Answer Sets Programs for Querying Inconsistent Databases: The Consistency Extractor System

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*— The *Consistency Extractor System* (*ConsEx*) is a general implementation of *consistent query answering*, i.e. the computation of consistent answers to queries posed to databases that may fail to satisfy certain desirable integrity constraints. The system is based on the specification of the repairs of the original instance as the stable models of disjunctive logic programs (aka. answer set programs). This paper describes the architecture and functionalities of the system, some of its theoretical foundations, the optimization of logic programs, and the interaction with DBMSs and *DLV*, as evaluator of logic programs. We also report on experimental results.**

I. Introduction

Integrity constraints (ICs) capture the semantics of a database instance, which is expected to satisfy them. Unfortunately, this is not always the case and we have to live with an inconsistent database [2]. As introduced in [2], *consistent query answering* (CQA) is the problem of characterizing and retrieving consistent answers to queries posed to inconsistent databases. Intuitively, an answer $\bar{a}$ to a query $\mathcal{Q}$ in a relational database instance D is *consistent* wrt a set IC of ICs if $\bar{a}$ is an answer (in the usual sense) to $\mathcal{Q}$ in every *repair* of D. Here, 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.

Disjunctive logic programs with stable model semantics [7] (aka. *answer set programs*) can be used to specify database repairs. There is a one-to-one correspondence between the stable models of the *repair program* and the database repairs, and the programs can be used to compute consistent answers. In *ConsEx* we use, implement, and optimize the repair programs introduced in [3], providing the most general methodology for CQA through logic programs. The repair semantics and the logic program semantics take into consideration possible occurrences of null values as they are used and found in real DBMSs that follow the SQL standard. Moreover, null values are also used to restore consistency wrt referential ICs.

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; thus, it can handle first-order queries in particular. Consistent answers to queries can be computed by evaluating queries against the repair programs, e.g. using the *DLV* system, that implements the cautions (or skeptical) stable model semantics of disjunctive logic programs [8].

Answer set programs capture the rather high intrinsic data complexity of CQA [2]. However, for several classes of ICs and queries, CQA has a lower complexity than the one of query evaluation against general answer set programs. Furthermore, the straightforward and naive evaluation of these programs may not be very efficient. Instead, repair programs may be evaluated applying the so-called *magic sets* (MS) techniques that transform the combination of the query and the repair program into a new program that can be evaluated more efficiently. These MS techniques have been developed for logic programs with stable model semantics [6]. The rewritten program contains a subset of the original rules in the program, those that are relevant to evaluate the query. *ConsEx* implements the MS methodology proposed in [4] for disjunctive repair programs with program constraints (cf. Section II). In *ConsEx* CQA improves considerably in comparison with the direct evaluation, and also shows quite a good performance and scalability. The experimental results are quite encouraging wrt the applicability of *ConsEx* in real database practice. We present here the architecture and main features of *ConsEx*, and also experimental results.

II. Database Repairs and Repair Programs

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 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* (UIC) is sentence of the form [3]: $\forall \bar{x}(\bigwedge_{i=1}^{m} P_i(\bar{x}_i) \rightarrow \bigvee_{j=1}^{n} Q_j(\bar{y}_j) \vee \varphi)$, where $P_i, Q_j \in \mathcal{R}$, $\bar{x} = \bigcup_{i=1}^{m} \bar{x}_i$, $\bar{y}_j \subseteq \bar{x}$, $m \geq 1$, and 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: $\bar{\forall} \bar{x}(P(\bar{x}) \wedge IsNull(x_i) \rightarrow \mathbf{false})$, where $x_i \in \bar{x}$ is in the position of the attribute that cannot take null values.

CQA implemented in *ConsEx* works for *RIC-acyclic* sets of UIC, RICs, and NNCs. When considering *RIC-acyclic* sets of ICs, there is a one-to-one correspondence between the stable models of the repair program and the database repairs [3]. Intuitively, a set of ICs is RIC-acyclic if there are no cycles involving RICs (cf. [4], [3] for details). 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.

When null values are introduced to restore consistency, it becomes necessary to modify the repair semantics introduced in [1], in order to give priority to null values over arbitrary domain constants when restoring consistency wrt RICs. This is achieved by modifying accordingly the notion of minimality as shown in the following example (cf. [3] for details).

Example 1: $D = \{P(a, null), P(b, c), R(a, b)\}$ 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 latter programs 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 $\mathbf{t_a}$ ($\mathbf{f_a}$) means that the atom is advised to made true (false) (i.e. inserted into (deleted) the database). 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. 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. [3] for the general form of the repair programs).

Example 2: Consider the database schema $\Sigma = \{S(x, y),\ R(x, y),\ T(x, y),\ W(x, y, z)\}$, the instance $D = \{S(a, c), S(b, c),\ R(b,\ c), T(a, null), W(null, b, c)\}$, and $IC = \{\forall xy(S(x, y) \rightarrow R(x, y)),\ \forall xy(T(x, y) \rightarrow \exists z W(x, y, z)),\ \forall xyz(W(x, y, z) \wedge IsNull(x) \rightarrow \mathbf{false})\}$.

[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.

The repair program $\Pi(D,\ IC)$ contains the following rules:[2]

1. $S(a, c).\quad S(b, c).\quad R(b, c).\quad T(a, null).\quad W(null, b, c).$
2. $\underline{S}(x, y, \mathbf{f_a}) \vee \underline{R}(x, y, \mathbf{t_a}) \leftarrow \underline{S}(x, y, \mathbf{t}^\star), R(x, y, \mathbf{f_a}).$
   $\underline{S}(x, y, \mathbf{f_a}) \vee \underline{R}(x, y, \mathbf{t_a}) \leftarrow \underline{S}(x, y, \mathbf{t}^\star),\ not\ R(x, y).$
3. $\underline{T}(x, y, \mathbf{f_a}) \vee \underline{W}(x, y, null, \mathbf{t_a}) \leftarrow \underline{T}(x, y, \mathbf{t}^\star),\ not\ aux(x, y).$
   $aux(x, y) \leftarrow \underline{W}(x, y, z, \mathbf{t}^\star),\ not\ \underline{W}(x, y, z, \mathbf{f_a}).$
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. $\underline{S}(x, y, \mathbf{t}^{\star\star}) \leftarrow \underline{S}(x, y, \mathbf{t}^\star),\ not\ \underline{S}(x, y, \mathbf{f_a}).$

   } (for R, T, 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. The rules in 3. specify the form of restoring consistency wrt the RIC: by deleting $T(x, y)$ or inserting $W(x, y, null)$. Rule 4. indicates how to restore consistency wrt the NNC: by eliminating $W(x, y, z)$. 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]

The program has two stable models:[4] $\mathcal{M}_1 = \{\ \underline{S}(a, c, \mathbf{t}^\star),\ \underline{S}(b, c, \mathbf{t}^\star),\ \underline{R}(b, c, \mathbf{t}^\star),\ \underline{T}(a,\ null, \mathbf{t}^\star),\ \underline{W}(null, b, c, \mathbf{t}^\star),\ \underline{W}(null, b, c, \mathbf{f_a}),\ R(a, c, \mathbf{t_a}),\ \underline{S}(a, c, \mathbf{t}^{\star\star}),\ \underline{S}(b, c, \mathbf{t}^{\star\star}),\ \underline{R}(b, c, \mathbf{t}^{\star\star}),\ R(a, c, \mathbf{t}^\star),\ \underline{R}(a, c, \mathbf{t}^{\star\star}),\ \underline{T}(a, null, \mathbf{t}^{\star\star})\},\ \mathcal{M}_2 = \{\ \underline{S}(a, c, \mathbf{t}^\star),\ \underline{S}(b, c, \mathbf{t}^\star),\ \underline{R}(b, c, \mathbf{t}^\star),\ \underline{T}(a,\ null, \mathbf{t}^\star),\ \underline{W}(null, b, c, \mathbf{t}^\star),\ \underline{W}(null, b, c, \mathbf{f_a}),\ \underline{S}(a, c, \mathbf{f_a}),\ \underline{S}(b, c, \mathbf{t}^{\star\star}),\ \underline{R}(b, c, \mathbf{t}^{\star\star}),\ \underline{T}(a, null, \mathbf{t}^{\star\star})\}$. 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 $\{S(a, c),\ W(null, b, c)\}$. If we concentrate on the underlined atoms in the stable models we obtain the repairs: $\{S(a, c),\ S(b, c), R(b, c),\ R(a, c),\ T(a, null)\}$ and $\{S(b, c), R(b, c),\ T(a, null)\}$, as expected. □

To compute consistent answers to a query $\mathcal{Q}$, the query is expressed as a logic program $\Pi(\mathcal{Q})$, where the positive literals of the form $P(\bar{s})$, with P an extensional predicate, are replaced by $\underline{P}(\bar{s}, \mathbf{t}^{\star\star})$, and negative literals of the form $not\ P(\bar{s})$ by $not\ \underline{P}(\bar{s}, \mathbf{t}^{\star\star})$. The query program is "run" together with program $\Pi(D, IC)$. In this way, CQA is translated into *cautious* or *skeptical* reasoning under the stable models semantics. For the repair program in Example 2, the Datalog query $\mathcal{Q}$: $Ans(x) \leftarrow S(b, x)$, becomes the program $\Pi(\mathcal{Q})$: $Ans(x) \leftarrow \underline{S}(b, x, \mathbf{t}^{\star\star})$. 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).

III. Architecture of the System

Figure 1 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.

[2]For simplification, we have omitted in the body of the rules in 2., and 3., conditions of the form $x \neq null$, which capture occurrences of null values in relevant attributes [3].

[3]For the program in this example, given the logical relationship between ICs, this phenomenon could happen only for predicate W, as analyzed in [4].

[4]The stable models are displayed without program facts.

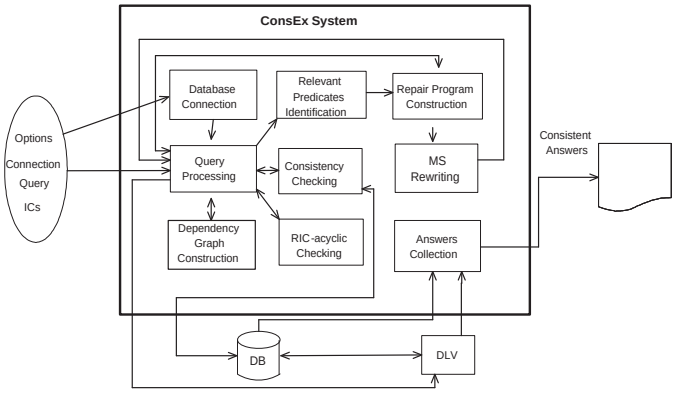

Fig. 1. ConsEx Architecture

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. In *ConsEx*, FO queries can be written as logic programs in *DLV* notation, or as queries in SQL. The former are non-recursive Datalog queries with weak negation and built-ins, which includes FO 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 checking the syntaxis of a query, the query program is generated.

For a given query, there might be ICs that are not related to the query. Moreover, their satisfaction or not by the given instance does not influence the consistent answers to the query. *ConsEx*, via the *Relevant Predicates Identification* module, analyzes the interaction between the predicates in the query and those in the ICs. This is done by appealing to a *dependency graph* $\mathcal{G}(IC)$ [4] that is generated by the *Dependency Graph Construction* module. For instance, the dependency graph for the ICs in Example 2 contains as nodes the predicates S, R, T, W, and the edges $(S, R), (T, W)$. 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 $\forall xy(S(x, y) \rightarrow R(x, y))$ (cf. [4] for more details).

Next, *ConsEx* checks (module *Consistency Checking*) if the database is consistent wrt the relevant ICs to the query. If this is the case, *ConsEx* evaluates the query directly on the original database instance, i.e. without computing repairs. For example, in Example 2 the database is inconsistent wrt $\forall xy(S(x, y) \rightarrow R(x, y))$. In consequence, in order to consistently answer the query, the repair program has to be generated.

The dependency graphs is also used to check if the set of ICs is *RIC*-acyclic. This is done by the *RIC-acyclic Checking* module. 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 them are generated by the system, and the database is not affected. The facts of the program are not imported from the database into *ConsEx*.

In *ConsEx* queries are evaluated efficiently by using the *magic sets* (MS) methodology presented in [4], that is an adaptation of the one in [6] (cf. [4] for details). That is, *ConsEx* transforms 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. This new *magic program*, with its own stable models, can be used to answer the original query more efficiently.

The *MS Rewriting* module generates the magic version of a program, which includes at the end appropriate database import sentences to retrieve tuples from the database, namely the tuples that are relevant to compute the consistent answers to the original query. For example, the program $\mathcal{MS}(\Pi)$ below is the magic version of the program Π consisting of the query program $Ans(x) \leftarrow S(b, x, \mathbf{t}^{\star\star})$ plus the repair program in Example 2.

Program $\mathcal{MS}(\Pi)$**:** $mg\_Ans^f.\ \ mg\_S^{bfb}(b, \mathbf{t}^{\star\star}) \leftarrow mg\_Ans^f.$
$mg\_S^{bfb}(x, \mathbf{t_a}) \leftarrow mg\_S^{bfb}(x, \mathbf{t}^{\star}).$
$mg\_S^{bfb}(x, \mathbf{t}^{\star}) \leftarrow mg\_S^{bfb}(x, \mathbf{t}^{\star\star}).$
$mg\_S^{bfb}(x, \mathbf{f_a}) \leftarrow mg\_S^{bfb}(x, \mathbf{t}^{\star\star}).$
$mg\_R^{bfb}(x, \mathbf{t_a}) \leftarrow mg\_S^{bfb}(x, \mathbf{f_a}).$
$mg\_S^{bfb}(x, \mathbf{t}^{\star}) \leftarrow mg\_S^{bfb}(x, \mathbf{f_a}).$
$mg\_R^{bfb}(x, \mathbf{f_a}) \leftarrow mg\_S^{bfb}(x, \mathbf{f_a}).$
$mg\_S^{bfb}(x, \mathbf{f_a}) \leftarrow mg\_R^{bfb}(x, \mathbf{t_a}).$
$mg\_S^{bfb}(x, \mathbf{t}^{\star}) \leftarrow mg\_R^{bfb}(x, \mathbf{t_a}).$
$mg\_R^{bfb}(x, \mathbf{f_a}) \leftarrow mg\_R^{bfb}(x, \mathbf{t_a}).$
$mg\_R^{bfb}(x, \mathbf{t_a}) \leftarrow mg\_R^{bfb}(x, \mathbf{t}^{\star}).$
$mg\_R^{bfb}(x, \mathbf{t}^{\star}) \leftarrow mg\_R^{bfb}(x, \mathbf{t}^{\star\star}).$
$mg\_R^{bfb}(x, \mathbf{f_a}) \leftarrow mg\_R^{bfb}(x, \mathbf{t}^{\star\star}).$
$Ans(x) \leftarrow mg\_Ans^f, S\_(b, x, \mathbf{t}^{\star\star}).$
$S\_(x, y, \mathbf{f_a}) \vee R\_(x, y, \mathbf{t_a}) \leftarrow mg\_S^{bfb}(x, \mathbf{f_a}), mg\_R^{bfb}(x, \mathbf{t_a}), S\_(x, y, \mathbf{t}^{\star}), R\_(x, y, \mathbf{f_a}).$
$S\_(x, y, \mathbf{f_a}) \vee R\_(x, y, \mathbf{t_a}) \leftarrow mg\_S^{bfb}(x, \mathbf{f_a}), mg\_R^{bfb}(x, \mathbf{t_a}), S\_(x, y, \mathbf{t}^{\star}),\ not\ R(x, y).$
$S\_(x, y, \mathbf{t}^{\star}) \leftarrow mg\_S^{bfb}(x, \mathbf{t}^{\star}), S\_(x, y, \mathbf{t_a}).$
$S\_(x, y, \mathbf{t}^{\star}) \leftarrow mg\_S^{bfb}(x, \mathbf{t}^{\star}), S(x, y).$
$R\_(x, y, \mathbf{t}^{\star}) \leftarrow mg\_R^{bfb}(x, \mathbf{t}^{\star}), R\_(x, y, \mathbf{t_a}).$
$R\_(x, y, \mathbf{t}^{\star}) \leftarrow mg\_R^{bfb}(x, \mathbf{t}^{\star}), R(x, y).$
$S\_(x, y, \mathbf{t}^{\star\star}) \leftarrow mg\_S^{bfb}(x, \mathbf{t}^{\star\star}), S\_(x, y, \mathbf{t}^{\star}),\ not\ S\_(x, y, \mathbf{f_a}).$
$R\_(x, y, \mathbf{t}^{\star\star}) \leftarrow mg\_R^{bfb}(x, \mathbf{t}^{\star\star}), R\_(x, y, \mathbf{t}^{\star}),\ not\ R\_(x, y, \mathbf{f_a}).$
$\leftarrow W\_(x, y, z, \mathbf{t_a}), W\_(x, y, z, \mathbf{f_a}).$

Notice that since $\mathcal{MS}(\Pi)$ contains rules related to predicates S, R only, the program constraint will be trivially satisfied. The import sentences are generated by inspection of the magic program (cf. [5] for details), identifying first in the rule bodies the extensional database atoms (they have no annotations 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. The generated import sentences for predicate S is $\#import(dbName,\ dbUser,\ dbPass,$ "`SELECT * FROM S WHERE ID = 'b'`", `S`$)$. This will retrieve into *DLV* only the corresponding subsets of the relations in the database. A similar import sentence is generated for relation

R.

The MS 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. Program $\mathcal{MS}(\Pi)$ 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.

IV. Experimental Evaluation

We 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. We use the version of *DLV* for Linux released on Jan 12, 2006.

The database schema was composed by eight relations, the set of ICs was composed of two primary key constraints, and three RICs. The databases instance D was composed of 6400 stored tuples. The number N of inconsistent tuples, i.e. those participating in an IC violation, varied between 20 and 400.[5]

Here, we report the execution time for two conjunctive queries. 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 2 (a) shows the running time for the first query, which 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. We can see that MS is faster than the straightforward evaluation. For $N = 200$, 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 2 (b) shows the execution time for a partially-ground query (e.g. $Ans(x) \leftarrow S(b, x)$). 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. As a consequence, less tuples are imported into *DLV*, and the ground instantiation of the MS program is reduced (wrt the original program).

V. Conclusions

ConsEx computes database repairs and consistent answers to first-order queries (and beyond) by evaluation of logic programs with stable model semantics. It also implements magic set techniques for disjunctive repair programs with program constraints [4]. *ConsEx* takes advantage of the 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, the instance is kept in the DBMS, and only the relevant data is imported into the logic programming system. *ConsEx* shows an excellent performance on query evaluation, which makes us think that CQA is viable and can be used in practical cases. In general, real databases do not contain such a high percentage of inconsistent data as those used in our experiments.

[5]The files used in the experiments are available at http://www.face.ubiobio.cl/∼mcaniupa/ConsEx

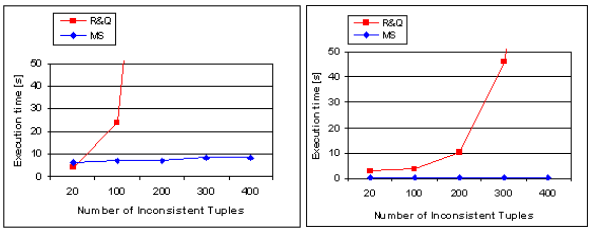

Fig. 2. Running Time for Conjunctive Queries

VI. Acknowledgments

Research supported by a 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 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. 18th ACM Symposium on Principles of Database Systems (PODS 99)*, ACM Press, 1999, pp. 68–79.

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

[3] Bravo, L. and Bertossi, L. Semantically Correct Query Answers in the Presence of Null Values. In *Pre-Proc. EDBT WS on Inconsistency and Incompleteness in Databases (IIDB 06)*, J. Chomicki and J. Wijsen (eds.), 2006, pp. 33–47.

[4] Caniupan, M. and Bertossi, L. Optimizing Repair Programs for Consistent Query Answering. In *Proc. 25th International Conference of the Chilean Computer Science Society (SCCC 2005)*, IEEE Computer Society Press, 2005, pp. 3–12.

[5] 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

[6] 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.

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

[8] 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.