 cont.) (a) For query $Ans(x) \leftarrow W_(x, y, z, \mathbf{t}^{\star\star})$, PC_Q contains only the program constraint: $\leftarrow W_(x, y, z, \mathbf{t_a}), W_(x, y, z, \mathbf{f_a})$, and $PC^{\star}_Q$ contains the rule $incoh_W(x, y, z) \leftarrow W_(x, y, z, \mathbf{t_a}), W_(x, y, z, \mathbf{f_a})$. Program $rel(PC^{\star}_Q, \Pi^{-c})$ contains the facts: $T(a, a)$, $W(b, b, b)$, and the following rules, among other ground rules for constant b, and combinations of constants a, b:

$T_(a, a, \mathbf{f_a}) \vee W_(a, a, null, \mathbf{t_a}) \leftarrow T_(a, a, \mathbf{t}^{\star}),\ not\ aux(a, a), a \neq null.$

$aux(a, a) \leftarrow W_(a, a, a, \mathbf{t}^{\star}),\ not\ W_(a, a, a, \mathbf{f_a}), a \neq null.$

$W_(a, a, a, \mathbf{f_a}) \leftarrow W_(a, a, a, \mathbf{t}^{\star}), a = null.$

$T_(a, a, \mathbf{t}^{\star}) \leftarrow T(a, a). \quad T_(a, a, \mathbf{t}^{\star}) \leftarrow T_(a, a, \mathbf{t_a}).$

$W_(a, a, a, \mathbf{t}^{\star}) \leftarrow W(a, a, a). \quad W_(a, a, a, \mathbf{t}^{\star}) \leftarrow W_(a, a, a, \mathbf{t_a}).$

(b) For query $Ans(x) \leftarrow S_(b, x, \mathbf{t}^{\star\star})$, the repair program does not contain relevant program constraints, and therefore, PC_Q and $PC^{\star}_Q$ are empty. □

Lemma 1. For program Π and query Q, the programs $rel(PC^{\star}_Q \cup Q, \Pi^{-c})$ and $rel(Q, \Pi^{-c})$ coincide as sets of rules (including the facts).

Proof. We can see that $rel(PC^{\star}_Q \cup Q, \Pi^{-c}) = rel(PC^{\star}_Q, \Pi^{-c}) \cup rel(Q, \Pi^{-c})$. Hence, it is sufficient to prove that $rel(PC^{\star}_Q, \Pi^{-c}) \subseteq rel(Q, \Pi^{-c})$.

First, if there are no relevant program constraints to answer the query, then program $PC^{\star}_Q$ has no rules, and $rel(PC^{\star}_Q, \Pi^{-c})$ is empty, and we have that $rel(PC^{\star}_Q \cup Q, \Pi^{-c})$ has the same rules as program $rel(Q, \Pi^{-c})$. Hence, we focalize on the case where there are program constraints which are relevant to answer the query.

PC_Q is not empty and program $PC^{\star}_Q$ has at least one rule of the form $incoh_P(\bar{x}) \leftarrow P_(\bar{x}, \mathbf{t_a}), P_(\bar{x}, \mathbf{f_a})$. It is easy to see that $rel_0(PC^{\star}_Q, \Pi^{-c})$ is composed by the rules of program Π^{-c} whose heads contain atoms of the form $P_(\bar{c}, \mathbf{t_a})$ or $P_(\bar{c}, \mathbf{f_a})$.

There are two cases to analyze: first predicate P is a query predicate and, second P is connected to a query predicate in the graph $\mathcal{G}(IC)$.

First, P is a query predicate. Therefore, $rel_0(Q, \Pi^{-c})$ is composed by interpretation rules of the form $P_(\bar{c}, \mathbf{t}^{\star\star}) \leftarrow P_(\bar{c}, \mathbf{t}^{\star}),\ not\ P_(\bar{c}, \mathbf{f_a})$. Then, $rel_1(Q, \Pi^{-c})$ is $rel_0(Q, \Pi^{-c})$ plus rules of Π^{-c} with head atoms of the form $P_(\bar{c}, \mathbf{t}^{\star})$ or $P_(\bar{c}, \mathbf{f_a})$. Then, $rel_2(Q, \Pi^{-c})$ is $rel_1(Q, \Pi^{-c})$ plus rules of Π^{-c} with head atoms of the form $P_(\bar{c}, \mathbf{t_a})$ or database facts of the form $P(\bar{c})$. Thus, we have that $rel_0(PC^{\star}_Q, \Pi^{-c}) \subseteq rel_2(Q, \Pi^{-c})$, and as a consequence, $rel(PC^{\star}_Q, \Pi^{-c}) \subseteq rel(Q, \Pi^{-c})$ holds.

Second, P is directly or indirectly connected to a query predicate. Therefore, there exists an i such that $rel_i(Q, \Pi^{-c})$ has rules with head atoms of the form $P_(\bar{c}, \mathbf{t_a})$ or $P_(\bar{c}, \mathbf{f_a})$, otherwise P is not connected to a query predicate. Thus, $rel_0(PC^{\star}_Q, \Pi^{-c}) \subseteq rel_i(Q, \Pi^{-c})$, and therefore $rel(PC^{\star}_Q, \Pi^{-c}) \subseteq rel(Q, \Pi^{-c})$ holds. □

Proposition 3. $rel(Q, \Pi^{-c}) \cup ground(PC_Q) \cup Q \equiv_{Ans_Q} \Pi(D, IC, Q)$.

Proof. First of all, each of the two programs involved can be split into the query program Q and the "bottom" subprogram related to the repair program [48]. In one case, $rel(Q, \Pi^{-c}) \cup ground(PC_Q)$, and in the other, $\Pi(D, IC)$. In consequence, the extensions for the predicates defined in Q, in particular Ans_Q, can be computed on top of the extensions of the predicates defined in the bottom subprograms, in particular, those with annotations $\mathbf{t}^{\star\star}$, which are those used by Q as "extensional" predicates.

First notice that, by Lemma 1, after imposing the program denials PC to both $rel(Q, \Pi^{-c}) \cup ground(PC_Q)$ and $\Pi(D, IC)$, no incoherent model will be kept, which, otherwise, could influence the answers to Q.

In consequence, it suffices to prove that $rel(Q, \Pi^{-c})$ and $\Pi^{-c}(D, IC)$ have the same extensions for the predicates that are relevant to Q for every database instance D. Program $\Pi^{-c}(D, IC)$ is stratified (cf. Proposition 2) and induces a stratification on its subprogram $rel(Q, \Pi^{-c})$ (and also a hierarchical splitting on both programs). This stratification can be used as the basis for an inductive proof. This is because the stable models can be computed bottom-up from the extensional database D, as the perfect models of the programs [52, 53, 31]. At level 0 of the stratification the two programs coincide since they use the same arbitrary instance D, as required by the notion of query equivalence. Now, assume that predicate P is relevant to Q, belongs to stratum k, and is defined in terms of predicates $R_1, \ldots, R_m$. Actually, in their predicated-annotation version (cf. Definition 12), the programs are non-recursive. So, we may assume that $R_1, \ldots R_m$ belong all to a lower stratum, say the $(k-1)$th, at which the two programs coincide for these predicates. Since $rel(Q, \Pi^{-c})$ contains all the rules to compute P from $R_1, \ldots R_m$, the extensions of P for both programs will coincide. □

Now, in Propositions 4 and 5 below we establish the relationship between the programs $\mathcal{MS}^{\leftarrow}(\Pi)$ and $rel(Q, \Pi^{-c})$. To prove these propositions we use Lemmas 2 and 3 below, and some auxiliary concepts that we now will define.

Definition 17. [29] For an interpretation M for a program Π without program constraints,[18] a predicate symbol P, and a set of interpretations I:
(a) $M[P]$ denotes the set of atoms in M whose predicate symbol is P.
(b) $\Pi[P]$ is the set of rules of Π whose head contains predicate P.
(c) $M[\Pi]$ is the set of atoms in M whose predicate symbol appears in the head of some rule in program Π.
(d) $I[P] = \{M[P] \mid M \in I\}$, and $I[\Pi] = \{M[\Pi] \mid M \in I\}$. □

The following lemmas can be obtained by a combination of results in [29] and [35]. These results hold for disjunctive programs, possibly with unstratified negation but without program constraints:[19]

Lemma 2. Let Π be a disjunctive, possibly unstratified, Datalog^{not} program without odd cycles (cf. Definition 11). For every stable model M' of $\mathcal{MS}(\Pi, Q)$, there exists a stable model M of $rel(Q, \Pi)$, such that $M = M'[rel(Q, \Pi)]$. □

Lemma 3. Let Π be a disjunctive, possibly unstratified, Datalog^{not} program without odd cycles. For every stable model M of $rel(Q, \Pi)$, there exists a stable model M' of $\mathcal{MS}(\Pi, Q)$, such that $M = M'[rel(Q, \Pi)]$. □

As mentioned earlier, the equalities between models in these lemmas refer to non-magic atoms only. We will use Lemmas 2 and 3 to prove that there exists a correspondence between the stable models of the rewritten program $\mathcal{MS}^{\leftarrow}(\Pi)$ and the stable models of $rel(Q, \Pi^{-c})$ (cf. Propositions 4 and 5). This is possible, because negation in repair programs (without their program constraints) does not occur in odd cycles (cf. Corollary 2).

Now, we need to prove that the correspondence between models still holds after reintroducing the program constraints, producing program $\mathcal{MS}^{\leftarrow}(\Pi)$ ($\mathcal{MS}(\Pi^{-c}) \cup PC$). We rely on the following:

[18] That is a set of ground atoms in the program's signature (or an extension of it) that can be used to interpret the program. It may not be a model of the program.
[19] Personal communication from the authors.

(a) By Lemmas 2 and 3, there is a correspondence between the stable models of $\mathcal{MS}(\Pi^{-c})$ and the stable models of $rel(Q, \Pi^{-c})$. (b) By Lemma 1, $\mathcal{MS}(\Pi^{-c})$ contains all the rules needed to check the relevant program constraints in PC_Q.

Proposition 4. For every stable model M' of $\mathcal{MS}^{\leftarrow}(\Pi)$, there exists a stable model M of $rel(Q, \Pi^{-c})$ that satisfies the program constraints in PC_Q, and $M = M'[rel(Q, \Pi^{-c})]$.

Proof. By contradiction, assume that there exists a stable model M' of $\mathcal{MS}^{\leftarrow}(\Pi)$ for which there is no stable model M of $rel(Q, \Pi^{-c})$ that simultaneously satisfies the program constraints in PC_Q and $M = M'[rel(Q, \Pi^{-c})]$.

Since $M' \in SM(\mathcal{MS}^{\leftarrow}(\Pi))$, and $\mathcal{MS}^{\leftarrow} = \mathcal{MS}(\Pi^{-c}) \cup PC$, it holds $M' \in SM(\mathcal{MS}(\Pi^{-c}))$. Now, by Lemma 2, there is a model M" of $rel(Q, \Pi^{-c})$ such that $M" = M'[rel(Q, \Pi^{-c})]$. Since there are no stable models of $rel(Q, \Pi^{-c})$ that satisfy the program constraints in PC_Q, M" does not satisfy PC_Q either. We have two cases:
(a) M" does not satisfy a program constraint pc in $PC \setminus PC_Q$. Since M" is a model of $rel(Q, \Pi^{-c})$, by Lemma 1, M" is a model of $rel(PC_Q^{\star} \cup Q, \Pi^{-c})$. Since pc is in $PC \setminus PC_Q$, there is no rule in $rel(PC_Q^{\star} \cup Q, \Pi^{-c})$ defining atoms that are relevant to pc. Thus, M" has no atoms relevant to pc and it cannot be violated. We have a contradiction.
(b) M" does not satisfy a program constraint pc in PC_Q, e.g. $pc:\ \leftarrow S_(x, \mathbf{t_a}), S_(x, \mathbf{f_a})$. Since atoms of the form $S_(\bar{a}, \mathbf{t_a}), S_(\bar{a}, \mathbf{f_a})$ are in M", and $M" = M'[rel\,(Q, \Pi^{-c})]$, it holds that $S_(\bar{a}, \mathbf{t_a})$, $S_(\bar{a}, \mathbf{f_a})$ are in M'. But M' satisfies PC_Q. We have a contradiction. □

Proposition 5. For every stable model M of $rel(Q, \Pi^{-c})$ that satisfies the program constraints in PC_Q, there exists a stable model M' of $\mathcal{MS}^{\leftarrow}(\Pi)$, such that $M = M'[rel(Q, \Pi^{-c})]$.

Proof. By Lemma 3, there exists a stable model M" of $\mathcal{MS}(\Pi^{-c})$, such that $M = M"[rel(Q, \Pi^{-c})]$. Since M satisfies the program constraints in PC_Q, M" also satisfies PC_Q. Thus, since M" satisfies the program constraints in PC_Q, it is also a model of $\mathcal{MS}(\Pi^{-c}) \cup PC_Q$. Now, since $\mathcal{MS}(\Pi^{-c})$ does not have rules for predicates in $PC \setminus PC_Q$, M" is also a model of $\mathcal{MS}^{\leftarrow}(\Pi) = \mathcal{MS}(\Pi^{-c}) \cup PC$. □

Proof of Theorem (1). Theorem 1 Soundness with respect to cautious query answering follows from Proposition 5, and completeness follows from Proposition 4. □

The MS methodology, based on first leaving the program constraints aside and adding them after the magic rewriting, always works in the case of repair programs. Basically, because the program constraints are of the form $\leftarrow P_(\bar{x}, \mathbf{t_a}), P_(\bar{x}, \mathbf{f_a})$, and only when there are rules in $\Pi(D, IC)$ defining both $P_(\bar{x}, \mathbf{t_a})$ and $P_(\bar{x}, \mathbf{f_a})$. In the program $\mathcal{MS}(\Pi^{-c}(D, IC, Q))$ we will still find all the rules defining P; then it will be possible to check the satisfiability of the program constraints by its models. Thus, the stable models of the MS program satisfy the program constraints.

The MS method does not necessarily work for more general disjunctive logic programs. Sometimes, even if the MS technique is applied to a disjunctive program with program constraints that does not have stable models, the method can produce a program with stable models. This might happen if the query is related with a part of the program which is consistent with respect to the program constraint. The MS method focuses on that part of the program to answer the query [44]. So, the non-satisfaction of the program constraints will not be detected by the latter. Actually, as the following example shows, the MS methodology we introduced above might not work for general logic programs (as opposed to repair programs) that do have stable models.

Example 14. For the database instance $\{R(a)\}$ and the program Π:

$Y(x) \leftarrow S(x).$ $\quad$ $S(x) \leftarrow R(x),\ not\ P(x).$

$P(x) \leftarrow R(x),\ not\ S(x).$ $\quad$ $\leftarrow Y(x).$

there is only one stable model: $\mathcal{M} = \{R(a), P(a)\}$. But for query $Ans(x) \leftarrow P(x)$, our MS methodology produces a program that has two stable models (shown here without magic atoms): $\mathcal{M}_1 = \{R(a), P(a), \underline{Ans(a)}\}$; and $\mathcal{M}_2 = \{R(a), S(a)\}$, and as a consequence there are no cautious answers to the query, even though $\langle a \rangle$ should be an answer. This is due to the fact that MS does not select the rule $Y(x) \leftarrow S(x)$, because predicate Y is not relevant to answer the query. Thus, when the constraint $\leftarrow Y(x)$ is put back into the program, it is satisfied even though it should not. In contrast, in the case of a repair program, a rule that is relevant to check the satisfiability of a program constraint is never left out of the rewritten program obtained via MS. □

5. System Architecture

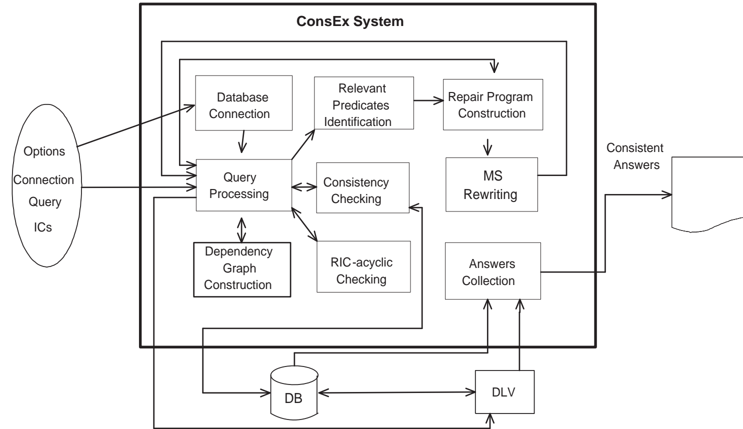
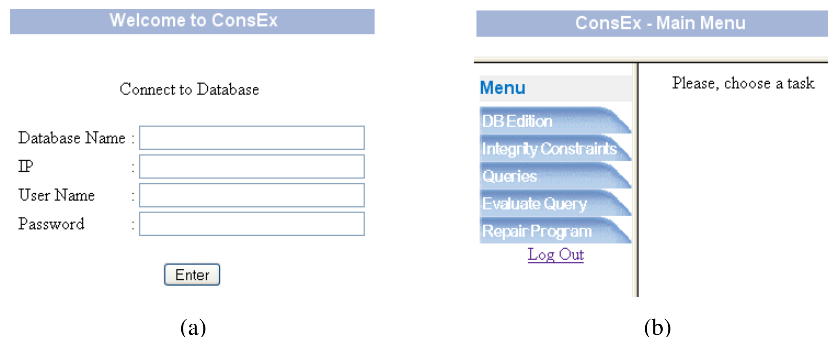


Figure 4: *ConsEx* Architecture

In Figure 4, that describes the general architecture of *ConsEx*, the *Database Connection* module receives the database parameters (database name, user and password) and connects to the database instance. We show in Figure 5 (a) the connection screen; and in Figure 5 (b), the main menu, obtained after connecting to the database.

Having input the database schema with its ICs, the system can be requested to check if the database instance is consistent. If not, one can request the generation of the repair program. However, before doing this, the *RIC-acyclic Checking* module uses the dependency graph to check if the set of ICs is *RIC*-acyclic. If it is not, the generation of the repair program is avoided, and a warning message is sent to the user. Otherwise, the *Repair Program Construction* module generates the repair program. It is constructed "on the fly", that is, all the annotations that appear in it are generated by the system, and the database is not affected. The facts of the program are not imported from the database into *ConsEx*. Instead, suitable sentences to import data are included into the repair program, as facilitated and understood by *DLV*.

The repair program may contain, for each extensional predicate P, the import sentence *#import* (*dbName*, *dbUser*, *dbPass*, "`SELECT * FROM P`", `P`), retrieving the tuples from relation P that



(a) (b)

Figure 5: *ConsEx*: Database Connection and Main Menu

will become the facts for predicate P in the program. As a result, when the program is evaluated by *DLV*, the database facts stored in the database *dbName* will be imported directly into the reasoning system. This is how DLV interacts with DBMS.

Of course, since we have not entered a query yet, this repair program will not be optimized with magic sets. The data import sentences just mentioned are not explicitly required, as shown before, when *ConsEx* is run with a query. Whatever data import statements are required, they are handled implicitly by the system. Most typically we will not be interested in performing a consistency check or generating the repair program without a concrete query at hand. However, the system allows us to do so if desired.

The *Query Processing* module receives the query and ICs; and coordinates the tasks needed to compute consistent answers. First, it checks queries for syntactic correctness. Currently in *ConsEx*, Datalog queries with negation can be written as logic programs in (rather standard) *DLV* notation or as queries in SQL. The former correspond to non-recursive Datalog queries with weak negation (*not*) and built-ins, which includes safe first-order queries. SQL queries may have disjunction (i.e. `UNION`), built-in literals in the `WHERE` clause, but neither negation nor recursion, i.e. unions of conjunctive queries with built-ins.[20]

After a query passes the syntax check, the query program is generated. For *DLV* queries, the query program is obtained by inserting the annotation $\mathbf{t}^{\star\star}$ into the literals in the bodies of the rules of the query that do not have a definition in the query program, but are defined in the repair program. For SQL queries, the query program is obtained by first translating queries into equivalent Datalog programs, and then by adding the annotation $\mathbf{t}^{\star\star}$ to the program rules, as for *DLV* queries.

Given a query, there might be ICs that are not related to the query. More precisely, their satisfaction or not by the given instance (and the corresponding portion of the repairs in the second case) does not influence the (standard or consistent) answers to the query. In order to capture the relevant ICs, the *Relevant Predicates Identification* module analyzes the interaction between the predicates in the query and those in the ICs by means of a *dependency graph*, which is generated by the *Dependency Graph Construction* module. We will use Example 7 to describe this feature and other system's components.

[20]We left as a future work the adaptation of the algorithms of *ConsEx* to include more general SQL queries, such as SQL queries with negation.

Figure 6 shows the dependency graph $\mathcal{G}(IC)$ for the ICs in Example 7. Then, for the query $Ans(x) \leftarrow S(b, x)$ the relevant predicates are S and R, because they are in the same component as the predicate S that appears in the query. Thus, the relevant IC to check is $\forall xy(S(x, y) \rightarrow R(x, y))$, which contains the relevant predicates.

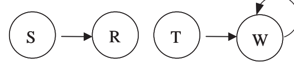


Figure 6: Dependency Graph $\mathcal{G}(IC)$ for ICs in Example 7

Next, *ConsEx* checks if the database is consistent with respect to the ICs that are relevant to the query. This check is performed by the *Consistency Checking* module, which generates an SQL query for each relevant IC, to check its satisfaction. For example, for the relevant IC $\forall xy(S(x, y) \rightarrow R(x, y))$ identified before, *ConsEx* generates the SQL query: `SELECT * FROM S WHERE (NOT EXISTS (SELECT * FROM R WHERE R.ID = S.ID AND R.NAME = S.NAME) AND ID IS NOT NULL AND NAME IS NOT NULL)`. This query is asking for violating tuples.

If the answer is empty, *ConsEx* proceeds to evaluate the given query directly on the original database instance, i.e. without computing repairs. For example, if the query is $\mathcal{Q}: \ Ans(x) \leftarrow S(b, x)$, the SQL query "`SELECT NAME FROM S WHERE ID='b'`", is generated by *ConsEx* and posed to D. However, in Example 7 we do have $\{(a, c)\}$ as the non-empty set of violations of the relevant IC. In consequence, the database is inconsistent, and, in order to consistently answer the query $\mathcal{Q}$, the repair program has to be generated. Remember that this program does not depend on these particular violating tuples, and can be used for any other query to be consistently answered from the inconsistent instance.

The *MS Rewriting* module generates the magic version of a repair program. It includes at the end appropriate database import sentences, which are generated by a static inspection of the magic program. This requires identifying first, in the rule bodies, the extensional database atoms (those not defined in the program, they show no annotation constants). Next, for each of these extensional atoms, it is checked if the magic atoms will have the effect of bounding their variables during the program evaluation. That is, it is checked if the constants appearing in the query will be pushed down to the program before query evaluation. For example, in the magic program $\mathcal{MS}^{\leftarrow}(\Pi, D, \mathcal{Q})$ for the query $Ans(x) \leftarrow S(b, x)$ shown in Section 4, the following rules contain extensional database atoms:
(a) $S_(x, y, \mathbf{t}^\star) \leftarrow magic_S_^{bfb}(x, \mathbf{t}^\star), S(x, y)$. (b) $R_(x, y, \mathbf{t}^\star) \leftarrow magic_R_^{bfb}(x, \mathbf{t}^\star), R(x, y)$.

In (a), the variable x in the extensional atom $S(x, y)$ will be bound during the evaluation due to the magic atom $magic_S_^{bfb}(x, \mathbf{t}^\star)$ appearing in the same body. This magic atom is defined in the magic program by the rule $magic_S_^{bfb}(x, \mathbf{t}^\star) \leftarrow magic_S_^{bfb}(x, \mathbf{t}^{\star\star})$, where atom $magic_S_^{bfb}(x, \mathbf{t}^{\star\star})$ is defined in its turn by the rule $magic_S_^{bfb}(b, \mathbf{t}^{\star\star}) \leftarrow magic_Ans^f$. Since $magic_Ans^f$ is always true in an MS program, $magic_S_^{bfb}(b, \mathbf{t}^{\star\star})$ will be true with the variable x in $S(x, y)$ eventually taking value b. As a consequence, the SQL query in the import sentence for predicate S will be: "`SELECT * FROM S WHERE  ID = 'b'`"

A similar static analysis can be done for rule (b), generating an import sentence for relation R. The generated import sentences will retrieve into *DLV* only the corresponding subsets of the relations in the database.

The resulting magic program is evaluated in *DLV*, that is automatically called by *ConsEx*, and the query answers are returned to the *Answer Collection* module, which formats the answers and returns them to the user as the consistent answers.

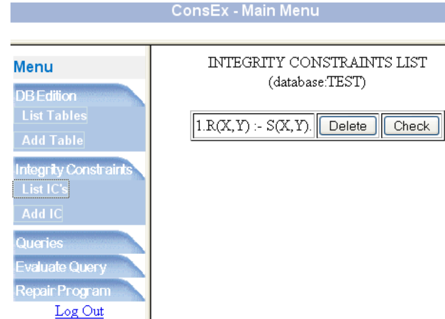


Figure 7: *ConsEx*: Integrity Constraints in Example 7

Moreover, *ConsEx* allows to check the satisfiability of the ICs stored in the system. (cf. Figure 7, option "Integrity Constraints"). As an illustration, the button "Check" besides the IC in Figure 7 performs the checking of the IC. This option is useful if a user wants to check if a specific constraint is satisfied by the database instance.

6. Experimental Evaluation

Several experiments on computation of consistent answers to queries were run with *ConsEx*. In particular, it was possible to quantify the gain in execution time by using magic sets instead of the direct evaluation of the repair programs.

6.1. Experimental Setup

The experiments were run on an Intel Pentium 4 PC, processor of 3.00 Ghz, 512 MB of RAM, and with Linux distribution UBUNTU 6.0. The database instance was stored in the IBM DB2 Universal Database Server Edition, version 8.2 for Linux. All the programs were run in the version of *DLV* for Linux released on Jan 12, 2006.

We considered a database schema with eight relations, and a set *IC* of ICs consisting of ten functional dependencies, and three RICs. In order to analyze scalability of CQA trough logic programs, we considered two databases instances D_1, and D_2, with 5,000 and 10,000 stored tuples, resp. The percentage N of inconsistent tuples, i.e. tuples participating in an IC violation varied between 1% and 10% of the data.[21]

Now, we report the execution time for three conjunctive queries, in both instances. The set *IC* contains the following FDs and RICs:

1. $\forall xyzsw(Passenger(x, y, z) \wedge Passenger(x, s, w) \rightarrow y = s)$.
2. $\forall xyzsw(Passenger(x, y, z) \wedge Passenger(x, s, w) \rightarrow z = w)$.
3. $\forall xyzwsmu(PlaneType(x, y, z, w) \wedge PlaneType(x, s, m, u) \rightarrow y = s)$.
4. $\forall xyzwsmu(PlaneType(x, y, z, w) \wedge PlaneType(x, s, m, u) \rightarrow z = m)$.
5. $\forall xyzwsmu(PlaneType(x, y, z, w) \wedge PlaneType(x, s, m, u) \rightarrow w = u)$.

[21]The files containing the database schema, ICs, the queries, and the instances used in the experiments are available in http://www.face.ubiobio.cl/~mcaniupa/ConsEx

6. $\forall xyz(Plane(x,y) \wedge Plane(x,z) \rightarrow y = z)$.
7. $\forall x_0 \cdots x_{16}\ (Flight(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) \wedge Flight(x_0, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_5, x_{16}) \rightarrow x_1 = x_9)$.
8. $\forall zx(Plane(z,x) \rightarrow \exists yuw PlaneType(x,y,u,w))$.
9. $\forall xyzws(Inspection(x,y,z,w,s) \rightarrow \exists u Plane(w,u))$.
10. $\forall x_0 \cdots x_8\ (Flight(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) \rightarrow \exists z Plane(x_8, z))$.

The three conjunctive queries are:

1. $Ans(x,y,z) \leftarrow Plane(x,y), PlaneType(y,w,z,s)$.
2. $Ans(x,w) \leftarrow Passenger(z,w,1), Flying(z,y,x)$.
3. $Ans \leftarrow Passenger(18, smith, 18)$.

The first query is open, i.e. with free variables and contains a join, and no constants. The second query is open, contains joins, and also constants, like the query in Section 4. The third query is boolean, i.e. it does not have free variables. All these queries fall in the class of *Tree*-queries for which CQA is tractable under key constraints [39]. However, since we are also considering RICs, which are repaired by inserting tuples with null values, it is not possible to use the polynomial time algorithm for CQA presented in [39].

6.2. Experimental Results

In the charts, *RQ* indicates the straightforward evaluation of the repair program combined with the query program, whereas its magic sets optimization is indicated with *MS*. Figure 8 shows the running time for the first query in the two instances. We can see that MS is faster than the straightforward evaluation of programs in both database instances. Indeed, for every percentage of inconsistency, the MS methodology returns answers in less than twenty seconds (in both database instances), while the straightforward evaluation returns answers after fifty seconds when the database presents 3% of inconsistent tuples, in database instance D_1, and after fifty seconds when the level of inconsistencies is higher than 1% in the second instance. Moreover, the execution time of the MS methodology is almost invariant with respect to percentage of inconsistency. Despite the absence of constants in the query, MS offers a substantial improvement because the magic program essentially keeps only the rules and relations that are relevant to the query, which reduces the ground instantiation of the program by *DLV*.

Figure 9 shows the execution time for the second, partially-ground query in both database instances. Again, MS computes answers much faster than the straightforward evaluation. In the first instance, for percentages smaller than 5% (and equal) of inconsistent tuples, MS returns answers in less than 25 seconds. When the level of inconsistencies is 10%, MS starts returning answers in 6 minutes. In the second instance, MS has a good performance for percentages of inconsistencies smaller than 5% (and equal). With a higher percentage, MS becomes slower, but it is still better than the straightforward evaluation. In particular, when the level of inconsistency is 10% of 10,000 tuples, MS returns answers after 1 hour. However, a direct evaluation of the programs produces answers after 6000 minutes (100 hours approx.). The better performance of MS is due to the occurrence of constants in the query, which the magic rules push down to the database relations. This causes less tuples to be imported into *DLV*, and the ground instantiation of the magic program is reduced (with respect to the original program).

Figure 10 shows the execution time for the third, boolean conjunctive query in both database instances. This query asks for a tuple that is involved in inconsistencies of a functional dependency, and therefore, the consistent answer to this query is *no*. In this case, the direct evaluation

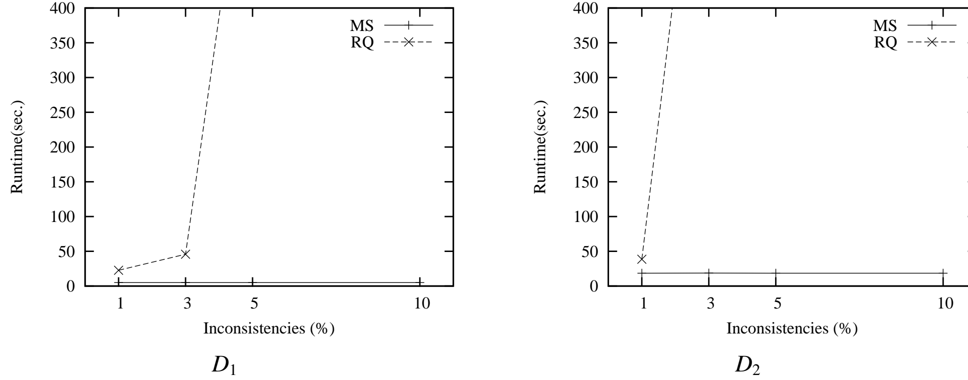


Figure 8: Running Time for the Conjunctive Query with Free Variables

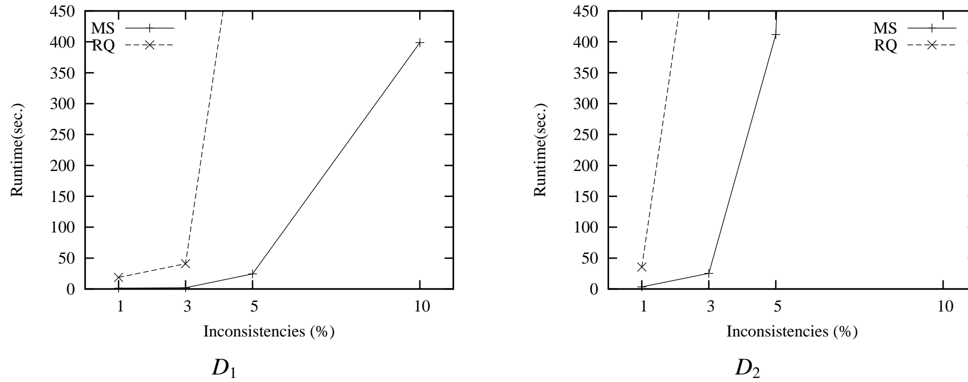


Figure 9: Running Time for the Partially-Ground Conjunctive Query with Free Variables

of the repair and query program, and the evaluation with magic sets show an excellent performance, but again, MS computes answers much faster than the straightforward evaluation. This good performance can be due to the fact that the atom $Passenger(18, smith, 18)$, in the query, is involved in inconsistencies, and then, the *Ans* atom is not true in every stable model of the programs. Then, DLV stops the computation of the query when it finds a stable model that does not contain *Ans*.

Furthermore, MS shows an excellent scalability for databases that present less than 5% of inconsistent tuples. For instance, MS computes answers to conjunctive queries with free variables, and to boolean queries from database instances D_1 and D_2 in less than thirty seconds, even with a database D_2 that contains twice as many tuples as D_1.

7. Conclusions

We have seen that the *ConsEx* system computes database repairs and consistent answers to first-order queries (and beyond) by evaluation of logic programs with stable model semantics

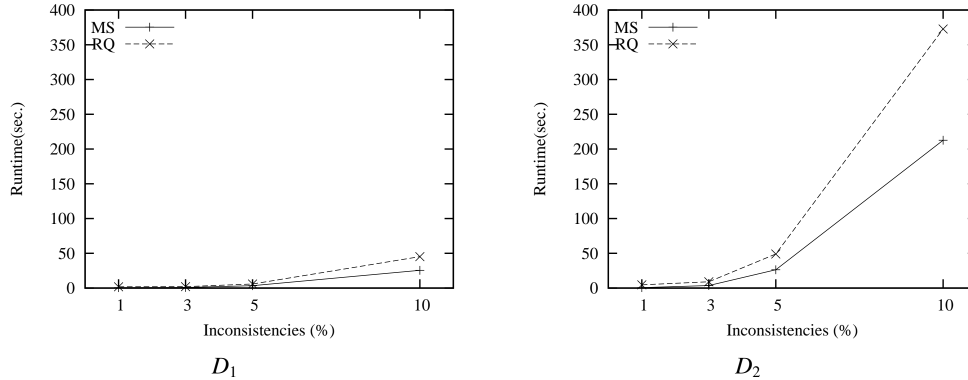


Figure 10: Running Time for the Boolean Conjunctive Query

that specify both the repairs and the query. In order to make query answering more efficient in practice, *ConsEx* implements a sound and complete magic set technique for disjunctive repair programs with program constraints. Moreover, *ConsEx* takes advantage of the smooth interaction between the logic programming environment and the database management systems (DBMS), as enabled by *DLV*. In this way, it is possible to exploit capabilities of the DBMS, such as storing and indexing, and the capabilities of *DLV* to compute stable models. Furthermore, bringing the whole database into *DLV*, to compute repairs and consistent answers, is quite inefficient. In our case, it is possible to keep the instance in the database, while only the relevant data is imported into the logic programming system.

The excellent performance exhibited by the magic sets techniques makes us think that CQA is viable and can be used in practical cases. Most likely, real databases do not contain such a high percentage of inconsistent data as those used in our experiments. However, we should notice that CQA has a high intrinsic data complexity [12, 27, 56]. In consequence, there are intrinsic limits on what a system like *ConsEx* (and others for CQA) can achieve in terms of performance.

The methodology for CQA based on repair logic programs as implemented in *ConsEx* is more general than those that have been proposed and implemented so far. It covers all the queries and ICs found in usual database practice and more. Implementations of other systems for CQA have been reported before. The *Queca* system [25] implements the query rewriting methodology presented in [4], and can be used with universal ICs with at most two database atoms (plus built-ins) and projection-free conjunctive queries. The system *Hippo* [27] implements first-order query rewriting based on graph-theoretic methods. It works for denial constraints and inclusion dependencies under a tuple deletion repair semantics, and projection-free conjunctive queries. The system *ConQuer* [38] implements CQA for key constraints and a non-trivial class of conjunctive queries with projections. Comparisons in terms of performance between *ConsEx* and these more specialized and optimized systems, for the specific classes of ICs and queries they can handle, still have to be made. Since these systems are specialized (actually, implemented) for certain classes of queries and ICs, they are likely to exhibit a better performance than *ConsEx* for their classes.

In *ConsEx*, consistency checking of databases with SQL null values and repairs that appeal to SQL null values both follow the precise and general semantics introduced in [16]. However,

when queries are answered in *ConsEx*, after the repair and MS programs have been generated, the query answer semantics is the usual logic programming semantics that treats nulls as any other constant. A semantics for query answering in the presence of SQL nulls that is compatible with null-based SQL IC satisfaction and repair semantics used in *ConsEx* is proposed in [17]. Its implementation in *ConsEx* corresponds to ongoing and future work. We also leave for future work the extension of CQA to broader classes of queries, in particular, to aggregate queries by means of logic programs as done in [23].

Surveys of CQA can be found in [10, 12, 28]. There are different repair semantics in the literature. Some comparisons between them can be found in [2, 50]. *ConsEx* implements the "classic" repair semantics based on minimality under set inclusion of sets of inserted or deleted tuples, namely the one introduced and investigated in [4], and the most studied in the CQA literature. A different repair semantics emerges if these sets of tuples is minimized under cardinality. In [5], although devoted mainly to the classic repair semantics, it is indicated how answer set programs with weak program constraints [47] can be used to specify repairs.

In [27] the database instance is assumed to be possibly incorrect but complete, then repairs are obtained by deletion of tuples only, i.e. the insertion of new tuples is not considered as an option to restore consistency. It is straightforward to write down repair programs to capture these repairs. The same applies to the repair semantics in [19], where the database instance is assumed to be possibly incorrect and incomplete, then functional dependencies are repaired by deletion, and referential ICs by adding arbitrary elements of the domain.

In [11, 14, 37, 45, 58, 59] minimality associated to repairs is based on the cardinality of the set of updates, i.e. changes of attributes values (as opposed to whole tuples). In [37] answer set programs are used to specify repairs of this kind for census databases.

Other repair policies have been proposed in the recent literature. For example, specifying *founded repairs* [24] by means of repairs programs as used by *ConsEx* seems to be easy to do. It should be possible to specify the *preferred repairs* introduced in [57] by adding to repair programs global preference criteria on the intended models. Preferences in answer programs have been investigated, e.g. in [33, 18].

In [51] the repairs of a database with respect to denial constraints are represented by a disjunctive database, which are finite sets of disjunctions of database facts. This approach is different from a specification via answer set programs. The latter is much more general since it can handle a broader class on ICs. Also a program layer for CQA can be easily added to the specification of repairs. However, doing CQA on top of a specification of repairs via a disjunctive database requires additional investigation.

It is important to mention that in the case of Data Warehouses (DWs), where efficiency in query answering is vital, it is almost impossible to compute consistent answers to queries based on database repairs. This is because DWs store terabytes of data; and, therefore, the computation of repairs may be unfeasible. For this kind of databases a minimal repair should be chosen to be materialized. In [13] CQA over inconsistent DWs with respect to a special class of constraints, called *strictness constraints*, is analyzed. Also, in this paper a canonical consistent instance is proposed, which is a new instance that consolidates information from all the repairs. In some cases, this new instance may be a good candidate to be materialized and used to compute approximate answers to queries.

Acknowledgements: This project is partially funded by FONDECYT, Chile, grant number 11070186. Part of this research was done during visits of L. Bertossi to the Universities of Concepcion and Bío-Bío (UBB). He has also supported by an NSERC Discovery Grant (#315682).

L. Bertossi is a Faculty Fellow of IBM Center for Advanced Studies (Toronto Lab.), and Adjunct Full Professor at the University of Concepcion (Chile). We are grateful to Claudio Gutiérrez and Pedro Campos, both from UBB, for their help with the implementation of algorithms and the interface of *ConsEx*. Conversations with Wolfgang Faber and Nicola Leone are very much appreciated.

References

[1] S. Abiteboul, O. Duschka, Complexity of Answering Queries using Materialized Views, in: Proceedings of the ACM Symposium on Principles of Database Systems (PODS'98), 1998, pp. 254–263.

[2] F. Afrati, P. Kolaitis, Repair Checking in Inconsistent Databases: Algorithms and Complexity, in: Proceedings of the International Conference on Database Theory (ICDT'09), 2009, pp. 31–41.

[3] K. R. Apt, H. A. Blair, A. Walker, Towards a Theory of Declarative Knowledge, 1988, Ch. Foundations of Deductive Databases and Logic Programming, pp. 89–148.

[4] M. Arenas, L. Bertossi, J. Chomicki, Consistent Query Answers in Inconsistent Databases, in: Proceedings of the ACM Symposium on Principles of Database Systems (PODS'99), 1999, pp. 68–79.

[5] M. Arenas, L. Bertossi, J. Chomicki, Answer Sets for Consistent Query Answering in Inconsistent Databases, Theory and Practice of Logic Programming 3 (4-5) (2003) 393–424.

[6] F. Bancilhon, D. Maier, Y. Sagiv, J. D. Ullman, Magic sets and other Strange Ways to Implement Logic Programs (extended abstract), in: Proceedings of the ACM Symposium on Principles of Database Systems (PODS'86), 1986, pp. 1–15.

[7] C. Beeri, R. Ramakrishnan, On the Power of Magic, in: Proceedings of the ACM Symposium on Principles of Database Systems (PODS'87), 1987, pp. 269–284.

[8] P. Barceló, L. Bertossi, Logic Programs for Querying Inconsistent Databases, in: Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL'03), Springer LNCS 2562, 2003, pp. 208–222.

[9] P. Barceló, L. Bertossi, L. Bravo, Characterizing and Computing Semantically Correct Answers from Databases with Annotated Logic and Answer Sets, in: Semantics in Databases, Springer LNCS 2582, 2003, pp. 1–27.

[10] L. Bertossi, J. Chomicki, Logics for Emerging Applications of Databases, Springer, 2003, Ch. Query Answering in Inconsistent Databases, pp. 43–83.

[11] L. Bertossi, L. Bravo, E. Franconi, A. Lopatenko, Complexity and Approximation of Fixing Numerical Attributes in Databases Under Integrity Constraints, in: Proceedings of the International Symposium on Database Programming Languages (DBPL'05), Springer LNCS 3774, 2005, pp. 262–278.

[12] L. Bertossi, Consistent Query Answering in Databases, ACM Sigmod Record (database principles column) 2 (35) (2006) 68–76.

[13] L. Bertossi, L. Bravo, M. Caniupan, Consistent Query Answering in Data Warehouses, in: Proceedings of the III Alberto Mendelzon International Workshop on Foundations of Data Management (AMW'09), Vol. 450, 2009, Arequipa, Peru.

[14] P. Bohannon, W. Fan, M. Flaster, R. Rastogi, A Cost-Based Model and Effective Heuristic for Repairing Constraints by value Modification, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'05), 2005, pp. 143–154.

[15] L. Bravo, L. Bertossi, Consistent Query Answering under Inclusion Dependencies, in: H. Lutfiyya, J. Singer, D. A. Stewart (Eds.), 14th Annual IBM Centers for Advanced Studies Conference (CASCON'04), IBM, 2004, pp. 202–216.

[16] L. Bravo, L. Bertossi, Semantically Correct Query Answers in the Presence of Null Values, in: Proceedings of the EDBT WS on Inconsistency and Incompleteness in Databases (IIDB'06), Springer LNCS 4254, 2006, pp. 336–357.

[17] L. Bravo, Handling Inconsistency in Databases and Data Integration Systems, Ph.D. thesis, Carleton University (2007). http://homepages.inf.ed.ac.uk/lbravo/Publications.htm

[18] G. Brewka, Preferences in Answer Set Programming, in: Current Topics in Artificial Intelligence, 2006, Springer LNCS 4177, pp. 1-10.

[19] A. Cali, D. Lembo, R. Rosati, On the Decidability and Complexity of Query Answering over Inconsistent and Incomplete Databases, in: Proceedings of the ACM Symposium on Principles of Database Systems (PODS'03), 2003, pp. 260–271.

[20] A. Cali, G. Gottlob, T. Lukasiewicz, A General Datalog-based Framework for Tractable Query Answering over Ontologies, in: Proceedings of the ACM Symposium on Principles of Database Systems (PODS'09), 2009, pp. 77–86.

[21] M. Caniupan, L. Bertossi, Optimizing Repair Programs for Consistent Query Answering, in: Proceedings of the

XXV International Conference on The Chilean Computer Science Society (SCCC'05), IEEE Computer Society, 2005, pp. 3–12.
[22] M. Caniupan, L. Bertossi, The Consistency Extractor System: Querying Inconsistent Databases using Answer Set Programs, in: Proceedings of the International Conference on Scalable Uncertainty Management (SUM'07), Springer LNCS 4772, 2007, pp. 74–88.
[23] M. Caniupan, Optimizing And Implementing Repair Programs for Consistent Query Answering In Databases, Ph.D. thesis, School of Computer Science, Carleton University (2007). http://www.face.ubiobio.cl/∼mcaniupa/publications.htm
[24] L. Caroprese, S. Greco, E. Zumpano, Ester, Active Integrity Constraints for Database Consistency Maintenance, IEEE Transactions on Knowledge and Data Engineering 21 (7) (2009) 1042–1058.
[25] A. Celle, L. Bertossi, Querying Inconsistent Databases: Algorithms and Implementation, in: Proceedings of the International Conference on Computational Logic (CL'00), Springer LNAI 1861, 2000, pp. 942–956.
[26] S. Ceri, G. Gottlob, L. Tanca, Logic Programming and Databases, Springer-Verlag New York, Inc., New York, NY, USA, 1990.
[27] J. Chomicki, J. Marcinkowski, Minimal-Change Integrity Maintenance using Tuple Deletions, Information and Computation 197 (1-2) (2005) 90–121.
[28] J. Chomicki, Consistent Query Answering: Five Easy Pieces, in: Proceedings of the International Conference on Database Theory (ICDT'07), Springer LNCS 4353, 2007, pp. 1–17.
[29] C. Cumbo, W. Faber, G. Greco, N. Leone, Enhancing the Magic-Set Method for Disjunctive Datalog Programs, in: Proceedings of the 20th International Conference on Logic Programming (ICLP'04), Springer LNCS 3132, 2004, pp. 371–385.
[30] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and Expressive Power of Logic Programming, ACM Computing Surveys 33 (3) (2001) 374–425.
[31] T. Eiter, G. Gottlob, On the Computational Cost of Disjunctive Logic Programming: Propositional Case, Journal Annals of Mathematics and Artificial Intelligence 15 (3-4) (1995) 257–456.
[32] T. Eiter, M. Fink, G. Greco, D. Lembo, Efficient Evaluation of Logic Programs for Querying Data Integration Systems, in: Proceedings of the International Conference on Logic Programming (ICLP'03), Springer LNCS 2916, 2003, pp. 163–177.
[33] T. Eiter, W. Faber, N. Leone, G. Pfeifer, Computing Preferred Answer Sets by Meta-Interpretation in Answer Set Programming, Theory and Practice of Logic Programs 3 (4-5) (2003) 463-498.
[34] T. Eiter, G. Gottlob, H. Mannila, Disjunctive Datalog, ACM Transactions on Database Systems 22 (3) (1997) 364–418.
[35] W. Faber, G. Greco, N. Leone, Magic Sets and their Application to Data Integration, Journal of Computer and System Sciences 73 (4) (2007) 584–609.
[36] R. Fagin, P. Kolaitis, R. Miller, L. Popa Data Exchange: Semantics and Query Answering, Theoretical Computer Science Volume 336 (1) (2005) 89–124.
[37] E. Franconi, A. Laureti-Palma, N. Leone, S. Perri, F. Scarcello, Census Data Repair: a Challenging Application of Disjunctive Logic Programming, in: Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'01), Springer LNCS 2250, 2001, pp. 561–578.
[38] A. Fuxman, E. Fazli, R. J. Miller, ConQuer: Efficient Management of Inconsistent Databases, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'05), 2005, pp. 155–166.
[39] A. Fuxman, R. J. Miller, First-Order Query Rewriting for Inconsistent Databases, Journal of Computer and System Sciences 73 (4) (2007) 610–635.
[40] M. Gelfond, V. Lifschitz, Classical Negation in Logic Programs and Disjunctive Databases, New Generation Computing 9 (3/4) (1991) 365–386.
[41] M. Gelfond, N. Leone, Logic Programming and Knowledge Representation: The A-Prolog Perspective, Artificial Intelligence 138 (1-2) (2002) 3–38.
[42] S. Greco, Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries, IEEE Transactions on Knowledge and Data Engineering 15 (2) (2003) 368–385.
[43] G. Greco, S. Greco, E. Zumpano, A Logical Framework for Querying and Repairing Inconsistent Databases, IEEE Transactions on Knowledge and Data Engineering 15 (6) (2003) 1389–1408.
[44] G. Greco, S. Greco, I. Trubitsyna, E. Zumpano, Optimization of Bound Disjunctive Queries with Constraints, Theory Pract. Log. Program. 5 (6) (2005) 713–745.
[45] S. Kolahi, L. V.S. Lakshmanan, On Approximating Optimum Repairs for Functional Dependency Violations, in: Proceedings of the International Conference on Database Theory (ICDT'09), 2009, pp. 53–62.
[46] D. Lembo, R. Rosati, M. Ruzzi, On the First-Order Reducibility of Unions of Conjunctive Queries over Inconsistent Databases, in: Proceedings of Current Trends in Database Technology (EDBT'06), Springer LNCS 4254, 2006, pp. 358–374.
[47] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, C. Koch, C. Mateis, S. Perri, F. Scarcello, The DLV System

for Knowledge Representation and Reasoning, ACM Transactions on Computational Logic 7 (3) (2006) 499–562.
[48] V. Lifschitz, H. Turner, Splitting a Logic Program, in: Proceedings of the International Conference on Logic Programming (ICLP'94), 1994, pp. 23–37.
[49] J. W. Lloyd, Foundations of Logic Programming, Springer Verlag, 1987.
[50] A. Lopatenko, L. Bertossi. Complexity of Consistent Query Answering in Databases under Cardinality-Based and Incremental Repair Semantics, in: Proceedings of the International Conference of Database Theory (ICDT'07), 2007, Springer LNCS 4353, pp. 179-193.
[51] C. Molinaro, J. Chomicki, J. Marcinkowski, Disjunctive Databases for Representing Repairs, to appear in Annals of Mathematics and Artificial Intelligence, preprint 0811.2117v1, 2008.
[52] T. C. Przymusinski, On the Declarative Semantics of Deductive Databases and Logic Programs, 1988, pp. 193–216.
[53] T. C. Przymusinski, Stable Semantics for Disjunctive Programs, New Generation Computing 9 (1991) 401–424.
[54] R. Reiter, Towards a Logical Reconstruction of Relational Database Theory, in: M. L. Brodie, J. Mylopoulos, J. W. Schmidt (Eds.), On Conceptual Modelling, Springer-Verlag, 1984, pp. 191–233.
[55] K. A. Ross, Modular Stratification and Magic Sets for Datalog Programs with Negation, Journal of the ACM 41 (6) (1994) 1216–1266.
[56] S. Staworko, J. Chomicki, Consistent Query Answers in the Presence of Universal Constraints, Information Systems 35 (1) (2010) 1–22.
[57] S. Staworko, J. Chomicki, J. Marcinkowski, Prioritized Repairing and Consistent Query Answering in Relational Databases, to appear in Annals of Mathematics and Artificial Intelligence, preprint arXiv:0908.0464v1, 2009.
[58] J. Wijsen, Condensed Representation of Database Repairs for Consistent Query Answering, in: Proceedings of the International Conference on Database Theory (ICDT'03), Springer LNCS 2572, 2003, pp. 378–393.
[59] J. Wijsen, Database Repairing Using Updates, ACM Transactions on Database Systems 30 (3) (2005) 722–768.
[60] J. Wijsen, On the Consistent Rewriting of Conjunctive Queries Under Primary Key Constraints, Information Systems 34 (7) (2009) 578–601.