The Consistency Extractor System: Querying Inconsistent Databases using Answer Set Programs

Monica Caniupan
Universidad del Bio-Bio
Departamento de Sistemas de Informacion
Concepcion, Chile.
mcaniupa@ubiobio.cl

Leopoldo Bertossi
Carleton University
School of Computer Science
Ottawa, Canada.
bertossi@scs.carleton.ca

Abstract. We present the *Consistency Extractor System* (*ConsEx*) that uses *answer set programming* to compute consistent answers to first-order queries posed to relational databases that may be inconsistent wrt their integrity constraints. Among other features, *ConsEx* implements a *magic sets* technique to evaluate queries via disjunctive logic programs with stable model semantics that specify the repair of the original database. We describe the methodology and the system; and also present some experimental results.

1 Introduction

For several reasons, databases may become inconsistent wrt certain integrity constraints (ICs) they are supposed to satisfy [1, 6]. However, in most of the cases only a small portion of the database violates the ICs, and the inconsistent database can still be queried and give us useful and correct information. In order to characterize this correct data, the notion of *consistent answer* to a query was introduced in [1], along with a mechanism for computing those answers.

Intuitively, an answer to a query $\mathcal{Q}$ in a relational database instance D is *consistent* wrt a set IC of ICs if it is an answer to $\mathcal{Q}$ in every repair of D, where a repair of D is an instance over the same schema that satisfies IC and is obtained from D by deleting or inserting a minimal set -under set inclusion- of whole database tuples. More precisely, if a database instance is conceived as a finite set of ground atoms, then for a repair D' of D wrt IC it holds: (a) D' satisfies IC, denoted $D' \models IC$, and (b) the symmetric difference $D \triangle D'$ is minimal under set inclusion [1].

The algorithm for consistent query answering (CQA) in [1] is based on a first-order query rewriting of the original query. The new query is posed to the original database, and the usual answers are the consistent answers to the original query. This algorithm has limitations wrt the class of ICs and queries it can handle. CQA based on first-order query rewriting was later extended [17, 22, 27], but it is still limited in its applicability (cf. Section 6), which is explained by the intrinsic data complexity of CQA (cf. [6, 7] for surveys in this direction).

In several papers [2, 25, 3, 4, 20, 9, 10], database repairs have been specified as the stable models of disjunctive logic programs with stable model semantics [23] (aka. *answer set programs*). It turns out that the data complexity of query evaluation from disjunctive logic programs with stable model semantics [19] matches the data complexity of CQA. In this line, the approach in [10] is the most general and also the more realistic, in the sense that it takes into consideration possible occurrences of null values and the way they are used in real database practice, and these null values are also used to restore consistency wrt referential ICs.

In *ConsEx* we implement, use and optimize the repair logic programs introduced in [10]. In consequence, *ConsEx* can be used for CQA wrt arbitrary universal ICs, acyclic

sets of referential ICs, and NOT-NULL constraints. The queries supported are Datalog queries with negation, which goes beyond first-order queries. Consistent answers to queries can be computed by evaluating queries against the repair programs, e.g. using the *DLV* system, that implements the stable model semantics of disjunctive logic programs [28].

The *ConsEx* system implements the most general methodology for CQA wrt the class of ICs and queries that can be handled. To achieve this goal, *ConsEx* computes and optimizes the logic programs that specify database repairs or represent queries. These programs are internally passed as inputs to *DLV*, which evaluates them. All this is done in interaction with IBM DB2 relational DBMS. *ConsEx* can be applied to relational databases containing NULL, and all the first-order ICs and (non-aggregate) queries used in database practice and beyond.

Using logic programs for CQA in a straightforward manner may not be the most efficient alternative. As shown in [12], a more efficient way to go is to apply the so-called *magic sets* (MS) techniques, that transform the combination of the query program and the repair program into a new program that, essentially, contains a subset of the original rules in the repair program, those that are relevant to evaluate the query.

Classically, MS optimizes the bottom-up processing of queries in deductive (Datalog) databases by simulating a top-down, query-directed evaluation [5, 15]. More recently, the MS techniques have been extended to logic programs with stable models semantics [21, 24, 18, 26]. In [12] it was shown how to adopt and adapt those techniques to our repair programs, resulting in a sound and complete MS methodology for the repair programs with program constraints. In Section 3, we briefly describe this particular MS methodology, which is the one implemented in the *ConsEx* system. In Section 5, we show that the use of MS in the evaluation of queries improves considerably the execution time of queries.

In this paper we describe both the methodologies implemented in *ConsEx* (more details about them can be found in [12]), and the features, functionalities, and performance of this system (again, more details and proofs of results can be found in [13]).

2 Preliminaries

We consider a relational database schema $\Sigma = (\mathcal{U}, \mathcal{R}, \mathcal{B})$, where $\mathcal{U}$ is the possibly infinite database domain with $null \in \mathcal{U}$, $\mathcal{R}$ is a fixed set of database predicates, each of them with a finite, and ordered set of attributes, and $\mathcal{B}$ is a fixed set of built-in predicates, like comparison predicates, e.g. $\{<, >, =, \neq\}$. There is a predicate $IsNull(\cdot)$, and $IsNull(c)$ is true iff c is $null$. Instances for a schema Σ are finite collections D of ground atoms of the form $R(c_1, ..., c_n)$, called *database tuples*, where $R \in \mathcal{R}$, and $(c_1, ..., c_n)$ is a *tuple* of constants, i.e. elements of $\mathcal{U}$. The extensions for built-in predicates are fixed, and possibly infinite in every database instance. There is also a fixed set IC of integrity constraints, that are sentences in the first-order language $\mathcal{L}(\Sigma)$ determined by Σ. They are expected to be satisfied by any instance for Σ, but they may not.

A *universal integrity constraint* is a sentence in $\mathcal{L}(\Sigma)$ that is logically equivalent to a sentence of the form [10]: $\forall \bar{x}(\bigwedge_{i=1}^{m} P_i(\bar{x}_i) \rightarrow \bigvee_{j=1}^{n} Q_j(\bar{y}_j) \vee \varphi)$, where $P_i, Q_j \in \mathcal{R}$, $\bar{x} = \bigcup_{i=1}^{m} \bar{x}_i$, $\bar{y}_j \subseteq \bar{x}$, and $m \geq 1$. Here φ is a formula containing only disjunctions of built-in atoms from $\mathcal{B}$ whose variables appear in the antecedent of the implication.

We will assume that there exists a propositional atom $\mathbf{false} \in \mathcal{B}$ that is always false in the database. Domain constants different from $null$ may appear in a UIC. A *referential integrity constraint* (RIC) is a sentence of the form:[1] $\forall \bar{x}(P(\bar{x}) \rightarrow \exists \bar{z}\, Q(\bar{y}, \bar{z}))$, where $\bar{y} \subseteq \bar{x}$ and $P, Q \in \mathcal{R}$. A *NOT NULL*-constraint (NNC) is a denial constraint of the form: $\forall \bar{x}(P(\bar{x}) \wedge IsNull(x_i) \rightarrow \mathbf{false})$, where $x_i \in \bar{x}$ is in the position of the attribute that cannot take null values.

Notice that our RICs contain at most one database atom in the consequent. E.g. tuple-generating joins in the consequent are excluded, and this is due to the fact that RICs will be repaired using null values (for the existential variables), whose participation in joins is problematic. It would be easy to adapt our methodology in order to include that kind of joins as long as they are repaired using other values in the domain. However, this latter alternative opens the ground for undecidability of CQA [11], which is avoided in [10] by using null values to restore consistency.

Based on the repair semantics and the logic programs introduced in [10], CQA as implemented in *ConsEx* works for *RIC-acyclic* sets of universal, referential, and NNCs. In this case, there is a one-to-one correspondence between the stable models of the repair program and the database repairs [10]. That a set of ICs is RIC-acyclic essentially means that there are no cycles involving RICs (cf. [12, 10] for details). For example, $IC = \{\forall x(S(x) \rightarrow Q(x)), \forall x(Q(x) \rightarrow S(x)), \forall x(Q(x) \rightarrow \exists y T(x, y))\}$ is RIC-acyclic, whereas $IC' = IC \cup \{\forall xy(T(x, y) \rightarrow Q(y))\}$ is not, because there is a cycle involving the RIC $\forall x(Q(x) \rightarrow \exists y T(x, y))$. In the following, we will assume that IC is a fixed, finite and RIC-acyclic set of UICs, RICs and NNCs. A database instance D is said to be *consistent* if it satisfies IC. Otherwise, it is *inconsistent* wrt IC.

In particular, RICs are repaired by tuple deletions or tuple insertion with null values. Notice that introducing null values to restore consistency makes it necessary to modify the repair semantics introduced in [1], which does not consider RICs or null values. This is needed in order to give priority to null values over arbitrary domain constants when restoring consistency wrt RICs. It becomes necessary to modify accordingly the notion of minimality associated to repair as shown in the following example (cf. [10] for details).

Example 1. The database instance $D = \{P(a, null), P(b, c), R(a, b)\}$ is inconsistent wrt IC: $\forall\ xy\ (P(x, y) \rightarrow \exists z R(x,\ z))$. There are two repairs: $D_1 = \{P(a, null),$ $P(b, c),\ R(a, b),\ R(b, null)\}$, with $\Delta(D, D_1) = \{R(b, null)\}$, and $D_2 = \{P(a, null),$ $R(a, b)\}$, with $\Delta(D, D_2) = \{P(b, c)\}$. For every $d \in \mathcal{U} \setminus \{null\}$, the instance $D_3 = \{P(a, null),\ P(b, c),\ R(a, b),\ R(b, d)\}$ is not a repair, because it is not minimal. □

Database repairs can be specified as stable models of disjunctive logic programs. The repair programs introduced in [10] build on the repair programs first introduced in [3] for universal ICs. They use annotation constants to indicate the atoms that may become true or false in the repairs in order to satisfy the ICs. Each atom of the form $P(\bar{a})$ (except for those that refer to the extensional database) receives one of the annotation constants. In $P\_(\bar{a}, \mathbf{t_a})$, the annotation $\mathbf{t_a}$ means that the atom is advised to made true (i.e. inserted into the database). Similarly, $\mathbf{f_a}$ indicates that the atom should be made false (deleted).[2]

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

[2] In order to distinguish a predicate P that may receive annotations in an extra argument from the same predicate in the extensional database, that does not contain annotations, the former is replaced by $P\_$.

For each IC, a disjunctive rule is constructed in such a way that the body of the rule captures the violation condition for the IC; and the head describes the alternatives for restoring consistency, by deleting or inserting the participating tuples (cf. rules 2. and 3. in Example 2).

Annotation $\mathbf{t}^\star$ indicates that the atom is true or becomes true in the program. It is introduced in order to keep repairing the database if there are interacting ICs; and e.g. the insertion of a tuple may generate a new IC violation. Finally, atoms with constant $\mathbf{t}^{\star\star}$ are those that become true in the repairs. They are use to read off the database atoms in the repairs. All this is illustrated in the following example (cf. [10] for the general form of the repair programs).

Example 2. Consider the database schema $\Sigma = \{S(ID, NAME),\ R(ID, NAME),\ T(ID, DEPTO),\ W(ID, DEPTO, SINCE)\}$, the instance $D = \{S(a, c), S(b, c), R(b, c), T(a, null), W(null, b, c)\}$, and $IC = \{\forall xy(S(x, y) \rightarrow R(x, y)), \forall xy(T(x, y) \rightarrow \exists z W(x, y, z)), \forall xyz(W(x, y, z) \wedge IsNull(x) \rightarrow \mathbf{false})\}$. 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. $\underline{S}(x, y, \mathbf{f_a}) \vee \underline{R}(x, y, \mathbf{t_a}) \leftarrow \underline{S}(x, y, \mathbf{t}^\star), R(x, y, \mathbf{f_a}), x \neq null, y \neq null.$
   $\underline{S}(x, y, \mathbf{f_a}) \vee \underline{R}(x, y, \mathbf{t_a}) \leftarrow \underline{S}(x, y, \mathbf{t}^\star),\ not\ R(x, y), x \neq null, y \neq null.$
3. $\underline{T}(x, y, \mathbf{f_a}) \vee \underline{W}(x, y, null, \mathbf{t_a}) \leftarrow \underline{T}(x, y, \mathbf{t}^\star),\ not\ aux(x, y), x \neq null, y \neq null.$
   $aux(x, y) \leftarrow \underline{W}(x, y, z, \mathbf{t}^\star),\ not\ \underline{W}(x, y, z, \mathbf{f_a}), x \neq null, y \neq null, z \neq null.$
4. $\underline{W}(x, y, z, \mathbf{f_a}) \leftarrow \underline{W}(x, y, z, \mathbf{t}^\star), x = null.$
5. $\underline{S}(x, y, \mathbf{t}^\star) \leftarrow S(x, y).$
   $\underline{S}(x, y, \mathbf{t}^\star) \leftarrow \underline{S}(x, y, \mathbf{t_a}).$
6. $\underline{S}(x, y, \mathbf{t}^{\star\star}) \leftarrow \underline{S}(x, y, \mathbf{t}^\star),\ not\ \underline{S}(x, y, \mathbf{f_a}).$

   } (Similarly for R, T and W)

7. $\leftarrow \underline{W}(x, y, z, \mathbf{t_a}), \underline{W}(x, y, z, \mathbf{f_a}).$

The rules in 2. establish how to repair the database wrt the first IC: by making $S(x, y)$ false or $R(x, y)$ true. Conditions of the form $x \neq null$ in the bodies are used to capture occurrences of null values in relevant attributes [10]. The rules in 3. specify the form of restoring consistency wrt 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 wrt 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 $\mathbf{f_a}$.[3] Relevant program constraints can be efficiently generated by using a *dependency graph* [12], which captures the relationship between predicates in the ICs (cf. Section 4).

The program has two stable models:[4] $\mathcal{M}_1 = \{\underline{S}(a, c, \mathbf{t}^\star), \underline{S}(b, c, \mathbf{t}^\star), R(b, c, \mathbf{t}^\star), \underline{T}(a, null, \mathbf{t}^\star), \underline{W}(null, b, c, \mathbf{t}^\star), \underline{W}(null, b, c, \mathbf{f_a}), R(a, c, \mathbf{t_a}), \underline{S}(a, c, \mathbf{t}^{\star\star}), \underline{S}(b, c, \mathbf{t}^{\star\star}), \underline{R}(b, c, \mathbf{t}^{\star\star}), R(a, c, \mathbf{t}^\star), \underline{R}(a, c, \mathbf{t}^{\star\star}), \underline{T}(a, null, \mathbf{t}^{\star\star})\}$, $\mathcal{M}_2 = \{\underline{S}(a, c, \mathbf{t}^\star), \underline{S}(b, c, \mathbf{t}^\star), \underline{R}(b, c, \mathbf{t}^\star), \underline{T}(a, null, \mathbf{t}^\star), \underline{W}(null, b, c, \mathbf{t}^\star), \underline{W}(null, b, c, \mathbf{f_a}), S(a, c, \mathbf{f_a}), \underline{S}(b, c, \mathbf{t}^{\star\star}), \underline{R}(b, c, \mathbf{t}^{\star\star}), \underline{T}(a, null, \mathbf{t}^{\star\star})\}$. Thus, consistency is recovered, according to $\mathcal{M}_1$ by inserting atom $R(a, c)$ and deleting atom $W(null, b, c)$; or, according to $\mathcal{M}_2$ by deleting atoms $\{S(a, c), W(null, b, c)\}$. Two repairs can be obtained by concentrating on the underlined atoms

[3] For the program in this example, given the logical relationship between the ICs, this phenomenon could happen only for predicate W, as analyzed in [12].

[4] In this paper, stable models are displayed without program facts.

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. □

As established in [4, 10], repair programs are a correct specification of database repairs wrt *RIC-acyclic* sets of UICs, RICs, and NNCs.

To compute consistent answers to a query $\mathcal{Q}$, the query is expressed (or simply given) as a logic program, e.g. as non-recursive Datalog program with weak negation and built-ins if $\mathcal{Q}$ is first-order [29]. In this program the positive literals of the form $P(\bar{s})$, with P an extensional predicate, are replaced by $P\_(\bar{s}, \mathbf{t}^{\star\star})$, and negative literals of the form $not\ P(\bar{s})$ by $not\ P\_(\bar{s}, \mathbf{t}^{\star\star})$. We obtain a query program $\Pi(\mathcal{Q})$, that is "run" together with the repair program $\Pi(D, IC)$. In this way, CQA becomes a form of *cautious* or *skeptical* reasoning under the stable models semantics. Notice that for a fixed set of ICs, the same repair program can be used with every instance (compatible with the schema) and with every query we want to answer consistently, so it can be generated once, and *ConsEx* will store it.

For the repair program in Example 2, the Datalog query $\mathcal{Q} : Ans(x) \leftarrow S(b, x)$, becomes the program $\Pi(\mathcal{Q})$ consisting of the rule $Ans(x) \leftarrow S\_(b, x, \mathbf{t}^{\star\star})$. The combined program $\Pi(D, IC, \mathcal{Q}) := \Pi(D, IC) \cup \Pi(\mathcal{Q})$ has two stable models, both of them containing the atom $Ans(c)$. Therefore, the consistent answer to $\mathcal{Q}$ is (c).

3 Magic Sets for Repair Programs

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*, with its own stable models, that can be used to answer the original query more efficiently. As shown in [12], the stable models of the magic program are relevant in the sense that they contain extensions for the predicates that are relevant to compute the query. Also, they are only partially computed, i.e. each of them can be extended to a stable model of the original program (ignoring the "magic" predicates introduced in the magic program). This happens because the magic program contains special auxiliary rules, the magic rules, that guide the course of query evaluation, avoiding unnecessary instantiation of rules and, as a consequence, 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 are expected to provide the same answers to the original query as the models of the program used as input to MS.

The magic sets techniques for logic programs with stable model semantics introduced in [21], for the non-disjunctive case but possibly unstratified negation, and in [24] (improved in [18]), with disjunction but stratified negation, are sound and complete, i.e. they compute all and only correct answers for the query. In [26] 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 programs constraints is to discard models of the rest of the program that make true the existential closure of $C(\bar{x})$.

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 could be directly applied to optimize our repair programs. However, as shown in [12] (cf. also [13] for details),

the following sound and complete MS methodology can be applied to repair programs (with program constraints): First, the program constraints are removed from the repair program. Next, a combination of the MS techniques in [18, 21] is applied to the resulting program. The disjunction is handled as in [18], and negation as in [21]. 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 [18, 21].[5] 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.

The MS techniques currently implemented in *DLV* cannot be applied to disjunctive programs with program constraints. On the other side, 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 (with program constraints) is not straightforward. *ConsEx*, that uses *DLV* for evaluation of logic programs, solves this problems as follows: First, *ConsEx* produces a magic program for the combination of the query and repair programs (as briefly mentioned above) without considering the program constraints. Next, the original program constraints are added to the magic program. Finally, this expanded magic program is given to *DLV* for evaluation, as any other logic program. This is the MS methodology implemented in the *ConsEx* system, which is correct for repair programs. An example below shows this process in detail.

The MS technique sequentially performs three well defined steps: *adornment, generation and modification*, which will be illustrated using Example 2 with the query program $Ans(x) \leftarrow S\_(b, x, \mathbf{t}^{\star\star})$.

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. First $\Pi(\mathcal{Q}) : Ans(x) \leftarrow S\_(b, x, \mathbf{t}^{\star\star})$ 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. Annotation constants are always bound.

The adorned predicate $S\_^{bfb}$ is used to propagate bindings (adornments) onto the rules defining predicate S, i.e. rules in 2., 5., and 6. 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. Extensional (base) predicates, e.g. S appearing as $S(x, y)$ in the first adorned rule, only bind variables and do not receive any annotation. Moreover, the adorned predicate $S\_^{bfb}$ propagates adornments over the disjunctive rules in 2. The adornments are propagated over the literals in the body of the rule, and to the head literal $R\_(x, y, \mathbf{t_a})$. Therefore, this rule becomes:[6] $S\_^{bfb}(x, y, \mathbf{f_a}) \vee R\_^{bfb}(x, y, \mathbf{t_a}), \leftarrow S\_^{bfb}(x, y, \mathbf{t}^{\star}), R\_^{bfb}(x, y, \mathbf{f_a})$. Now, the new adorned predicate $R\_^{bfb}$ also has to be processed, producing adornments on rules defining predicate R. The output of this step is an *adorned program* that contains only adorned rules.

[5] Personal communication from Wolfgang Faber. Actually, this combination is the MS technique implemented in *DLV*. Correctness is guaranteed for disjunctive programs with unstratified negation appearing in even cycles, which is what we need.

[6] For simplification purposes, conditions of the form $x \neq null$ are omitted from the disjunctive rules.

The iterative process of passing bindings is called *sideways information passing strategies* (SIPS) [5]. There may be different SIPS strategies, but any SIP strategy has to ensure that all of the body and head atoms are processed. We follow the strategy adopted in [18], which is implemented in *DLV*. According to it, only extensional predicates bind new variables, i.e. variables that do not carry a binding already. As an illustration, suppose we have the adorned predicate P^{fbf} and the rule $P(x, y, z) \vee T(x, y) \leftarrow R(z), M(x, z)$, where R is a extensional predicate. The adorned rule is $P^{fbf}(x, y, z) \vee T^{fb}(x, y) \leftarrow R(z), M^{fb}(x, z)$. Notice that variable z is free according to the adorned predicate P^{fbf}. However, the extensional atom $R(z)$ binds this variable, and propagates this binding to $M(x, z)$, where z becomes $bound$, producing the adorned predicate M^{fb}.

The next step is the *generation of magic rules*; those that will direct the computation of the stable models of the rewritten program obtained in the previous step. For each adorned atom P^A in the body of an adorned non-disjunctive rule, a magic rule is generated as follows: (a) The head of the magic rule becomes the magic version of P^A, i.e. $magic\_P^A$, from which all the variables labelled with f in A are deleted. (b) The literals in the body of the magic rule become the magic version of the adorned rule head, followed by the literals (if any) that produced bindings on atom P^A. For example, for the adorned literal $S^{bfb}(x, y, \mathbf{t_a})$ 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 $magic\_S^{bfb}(x, \mathbf{t_a}) \leftarrow magic\_S^{bfb}(x, \mathbf{t}^\star)$. 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, for the rule $S^{bfb}(x, y, \mathbf{f_a}) \vee R^{bfb}(x, y, \mathbf{t_a}) \leftarrow S^{bfb}(x, y, \mathbf{t}^\star), R^{bfb}(x, y, \mathbf{f_a})$, we have two non-disjunctive rules: (a) $S^{bfb}(x, y, \mathbf{f_a}) \leftarrow R^{bfb}(x, y, \mathbf{t_a}), S^{bfb}(x, y, \mathbf{t}^\star), R^{bfb}(x, y, \mathbf{f_a})$; and (b) $R^{bfb}(x, y, \mathbf{t_a}) \leftarrow S^{bfb}(x, y, \mathbf{f_a}), S^{bfb}(x, y, \mathbf{t}^\star), R^{bfb}(x, y, \mathbf{f_a})$. There are three magic rules for rule (a): $magic\_R^{bfb}(x, \mathbf{t_a}) \leftarrow magic\_S^{bfb}(x, \mathbf{f_a})$; $magic\_S^{bfb}(x, \mathbf{t}^\star) \leftarrow magic\_S^{bfb}(x, \mathbf{f_a})$; 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. This corresponds to the magic version of the Ans predicate from the adorned query rule, e.g. for rule $Ans^f(x) \leftarrow S^{bfb}(x, y, \mathbf{t}^{\star\star})$, the magic seed atom is $magic\_Ans^f$.

The last phase is the *modification step*, where magic atoms constructed in the generation stage are included in the body of adorned rules. Thus, 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 rule $S^{bfb}(x, y, \mathbf{f_a}) \vee R^{bfb}(x, y, \mathbf{t_a}) \leftarrow S^{bfb}(x, y, \mathbf{t}^\star), R^{bfb}(x, y, \mathbf{f_a})$, 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 $S^{bfb}(x, y, \mathbf{f_a}) \vee R^{bfb}(x, y, \mathbf{t_a}) \leftarrow magic\_S^{bfb}(x, \mathbf{f_a}), magic\_R^{bfb}(x, \mathbf{t_a}), S^{bfb}(x, y, \mathbf{t}^\star), R^{bfb}(x, y, \mathbf{f_a})$. From the modified rules the rest of the adornments are now deleted. Thus, the previous modified rule becomes $S(x, y, \mathbf{f_a}) \vee R(x, y, \mathbf{t_a}) \leftarrow magic\_S^{bfb}(x, \mathbf{f_a}), magic\_R^{bfb}(x, \mathbf{t_a}), S(x, y, \mathbf{t}^\star), R(x, y, \mathbf{f_a})$.

The final, rewritten, magic program consists of the magic and modified rules, the *magic seed atom*, and the facts of the original program. In our case, it also contains the set of original program constraints that were not touched during the application of MS. Since in the MS program only magic atoms have adornments, the program constraints can be added as they come to the program. The program $\mathcal{MS}(\Pi)$ below is the magic

program for the program Π consisting of the query program $Ans(x) \leftarrow \underline{S}(b, x, \mathbf{t}^{\star\star})$ plus the repair program in Example 2.

Program $\mathcal{MS}(\Pi)$**:** $magic\_Ans^f$.

$magic\_\underline{S}^{bfb}(b, \mathbf{t}^{\star\star}) \leftarrow magic\_Ans^f$.
$magic\_\underline{S}^{bfb}(x, \mathbf{t_a}) \leftarrow magic\_\underline{S}^{bfb}(x, \mathbf{t}^{\star})$.
$magic\_\underline{S}^{bfb}(x, \mathbf{t}^{\star}) \leftarrow magic\_\underline{S}^{bfb}(x, \mathbf{t}^{\star\star})$.
$magic\_\underline{S}^{bfb}(x, \mathbf{f_a}) \leftarrow magic\_\underline{S}^{bfb}(x, \mathbf{t}^{\star\star})$.
$magic\_\underline{R}^{bfb}(x, \mathbf{t_a}) \leftarrow magic\_\underline{S}^{bfb}(x, \mathbf{f_a})$.
$magic\_\underline{S}^{bfb}(x, \mathbf{t}^{\star}) \leftarrow magic\_\underline{S}^{bfb}(x, \mathbf{f_a})$.
$magic\_\underline{R}^{bfb}(x, \mathbf{f_a}) \leftarrow magic\_\underline{S}^{bfb}(x, \mathbf{f_a})$.
$magic\_\underline{S}^{bfb}(x, \mathbf{f_a}) \leftarrow magic\_\underline{R}^{bfb}(x, \mathbf{t_a})$.
$magic\_\underline{S}^{bfb}(x, \mathbf{t}^{\star}) \leftarrow magic\_\underline{R}^{bfb}(x, \mathbf{t_a})$.
$magic\_\underline{R}^{bfb}(x, \mathbf{f_a}) \leftarrow magic\_\underline{R}^{bfb}(x, \mathbf{t_a})$.
$magic\_\underline{R}^{bfb}(x, \mathbf{t_a}) \leftarrow magic\_\underline{R}^{bfb}(x, \mathbf{t}^{\star})$.
$magic\_\underline{R}^{bfb}(x, \mathbf{t}^{\star}) \leftarrow magic\_\underline{R}^{bfb}(x, \mathbf{t}^{\star\star})$.
$magic\_\underline{R}^{bfb}(x, \mathbf{f_a}) \leftarrow magic\_\underline{R}^{bfb}(x, \mathbf{t}^{\star\star})$.
$Ans(x) \leftarrow magic\_Ans^f, \underline{S}(b, x, \mathbf{t}^{\star\star})$.

$\underline{S}(x, y, \mathbf{f_a}) \vee \underline{R}(x, y, \mathbf{t_a}) \leftarrow magic\_\underline{S}^{bfb}(x, \mathbf{f_a}), magic\_\underline{R}^{bfb}(x, \mathbf{t_a}), \underline{S}(x, y, \mathbf{t}^{\star}), \underline{R}(x, y, \mathbf{f_a})$.
$\underline{S}(x, y, \mathbf{f_a}) \vee \underline{R}(x, y, \mathbf{t_a}) \leftarrow magic\_\underline{S}^{bfb}(x, \mathbf{f_a}), magic\_\underline{R}^{bfb}(x, \mathbf{t_a}), \underline{S}(x, y, \mathbf{t}^{\star}),\ not\ R(x, y)$.
$\underline{S}(x, y, \mathbf{t}^{\star}) \leftarrow magic\_\underline{S}^{bfb}(x, \mathbf{t}^{\star}), \underline{S}(x, y, \mathbf{t_a})$. $\underline{S}(x, y, \mathbf{t}^{\star}) \leftarrow magic\_\underline{S}^{bfb}(x, \mathbf{t}^{\star}), S(x, y)$.
$\underline{R}(x, y, \mathbf{t}^{\star}) \leftarrow magic\_\underline{R}^{bfb}(x, \mathbf{t}^{\star}), \underline{R}(x, y, \mathbf{t_a})$. $\underline{R}(x, y, \mathbf{t}^{\star}) \leftarrow magic\_\underline{R}^{bfb}(x, \mathbf{t}^{\star}), R(x, y)$.
$\underline{S}(x, y, \mathbf{t}^{\star\star}) \leftarrow magic\_\underline{S}^{bfb}(x, \mathbf{t}^{\star\star}), \underline{S}(x, y, \mathbf{t}^{\star}),\ not\ \underline{S}(x, y, \mathbf{f_a})$.
$\underline{R}(x, y, \mathbf{t}^{\star\star}) \leftarrow magic\_\underline{R}^{bfb}(x, \mathbf{t}^{\star\star}), \underline{R}(x, y, \mathbf{t}^{\star}),\ not\ \underline{R}(x, y, \mathbf{f_a})$.
$\leftarrow \underline{W}(x, y, z, \mathbf{t_a}), \underline{W}(x, y, z, \mathbf{f_a})$.

Notice that $\mathcal{MS}(\Pi)$ 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 will be trivially satisfied. Program $\mathcal{MS}(\Pi)$ (with the same facts of the original repair program) has only one stable model: $\mathcal{M} = \{\underline{S}(b, c, \mathbf{t}^{\star}), \underline{S}(b, c, \mathbf{t}^{\star\star}),\ Ans(c)\}$ (displayed here without the magic atoms), which indicates through its Ans predicate that (c) 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 $\Pi(D, IC, \mathcal{Q})$. More precisely, except for the magic atoms, model $\mathcal{M}$ is contained in every model of the original repair program $\Pi(D, IC, \mathcal{Q})$ (cf. Section 2). [7]

4 System Description

In Figure 1, 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 2 (a) the connection screen; and in Figure 2 (b), the main menu, obtained after connecting to the database.

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*, first-order queries 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 and built-ins, which includes 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.

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

[7] In [13] it has been shown that the magic program, and the original repair program are *query equivalent* under both brave and cautious reasoning.

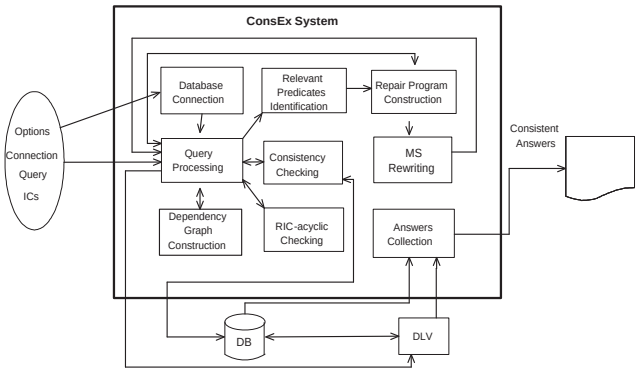

Fig. 1. *ConsEx* Architecture

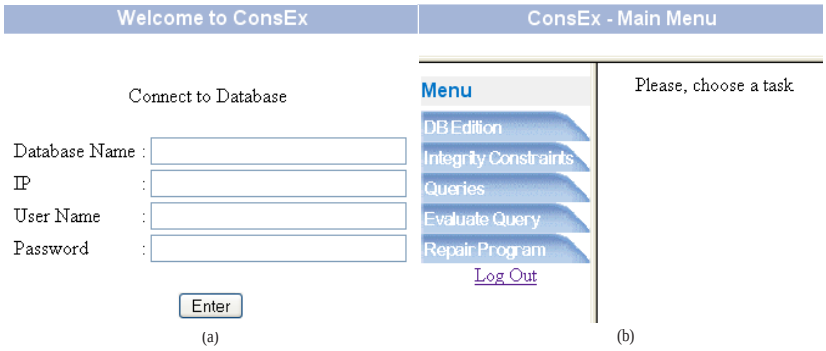

Fig. 2. *ConsEx*: Database Connection and Main Menu

by first translating queries into equivalent Datalog programs, and then by adding the annotation $\mathbf{t}^{\star\star}$ to the program rules as for the *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* [12], which is generated by the *Dependency Graph Construction* module. We can use our running example to describe this feature and other system's components.

The dependency graph $\mathcal{G}(IC)$ for the ICs in Example 2 contains as nodes the predicates S, R, T, W, and the edges $(S, R), (T, W)$. 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 (cf. [12] for more details).

Next, *ConsEx* checks if the database is consistent wrt 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)`, 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 2 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.

The *RIC-acyclic Checking* module uses the dependency graph to check if set of ICs is *RIC*-acyclic. If it is, the generation of programs is avoided, and a warning message is sent to the user. Otherwise, the *Repair Program Construction* module generates the repair program, which 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 will become the facts for predicate P in the program. As a result, when the program is evaluated by *DLV*, the database facts will be imported directly into the reasoning system. These data import sentences are required at this stage only if *ConsEx* will run the original repair program without any magic sets optimization, which is an option given by the system.

The *MS Rewriting* module generates the magic version of a 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 (they have 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 $\mathcal{MS}(\Pi)$ for the query $Ans(x) \leftarrow S(b,x)$ shown in Section 3, the following rules contain database atoms: (a) $S\_(x,y,\mathbf{t}^\star) \leftarrow magic\_S\_^{bfb}(x,\mathbf{t}^\star), \ S(x,y)$; and (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.

5 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 when using magic sets instead of the direct evaluation of the repair programs. 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 of ICs composed of two primary key constraints, and three RICs. In order to analyze scalability of CQA trough logic programs, we considered two databases instances D_1, and D_2, with 3200 and 6400 stored tuples, resp. The number N of inconsistent tuples, i.e. participating in an IC violation varied between 20 and 400.[8]

Here, we report the execution time for two conjunctive queries, in both instances. The first query is of the form, $Ans(\bar{x}) \leftarrow P(\bar{y}), R(\bar{z})$, with $\bar{x} \subseteq \bar{y} \cup \bar{z}$, with free variables (an open query), joins ($\bar{y} \cap \bar{z} \neq \emptyset$), and no constants. The second query contains joins and is also partially-ground, like the query used in Section 3. Both queries fall in the class of *Tree*-queries for which CQA is tractable under key constraints [22]. 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 [22]. Even more, it is not clear that the tractability result in [22] carries over to the queries and ICs used in our experiments.

In the charts, *R&Q* indicates the straightforward evaluation of the repair program combined with the query program, whereas its magic sets optimization is indicated with *MS*. Figure 3 shows the running time for the first query in the two instances. We can see that MS is faster than the straightforward evaluation. For $N = 200$ (in both database instances), the MS methodology returns answers in less than ten seconds, while the straightforward evaluation returns answers after one minute. Moreover, the execution time of the MS methodology is almost invariant wrt percentage of inconsistency. Despite the absence of constants in the query, MS still 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 4 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 this case, MS has an even better performance 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 (wrt the original program).

Furthermore, MS shows an excellent scalability. For instance, MS computes answers to queries from database instances D_1 and D_2 in less than ten seconds, even with a database D_2 that contains twice as many tuples as D_1.

[8] 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

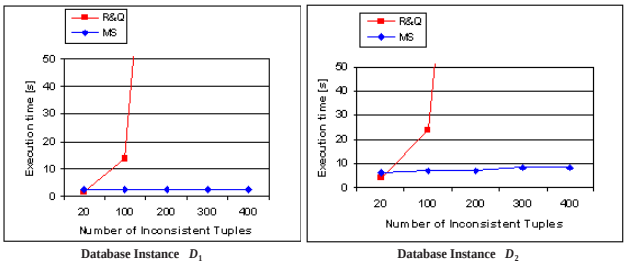

Fig. 3. Running Time for the Conjunctive Query with Free Variables

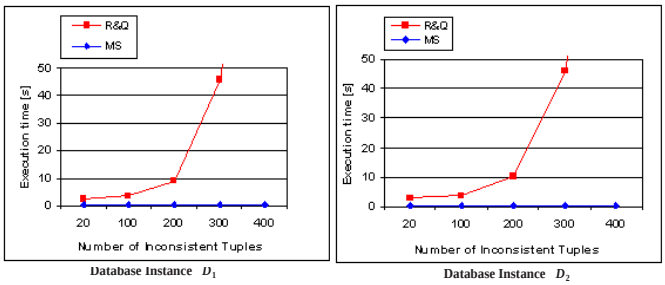

Fig. 4. Running Time for the Partially-Ground Conjunctive Query with Free Variables

6 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 that specify both the repairs and the query. In order to make query answering more efficient in practice, *ConsEx* implements sound and complete magic set techniques for disjunctive repair programs with program constraints [12]. 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. 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 methodology for CQA based on repair logic programs is general enough to cover all the queries and ICs found in database practice (and more). On the other side, we know that CQA has a high intrinsic data complexity [16, 7]. 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.

Implementations of other systems for CQA have been reported before. The *Queca* system [14] implements the query rewriting methodology presented in [1], and can be used with universal ICs with at most two database atoms (plus built-ins) and projection-

free conjunctive queries. The system *Hippo* [17] 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* [22] 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.

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 [10]. However, when queries are answered in *ConsEx*, 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 the IC satisfaction and repair semantics used in *ConsEx* is proposed in [8]. Its implementation in *ConsEx* is left for 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 [13].

Acknowledgements: Research supported by an NSERC Discovery Grant, and the University of Bio-Bio (UBB-Chile) (Grant DIUBB 076215 4/R). L. Bertossi is Faculty Fellow of IBM Center for Advanced Studies (Toronto Lab.). 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] Arenas, M., Bertossi, L. and Chomicki, J. Consistent Query Answers in Inconsistent Databases. In *Proc. ACM Symposium on Principles of Database Systems (PODS 99)*, ACM Press, 1999, pp. 68–79.

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

[3] Barcelo, P. and Bertossi, L. Logic Programs for Querying Inconsistent Databases. In *Proc. 5th International Symposium on Practical Aspects of Declarative Languages (PADL 03)*, Springer LNCS 2562, 2003, pp. 208–222.

[4] Barcelo, P., Bertossi, L. and Bravo, L. Characterizing and Computing Semantically Correct Answers from Databases with Annotated Logic and Answer Sets. In *Semantics of Databases*, Springer LNCS 2582, 2003, pp. 1-27.

[5] Bancilhon, F., Maier, D., Sagiv, Y. and Ullman, J. Magic Sets and Other Strange Ways to Implement Logic Programs (extended abstract). In *Proc. 5th ACM Symposium on Principles of Database Systems (PODS 86)*, ACM Press, 1986, pp. 1–15.

[6] Bertossi, L. and Chomicki, J. Query Answering in Inconsistent Databases. In *Logics for Emerging Applications of Databases*. Springer, 2003, pp. 43-83.

[7] Bertossi, L. Consistent Query Answering in Databases. *ACM Sigmod Record*, June 2006, 35(2):68-76.

[8] Bravo, L. Handling Inconsistency in Databases and Data Integration Systems. PhD. Thesis, Carleton University, Department of Computer Science, 2007, http://homepages.inf.ed.ac.uk/lbravo/Publications.htm

[9] Bravo, L. and Bertossi, L. Consistent Query Answering under Inclusion Dependencies. In *14th Annual IBM Centers for Advanced Studies Conference (CASCON 2004)*, 2004, pp. 202–216.

[10] Bravo, L. and Bertossi, L. Semantically Correct Query Answers in the Presence of Null Values. In *Current Trends in Database Technology - EDBT 2006*, Springer LNCS 4254, 2006, pp. 33-47.

[11] Cali, A., Lembo, D. and Rosati, R. On the Decidability and Complexity of Query Answering over Inconsistent and Incomplete Databases. In *Proc. ACM Symposium on Principles of Database Systems (PODS 03)*, ACM Press, 2003, pp. 260–271.

[12] Caniupan, M. and Bertossi, L. Optimizing Repair Programs for Consistent Query Answering. In *Proc. International Conference of the Chilean Computer Science Society (SCCC 05)*, IEEE Computer Society Press, 2005, pp. 3–12.

[13] Caniupan, M. Optimizing and Implementing Repair Programs for Consistent Query Answering in Databases. PhD. Thesis, Carleton University, Department of Computer Science, 2007, http://www.face.ubiobio.cl/∼mcaniupa/publications.htm

[14] Celle, A. and Bertossi, L. Querying Inconsistent Databases: Algorithms and Implementation. In *Computational Logic - CL 2000*, Springer LNCS 1861, 2000, pp. 942-956.

[15] Ceri, S., Gottlob, G. and Tanca, L. *Logic Programming and Databases*. Springer-Verlag, 1990.

[16] Chomicki, J. and Marcinkowski, J. On the Computational Complexity of Minimal-Change Integrity Maintenance in Relational Databases. In *Integrity Tolerance*, Springer LNCS 3300, 2004, pp. 119-150.

[17] Chomicki, J., Marcinkowski, J. and Staworko, S. Computing Consistent Query Answers using Conflict Hypergraphs. In *Proc. 13th ACM International Conference on Information and Knowledge Management (CIKM 04)*, ACM Press, 2004, pp. 417–426.

[18] Cumbo, C., Faber, W., Greco, G. and Leone, N. Enhancing the Magic-Set Method for Disjunctive Datalog Programs. In *Proc. 20th International Conference on Logic Programming (ICLP 04)*, Springer LNCS 3132, 2004, pp. 371–385.

[19] Dantsin, E., Eiter, T., Gottlob, G. and Voronkov, A. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 2001, 33(3):374-425.

[20] Eiter, T., Fink, M., Greco, G. and Lembo, D. Efficient Evaluation of Logic Programs for Querying Data Integration Systems. In *Proc. 19th International Conference on Logic Programming (ICLP 03)*, Springer LNCS 2916, 2003, pp. 163–177.

[21] Faber, W., Greco, G. and Leone, N. Magic Sets and their Application to Data Integration. *Journal of Computer and System Sciences*, 2007, 73(4):584–609.

[22] Fuxman, A., Fazli, E. and Miller, R.J. ConQuer: Efficient Management of Inconsistent Databases. *Proc. ACM International Conference on Management of Data (SIGMOD 05)*, ACM Press, 2005, pp. 155-166.

[23] Gelfond, M. and Lifschitz, V. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 1991, 9:365–385.

[24] Greco, S. Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. In *IEEE Transac. on Knowledge and Data Eng.*, 2003, 15(2):368–385.

[25] Greco, G., Greco, S. and Zumpano, E. A Logical Framework for Querying and Repairing Inconsistent Databases. *IEEE Transactions on Knowledge and Data Eng.*, 2003, 15(6):1389–1408.

[26] Greco, G., Greco, S., Trubtsyna, I. and Zumpano, E. Optimization of Bound Disjunctive Queries with Constraints. *Theory and Practice of Logic Programming*, 2005, 5(6):713–745.

[27] Lembo, D., Rosati, R. and Ruzzi, M. On the First-Order Reducibility of Unions of Conjunctive Queries over Inconsistent Databases. In *Current Trends in Database Technology - EDBT 2006*, Springer LNCS 4254, 2006, pp. 358-374.

[28] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 2006, 7(3):499–562.

[29] Lloyd, J.W. *Foundations of Logic Programming*. Second ed., Springer-Verlag, 1987.